Universidade do Minho
Escola de Engenharia

Vítor Emanuel Gonçalves Fernandes

Concurrent semantics for probabilistic and quantum languages

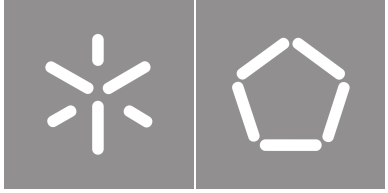
The MAP-i Doctoral Programme in Informatics, of the Universities of Minho, Aveiro, and Porto

Universidade do Minho

December, 2025

Concurrent semantics for probabilistic and quantum languages
Vítor Fernandes
UMinho | 2025

REPÚBLICA PORTUGUESA | CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR
fct Fundação para a Ciência e a Tecnologia
université PARIS-SACLAY



Universidade do Minho
Escola de Engenharia

Vítor Emanuel Gonçalves Fernandes

Concurrent semantics for probabilistic and quantum languages

The MAP-i Doctoral Programme in Informatics, of the Universities of Minho, Aveiro, and Porto



Universidade do Minho





Work developed under the supervision of:
Luís Soares Barbosa

Renato Neves

December, 2025

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:



CC BY-NC-ND

https://creativecommons.org/licenses/by-nc-nd/4.0/

Acknowledgements

This is it, the end of another chapter. This road proved to be very harsh, very difficult, almost impossible to walk. However, with the right support, everything is possible. It seems that I'm going to enumerate a lot of people. If I forget someone, please forgive me.

I would like to start by thanking my supervisors Renato Neves, Luís Soares Barbosa, and Benoît Valiron for their help and guidance. I would also like to highlight Marc de Visme's collaboration, since "from the shadows we served the light".

During my PhD, I spent most of the time in Portugal, and had the pleasure of spending a few months in France. So, I will write the following acknowledgements in Portuguese, and I'll try to write something in French as well.

Para todos os meus amigos, companheiros e colegas que me acompanharam neste percurso, nem sempre fácil, um muito obrigado. Creio que sem o vosso apoio, este caminho teria sido mais difícil do que aquilo que foi. Um obrigado aos meus velhos amigos (Peão, Pedró, Valoura, Paulinho, Castelar, Wanny, Brandão, Jota, Luís) por ouvirem todas as coisas sem sentido que ia dizendo, e continuo a dizer. Um obrigado às pessoas que conheci durante o curso e que me acompanharam até esta etapa da vida (Sequeira, Micha, Sofia, Sara, Carolina, Lucas, Paulinho, Beta, José). E depois, um obrigado às pessoas que conheci durante o doutoramento (Juliana, Diego, Alexandra, Mafalda, Nico, Nori, Bruna, Henrique, Inês, Wagner, Timmie e Elisabeth). Partilhámos muitos momentos no laboratório, bons e maus, tristes e alegres, como tudo na vida. A partilha destes momentos, em grupo, tornou este caminho cheio de pedras mais agradável de fazer.

En France, j'ai connu beaucoup de personnes qui m'ont aidé pendant une période très difficile (Benoît, Marc, Vladimir, Louis, Dong-Ho Lee, James, Tim, Renaud, Pablo et Pablo, Thea, Mathieu, Marin, Nicolas, Julien, Octave, Jérôme, Kostia). Merci pour toute la bière et le fromage, et merci de m'avoir fait découvrir l'escalade de bloc ("bouldering").

Agora vamos à famelga. Durante o período em França tive um apoio fantástico dos meus tios e primos que lá se encontram (Natália, Carlos, Paulo, Isilda, Julião, Pierre, Alexandra, Laura, Lucas, Jayden, Hayley, Mathieu, Gui, Rosa Maria, Camila). Desde plantar batatas até aos almoços de domingo, tudo serviu para arejar a cabeça e descansar. Um muito obrigado a todos.

Finalmente, quero agradecer aos meus pais (Maria e Manuel) e aos meus irmãos (Carlos e David) por me escutarem o meu silêncio. Obrigado por todo o vosso apoio. Gostaria também de agradecer aos pequenotes da família (Zé Pedro, Camila, Dinis e Eva), por me darem alegrias só por vos ver. E claro, não podia faltar a estrela da companhia, a minha afilhada Clara. Obrigado pisquena por todas as brincadeiras que fizemos e vamos continuar a fazer.

Last but not least, would like to thank INESC/HASLAB for all the support given throughout this period.

This work was financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology), under the scope of a Ph.D. Grant with reference 2020.05715.BD.

Raios de luz fintam as nuvens e iluminam este dia sombrio.
Amanhã, será diferente, mas, hoje, os nossos corações choram a tua partida.
Idália, minha avó e tua esposa, encontrarás e, de novo, vivereis o amor que vos uniu.
Melgaço, teu lar, e França, teu ganha-pão, são lugares que partilhamos.
Ulteriormente, serás recordado pela pessoa que foste e pelo que pelos outros fizeste.
Nunca esqueceremos a tua mensagem de vida.
Do teu rosto sério, olhar expressivo e sorriso contagiante emanavam respeito e carinho.
Ora bolas, lá se vão os noventa!

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, december 2025

Vítor Emanuel Gonçalves Fernandes

Abstract

This thesis explores both theoretical and practical aspects of concurrent programming languages that integrate probabilistic and/or quantum computations.

On the theoretical side, we investigate how event structures, a model for concurrency, can be used to furnish concurrent languages with denotational semantics. Event structures are studied in three distinct settings: non-deterministic, probabilistic, and quantum computation. Based on these three realms, we successively extend a core imperative-like concurrent language with non-deterministic choice, probabilistic choice, and standard quantum operations. For each extension, we define both operational and denotational semantics and establish their equivalence through soundness and adequacy results. Soundness ensures that every behavior obtainable through the operational semantics is reflected denotationally, showing that the denotational semantics correctly abstracts the computations. Conversely, adequacy guarantees that whenever the denotational semantics yields a value, this value can be produced by the operational semantics, demonstrating that the operational semantics explicitly realises what the denotational semantics predicts.

On the practical side, motivated by the growing interplay between classical and quantum computers, we integrate non-deterministic choice, probabilistic choice, and quantum operations into a single language, further extended with a synchronization primitive. We develop an interpreter for this language and test it on several illustrative examples, including three versions of the Dining Philosophers problem: concurrent-only, probabilistic, and quantum. Using our tool, we demonstrate that a result stated in the literature about the quantum Dining Philosophers problem does not hold, leaving open the question of whether an efficient algorithm for solving it exists.

Keywords Concurrency, probabilistic programming, quantum programming, program semantics, Haskell

Resumo

Esta tese explora tanto os aspectos teóricos como os prácticos de linguagens de programação concorrentes que integram computações probabilísticas e/ou quânticas.

Do ponto de vista teórico, investigámos de que forma *event structures*, um modelo para concorrência, podem ser utilizadas para equipar linguagens concorrentes com uma semântica denotacional. As *event structures* são estudadas em três contextos distintos: computação não determinística, probabilística e quântica. Com base nestes três domínios, estendemos sucessivamente uma linguagem concorrente do género imperativa com a escolha não determinística, a escolha probabilística e operações quânticas usuais. Para cada extensão, definimos tanto a semântica operacional como a semântica denotacional, e estabelecemos a sua equivalência através de resultados de *soundness* e *adequacy*. *Soundness* garante que cada comportamento obtido através da semântica operacional é refletido denotacionalmente, mostrando que a semântica denotacional abstrai correctamente as computações. Por outro lado, *adequacy* assegura que, sempre que a semântica denotacional produz um valor, esse valor pode ser obtido pela semântica operacional, demonstrando que a semântica operacional concretiza explicitamente o que a semântica denotacional prevê.

Do ponto de vista prático, motivados pela crescente interação entre computadores clássicos e quânticos, integramos a escolha não determinística, a escolha probabilística e operações quânticas numa única linguagem, posteriormente estendida com uma primitiva de sincronização. Desenvolvemos um interpretador para esta linguagem e testámo-lo com vários exemplos ilustrativos, incluindo três versões do problema dos *Dining Philosophers*: apenas concorrente, probabilística e quântica. Com o auxílio da nossa ferramenta, demonstrámos que um resultado apresentado na literatura sobre a versão quântica do problema dos *Dining Philosophers* não se verifica, deixando em aberto a questão sobre a existência de um algoritmo eficiente para o resolver.

Palavras-chave Concorrência, programação probabilística, programação quântica, semântica de programas, Haskell

Contents

1 Introduction **1**

1.1 Motivation and Context 1

1.2 Contributions 5

1.3 Structure of the thesis 7

1.3.1 Languages used throughout the thesis 10

2 Quantum computing in a nutshell **14**

I Concurrent Semantics Through the Lens of Event Structures **24**

3 Setting the stage **27**

3.1 Domain Theory: minimum requirements 27

3.2 Event Structures 28

3.3 Probabilistic Event Structures 37

3.4 Unitary Event Structures 39

3.5 Core language 42

4 Adding non-deterministic choice **46**

4.1 Language 46

4.2 Constructions on Event Structures 48

4.3 Introducing cyclic behavior 57

4.4 Soundness and Adequacy 64

5 Adding probabilistic choice **68**

5.1 Language 68

5.2 Constructions on Probabilistic Event Structures 71

5.3 Introducing cyclic behavior . . . 79
5.4 Soundness and Adequacy . . . 84

6 Adding quantum commands 88
6.1 Language . . . 88
6.2 Constructions on Unitary Event Structures . . . 91
6.3 Introducing cyclic behavior . . . 97
6.4 Unitary Event Structures with initial state . . . 100
6.5 Soundness and Adequacy (with initial state) . . . 111

7 Closing the stage 115
7.1 Related Work . . . 115
7.2 Future Work . . . 116
7.3 Conclusion . . . 118

II A concurrent (quantum) language and its interpreter 119

8 Setting the stage 122
8.1 Adding states to the core language . . . 122
8.1.1 Towards a denotational semantics with transition traces . . . 125
8.2 Equipping the core language with non-deterministic, probabilistic, and control flow commands . . . 128
8.2.1 Towards a denotational semantics with $P_{TM}V$ . . . 131
8.3 Haskell in a nutshell . . . 144

9 Adding a mechanism of synchronization 149
9.1 Extending and instantiating the core language . . . 149
9.1.1 Setting up the quantum framework . . . 151
9.2 Fully Quantum Instantiation . . . 155
9.2.1 Language and operational semantics . . . 155
9.2.2 Putting the language to test . . . 160

10 An interpreter for the concurrent (quantum) language 167
10.1 Brief description . . . 167

10.2 Implementation of the semantics . . . 174
10.3 The limits of the tool . . . 191
10.4 The interpreter at work . . . 194
10.4.1 Dining Philosophers . . . 196
10.4.2 Monty Hall . . . 200
10.4.3 Probabilistic Dining Philosophers . . . 204
10.4.4 Quantum Teleportation . . . 208
10.4.5 Grover's algorithm . . . 211
10.4.6 Superdense Coding with Grover's Algorithm . . . 213
10.4.7 Quantum Dining Philosophers . . . 214

11 Closing the stage **224**
11.1 Related Work . . . 224
11.2 Future Work . . . 227
11.3 Conclusion . . . 227

Appendix **240**

A Proofs of Part I **241**
A.1 Proofs of Chapter 4 . . . 241
A.2 Proofs of Chapter 5 . . . 270
A.3 Proofs of Chapter 6 . . . 298

B Parser details **317**
B.1 Implementation of the parser . . . 317

C Proofs of Part II **338**
C.1 Proofs of Chapter 10 . . . 338

D Data collected for testing the tool **356**

E Calculations for the Quantum Dining Philosophers **357**

List of Figures

1 Scheme of the languages used in this thesis 13

2 Matrix representation of single qubit unitary gates 19
3 Matrix representation of two-qubit unitary gates 19
4 Matrix representation of Toffoli gate 20
5 Measure scheme 21
6 Summary of Part I 26

7 Representation of $Music$, $PCOn$, and $Work$ through boxes 30
8 Representation of $Music$, $PCOn$, $Work$, $PCOff$, and $Rest$ through boxes 30
9 Drawing of an invalid event structure 33
10 Correct drawing of an event structure 33
11 Example of an event structure 35
12 Broken map of event structures 36
13 Map of event structures 37
14 Example of a probabilistic event structure 39
15 Example of a unitary event structure 42
16 Rules of the small-step operational semantics 44
17 Rules of the big-step operational semantics 44
18 Labeled transition system of $(a\,;\,b)\,||\,c$ 45

19 Rules of the small-step operational semantics 47
20 Labeled transition system of $(a\,;\,b)$ or $(c\,||\,d)$ 47
22 Event structure E_1 49
23 Event structure E_2 49
24 Parallel composition of event structures E_1 and E_2 50
25 Event structure E_1 51

26 Event structure E_2 . . . 51
27 Non-deterministic composition of event structures E_1 and E_2 . . . 51
28 Event structure of $Music \,||\, ((PCOn \,;\, Work)$ or $(PCOff \,;\, Rest))$ . . . 52
29 Event structure E_1 . . . 54
30 Event structure E'_1 . . . 54
31 Event structure E_2 . . . 54
32 Event structure $\mathsf{E}_1 \,;\, \mathsf{E}_2$ . . . 54
33 Event structure $\mathsf{E}'_1 \,;\, \mathsf{E}_2$ . . . 54
34 An event structure . . . 56
35 Event structure resultant from removing d . . . 56
36 Event structure resultant from removing b . . . 56
37 Free and bound variables of commands . . . 59
38 Unrolling the execution of $\mu X.(skip$ or $a \,;\, X)$ . . . 60
39 Unrolling the event structure of $[\![\mu X.(skip$ or $a \,;\, X)]\!]_\gamma$ . . . 64
40 Event structure of $[\![l \,;\, (a$ or $b) \,;\, c]\!]$ . . . 66
41 Event structure of $[\![l \,;\, (a$ or $b) \,;\, c]\!] \backslash l$ . . . 66
42 Event structure of $[\![(a \,;\, b) \,||\, c]\!]$ . . . 67

43 Rules of the probabilistic small-step operational semantics . . . 69
44 Rules of the big-step operational semantics . . . 70
45 Segala Automaton of $(a \,;\, b) \,+_p\, (c \,||\, d)$ . . . 71
46 Probabilistic event structure P_1 . . . 73
47 Probabilistic event structure P_2 . . . 73
48 Probabilistic event structure $\mathsf{P}_1 \,+_p\, \mathsf{P}_2$ . . . 73
49 Probabilistic event structure of $Music \,||\, ((PCOn \,;\, Work) \,+_p\, (PCOff \,;\, Rest))$ . . . 75
50 Probabilistic event structure of $(a \,+_p\, b) \,||\, c$ . . . 76
51 Probabilistic event structure of $(a \,||\, c) \,+_p\, (b \,||\, c)$ . . . 77
52 ProbES P_1 . . . 77
53 ProbES P_2 . . . 77
54 ProbES P_3 . . . 77
55 Fragment of the execution of $\mu X.(skip \,+_p\, X)$ . . . 80
56 Unrolling the probabilistic event structure $[\![\mu X.(X \,+_p\, skip)]\!]$ . . . 84

57 Rules of the small-step operational semantics . . . 90
58 Labeled transition system of $H(1)\,;\,\mathtt{M}(1, Z(1), Id(2)\,||\,X(1))$ . . . 91
59 Unitary event structure U_1 . . . 93
60 Unitary event structure U_2 . . . 93
61 Unitary event structure $\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2)$ . . . 94
62 Unitary event structure of $[\![H(1)\,;\,\mathtt{M}(1, Z(1), Id(2)\,||\,X(1))]\!]$ . . . 94
63 Fragment of the execution of $while\ \mathtt{M}(n, H(n))$ . . . 98
64 Unrolling the unitary event structure $[\![while\ \mathtt{M}(n, H(n))]\!]$ . . . 101
65 Unitary event structure U . . . 104
66 Covering chains of U . . . 105
67 Fragment of the covering chains of U . . . 105
68 Unitary event structure $\tilde{\mathsf{U}}_y$ . . . 105
69 Unitary event structure $\hat{\mathsf{U}}_y$ . . . 105

70 Rules for the simple concurrent imperative language . . . 123
71 Rules of the big-step semantics . . . 123
72 Execution tree of $\langle a\,||\,(b\,;\,c),\ s\rangle$ . . . 124
73 Rules of the operational semantics . . . 130
74 Big-step operational semantics . . . 131
75 Execution tree of $(a\ +_p\ skip)\,||\,a$ . . . 139

76 Rules for integer expressions . . . 150
77 Rules for Boolean expressions . . . 151
78 New rules added to the operational semantics from Figure 73 . . . 152
79 Rules for quantum atomic actions . . . 154
80 Rules of the small-step operational semantics . . . 159
81 Rules of the big-step semantics . . . 160
82 Execution tree of $\langle\big(X(2)\,;\,\mathtt{await}\ \ (2, \mathtt{M}(1, skip, skip))\big)\,||\,H(1),\ |00\rangle\rangle$ . . . 161
83 Code describing the behavior of Philosopher$_i$ . . . 162

84 Tool workflow in a nutshell . . . 168
85 Histogram for the variable `x1` . . . 170
86 Histogram for the variable `x2` . . . 170
87 Histogram for the variables `x1` and `x2` . . . 171

88 Unwrapping the different constructors from `StTQC a` . . . 178
89 `runSem` vs `runHist` . . . 193
90 `Tool` vs `QIO` . . . 195
91 Setting of the three philosophers . . . 196
92 Code describing the behavior of Philosopher$_i$. . . . 197
93 Monty Hall code . . . 202
94 Histogram of the Monty Hall problem where the contestant opts to change his initial guess 203
95 Histogram of the Monty Hall problem where the contestant can choose between changing or not his initial guess . . . 203
96 Code describing the behavior of Philosopher$_i$. . . . 206
97 Non-deterministic quantum teleportation protocol . . . 209
98 Histogram of the non-deterministic quantum teleportation protocol . . . 210
99 Grover's algorithm code . . . 211
100 Histogram of Grover's algorithm showing the frequency with which each possible target (i, j) was selected. . . . 213
101 Histogram of Grover's algorithm showing the frequency with which each outcome $(x1, x2)$ was measured. . . . 213
102 Superdense Coding with Grover's code . . . 214
103 Histogram of the superdense coding combined with Grover's algorithm . . . 214
104 Code describing the behavior of Philosopher$_i$ . . . 219

105 Definition of data type E . . . 317
106 Generic parser of multiplication, division, addition, and subtraction expressions . . . 322
107 Definition of data type B . . . 323
108 Definition of data type C . . . 327

List of Tables

1 Execution time of the Dining Philosophers program for each scheduler 200

2 Execution time for the Monty Hall 204

3 Execution time of the Probabilistic Dining Philosophers for each scheduler 208

4 Execution time of the Quantum Dining Philosophers for each scheduler 221

5 Amplitudes of the states that should cancel out 223

6 Execution times of BigStep and Histogram semantics. 356

7 Execution times of the tool and QIO for increasing numbers of qubits. 356

8 Amplitudes of the states that should cancel out 361

Chapter 1

Introduction

1.1 Motivation and Context

Concurrency is ubiquitous. Nowadays, mobile phones, tablets, computers, cars, and many other everyday devices relay on it. For example, modern smartphones or in-car systems allow users to listen to music while instructions are giving to drive to some destination. At the hardware level, the processes responsible for these tasks are continuously interrupted and resumed at high speed, creating the illusion of simultaneous execution.

Probability, like concurrency, is all around us, whether in the uncertainty of natural phenomena or in the devices we use in our daily lives. For instance, in quantum mechanics, the position of a particle is known only with a certain probability rather than as an exact value. Large Language Models (LLMs) such as ChatGPT, Gemini, and Grok are also built upon probabilistic foundations. Similarly, text prediction, *i.e.* suggesting word completions or predicting the next word to write, is also based on probabilistic principles. Probability is also often combined with concurrency, where *e.g.* it plays a fundamental role in breaking the symmetry of concurrent programs. Although symmetry is crucial in the physical world, it is undesirable in concurrent programs, as it often leads to deadlocks – *i.e.* states in which programs cease to make progress.

At last, but not least, we have quantum computation. Inspired by Richard Feynman's insight [40] [1] that classical computers cannot efficiently simulate quantum systems, quantum computing has gained theoretical and practical attention. Quantum algorithms such as Grover's [48], to perform searches in unstructured data, and Shor's [97], providing an efficient way to factorize prime numbers, demonstrate notable speedups over classical counterparts. While general-purpose quantum computers remain under development, major companies including IBM, Google, Microsoft, IonQ, and Quantinuum are actively com-

[1] "Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy." [40]

peting to build scalable quantum machines. Furthermore, quantum communication protocols, such as BB84 [13], already offer enhanced security guarantees over classical methods. This shows that combining quantum effects with concurrency can lead to robust and secure systems.

Concurrent and probabilistic systems already impact our daily lives. Quantum systems are expected to have similar impact in the near future, not necessarily in the form of quantum laptops, which are unlikely due to physical constraints, but in high-impact domains such as drug discovery, optimization, and secure communication. Given the influence these systems have, or are projected to have, on our society, it is essential to develop methods that allow reasoning formally about their behavior. The current state of quantum computers was already lived by classical computers, which motivated by the need of rigor led to the development of the field of *formal methods*. Formal methods is concerned with the specification, analysis, and verification of software and hardware systems using rigorous mathematical techniques. At the heart of formal methods is the notion of *semantics*, which gives non ambiguity meaning to programs and supports precise reasoning about their behavior.

The word *semantics*, as we know it today, comes from the Ancient Greek *sema*, meaning "sign", and its relative adjective *semantikos*, meaning "significant". Making use of redundancy, the meaning of semantics is, quite literally, "to give meaning". In computer science, semantics was first used by Floyd [41] who attempted to interpret programs from a logical point of view. In this thesis, we use the following approaches to semantics:

- Operational Semantics: Describes how programs are executed on a computer, by introducing a set of rules that specify how a program should be evaluated. Operational semantics is usually defined via: a small-step approach (which describes individual computation steps) and a big-step approach (which describes the overall evaluation result).

- Denotational Semantics: Interprets programs as mathematical objects, such as functions, sets, traces, or vector spaces, abstracting away from implementation details. Because programs are represented as mathematical objects, we obtain, for free, a natural notion of program equivalence, inherited from the underlying mathematical structures used in their interpretation.

Semantics is not only useful to give meaning to programs, but also to verify if a program behaves as expected. This becomes particularly important in languages that include computational effects such as concurrency, probability, and quantum, due to their increased complexity. A feature of concurrent programs is that they can interact with one another, and this interaction makes it very difficult to predict their outcomes. For example consider the concurrent execution of an atomic action a together with a

sequential composition of two atomic actions b and c, *i.e.* $a \,||\, (b\,;\, c)$. The possible execution paths are: first execute a, then b and c, or first execute b, then c and then a or first execute b then execute a and then execute c. In this last execution path we see that the computation of the sequential command was interfered with the execution of the atomic action a. Although only demonstrated with a simple program, it is straightforward to deduce that the number of interactions may increase exponentially with program's size. The output of probabilistic programs is, as expected, probabilistic. For example consider a language supporting a probabilistic choice operator, $C_1 \,+_p\, C_2$ such that C_1 has probability p to occur while C_2 has probability $1 - p$ of happening. By applying $a \,+_p\, b$ to an initial state v yields two possible final states: av with probability p, and bv with probability $1 - p$. In the case of quantum programs, where superposition and entanglement are unique features, measurements cause a quantum state to collapse into one of several possible outcomes. For example, measuring a qubit in a superposition state like $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ results in $|0\rangle$ or $|1\rangle$, each with probability 0.5. Furthermore, entanglement between qubits can propagate across processes, making it harder to predict the global behavior of the system. Since concurrent, probabilistic, and quantum computation combined is so important, it is only natural that a corresponding unified semantics is developed.

Concurrent semantics is usually explained from two different paradigms: interleaving and "true concurrency". The former aims to sequentialize the concurrent execution of programs, while the latter seeks to establish a notion of causality between the actions performed along a computation. For example, given two atomic actions a and b, the interleaving paradigm interprets their concurrent execution as either executing a followed by b, or executing b followed by a. On other hand, the true concurrency paradigm just asserts that a and b are not causally related. In the interleaving paradigm, examples of semantic models include resumptions [49, 90, 108] and transition traces [88, 21]. Both models were used to furnish imperative shared-variable languages with a denotational semantics [49, 88, 21]. In the true concurrency paradigm, event structures [115, 117, 118] are a well-known model, which has been used to furnish process algebras with a denotational semantics. An event structure can be seen as a partial order together with a conflict relation, where conflicts behave analogously to a non-deterministic choice operator in programming languages. That is, if a and b are events in conflict, then either a or b can occur in a computation.

When combining probability with concurrency, it is necessary to develop models that accommodate both effects. On the operational side, a well-known model is Segala automata [95], whose main feature is that non-deterministic choices are resolved before probabilistic ones. On the denotational side, several models have been proposed, most of which rely on probabilistic powerdomains [59, 106, 73, 74].

These models belong to the interleaving paradigm, and, as far as we know, the only works that provide a shared-variable language with both a probabilistic and a concurrent operator equipped with a denotational semantics are those by Claire Jones [59], Renato Neves [80], and Noam Zilberstein [123]. At the core of these works lies the resumptions model. Within the true concurrency paradigm, event structures have been extended to the probabilistic setting [61, 110, 119]. However they are mainly used as foundations for more complex theories, such as game semantics [87]. Nevertheless, we should also mention the work of Marc de Visme [26], who modified Winskel's definition of probabilistic event structures to furnish a probabilistic extension of CCS [9] featuring concurrency, non-deterministic choice, and probabilistic choice.

An advantage of developing models that account for both concurrency and probability is to open the door to modeling concurrent quantum languages. This is because, in a quantum language, measurements give rise to probabilities, as we have already seen. Hence, it is expected that some of the previously mentioned models can be used as a denotational semantics for a concurrent quantum language. An example of such a model is the one developed in [80] (and, as far as our knowledge goes, it is the only one). Although it is well known that measurements give rise to probabilistic behavior, it is also possible to hide such probabilities so that measurements appear as non-deterministic actions. When this happens, concurrent probabilistic models may no longer be adequate to interpret concurrent quantum languages, and new models need to be defined. This is the case of event structures, where Winskel introduced quantum event structures [119]. However, similarly to probabilistic event structures, quantum event structures are mainly used as the backbone of more complex theories [25].

The development of operational and denotational semantics is essential for reasoning about programs. For simple programs, such reasoning can be made by hand. However, as program complexity increases, verifying every behavior manually becomes intractable. For instance, a concurrent system composed of N processes, each executing M distinct atomic actions can have up to $\frac{(N\times M)!}{(M!)^N}$ different possible states (if we let $N = 3$ and $M = 2$, there are 90 possible states). Hence, it is necessary to develop tools that are not only practical and executable but also grounded in formal semantic models. Such tools can simulate programs faithfully, support verification, and serve as experimental platforms for language design. This ensures that the tool's behavior reflects the theoretical model and can be trusted for debugging, validation, and reasoning.

There already exist several tools that follow the approach suggested above, *i.e.* tools based on formal semantics. Examples include mCRL2 [47], CADP [39], PRISM [64], and QWire [89], among others. The first two focus exclusively on verifying the behavior of concurrent systems. PRISM combines probability

with concurrency, enabling the verification of randomized concurrent programs. Finally, QWire focuses solely on verifying the behavior of quantum programs. In addition, there has also been work on developing tools that combine concurrency with quantum computation. Examples include the implementation of an interpreter for the LanQ language [75], and the implementation in Maude of a concurrent quantum language [55].

Despite the progress made in the development of these tools, none of them considers concurrent commands, non-deterministic choice, probabilistic choice, and quantum commands within a unified framework. This creates a gap between theory and practice, in which the richness of formal semantics is not fully exploited in tool development.

This thesis aims to fill these gaps by: (i) using event structures and their variants as the primary model to furnish imperative-like languages with a denotational semantics and (ii) developing a tool, based on a formal semantics, that unifies concurrent commands, non-deterministic choice, probabilistic choice, and quantum commands within a single framework.

1.2 Contributions

This thesis contributes both theoretically and practically to the study of mixing concurrency with probabilistic and quantum computation.

On the theoretical side, we expand the use of event structures, a true-concurrent model used for reasoning about concurrency, to serve as a denotational semantics for imperative-like languages across three distinct computational paradigms: non-deterministic, probabilistic, and quantum. We start with a core concurrent language and, for each of the mentioned paradigms, we extend the core language with non-deterministic choice, probabilistic choice, and quantum commands composed of unitary gates and measurements. For each extension, we provide a denotational semantics based on the appropriate form of event structures. We then prove soundness, showing that behaviors observable in the operational semantics are captured by the denotational semantics, and adequacy, showing that behaviors represented in the denotational semantics can be observed through the operational semantics. In the quantum setting, we restrict Winskel's definition of quantum event structures, leading to unitary event structures. This refinement enables us to extend a result obtained by Winskel [119, Theorem 3], which established that quantum event structures without conflicting events and furnished with an initial quantum state correspond to probabilistic event structures. We show that this correspondence still holds, even in the presence of conflicting events, provided we use unitary event structures. This work was published in the proceedings

of "The 22nd Asian Symposium on Programming Languages and Systems" [38].

On the practical side, we present a tool developed in collaboration with a M.Sc. student (Inês Dias [31]), co-supervised by the author of this thesis. The language and operational semantics are based on [80]. In that work, the language featured generic atomic actions, as well as non-deterministic choice and probabilistic choice. To implement the language in our tool, we instantiate these actions as deterministic assignments, unitary gates, measurements, and qubit resets, thereby obtaining a unified framework that accommodates both classical and quantum atomic operations. We also extend the instantiated language with a classical synchronization primitive, the $await$ command [21], together with its corresponding rule in the small-step operational semantics. The big-step semantics employs schedulers to resolve the nondeterminism arising from concurrent computations.

Using our tool, and to the best of our knowledge, we implemented and simulated for the first time the efficient quantum solution to the Dining Philosophers problem developed by Aharonov [5]. We performed this simulation using the $await$ command to ensure that no two philosophers pick up the same "quantum chopstick" simultaneously. The efficient solution in [5] is claimed to be deterministic, *i.e.* the problem is guaranteed to be solved in finite time.[2] This solution relies on a simplification of the quantum leader election protocol [104], where the symmetry-breaking step is executed only once, and the resulting outcome determines which fork each philosopher picks up. However, when we simulated the algorithm with three philosophers, we observed that symmetry was not broken, yielding a non-zero probability of deadlock. This contradicts the claim made in [5, Theorem 3], which asserts that the algorithm deterministically solves the quantum Dining Philosophers problem efficiently. A manuscript for this work is currently being prepared.

During the PhD, a collaboration with Manisha Jain, another PhD student arose. The results of this collaboration can be summarized as follows:

- Adding concurrency to Quantum Dynamic Logic [54]
  - In this paper we introduced a quantum concurrent language with the $await$ primitive, taking into account only quantum variables, together with a corresponding quantum dynamic logic. To the best of our knowledge, this was the first time the $await$ primitive was studied in the quantum setting. We also developed some simple examples to illustrate how the logic can be used to reason about properties of concurrent quantum programs.
- Exploring a quantum programming language with concurrency [55]

[2] In [5], Aharonov considers two types of solutions: the *randomized* solution, which solves the problem with some probability (possibly even with probability 1), and the *exact* (or deterministic) solution, which always solves the problem in finite time, for every execution.

- This paper extends the work presented in [54]. We begin by modifying the syntax of the measurement command and its operational semantics, and then we animate the modified language in Maude. In [54], the measurement command is written as $M_b(n)$, meaning "measure qubit n under basis b". For instance, if b is the computational basis, *i.e.* $\{|0\rangle, |1\rangle\}$, then measuring qubit n collapses it to either $|0\rangle$ or $|1\rangle$. A different view is adopted in [55], where measurement is expressed akin to a classical if-then-else, $\mathtt{M}(n, C_1, C_2)$. Here, the measurement is performed in the computational basis and, if the outcome is $|0\rangle$, we execute the command C_1, otherwise C_2 is executed. Using the Maude animation of the modified language, we encoded well-known quantum algorithms in the language, executed them, and verified that they behave as expected. In addition, we applied the $await$ constructor in various examples to prevent undesired behaviors arising from concurrency.

- A quantum programming language with concurrency and its dynamic logic [56]

  - This paper is an extended version of [54]. It builds on the language introduced in [55] and presents its associated quantum dynamic logic. The introduction of the synchronization primitive, *i.e.* the $await$ command, is motivated by a simplified version of the quantum Dining Philosophers problem. The manuscript has been submitted to a journal.

1.3 Structure of the thesis

The thesis is divided into two parts and it is constituted by eleven chapters, with two of them being independent of the two parts. Chapter 1 introduces the reader to this thesis. It does so by presenting the problems we tackled and the motivation behind them. Furthermore, this chapter also summarizes our contributions and the publications that originated from this work, as well as the languages used throughout the thesis. Chapter 2 provides a brief overview of the essential concepts related to quantum computation. This part is intended for readers who are not familiar with the subject. The next two parts constitute our main contributions. Due to their importance, they are presented with greater rigor and detail.

- Part I approaches how to furnish concurrent languages, extended with different commands (non-deterministic choice, probabilistic choice, and quantum commands), with a denotational semantics based on event structures. This part consists on five chapters:

  - Chapter 3

We introduce the necessary background for the work developed in this part. Section 3.1 gives a minimal overview of domain theory. We then present event structures in their different realms: non-deterministic (Section 3.2), probabilistic (Section 3.3), and quantum (Section 3.4). The chapter concludes with Section 3.5, which introduces a core language that, as its name suggests, serves as the foundation for this thesis. It is around this core language that our contributions are built.

- Chapter 4

  In Section 4.1, we extend the core language with a non-deterministic command, followed by the corresponding extension of the operational semantics. Section 4.2 defines how the various operators of the language are represented using event structures, ending up with a denotational semantics based on event structures. Section 4.3 explores recursion and updates both the operational and denotational semantics accordingly. The chapter concludes with results of soundness and adequacy in Section 4.4.

- Chapter 5

  Following a similar structure to the previous chapter, Section 5.1 extends the core language with a probabilistic command and adapts the operational semantics to the probabilistic setting. Section 5.2 defines the operators of the language as probabilistic event structures, leading to a denotational semantics for the extended language. Section 5.3 revisits cyclic behavior and updates the operational and denotational semantics. The chapter ends with Section 5.4, which presents the results of soundness and adequacy.

- Chapter 6

  In Section 6.1, we extend the core language with quantum commands and update the operational semantics. Section 6.2 defines the operators of the extended language in terms of unitary event structures. Section 6.3 explores cyclic behavior through a while loop and updates the operational and denotational semantics. Section 6.4 extends [119, Theorem 3], which connects quantum event structures with an initial state to probabilistic event structures. This part concludes with Section 6.5, which presents the results of soundness and adequacy, both with and without initial states.

- Chapter 7

  This chapter concludes Part I. Section 7.1 presents related work, Section 7.2 talks about future work, and Section 7.3 sums up the work developed in this part.

- Part II includes a synchronization primitive in a concurrent (quantum) language [80], and builds a Haskell interpreter for it. This part is constituted by four chapters:

  - Chapter 8

    We begin by introducing the necessary background for this part. Section 8.1 adds states to the core language and updates the operational semantics accordingly. Section 8.2 extends the core language with non-deterministic, probabilistic, and control-flow commands, leading to the language described in [80], and presents its operational semantics. Section 8.3 provides a minimal introduction to Haskell, as well as to the mtl [45] package.

    The following sections are part of an exploratory work. Section 8.1.1 presents the transition traces model [88, 21], used to define a denotational semantics for the core language with states. Section 8.2.1 extends transition traces to the probabilistic setting and attempts to provide a denotational semantics for the language in [80], without cyclic behavior.

  - Chapter 9

    Section 9.1 extends the language from Section 8.2 with a synchronization command and instantiates the extended language with deterministic assignments. Section 9.1.1 makes explicit the quantum operations considered (unitary gates, measurement, and reset). Section 9.2 presents a fragment of the work developed in collaboration with Manisha Jain, more specifically the contribution of the author of this thesis, which consists in the design of the language (adding quantum commands to the core language) and the operational semantics adopted for that work.

  - Chapter 10

    Section 10.1 provides an overview of the tool, complemented by Section 10.2, which details the implementation of the semantics. Section 10.3 discusses the tool's limitations, and Section 10.4 concludes this chapter and this part by executing various examples using the developed tool.

  - Chapter 11

    This chapter concludes Part II. In Section 11.1 we present related work, Section 11.2 talks about future work, and Section 11.3 sums up the work developed in this part.

1.3.1 Languages used throughout the thesis

Throughout this document we work with several different languages. For convenience, we summarize them here, list their syntactic forms, and indicate where each one appears. Note that only one of these languages (point 8 in the following enumeration) does not consider cyclic behavior. Despite this, we focus solely on the commands associated with non-deterministic, probabilistic, and quantum effects.

1. Core Language

   The core language, introduced in Section 3.5, is:

   $$C ::= skip \mid a \in Act \mid C\,;C \mid C \,||\, C$$

   In its essence, the core language is a fragment of an imperative language [92, Section 2], where we consider atomic actions instead of assignments, extended with a concurrency operator. Since event structures are suitable to model languages without the notion of state, the operational semantics of the core language ignores states and is given as a labeled transition system, where transitions are labeled by the triggering actions.

   The remaining languages introduced in this thesis all extend, in different ways, this core language.

2. Core Language extended with non-deterministic choice

   Adding non-deterministic choice yields the language of Section 4.1:

   $$C ::= skip \mid a \in Act \mid C\,;C \mid C \,||\, C \mid C \,\mathsf{or}\, C$$

   Its operational semantics is again a labeled transition system. We then define a denotational semantics for this language using (prime) event structures [115]. Event structures allow us to explicitly manipulate causality and conflict, thus being appropriate for furnishing this language with a denotational semantics.

3. Core Language extended with probabilistic choice

   Adding probabilistic choice yields the language of Section 5.1:

   $$C ::= skip \mid a \in Act \mid C\,;C \mid C \,||\, C \mid C \,+_p\, C$$

   where $p \in]0,1[$. This language combines concurrency with probabilities. As a result, labeled transition systems are no longer adequate, and we use labeled Segala automaton instead. The

rationale behind Segala automaton is that non-deterministic transitions are resolved before probabilistic ones. Similarly to the non-deterministic expansion, here we give a denotational semantics based on probabilistic event structures [119, Definition 1].

4. Core Language extended with quantum commands

   Extending the core language with quantum commands, originates the language shown in Section 6.1:

   $$C ::= skip \mid U(\tilde{n}) \mid C\,;\, C \mid C \,||\, C \mid \mathtt{M}(n, C_1, C_2)$$

   Here, the operational semantics is again a labeled transition system. This entails that measurements, although inherently probabilistic, are treated as non-deterministic transitions. Following the same spirit as previously, we define a denotational semantics for this language using unitary event structures (see Definition 3.4.1).

5. Core Language with states

   The quantum extension above (point 4) is the final language considered in the context of event structures (Part I). Furthermore, the operational semantics considered in those languages were stateless. In order to build an interpreter that assists us in the reasoning of complex programs, we add states to the operational semantics of the core language in Section 8.1. The syntax is the same as the core language in point 1.

   As a side note, Park [88] and Brookes [21] furnish a related language (core language with states) with a denotational semantics based on transition traces.

6. Core Language with states extended with non-deterministic, probabilistic, and quantum commands

   By adding non-deterministic choice, probabilistic choice, and quantum commands, we obtain the language considered in [80], used in Section 8.2:

   $$C ::= skip \mid a \in Act \mid C \;+_p\; C \mid C\,;\, C \mid C \,\mathsf{or}\, C \mid C \,||\, C$$

   where $p \in [0,1] \cap \mathbb{Q}$. Since deterministic assignments and quantum commands are treated as atomic actions, we keep them under the hood, *i.e.* we do not instantiate them in the syntax. The operational semantics is described in terms of a Segala automata and uses schedulers to resolve nondeterminism.

   This language was then used to perform a preliminary study: to extend transition traces with probabilities and to furnish this language with a denotational semantics based on probabilistic transition traces.

7. Extending the previous language (point 6) with an await command

   With the addition of the await command to the language in [80], we also found useful to include an atomic constructor (Section 9.1), which schedulers use to prioritize the commands wrapped by it. The resulting language is:

   $$C ::= skip \mid a \in Act \mid C \ +_p \ C \mid C \,;\, C \mid C \,\mathsf{or}\, C \mid C \,||\, C \mid \texttt{await}\ \ b\ \ \texttt{do}\ \ \{C\} \mid \texttt{Atom}(C)$$

   This language is then instantiated (atomic actions are considered to be deterministic assignments, unitary gates, measurements, and qubit reset), and a Haskell interpreter is built for this instantiation.

8. Core language with states extended with non-deterministic choice, quantum commands, and await

   This language (Section 9.2.1), although not the last to be considered in the thesis, is the last to be shown here. It extends the core language with states by adding non-deterministic choice, quantum commands, and the await command. Furthermore, this is the only language that does not take into consideration cyclic behavior.

   $$C ::= skip \mid U(\vec{n}) \mid \texttt{M}(n, C_1, C_2) \mid C_1 \,;\, C_2 \mid C_1 \,\mathsf{or}\, C_2 \mid C_1 \,||\, C_2 \mid \texttt{await}\ \ (n, D)$$
   $$D ::= skip \mid U(\vec{n}) \mid \texttt{M}(n, D_1, D_2) \mid D_1 \,;\, D_2$$

   The operational semantics is based on a transition system, where transitions caused by measurements are seen as non-deterministic transitions instead of probabilistic ones. Hence, we can ignore the use of schedulers. Because of this design choice, the commands allowed inside the await construct (the grammar for D) are restricted. In contrast, in the language of point 7, any command is permitted inside `await`, with the scheduler responsible for resolving the resulting nondeterminism.

Figure 1 summarizes these languages in a visual fashion.

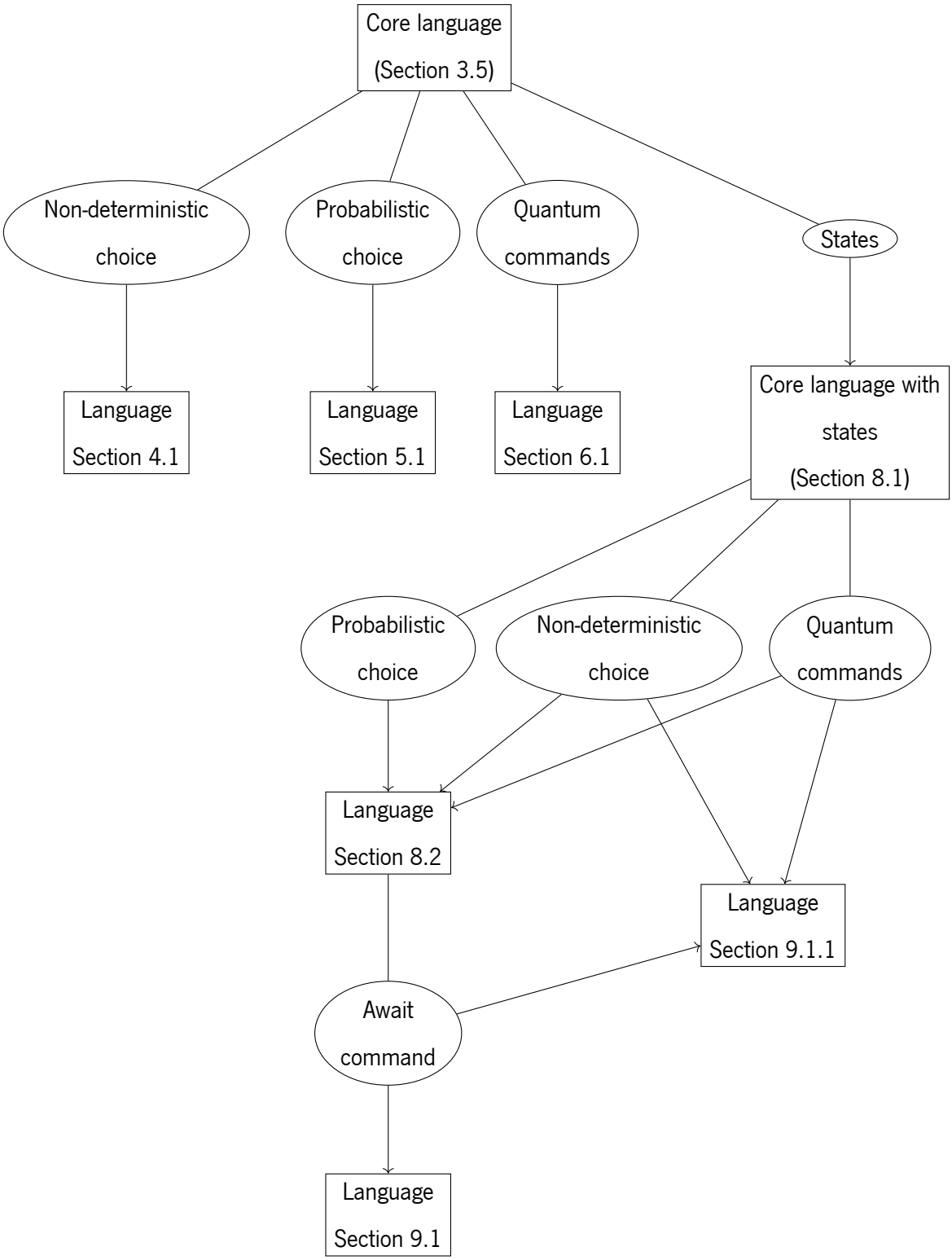


Figure 1: Scheme of the languages used in this thesis

Chapter 2

Quantum computing in a nutshell

Quantum computing is an emerging field in computer science that uses the laws of quantum mechanics to solve computational problems. Inspired by Richard Feynman's famous remark [40]:

> "Nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy."

quantum computing has evolved from a theoretical concept into an active area of research and technological development. Its hype is no surprise: quantum computers promise substantial improvements both in modeling physical systems (useful in chemistry, for instance, when searching for molecules with desirable properties) and in identifying patterns or structures in complex data (as in biological applications such as protein folding).

Unlike classical computers, which use *bits* as the basic unit of information, quantum computers use *qubits*. While a classical bit is either 0 or 1, a qubit can be in state 0, 1, or any superposition of these two states. This capability enables quantum computers to perform certain tasks more efficiently than their classical counterparts. For example, the Deutsch algorithm can determine whether a function $f : \{0,1\} \to \{0,1\}$ is balanced ($f(0) \neq f(1)$) or constant ($f(0) = f(1)$) using only a single evaluation, whereas a classical algorithm requires two [27]. When qubits are in superposition, they are "simultaneously" in all states contributing to that superposition. However, upon observation we obtain only one classical outcome. This occurs because observing the system causes the quantum state to collapse to one of the basis states with a probability determined by its amplitudes (complex numbers associated with each state). This process is known as *measurement*. Quantum computers also exploit a counter-intuitive phenomenon called *entanglement*. Entangled qubits share correlations that cannot be explained by classical physics: measuring one qubit yields information about the other, even when they are physically separated.

Another important aspect of quantum computing concerns global and relative phases. A qubit lives in a complex vector space, and measurement probabilities depend only on the squared absolute values

of its amplitudes. As a result, multiplying a state by an overall complex phase $e^{i\theta}$ does not change any observable quantities, and such global phases are ignored. In contrast, relative (local) phases between components of a superposition are physically meaningful: they determine how amplitudes combine, enabling constructive or destructive interference. Interference is a fundamental resource in many quantum algorithms, such as Grover's search algorithm [48], where it is used to amplify the probability of obtaining a desired result.

In this chapter, we review a few basic concepts of quantum computation introduced above. For readers interested in a deeper understanding of quantum computation, a recommended source is [82].

Dirac's Notation

Dirac [33] introduced a notation that is used in quantum mechanics: a vector is written as a *ket*, $|\cdot\rangle$, and its conjugate transpose, denoted with †, is written as a *bra*, $\langle\cdot|$. Due to their names, this notation is also known as the *braket* notation.

A *ket* can be represented by a column vector

$$|\psi\rangle = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$$

and the corresponding *bra* by a row vector, specifically the complex conjugate of $|\psi\rangle$.

$$\langle\psi| = |\psi\rangle^\dagger = [c_1^* \cdots c_n^*]$$

where c^* is the complex conjugate of c.

Using the *braket* notation, the inner product of two vectors $|\psi\rangle$ and $|\phi\rangle$ is expressed as

$$\langle\psi|\phi\rangle = [c_1^* \cdots c_n^*] \cdot \begin{bmatrix} d_1 \\ \vdots \\ d_n \end{bmatrix} = \sum_{i=1}^{n} c_i^* \cdot d_i$$

Hilbert Space

The mathematical framework underlying quantum computing and quantum information theory is that of a *Hilbert space* over the complex numbers, $\mathbb{C}$, as discussed in [112].

A finite-dimensional Hilbert space [1], denoted as $\mathcal{H}$, is a vector space equipped with an inner product. The inner product is a mapping from $\mathcal{H} \times \mathcal{H} \to \mathbb{C}$ satisfying the following properties:

[1] In this thesis we work with finite-dimensional Hilbert spaces. Thus, we avoid the additional technicalities that arise in the infinite-dimensional setting.

1. $\langle\psi|\psi\rangle \geq 0$ for any $|\psi\rangle \in \mathcal{H}$, with equality iff $|\psi\rangle = 0$

2. $\langle\phi|\psi\rangle = \langle\psi|\phi\rangle^*$

3. $\langle\phi|\lambda_1\psi_1 + \lambda_2\psi_2\rangle = \lambda_1\langle\phi|\psi_1\rangle + \lambda_2\langle\phi|\psi_2\rangle$

where $\lambda_1, \lambda_2 \in \mathbb{C}$ and $|\phi\rangle, |\psi\rangle, |\psi_1\rangle, |\psi_2\rangle$ are vectors in $\mathcal{H}$. A vector space equipped with an inner product is called *inner product space*.

Any two vectors, $|\psi\rangle$ and $|\phi\rangle$, are *orthogonal* if their inner product equals zero:

$$\langle\psi|\phi\rangle = 0$$

The norm of a vector is:

$$|| \, |\psi\rangle \, || = \sqrt{\langle\psi|\psi\rangle}$$

When the norm is equal to one the vector is said to be a *unit vector*.

An *orthonormal basis* in a Hilbert space $\mathcal{H}$ is a set of unit vectors $|i\rangle$ which are pairwise orthogonal and span the whole space $\mathcal{H}$ (every vector can be written as a linear combination of the basis vectors). The dimension of a Hilbert space is the number of vectors that compose a basis.

Tensor Product

The *tensor product*, denoted by $\otimes$, is a mathematical operation that combines vector spaces. For example, if one considers a vector space V with dimension n and another vector space W with dimension m, the tensor product of these two vector spaces, $V \otimes W$, is another vector space whose dimension, $dim(V \otimes W)$, is equal to $n \times m$. In this new vector space, a vector corresponds to a linear combination of the tensor product of vectors from V and W, $|v\rangle \otimes |w\rangle = |v \otimes w\rangle = |vw\rangle$. Notably, if $|i\rangle$ is a basis in V and $|j\rangle$ is a basis in W, then $|i \otimes j\rangle$ is a basis for $V \otimes W$. The tensor product has the following properties:

1. For an arbitrary scalar z and elements $|v\rangle$ of V and $|w\rangle$ of W,

$$z(|v\rangle \otimes |w\rangle) = (z|v\rangle) \otimes |w\rangle = |v\rangle \otimes (z|w\rangle)$$

2. For arbitrary $|v_1\rangle$ and $|v_2\rangle$ in V and $|w\rangle$ in W,

$$(|v_1\rangle + |v_2\rangle) \otimes |w\rangle = |v_1\rangle \otimes |w\rangle + |v_2\rangle \otimes |w\rangle$$

3. For arbitrary $|v\rangle$ in V and $|w_1\rangle$ and $|w_2\rangle$ in W,

$$|v\rangle \otimes (|w_1\rangle + |w_2\rangle) = |v\rangle \otimes |w_1\rangle + |v\rangle \otimes |w_2\rangle$$

The tensor product applies in particular to matrices, and in this specific context is known as the *Kronecker product*. Consider two matrices, A and B, with dimensions $m \times n$ and $p \times q$, respectively. The Kronecker product of these two matrices is:

$$A \otimes B = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \otimes \begin{bmatrix} b_{11} & \cdots & b_{1q} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pq} \end{bmatrix} =$$

$$= \begin{bmatrix} a_{11} \begin{bmatrix} b_{11} & \cdots & b_{1q} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pq} \end{bmatrix} & \cdots & a_{1n} \begin{bmatrix} b_{11} & \cdots & b_{1q} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pq} \end{bmatrix} \\ \vdots & \ddots & \vdots \\ a_{m1} \begin{bmatrix} b_{11} & \cdots & b_{1q} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pq} \end{bmatrix} & \cdots & a_{mn} \begin{bmatrix} b_{11} & \cdots & b_{1q} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pq} \end{bmatrix} \end{bmatrix}$$

The dimension of the resulting matrix is $m \times p$ by $n \times q$.

Qubits

Qubits are to quantum computation what bits are to classical computation. Concretely, qubits are the most elementary form of information in quantum computation. Formally, a single qubit is described via the two-dimensional Hilbert space, denoted $\mathcal{H}^2$. The set of vectors $\{|0\rangle, |1\rangle\}$ forms the so-called *computational basis*, where

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The general state of a single qubit is obtained as a linear combination of the basis vectors:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where $\alpha, \beta \in \mathbb{C}$ are known as probability amplitudes and the equation $|\alpha|^2 + |\beta|^2 = 1$ holds. The values $|\alpha|^2, |\beta|^2$ are the probabilities of measuring $|0\rangle$ or $|1\rangle$, respectively. Furthermore, whenever both

$\alpha \neq 0$ and $\beta \neq 0$, we say that the qubit is in a *superposition state*. This is a fundamental difference between quantum computation and classical computation, since bits cannot be in a superposition state.

If we consider a system of n qubits, its state space is given by the tensor product of the Hilbert spaces:

$$\mathcal{H}^{2^n} = \bigotimes_{i=1}^{n} \mathcal{H}^2$$

For example, a two-qubit system has a Hilbert space $\mathcal{H}^{2\times 2}$ and a basis given by $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. A general state can then be written as:

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$$

where $\alpha, \beta, \gamma, \delta \in \mathbb{C}$, $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$, and $|\alpha|^2,\ |\beta|^2,\ |\gamma|^2$ and $|\delta|^2$, are the probabilities of measuring $|00\rangle,\ |01\rangle,\ |10\rangle$ and $|11\rangle$, respectively.

The states represented by multiple qubits are denominated either as *separable states* or as *entangled states*. A separable state can be written as a linear combination of tensor products. For example:

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$$

Entangled states, in contrast, cannot be expressed in this form. An example of an entangled state for a two-qubit system is the Bell state:

$$|\beta_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Unitary Operators

A *unitary operator* respects the following equation:

$$U^\dagger U = UU^\dagger = Id$$

where $U^\dagger$ is the conjugate transpose (also called *Hermitian conjugate*) of U, and Id is the identity operator.

In closed quantum systems, *i.e.* systems that do not interact with the environment, the evolution of a state is described via a unitary operator. For instance, if a system evolves from an initial state $|\psi\rangle$ to a final state $|\phi\rangle$ under a unitary operator U, this evolution is expressed as: $U(|\psi\rangle) = U|\psi\rangle = |\phi\rangle$.

For single qubits, common unitary operators include the *Pauli operators* (Id, X, Y, Z), the *Hadamard gate* (H), and the *phase gates* ($Ph(\phi)$). For two-qubit systems, we have the *SWAP* gate and various *controlled gates*, such as the *controlled-NOT* ($CNOT$), the *controlled-Z* (CZ), and the *controlled-phase shift* ($CPh(\phi)$). Finally, for three-qubit systems, the *Toffoli gate* (TOF), also known as the *controlled-CNOT* ($CCNOT$), is a notable example. Figures 2, 3, and 4 show the matrix representations of these unitary gates.

$$Id = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad Z = Ph(\pi) = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$S = Ph\left(\frac{\pi}{2}\right) = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \qquad Ph(\phi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix} \qquad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Figure 2: Matrix representation of single qubit unitary gates

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

$$CPh(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\phi} \end{bmatrix}$$

Figure 3: Matrix representation of two-qubit unitary gates

Unitary gates can also be represented via outer products. For instance:

$$X = |0\rangle\langle 1| + |1\rangle\langle 0|, \qquad CNOT = |0\rangle\langle 0| \otimes Id + |1\rangle\langle 1| \otimes X$$

The equational forms of CZ and $CPh(\phi)$ are similar to that of $CNOT$. They can be obtained by replacing X with Z and $Ph(\phi)$, respectively.

Let us now briefly describe the behavior of these gates. The identity gate, as its name suggests, leaves the state unchanged. The X gate acts as a logical negation: it flips $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$. The Y gate performs a similar transformation, mapping $|0\rangle$ to $i|1\rangle$ and $|1\rangle$ to $-i|0\rangle$. The Z, S, and $Ph(\phi)$ gates are *phase gates*, which add a phase factor whenever the qubit state is in the state $|1\rangle$. The Hadamard gate creates superposition between the states $|0\rangle$ and $|1\rangle$, *i.e.* $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. Furthermore, the Hadamard gate can also destroy the superposition, since by unitarity $HH|0\rangle = H^\dagger H|0\rangle = |0\rangle$.

$$TOF = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 4: Matrix representation of Toffoli gate

The $SWAP$ gate changes the position of two qubits, for instance $|10\rangle \mapsto |01\rangle$.

The controlled gates act conditionally on one (or more) control qubits, modifying a target qubit when the control conditions are met. Since the behavior of the controlled gates is similar, only changing the gate to be applied whenever the condition is met, let us consider the $CNOT$ gate. When applying a $CNOT$ to a two-qubit system, we consider the first qubit as the control qubit and the second qubit as the target qubit. If the control qubit is in state $|0\rangle$, the target qubit remains unchanged. Otherwise, if the control qubit is in state $|1\rangle$, the target qubit is changed. As an example, consider the effect that a $CNOT$ has on a general two-qubit state $|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$:

$$CNOT|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|11\rangle + \delta|10\rangle$$

At last, the Toffoli gate is a controlled-CNOT, *i.e.* it applies an X gate to the target qubit only when both control qubits are in the state $|1\rangle$.

Let us now see how single qubit gates are applied to multi-qubit systems. Consider we have a two-qubit system, $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, and we want to apply a X gate on the first qubit and a Z gate on the second. This is achieved using the tensor product:

$$\begin{aligned} X \otimes Z|\psi\rangle &= (X \otimes Z)\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\ &= (X \otimes Z)\frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle) \\ &= \frac{1}{\sqrt{2}}[(X \otimes Z)|0\rangle \otimes |0\rangle + (X \otimes Z)|1\rangle \otimes |1\rangle] \end{aligned}$$

$$= \frac{1}{\sqrt{2}}[X|0\rangle \otimes Z|0\rangle + X|1\rangle \otimes Z|1\rangle]$$
$$= \frac{1}{\sqrt{2}}(|10\rangle - |01\rangle)$$

Measurement

A *measurement* is an operation that allows us to extract information from a quantum system. For example, we previously saw that a single qubit can be in a superposition state, *i.e.* $\alpha|0\rangle + \beta|1\rangle$. In such a situation, it is not known whether the qubit is in state $|0\rangle$ or state $|1\rangle$. To determine the state of the qubit, we must perform a measurement. By doing so, the system collapses into one of the two possible states: with probability $|\alpha|^2$, it collapses to $|0\rangle$, and with probability $|\beta|^2$, it collapses to $|1\rangle$.

More formally, a measurement is defined by a collection of *measurement operators* $\{M_m\}$, where m labels the possible measurement outcomes. These operators must satisfy the *completeness relation*:

$$\sum_m M_m^\dagger M_m = Id$$

The probability of m being the measurement outcome of a state $|\psi\rangle$ is given by:

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle$$

The state of the system after the measurement is:

$$\frac{M_m|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}}$$

To better understand how a measurement works, consider the following example.

Example 2.0.1. In Figure 5 we have a two-qubit state $|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$ and all possible outcomes when first measuring the left qubit and then the right qubit.

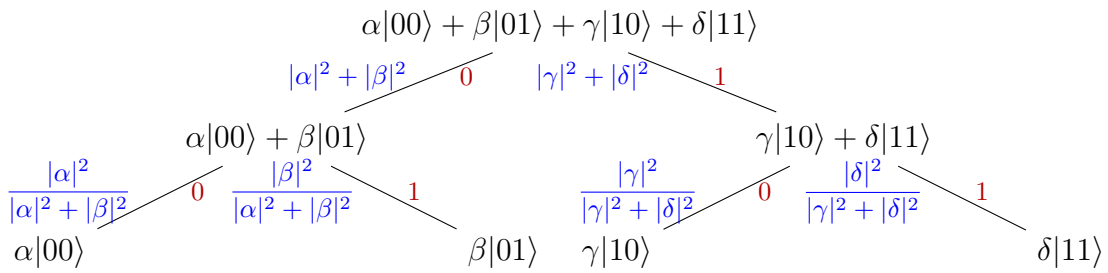


Figure 5: Measure scheme

In Figure 5, the measurement outcomes are shown in red, and their respective probabilities are shown in blue. Note that the states appearing after each measurement are not normalized.

Density Operators

Another way to represent quantum states is by using *density operators* (also called *density matrices*).

Let us consider we have at our disposal n qubits. While a pure state $|\psi\rangle$ is represented by a state vector, a density operator, denoted by ρ, is written as an *ensemble of pure states*, $\{p_i, |\psi_i\rangle\}_{i=1}^n$, as follows:

$$\rho = \sum_{i=1}^{n} p_i |\psi_i\rangle\langle\psi_i|$$

where p_i is the probability of the system being in the pure state $|\psi_i\rangle$.

A density operator can represent both pure and mixed states. To distinguish between them, the following criterion is used:

1. $tr(\rho^2) = 1 \ \rightarrow$ pure states
2. $tr(\rho^2) < 1 \ \rightarrow$ mixed states

where tr is the trace of a $N \times N$ square matrix (*i.e.* the sum of its diagonal elements):

$$tr(A) = \sum_{i=1}^{N} A_{ii}$$

Another fact about density operators is that one density operator can represent different state vectors, as shown in the following example.

Example 2.0.2. Consider two different state vectors $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|\phi\rangle = -|\psi\rangle$. Now let us construct the density operator of both state vectors. For $|\psi\rangle$ the density operator is $\rho_1 = \frac{1}{2}(|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + |1\rangle\langle 1|)$ and for $|\phi\rangle$ the density operator is $\rho_2 = \frac{1}{2}(|0\rangle\langle 0| + |0\rangle\langle 1| + |1\rangle\langle 0| + |1\rangle\langle 1|)$. Here, we see that $\rho_1 = \rho_2$, which means that two different state vectors can be represented by the same density operator.

This example also shows that density operators ignore *global phases*. Note that the only difference between $|\psi\rangle$ and $|\phi\rangle$ is a global phase factor.

Formally, a density operator is defined as follows:

Definition 2.0.3. An operator ρ is the density operator associated to some ensemble $\{p_i, |\psi_i\rangle\}_{i=1}^n$ iff it satisfies the conditions:

1. $tr(\rho) = 1$

2. ρ is a positive operator

where a positive operator is an operator with non-negative eigenvalues.

Whenever a positive operator ρ has a trace less than or equal to one, *i.e.* $tr(\rho) \leq 1$, we call it *partial density operator*.

A unitary operator U acts on a density operator ρ as follows:

$$U\rho U^\dagger$$

Since a density operator is expressed as an ensemble of pure states, the action of a unitary operator is written as:

$$U\rho U^\dagger = U\sum_i p_i|\psi_i\rangle\langle\psi_i|U^\dagger = \sum_i p_i U|\psi_i\rangle\langle\psi_i|U^\dagger$$

After a measurement, $M_m^\dagger \rho M_m$, the probability of m being the measurement outcome is given by:

$$p(m) = tr(M_m^\dagger M_m \rho)$$

The state of the system after the measurement is:

$$\frac{M_m^\dagger \rho M_m}{tr(M_m^\dagger M_m \rho)}$$

We finish this chapter by defining the notion of *partial trace*. Recall that a quantum system can be composed of smaller subsystems, represented by the tensor product of their respective Hilbert spaces. If we are interested in extracting information only from one subsystem (without measuring the entire system), we can perform a *local measurement* using the partial trace. For instance, let ρ be a density operator of a system composed of two subsystems A and B, such that $\rho = \rho_A \otimes \rho_B$. The partial trace over the subsystem B is defined by:

$$tr_B(|a_1\rangle\langle a_2| \otimes |b_1\rangle\langle b_2|) = |a_1\rangle\langle a_2|tr(|b_1\rangle\langle b_2|)$$

where $|a_1\rangle$ and $|a_2\rangle$ are any two vectors in the state space of A, and $|b_1\rangle$ and $|b_2\rangle$ are any two vectors in the state space of B. Furthermore, the partial trace should be linear in its input.

Part I

Concurrent Semantics Through the Lens of Event Structures

Prologue

Event structures are defined in three realms: non-deterministic, probabilistic, and quantum.

We will start with a core concurrent language. Then for each realm, we add a corresponding command to the core language.

We first consider event structures in the non-deterministic realm, extending the core language with a non-deterministic command. We define both small-step and big-step operational semantics, where transitions are labeled by the actions that trigger them. Then, we define sequential, concurrent, and non-deterministic composition in terms of event structures and give a denotational semantics by interpreting such commands in this model. Next, we enrich the language with cyclic behavior and develop the corresponding operational and denotational semantics. Finally, we prove equivalence between the two semantics, *i.e.* we show that words (sequences of triggered actions obtained via big-step semantics) correspond to covering chains (finite sequences of events representing computational steps).

Next, we move to the probabilistic realm, where we introduce the probabilistic choice, $C_1 +_p C_2$. Operationally the latter means we execute C_1 with probability p and C_2 with probability $1 - p$. We follow the same methodology as in the non-deterministic case: defining operational semantics (small-step and big-step), giving a denotational semantics in terms of probabilistic event structures, enrich the language with cyclic behavior, and proving equivalence. In this case equivalence is established by showing that words and covering chains corresponding to each other carry the same probability.

We then proceed to the quantum realm, where atomic actions are considered to be unitary gates and we add a measurement command, $\mathtt{M}(q, C_1, C_2)$ that behaves similarly to an if-then-else statement: if the measurement of q returns 0 then we execute C_1, otherwise we execute C_2. In this part, we restrict Winskel's definition of quantum event structures [119, Definition 2]. This allows us to extend a result previously obtained by Winskel. We call the restricted structures "unitary event structures". We then repeat the same steps as before: defining operational and denotational semantics, including cyclic behavior, and proving equivalence between the semantics (the equivalence between semantics is similar to the one formulated in the non-deterministic realm). As mentioned before, the restriction of Winskel's quantum event structures allows us to extend a result previously stated by Winskel: a quantum event structure without events in conflict and with an initial state is a probabilistic event structure. With unitary event structures, the previous result can be stated as: unitary event structures with an initial state is a probabilistic event structure.

Figure 6 (read from left to right) summarizes Part I. *Lang*, *pLang*, and *qLang* are the languages

considered in Section 4.1 (non-deterministic realm), Section 5.1 (probabilistic realm), and Section 6.1 (quantum realm), respectively. *LTS* and *LSA* stand for Labeled Transition System and Labeled Segala Automaton, respectively.

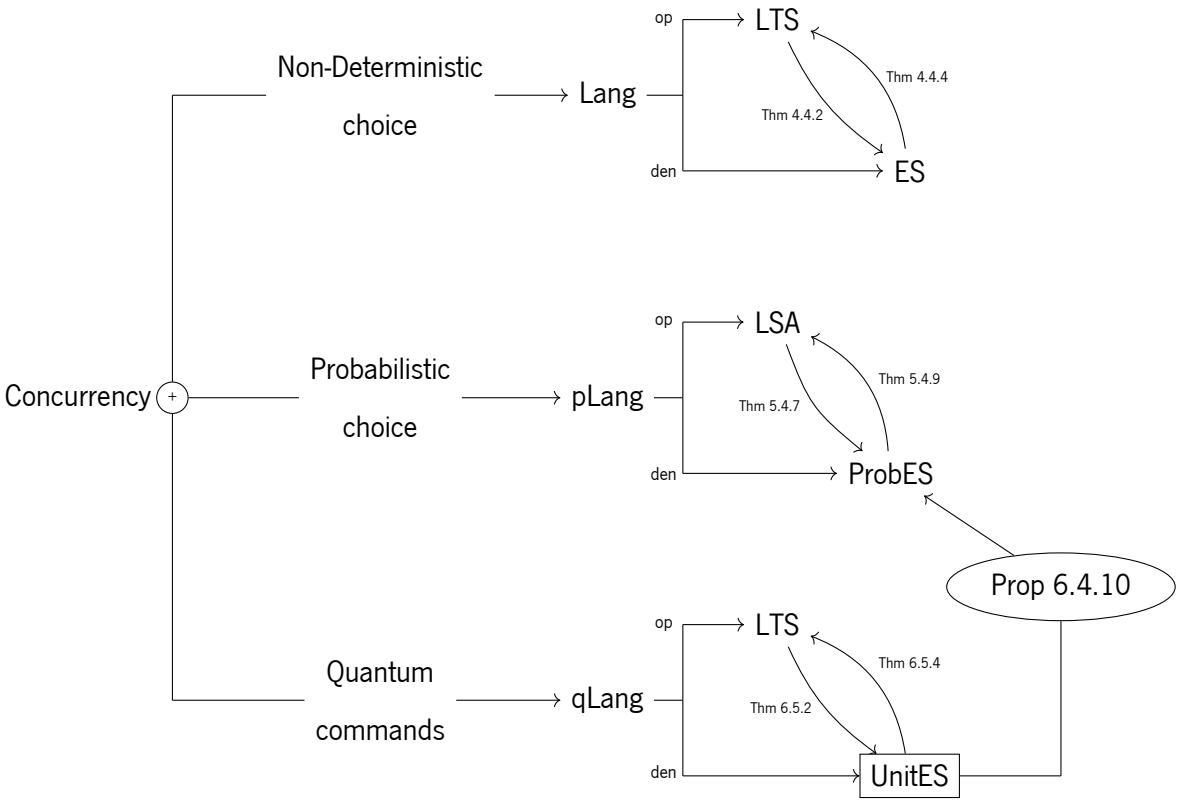


Figure 6: Summary of Part I

Chapter 3

Setting the stage

3.1 Domain Theory: minimum requirements

Domain Theory was introduced by Dana Scott to provide a denotational semantics for the λ-calculus. A central idea in Domain Theory is to interpret recursive programs using fixpoints. In this section, we present only the concepts necessary for our purposes. We refer the interested reader to standard references such as [3, 44] for a more detailed treatment.

We begin with the notion of a partial order.

Definition 3.1.1. Let X be a set equipped with a binary relation $\leq$. The pair $(X, \leq)$ is called a partially ordered set (or poset) if the following conditions hold for all $x, y, z \in X$:

$$x \leq x \qquad \text{(Reflexivity)}$$

$$x \leq y \text{ and } y \leq z \Rightarrow x \leq z \qquad \text{(Transitivity)}$$

$$x \leq y \text{ and } y \leq x \Rightarrow x = y \qquad \text{(Antisymmetry)}$$

In a poset X, an element $x \in X$ is called the least element (or bottom element, denoted by $\perp$) if for all $y \in X$, we have $x \leq y$.

Let X be a poset and $Y \subseteq X$. An element $x \in X$ is said to be an upper bound of Y if $y \leq x$ for all $y \in Y$. If x is the smallest element among all upper bounds of Y, it is called the least upper bound (or supremum) of Y, and is denoted by $\bigsqcup Y$.

A subset $Y \subseteq X$ is called a chain if for all $x, y \in Y$, either $x \leq y$ or $y \leq x$. If the chain is isomorphic to the natural numbers, we refer to it as an ω-chain.

Let $(X, \leq)$ and $(Y, \leq)$ be posets. A function $f : X \to Y$ is called monotonic if for all $x, y \in X$ with $x \leq y$, we have $f(x) \leq f(y)$.

A subset $Y \subseteq X$ is called directed if it is non-empty and every finite subset of Y has an upper bound in Y. The least upper bound of a directed set Y is often denoted by $\bigsqcup^{\uparrow} Y$.

A directed complete partial order (or dcpo) is a poset D in which every directed subset has a least upper bound. Given dcpo's D and E, a function $f : D \to E$ is said to be continuous if it is monotonic and, for every directed subset $X \subseteq D$, we have

$$f\left(\bigsqcup^{\uparrow} X\right) = \bigsqcup^{\uparrow} f(X).$$

The following result enables the interpretation of recursive behavior via fixpoints [92, Proposition 2.5]:

Proposition 3.1.2. Let D be a dcpo with a least element $\bot$, and let $f : D \to D$ be a continuous function. Then

$$x = \bigsqcup_{n=0}^{\infty} f^n(\bot)$$

is the least fixpoint of f.

In other words, this result tells us that we can construct the least fixpoint of a continuous function by iteratively applying f starting from the bottom element:

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdots \sqsubseteq f^n(\bot) \sqsubseteq \cdots$$

The least upper bound of this ascending chain is the least fixpoint of f.

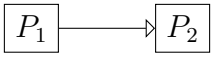
3.2 Event Structures

Event structures are a model for concurrent systems developed by Winskel in the 1980s [115]. They focus on describing the relations between events in a concurrent system. In this model, events can be causally related, in conflict, or concurrent. Although event structures capture the relations between events, they do not specify which events can occur together in a computation. Hence, it is necessary to define *configurations* alongside an event structure. Configurations represent valid sets of events that occur in a computation. Informally, configurations consist of events that are either causally dependent or concurrent, but never in conflict.

As the name suggests, event structures focus on *events*. Events are the elementary components of a broader entity known as a *process*, which describes how something works. Here, we alert the reader that the level of abstraction we adopt significantly influences our interpretation. For instance, when describing the process of a person getting ready for work, listening to music may be considered a single elementary action. However, if we focus specifically on listening to music, we can break it down further: choosing a device, selecting a song, or picking an album or artist. Thus, depending on the level of abstraction, an

event may be viewed as either a fundamental unit of a process or a process itself. For simplicity, we shall consider the former.

We now explore different ways processes can be composed. Since processes are built from events, process composition principles also apply to events. To visualize processes, we represent them as boxes. These processes can be combined using sequential, concurrent, or non-deterministic operators. Sequential composition is depicted by two boxes connected by an arrow, representing a causal dependency. Specifically, if we have two processes, P_1 and P_2, and an arrow points from P_1 to P_2, this means that P_2 can only occur after P_1 has taken place.



If we replace the arrow with a wiggly line, we non-deterministically compose the two boxes, indicating that the process can execute P_1 or P_2. In other words, during the execution, the process must choose one to execute, while discarding the other.



At last, when there is nothing between the two boxes, they are composed in parallel. Moreover, since we are in a true-concurrent setting, the two boxes are executed "simultaneously" during the execution of the process.



To create more complex processes, we can combine different composing methods. Let us explore an example to illustrate this. Imagine a person is listening to his favorite music while turning on his computer to start another day at work. This process describes the actions involved and their relations in getting ready to work. Specifically, the process consists of three events: listening to music, turning on the computer, and starting to work. Listening to music is independent of both turning on the computer and working, *i.e.* the person can listen to music while turning on the computer or while working. However, the person must first turn on the computer to start working, establishing a relationship between starting to work and turning on the computer. This relationship is *causal*, because without turning on the computer, the person cannot start working. Now, let us represent this process using boxes (see Figure 7):

Note that the $Work$ box is also in parallel with the $Music$ box, which aligns with our earlier explanation: listening to music and working are events that can occur simultaneously. If we transform this process into a program and run it on a computer, the computer will perform this program in one of the three possible ways:

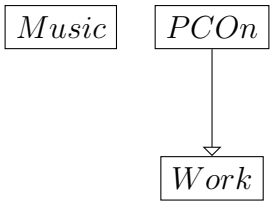


Figure 7: Representation of $Music$, $PCOn$, and $Work$ through boxes

1. $Music\,;\, PCOn\,;\, Work$
2. $PCOn\,;\, Music\,;\, Work$
3. $PCOn\,;\, Work\,;\, Music$

In the first possibility, we start by listening to music, turn on the computer, and finally begin working. In the second possibility, we switch the order of listening to music and turning on the computer. In the last possibility, the final action is starting to listen to music (perhaps to avoid falling asleep while working). We refer to these possibilities as *computations*. Furthermore, note that this corresponds to an interleaving interpretation of the program.

Now, let us add the possibility of being lazy. In this case, we consider the option of leaving the computer turned off to rest. We then add two more events to our example: turning off the computer and resting. This is depicted, in terms of boxes, in Figure 8:

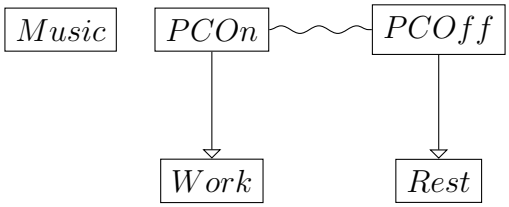


Figure 8: Representation of $Music$, $PCOn$, $Work$, $PCOff$, and $Rest$ through boxes

There is one important point to highlight. The boxes $Work$ and $Rest$ are not connected by a wiggly line. This is because the conflict is inherited from the previous level. In other words, if we turn on the computer, we will work and cannot rest. So, turning on the computer prevents resting. On the other hand, if we keep the computer turned off, we will rest but cannot work. Therefore, working becomes an invalid option when the computer is off. So, in order to obtain the possible computations for this new process, we need to add three new computations to the previous ones:

4. $Music\,;\, PCOff\,;\, Rest$
5. $PCOff\,;\, Music\,;\, Rest$

6. $PCOff\,;\,Rest\,;\,Music$

Note that in all the computations, the actions of turning on the computer and working do not appear in the same computations as keeping the computer turned off and resting appear.

Now, the reader might ask how everything we discussed about the boxes relates to event structures. First, event structures can be drawn, by illustrating the different relations (causal, conflict, and concurrent) between events, as shall be seen later. We then use this to discuss the relation with what we have covered so far. As previously stated, event structures consist of events that can be causally related, in conflict, or concurrent. If events are causally related, they are connected by an arrow. If they are in conflict, there is either a wiggly line between them, or the conflict is inherited from events that were previously in conflict. Finally, if events are concurrent, there is nothing between them, *i.e.* they are neither causally related nor in conflict.

Informally, an event structure [118] is composed of a set of events, together with a notion of causality given by a partial order on events (if $e \leq e'$ then e' can only occur if e already occurred), and a notion of conflict between events (if $e\#e'$ then either e occurs or e' occurs, which is a behavior similar to a non-deterministic choice). An alternative perspective on event structures is seeing it as a partial order together with a conflict relation.

Based on the informal description, let us delve into the formal details of event structures and related concepts. The concepts of causality and conflict between events are at the core of the definition of an event structure. In its simplest definition, both concepts are captured by a binary relation that satisfies certain conditions. In the literature, this type of event structure (Definition 3.2.1) is called a *prime event structure*. This is the definition we will use, as it allows us to identify events that are causally related or in conflict directly. We also want to point out that by modifying the definitions of causality or conflict, we can derive new but equivalent definitions of event structures [115, 116, 118, 117]. Throughout this document, we will use the term "event structures" rather than "prime event structures", only resorting to the latter when necessary.

Definition 3.2.1 (Event Structures)**.** Define an *event structure* to be a structure E $= (E, \leq, \#)$ consisting of a set E of events, which are partially ordered by $\leq$, the *causal dependency relation*, and a binary, symmetric, irreflexive relation $\# \subseteq E \times E$, the *conflict relation*, satisfying:

1. $\{e' \mid e' \leq e\}$ is finite
2. $(e\#e') \wedge (e' \leq e'') \Rightarrow e\#e''$

for all $e, e', e'' \in E$.

Intuitively, Condition 1 tells that if an event e is to occur, it must occur after a finite number of events. This is equivalent to saying that the set of events that e causally depends on must be finite. Condition 2 indicates that the conflict relation is inherited. Concretely, if two events e and e' are in conflict, and an event e'' causally depends on e', then e'' must also be in conflict with e.

When giving the intuition of an event/process, we used arrows and wiggly lines to represent sequential and non-deterministic compositions, respectively. However, in the definition of event structures, we use $\leq$ and #. So what is the relation between arrows and wiggly lines with $\leq$ and #, and why do we use the former with drawings instead of the latter? Since the causal relation is a partial order, we know it is transitive. Furthermore, the conflict relation is hereditary over events. Hence, if we were to represent an event structure using $\leq$ and #, we would have to add redundant information, making the drawing of the event structure hard to understand. To simplify this, we find it helpful to introduce some known concepts [26]: *immediate causality*, represented by $\rightarrowtriangle$, and *minimal conflict*, represented by $\sim\!\!\!\cdot$. Let E be an event structure such that $e_1, e_2, e'_1, e'_2 \in E$. We say:

- $e_1 \rightarrowtriangle e_2$ iff $e_1 < e_2$ and $\nexists e_3 \in E \,.\, e_1 < e_3 < e_2$
- $e_1 \sim\!\!\!\cdot\, e_2$ iff $e_1 \# e_2$ and whenever $e'_1 \leq e_1$, $e'_1 \# e'_2$, and $e'_2 \leq e_2$ we have $e_1 = e'_1$ and $e_2 = e'_2$

Note that the causal relation is the reflexive and transitive closure of the immediate causality relation. Furthermore the conflict relation can be derived from the minimal conflict relation. The intuition behind these relations is that the immediate causality relation represents events that are causally related, with no other events between them. Since the conflict relation is inherited, the minimal conflict aims to identify the source of the conflict. In other words, it helps to identify where the conflict between events starts.

In Definition 3.2.1, we saw that events can be causally related or in conflict. However, we also said that events can be concurrent. Informally, concurrent events are neither causally related nor in conflict. From this informal view, it becomes clearer why boxes composed in parallel had nothing between them. Formally, concurrent events are defined as follows:

Definition 3.2.2 (Concurrent Event)**.** Two events, e, e' are said to be *concurrent* iff

$$\neg(e \leq e') \wedge \neg(e' \leq e) \wedge \neg(e \# e')$$

A consequence of using prime event structures is that the causal history of each event is unique [118]. Consequently, if a given action a can be triggered in different ways, then, in terms of event structures, a will be represented by several distinct events corresponding to the different ways in which it can occur. The next example illustrates this situation.

Example 3.2.3. Consider a scenario where an action c can be activated in two distinct and incompatible ways: either by first executing a or by first executing b. A first attempt to draw this scenario as an event structure would be as Figure 9 (the name of events is the same as the name of the actions, a convention we shall use).

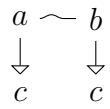


Figure 9: Drawing of an invalid event structure

However, Figure 9 does not represent a valid event structure because of how event c is handled. This event causally depends on a and b, which are conflicting events. According to Definition 3.2.1, the conflict relation is inherited. As a result, c should be in conflict with itself. However, an event cannot be in conflict with itself. To solve this problem, a copy for each possible way c is activated is made. In this case, since c is activated in two different ways, we shall have two copies, c_1 and c_2, as depicted in Figure 10.

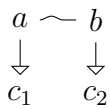


Figure 10: Correct drawing of an event structure

In general, whenever an event is activated in different ways, a copy for each possible causal history of the event is made. This is particularly important when cyclic behaviors are considered, since the same actions are repeated, but its events are all different.

Earlier, it was mentioned that event structures can also capture the notion of a computation. To do this, event structures have the notion of *configurations*, which intuitively are subsets of the set of events where no events are in conflict. Furthermore, if an event e' is in the configuration, then all the events that e' causally depends on must also be in the configuration. Formally:

Definition 3.2.4 (Configuration)**.** A configuration is a subset of the set of events, $x \subseteq E$, that are

conflict-free: $\forall e, e' \in x \;.\; \neg(e \# e')$

down-closed: $\forall e, e' \;.\; e' \leq e \wedge e \in x \Rightarrow e' \in x$

We then denote by $\mathcal{C}(\mathrm{E})$ the set of finite configurations [1].

[1] We do not consider infinite event structures, whose set is denoted by $\mathcal{C}^{\infty}(\mathrm{E})$, since in this part of the thesis, we are only concerned with computations that finish, which are represented by finite configurations

Since a configuration is a subset of the set of events, the empty set is also a configuration. It represents the absence of any action or the beginning of a computation. From the empty set, we can begin constructing configurations by adding one event at a time, then adding another event to the previous set, and so on, until no more events can be added. This way of building configurations is known as *covering chain*, which we formally define next:

Definition 3.2.5 (Covering chain)**.** Let E $= (E, \leq, \#)$ be an event structure, $e \in E$, and $x \in \mathcal{C}(\mathsf{E})$. Denote by $x \xrightarrow{e}\!\!\!\!\!\subset x \cup \{e\}$ if $e \notin x$ and $(x \cup \{e\}) \in \mathcal{C}(\mathsf{E})$. A *covering chain* on a configuration $x \in \mathcal{C}(\mathsf{E})$ is a finite sequence of events $e_1 e_2 \ldots e_n$ such that

$$\emptyset \xrightarrow{e_1}\!\!\!\!\!\subset x_1 \xrightarrow{e_2}\!\!\!\!\!\subset x_2 \xrightarrow{e_3}\!\!\!\!\!\subset \ldots \xrightarrow{e_n}\!\!\!\!\!\subset x_{n+1} = x$$

Especially when working with probabilistic event structures, we found it helpful to use a notation similar to that of a covering chain. This notation allows us to say that a configuration *covers* another configuration, *i.e.* a configuration is "inside" another. Formally:

Definition 3.2.6 (Cover)**.** Let E be an event structure and $x, y \in \mathcal{C}(\mathsf{E})$. Say that y covers x, pictured as $x \multimap\!\!\!\!\subset y$, if $x \subset y$ with nothing in between ($\nexists z \,.\, x \subset z \subset y$).

When writing a cover, $y \multimap\!\!\!\!\subset x_1, \ldots, x_n$ (read as $y \multimap\!\!\!\!\subset x_1$, ..., $y \multimap\!\!\!\!\subset x_n$), it is sometimes useful to explicitly enumerate the events added to the configuration y. For that we use the notation $y \xrightarrow{e_1,\ldots,e_n}\!\!\!\!\!\subset x_1, \ldots, x_n$ whenever $y \multimap\!\!\!\!\subset x_1, \ldots, x_n$. Furthermore, when showing the equivalence between the operational and denotational semantics, we simplify the representation of covering chains. We let the sequence of events $e_1 e_2 \ldots e_n$ be denoted by ω, *i.e.* $\omega = e_1 e_2 \ldots e_n$ and denote $\emptyset \xrightarrow{e_1}\!\!\!\!\!\subset x_1 \xrightarrow{e_2}\!\!\!\!\!\subset x_2 \xrightarrow{e_3}\!\!\!\!\!\subset \ldots \xrightarrow{e_n}\!\!\!\!\!\subset x_{n+1} = x$ by $\emptyset \xrightarrow{w}\!\!\!\!\!\subset x$.

If on one side of the spectrum we have the beginning of a computation, represented by the empty configuration, then on the other side we have a finished computation, represented by a *maximal configuration*. Intuitively, a configuration x is maximal if it does not cover any other configuration. Formally:

Definition 3.2.7 (Maximal configuration)**.** Let E be an event structure and $x \in \mathcal{C}(\mathsf{E})$. Say that x is a maximal configuration iff $\nexists y \in \mathcal{C}(\mathsf{E})$ such that $x \multimap\!\!\!\!\subset y$. Denote by $\mathcal{C}_{\mathsf{max}}(\mathsf{E})$ the set of maximal configurations.

An interesting property of maximal configurations is that two distinct maximal configurations have events in conflict. This is useful when showing the validity of sequential composition's definition in the probabilistic realm (Lemma 5.2.2).

Lemma 3.2.8. Let E be an event structure and $x, x' \in \mathcal{C}_{\max}(\mathsf{E})$. If $x \neq x'$ then $\exists e \in x, e' \in x'$ such that $e\#e'$.

Proof. Let $x, x' \in \mathcal{C}_{\max}(\mathsf{E})$ such that $x \neq x'$.

Pick $e \in x \backslash x' = \{e \in x \mid e \notin x'\}$ and let $M = \{a \in E \mid a \leq e \text{ and } a \notin x'\}$ be the set of events that e causally depends on and not in x'. Clearly we have $M \neq \emptyset$, since $e \in M$. By Definition 3.2.1, $\{e' \mid e' \leq e\}$ is finite. Thus we can choose a minimal in M with respect to $\leq$, *i.e.* $\forall a' < a \,.\, a' \notin M$ (in other words, choose the first event a that e causally depends on that is not in x'). Then every $a' \in E$ such that $a' < a$ is in x', *i.e.* $a' \in x'$. Thus, $x' \cup \{a\}$ is downward closed. Since x' is maximal, $x' \cup \{a\}$ cannot be a configuration, hence it must fail conflict-freeness: $\exists e' \in x'$ such that $a\#e'$. By hereditary of conflict, $a \leq e$ and $a\#e'$, we have $e\#e'$. □

Let us delve into another example to consolidate the concept of a configuration and its related notions.

Example 3.2.9. Figure 11 shows an event structure with four events, a, b, c, and d, where: b causally depends on a, c and d are concurrent events which are in conflict with a, and consequently also with b. The set of configurations, *i.e.* the set of possible computations, is $\{\emptyset, \{a\}, \{c\}, \{d\}, \{a, b\}, \{c, d\}\}$, with $\{a, b\}$ and $\{c, d\}$ being maximal configurations. Note that the configuration $\{c, d\}$, which is composed of two concurrent events, has two possible covering chains: $\emptyset \xrightarrow{d} \{d\} \xrightarrow{c} \{c, d\}$ and $\emptyset \xrightarrow{c} \{c\} \xrightarrow{d} \{c, d\}$. Moreover, $\{b\}$ is not a configuration because it is not down-closed, *i.e.* b causally depends on a but $a \notin \{b\}$.

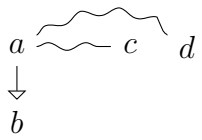


Figure 11: Example of an event structure

The last concept we will introduce in this section is that of a map of event structures, which can be total or partial. Its usefulness will be demonstrated when showing that unitary event structures with an initial state correspond to probabilistic event structures (Section 6.4). Intuitively, a map of event structures $f : \mathsf{E} \to \mathsf{E}'$ tells how the occurrence of an event e in E implies the occurrence of $f(e) = e'$ in E′. To be more precise, if x in E, *i.e.* $x \xrightarrow{e} x \cup \{e\}$ then $f[x] \xrightarrow{f(e)} f[x \cup \{e\}]$ in E′, whenever $f(e)$ is defined. In words, this means that if an event e can be added to a configuration x to form a new configuration $x \cup \{e\}$ in E, then the same behavior holds in E′ when $f(e)$ is defined. Formally, a map of event structures is defined as follows:

Definition 3.2.10 (Map event structures)**.** Let E $= (E, \leq, \#)$ and E′ $= (E', \leq', \#')$ be event structures. A partial/total map f from E to E′ is a partial/total function $f : E \rightharpoonup E'$ such that:

(Configuration Preserving) $\forall x \in \mathcal{C}(\mathsf{E}) \Rightarrow f[x] \in \mathcal{C}(\mathsf{E}')$

(Locally injective) $\forall (a \neq b) \in x \in \mathcal{C}(\mathsf{E}),$ if f is defined in both then $f(a) \neq f(b)$

where $f[x] = \{f(e) \mid e \in x,\ f(e)$ defined$\}$

The preserving configuration condition is self-explanatory. It suffices to recall what was previously said, *i.e.* that if an event e is added to a configuration x to form a new configuration $x \cup \{e\}$ in E, then the same behavior holds in E′, when $f(e)$ is defined. The locally injective condition assures that no two distinct events, under the same configuration, are mapped to the same image event. This prevents concurrent events, or causally dependent events from being mapped to the same event. To understand why this condition is necessary, let us explore an example illustrating a spurious map between event structures.

Example 3.2.11. Consider the event structures in Figure 12 linked by a broken map of event structures.

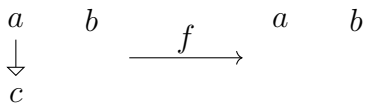


Figure 12: Broken map of event structures

The function f maps c to a and applies the identity on a and b. Let us denote the event structure on the left by E and the one on the right by E′. The set of configurations of E is $\{\emptyset, \{a\}, \{b\}, \{a, c\}, \{a, b\}, \{a, b, c\}\}$ and the set of configurations of E′ is $\{\emptyset, \{a\}, \{b\}, \{a, b\}\}$.

It is not difficult to check that the configurations of E are preserved. However, the map becomes invalid because it does not respect the local injective condition. Let us consider two specific configurations from E: $\{a, b\}$ and $\{a, c\}$. The local injective condition is satisfied for $\{a, b\}$, since a and b belong to $\{a, b\}$ and are mapped to themselves. On the other hand, the configuration $\{a, c\}$ fails to satisfy the local injective condition, since a and c are mapped to a, *i.e.* causally related events are being mapped to the same image event. Thus, different events are mapped to the same event, which contradicts the locally injective condition.

Another way to understand the local injective condition is by looking at the size of the configurations, especially when the map is total. If the map is total, the size of the configurations must remain the same. In other words, if a configuration has n events in E, the corresponding configuration in E′ must also have n events.

We now develop an example that shows a valid map of event structures.

Example 3.2.12. In Figure 13, we have a map of event structures f that maps a to itself and the conflicting events c and b to d. The map f is total, and it is valid because it preserves configurations and obeys local injectivity. Each configuration in E is mapped to a configuration in E′, and distinct events within the same configuration are mapped to distinct events. Furthermore, since f is total, we notice that the size of covering chains are preserved. Consider the covering chain $\emptyset \xrightarrow{c} \subset \{c\} \xrightarrow{a} \subset \{a, c\}$. The respective covering chain after applying f is $\emptyset \xrightarrow{d} \subset \{d\} \xrightarrow{a} \subset \{a, d\}$. This is only possible because the local injective condition ensures that different events in the same configuration must map to distinct events.

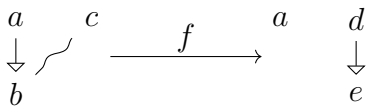


Figure 13: Map of event structures

Although the event structure on the right has an event e that is not in the image of f, this does not invalidate the map. The reason is that e is not causally required by any event in the image of f.

In Section 3.3 and Section 3.4 we will see how versatile event structures are.

3.3 Probabilistic Event Structures

This section describes how to endow event structures with a probabilistic behavior. Probabilistic event structures [119] are event structures together with a function on configurations, $v : \mathcal{C}(\mathsf{E}) \to [0, 1]$, which assigns 1 for the empty set, jointly with a *monotonicity* condition. The intuition behind $v(x)$ is the probability of reaching at least configuration x.

The monotonicity condition in [119, Definition 1] is described in terms of *drop functions*, which will be necessary when showing that unitary event structures with an initial state corresponds to probabilistic event structures (Section 6.4). Define drop functions $d_v^{(n)}[y\,;\, x_1, \ldots x_n] \in \mathbb{R}$ for $y, x_1, \ldots x_n \in \mathcal{C}(\mathsf{E})$ with $y \subseteq x_1, \ldots, x_n$, by induction, taking

$$d_v^{(0)}[y\,;\,] = v(y)$$
$$d_v^{(n)}[y\,;\, x_1, \ldots, x_n] = d_v^{(n-1)}[y\,;\, x_1, \ldots, x_{n-1}] - d_v^{(n-1)}[x_n\,;\, x_1 \cup x_n, \ldots, x_{n-1} \cup x_n]$$

for $n > 0$. Intuitively, a drop function $d_v^{(n)}[y\,;\, x_1, \ldots x_n]$ indicates the probability of reaching at least configuration y without reaching any of the $x_1, \ldots, x_n$, with $y \subseteq x_1, \ldots, x_n$.

According to [119, Proposition 1], the drop function can be described based on the inclusion-exclusion principle for sets [2] (Equation 3.1). We make use of such result to present the definition of probabilistic event structures.

Definition 3.3.1 (Probabilistic event structure)**.** Let $\mathsf{E} = (E, \leq, \#)$ be an event structure. A configuration-valuation on E is a function $v : \mathcal{C}(\mathsf{E}) \to [0, 1]$ such that $v(\emptyset) = 1$ and $\forall y, x_1, \ldots x_n \in \mathcal{C}(\mathsf{E})$ such that $y \subseteq x_1, \ldots, x_n$

$$v(y) - \sum_{\emptyset \neq I \subseteq \{1,\ldots,n\}} (-1)^{|I|+1} v\left(\bigcup_{i \in I} x_i\right) \geq 0 \tag{3.1}$$

where $v(x) = 0$ whenever $x \notin \mathcal{C}(\mathsf{E})$.

A probabilistic event structure, $\mathsf{P} = (\mathsf{E}, v)$, comprises an event structure $\mathsf{E} = (E, \leq, \#)$ together with a configuration-valuation $v : \mathcal{C}(\mathsf{E}) \to [0, 1]$.

From Equation 3.1, there are some conclusions that we can make [26]:

- The configuration-valuation is decreasing:

$$x \subseteq y \implies v(x) \geq v(y)$$

- Events in conflict have a conditional probability whose sum is less than or equal to one:

$$\forall 1 \leq i \leq n,\ x \xrightarrow{e_i} x_i \text{ and } \forall 1 \leq i < j \leq n,\ e_i \# e_j \implies \sum_{i=1}^{n} \frac{v(x \cup \{e_i\})}{v(x)} \leq 1$$

The first conclusion taken from Equation 3.1 is trivial. Intuitively it says that as further we advance in the computation, the probabilities either stay the same or decrease. The second conclusion can be illustrated by considering two conflicting events e and e'. Now consider $x \xrightarrow{e} x \cup \{e\}$ and $x \xrightarrow{e'} x \cup \{e'\}$. Clearly we have $x \subseteq x \cup \{e\}, x \cup \{e'\}$. From Equation 3.1 we then derive the following:

$$\begin{aligned} v(x) - v(x \cup \{e\}) - v(x \cup \{e'\}) \geq 0 &\Leftrightarrow v(x) \geq v(x \cup \{e\}) + v(x \cup \{e'\}) \\ &\Leftrightarrow 1 \geq \frac{v(x \cup \{e\}) + v(x \cup \{e'\})}{v(x)} \\ &\Leftrightarrow \frac{v(x \cup \{e\}) + v(x \cup \{e'\})}{v(x)} \leq 1 \end{aligned}$$

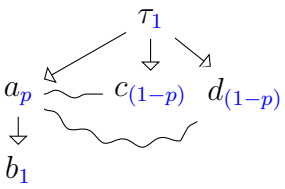
This can be interpreted as a conditional probability: given that configuration x has occurred, the probability of subsequently executing either e or e' cannot exceed 1.

Example 3.3.2 illustrates probabilistic event structures.

[2] The inclusion-exclusion principle for sets is a method to count the number of elements in the union of possibly overlapping finite sets

Example 3.3.2. Figure 14 shows a probabilistic event structure very similar to the event structure in Figure 11, the only difference being the addition of a new event τ, for which the events a, c, and d are causally dependent. The event τ is used to indicate that the events that are causally immediate to it, *i.e.* $\tau \rightarrowtriangle a$, $\tau \rightarrowtriangle c$, and $\tau \rightarrowtriangle d$ arose from a probabilistic choice. Consequently, their probabilities are not 1, as can be seen by the configuration-valuation. Furthermore, in the drawing of the event structure, we subscript the probability of each event to make it easier to understand where probabilities came from.

The set of configurations is composed of $\{\emptyset, \{\tau\}, \{\tau, a\}, \{\tau, c\}, \{\tau, d\}, \{\tau, a, b\}, \{\tau, c, d\}\}$, where $\{\tau, a, b\}$ and $\{\tau, c, d\}$ are maximal configurations with probability p and $1 - p$, respectively.



$$v(x) = \begin{cases} p & \text{if } a \in x \\ 1 - p & \text{if } c \in x \text{ or } d \in x \\ 1 & \text{otherwise} \end{cases}$$

Figure 14: Example of a probabilistic event structure

3.4 Unitary Event Structures

This section shows how event structures are adapted to support quantum behavior. A quantum event structure [119, Definition 2] is an event structure together with a function that maps events to unitary operators or projections on a finite-dimensional Hilbert space $\mathcal{H}$. Additionally, operators of concurrent events must commute in quantum event structures and the latter support the notion of an initial state considered to be a density operator.

The definition of quantum event structures [119, Definition 2] does not mention probabilities, even though probabilities arise as a result of measuring quantum states. So, how do probabilities arise in quantum event structures? The key lies in the combination of the initial state together with quantum operators associated with the events in a configuration (recall that a configuration captures the notion of a computation in event structures). By applying the sequence of operators associated with the events in a configuration to the initial state, we evolve the state along that computational path. This process allows us to extract the probabilities related to measurement outcomes. Winskel formalizes this idea [119, Theorem 3]. He shows that a quantum event structure without conflicting events – also called an elementary quantum event structure – can be transformed into a corresponding probabilistic event structure.

To drop the elementary condition imposed by Winskel in [119, Theorem 3], we add two new conditions

to Winskel's definition of quantum event structures [119, Definition 2] (in Section 6.4 we show how these additional constraints allow us to remove the elementary condition). First, we impose the minimal conflict to be transitive. Secondly, the sum of the operators of events in minimal conflict, or an event itself, should be a unitary operator. The intuition behind the restrictions is to consider events in minimal conflicts as measurements. We call the resultant structure *unitary event structures*. Why do we add these restrictions? The reason is to ensure that Equation 3.1 holds when considering unitary event structures, jointly with an initial state, as probabilistic event structures. Recall from Equation 3.1 that conflicting events have a conditional probability no greater than one. Now, let us consider two events in conflict, e and e', such that $Q(e) = H = Q(e')$. Taking $|0\rangle$ as the initial state, the configuration-valuations of $\{e\}$ and $\{e'\}$ are $v(\{e\}) = 1 = v(\{e'\})$. It is straightforward to check that Equation 3.1 does not hold for $\emptyset \subseteq \{e\}, \{e'\}$:

$$v(\emptyset) - v(\{e\}) - v(\{e'\}) \geq 0 \Leftrightarrow 1 - 1 - 1 \geq 0 \Leftrightarrow -1 \geq 0$$

Now, consider $Q(e) = P_0$ and $Q(e') = P_1$, *i.e.* projections onto $|0\rangle$ and $|1\rangle$, respectively. For any initial state ρ, their configuration-valuations are given by $v(\{e\}) = p_0^\rho$ and $v(\{e'\}) = p_1^\rho$, *i.e.* the probabilities of obtaining $|0\rangle$ or $|1\rangle$ when measuring ρ. Note that $p_0^\rho + p_1^\rho = 1$. Now Equation 3.1 holds for $\emptyset \subseteq \{e\}, \{e'\}$:

$$v(\emptyset) - v(\{e\}) - v(\{e'\}) \geq 0 \Leftrightarrow 1 - p_0^\rho - p_1^\rho \geq 0 \Leftrightarrow 1 - 1 \geq 0 \Leftrightarrow 0 \geq 0$$

However, by being complacent with the satisfaction of Equation 3.1 we lose the ability to interpret non-deterministic choice in unitary event structures. For instance, the program H or X or Z would be represented as $e_H \sim\!\!\sim e_X \sim\!\!\sim e_Z$. Since minimal conflict is not transitive, this fails to form a unitary event structure.

To define unitary event structures, we use the equivalence class of an event e, which is composed of itself and by the events in which e is in minimal conflict, *i.e.* $[e] = \{e' \mid e = e',\ e \sim\!\!\sim e'\}$.

Definition 3.4.1 (Unitary Event Structure)**.** A unitary event structure over a finite-dimensional Hilbert space $\mathcal{H}$, is a pair $(\mathsf{E},\ Q : E \to Op(\mathcal{H}))$ comprised of an event structure $\mathsf{E} = (E, \leq, \#)$, where Q maps events $e \in E$ to either projections or unitary operators on $\mathcal{H}$ such that:

- $\forall e_1, e_2 \in E,\ e_1$ co $e_2 \Rightarrow Q(e_1)Q(e_2) = Q(e_2)Q(e_1)$
- $\sim\!\!\sim$is transitive
- $\forall e \in E,\ \sum_{e' \in [e]} Q(e')$ is unitary

Given a finite configuration, $x \in \mathcal{C}(\mathsf{E})$, define the operator A_x as the composition $Q_{e_n} Q_{e_{n-1}} \dots Q_{e_2} Q_{e_1}$ for some covering chain $\emptyset \overset{e_1}{\subset} x_1 \overset{e_2}{\subset} x_2 \dots \overset{e_n}{\subset} x_n$ in $\mathcal{C}(\mathsf{E})$, with $x_n = x$. We additionally set $A_\emptyset = Id : \mathcal{H} \to \mathcal{H}$ for the initial configuration. An *initial state* is given by a density operator ρ on $\mathcal{H}$.

We now show that the operator A_x in Definition 3.4.1 is well-defined. For that, we need an auxiliary definition. Let $\alpha = \dots, e, e', \dots$ be a finite event sequence. Denote $Q(\alpha)$ be the ordered product of operators along α, according to Definition 3.4.1, *i.e.* $Q(\alpha) = \dots Q(e')Q(e) \dots$ Furthermore, we also need a lemma stating that the operators for any two sequence of events that differ from swapping adjacent concurrent events are the same.

Lemma 3.4.2. Let $\alpha = \dots, e, e', \dots$ and $\beta = \dots, e', e, \dots$ be two finite event sequences that differ only by swapping adjacent concurrent events e co e'. Then $\dots Q(e')Q(e) \cdots = \dots Q(e)Q(e') \dots$.

Proof. Let $\alpha = \dots, e, e', \dots$ and $\beta = \dots, e', e, \dots$ be two finite event sequences such that e co e'. Let $Q(\alpha) = \dots Q(e')Q(e) \dots$ and $Q(\beta) = \dots Q(e)Q(e') \dots$. Since e co e' then $Q(e')Q(e) = Q(e)Q(e')$. Thus

$$Q(\alpha) = \dots Q(e')Q(e) \cdots = \dots Q(e)Q(e') \cdots = Q(\beta)$$

□

As stated by Winskel [119], any two covering chains of a configuration x are Mazurkiewicz trace equivalent, *i.e.* obtainable, one from the other, by successively interchanging concurrent events. This fact together with Lemma 3.4.2 are crucial for showing that A_x is well-defined.

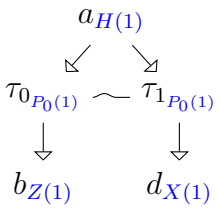
Proposition 3.4.3. Let $x \in \mathcal{C}(\mathsf{E})$. Then A_x is well-defined, *i.e.* A_x is independent from the chosen covering chain of x.

Proof. Let $x \in \mathcal{C}(\mathsf{E})$ and α, β be two covering chains of x. From Winskel [119], we know that any two covering chains of a configuration x are obtainable, one from another, by successively swapping concurrent events (since covering chains are by definition finite, then the interchanges performed are also finite). By applying Lemma 3.4.2 at each swap, the associated operator is preserved at each step. Hence $Q(\alpha) = Q(\beta)$, and therefore A_x is independent of the chosen covering chain. The base case, $A_\emptyset = Id$ holds by definition. □

Example 3.4.4 is designed for the reader to get used to unitary event structures.

Example 3.4.4. Figure 15, depicts a unitary event structure composed of the events a, τ_0, τ_1, d, and b, where the subscript is the respective associated quantum operator. a is the initial event, followed by τ_0, which leads to b, and τ_1, which leads to d. Note that τ_0 and τ_1 are in conflict (specifically, minimal conflict) and that their associated quantum operations form a measurement. Furthermore, since the conflict relation is hereditary, d and b are in conflict.

The set of configurations is $\{\emptyset, \{a\}, \{a, \tau_0\}, \{a, \tau_1\}, \{a, \tau_0, b\}, \{a, \tau_1, d\}\}$. As said in Definition 3.4.1, from a configuration x we have as operator A_x. For example, if we consider the maximal configurations $\{a, \tau_0, b\}$ and $\{a, \tau_1, d\}$, the respective operators are ZP_0H and XP_1H. The former applies the Hadamard gate to qubit 1, projects it to $|0\rangle$, and then applies the Z gate. After applying the Hadamard gate, the latter projects the qubit to $|1\rangle$ and then applies the X gate.



$$Q(a) = H(1),$$
$$Q(\tau_0) = P_0(1),\ Q(\tau_1) = P_1(1),$$
$$Q(b) = Z(1),$$
$$Q(d) = X(1)$$

Figure 15: Example of a unitary event structure

3.5 Core language

For Part I, our *modus operandi* is to consider a core language that will be adapted according to the different types of event structure – Definition 3.2.1, Definition 3.3.1, and Definition 3.4.1. In sum, each type of event structure captures a certain kind of behavior, and for each behavior we add to the core language a matching command.

The core language is a fragment of an imperative language [92, Section 2], where we consider atomic actions instead of assignments, extended with a concurrency operator. We then assume a set of atomic actions, denoted by Act. Atomic actions are represented by letters $a, b, c, \ldots$, which may also possess integers in subscript. We let Com be the set of commands formed by the core language. These commands are formed according to the following BNF syntax:

$$C ::= skip \mid a \in Act \mid C\,;\,C \mid C \,||\, C$$

As we can see, the core language only contemplates sequential composition and concurrent composition as primordial operators. In Chapter 4, we deal with non-deterministic behavior, which leads us to extend

the core language with a non-deterministic command, resulting in the language presented in Section 4.1. In Chapter 5, we deal with probabilistic behavior. Following the same line of thought as for the non-deterministic case, we now add to the core language a probabilistic command, which gives us the language presented in Section 5.1. A similar reasoning is also applied when considering quantum behavior in Chapter 6, where we introduce a measurement command to the core language and consider the set of atomic actions to be composed of unitary gates.

Recall that the core language is a fragment of an imperative language [92, Section 2] extended with a concurrency operator. In an imperative setting, the evaluation of a program is commonly accompanied by a memory that is updated as execution progresses. However, event structures focus on capturing the relationships between actions in a computation rather than tracking memory changes. Hence, they are more suitable to model process algebras, in which no state is considered and transitions are labeled according to the actions that triggered them. At first glance, using event structures to model an imperative language may seem contradictory. However, we can overcome such a contradiction by adopting a process algebra approach: when evaluating programs, we label the transitions by the actions that triggered the transitions. This approach allows us to record a sequence of actions, which we can see as a "tape", that represents the computation history. Given an initial state, we can then perform the recorded actions on the initial state to determine the final state of the execution.

Given the intuition on how to evaluate programs, let us formalize it. To do that, we found it useful to consider a special symbol, denoted by ✓, that signals the end of a computation. This special symbol is not considered as a command, and hence it is not included in the grammar above because that would permit programs like "a ; ✓ ; b". This program would be syntactically valid. However, if we were to have a rule ✓ in the operational semantics, after "reading" the ✓, the program would halt without having computed all of its actions, which is counterintuitive. For the operational semantics, we also need a set of labels, defined as $L = Act \cup \{sk\}$ [3], which is ranged by l. The small-step semantics is then defined as the smallest relation $\longrightarrow \subseteq Com \times L \times (Com \cup \{\checkmark\})$ obeying the rules in Figure 16.

The operational semantics defined in Figure 16 is inspired by CCS [72], where commands do not have associated states and transitions are labeled by the triggering actions. It is also influenced by an imperative language [92, Chapters 6 and 8], where both actions $skip$ and a lead to the terminal command (or, as in [92, Chapter 6], to a final state). Note that if we had associated states with our commands, we could return the final state instead of ✓. Intuitively, the small-step semantics stamps a tape with the action that triggered the computation.

[3] sk is the label for the command $skip$

$$skip \xrightarrow{sk} \checkmark \qquad a \xrightarrow{a} \checkmark \qquad \frac{C_1 \xrightarrow{l} \checkmark}{C_1 \,;\, C_2 \xrightarrow{l} C_2} \qquad \frac{C_1 \xrightarrow{l} C_1'}{C_1 \,;\, C_2 \xrightarrow{l} C_1' \,;\, C_2}$$

$$\frac{C_1 \xrightarrow{l} \checkmark}{C_1 \,||\, C_2 \xrightarrow{l} C_2} \qquad \frac{C_1 \xrightarrow{l} C_1'}{C_1 \,||\, C_2 \xrightarrow{l} C_1' \,||\, C_2} \qquad \frac{C_2 \xrightarrow{l} \checkmark}{C_1 \,||\, C_2 \xrightarrow{l} C_1} \qquad \frac{C_2 \xrightarrow{l} C_2'}{C_1 \,||\, C_2 \xrightarrow{l} C_1 \,||\, C_2'}$$

Figure 16: Rules of the small-step operational semantics

Let us explain the rules in Figure 16. The first two rules are straightforward. Suppose the command is a $skip$ or an atomic action a. In that case, the transition is labeled with either sk or a, respectively, and the computation finishes, indicated by the ✓ symbol. We have two rules for sequential composition based on the behavior of C_1. If C_1 transitions to the terminal symbol by executing l, then the next computation comes from C_2. Otherwise, if the execution of l reduces C_1 to C_1' then we compose what is left to compute with C_2 and the next command to be evaluated is $C_1' \,;\, C_2$. Parallel composition follows a similar reasoning. However, the command to be evaluated can be either C_1 or C_2.

We also furnish the core language with a big-step operational semantics (Figure 17). Building on the previous intuition, the big-step semantics will stamp the tape with n actions that occurred during a computation. More formally, these n actions form a sequence of labels, *i.e.* a *word* of labels. The corresponding grammar of words is defined by:

$$\omega ::= l \mid l : \omega$$

where $l : \omega$ appends l to the beginning of ω. A word can also be seen as an element of L^+, *i.e.* a non-empty finite sequence.

Define the big-step transition, $\twoheadrightarrow \subseteq Com \times L^+ \times (Com \cup \{\checkmark\})$, where n is the length of the words, obeying the rules in Figure 17:

$$\frac{C \xrightarrow{l} C'}{C \xrightarrow{l} \!\!\!\!\twoheadrightarrow C'} \qquad \frac{C \xrightarrow{l} C'' \qquad C'' \xrightarrow{\omega'} \!\!\!\!\twoheadrightarrow C'}{C \xrightarrow{l:\omega'} \!\!\!\!\twoheadrightarrow C'}$$

Figure 17: Rules of the big-step operational semantics

The relation $\xrightarrow{\omega} \!\!\!\!\twoheadrightarrow$ can be seen as the transitive closure of $\xrightarrow{l}$, where the left rule represents the base case and the right rule represents the transitive part.

To better understand how the semantics works, let us focus on the following example, which also aims to illustrate how concurrency leads to interference.

Example 3.5.1. The initial program is $(a\,;\,b)\,||\,c$, from which we have two possible transitions: either by a or by c. If we transit by c we go to the command $a\,;\,b$, where we execute a followed by b to complete the computation. On the other hand, if we transition by a, we reach the command $b\,||\,c$, which allows two transitions: first b and then c or first c and then b. Note that when we first execute a, then c, and then b, the computation of the sequential composition $a\,;\,b$ is being interfered by the computation of c. This is a phenomenon that happens when we take into consideration a language that has a concurrent operator.

By recurring to the execution tree in Figure 20 together with the above explanation, we can straightforwardly deduce the words that can be formed by the big-step semantics: a, c, ab, ac, ca, abc, acb, and cab.

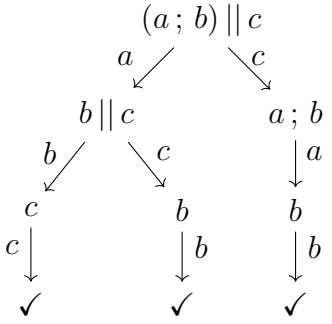


Figure 18: Labeled transition system of $(a\,;\,b)\,||\,c$

In the upcoming chapters, we omit many proofs from the main text and put them in appendix. These proofs can be consulted by the reader at will. Concretely, the proofs of Chapter 4 can be found in Appendix A.1, the proofs of Chapter 5 can be found in Appendix A.2, and the proofs of Chapter 6 can be found in Appendix A.3. Furthermore, to keep the notation as simple as possible, whenever new commands are added to the core language, the set of commands will still be called Com.

Chapter 4

Adding non-deterministic choice

4.1 Language

We start using event structures to give a denotational semantics to different languages by considering a language with non-deterministic behavior. To do this, we extend the core language, in Section 3.5, with non-deterministic choice. Recall that we consider a set of atomic actions Act ranged over by a. The following grammar gives the commands we can form, when extending the core language with the non-deterministic command (we postpone the introduction of cyclic behavior to Section 4.3):

$$C ::= skip \mid a \in Act \mid C\,;\,C \mid C \,||\, C \mid C \,\mathsf{or}\, C$$

where $skip$ is a command that does nothing; a is an atomic action from the set of atomic actions Act; $C\,;\,C$ is the usual sequential composition of programs; $C \,||\, C$ is the parallel composition of commands; $C \,\mathsf{or}\, C$ represents the non-deterministic choice.

Regarding the operational semantics, and since concurrency also entails non-determinism, the small-step semantics is still defined as the smallest relation $\longrightarrow \subseteq Com \times L \times (Com \cup \{\checkmark\})$. Hence, we need to extend the rules in Figure 16 with the rules concerning the non-deterministic command. In order to avoid going back and forth when seeing the small-step operational rules, Figure 19 agglomerates the rules in Figure 16 together with the rules concerning the non-deterministic command.

Except for the non-deterministic choice, all the other rules were already explained in Section 3.5. Hence, here we shall explain the rules concerning the non-deterministic choice. This operator has the same specificity as parallel composition, in which the command to be evaluated can either be C_1 or C_2. However, unlike sequential and parallel composition, when one command is performed, the other is discarded. For example, if C_1 is executed and either results in $\checkmark$ or reduces to C_1', then $C_1 \,\mathsf{or}\, C_2$ evaluates to either $\checkmark$ or C_1', respectively.

The language in this section is also furnished with a big-step operational semantics. With a similar

$$skip \xrightarrow{sk} \checkmark \qquad a \xrightarrow{a} \checkmark \qquad \frac{C_1 \xrightarrow{l} \checkmark}{C_1 \,;\, C_2 \xrightarrow{l} C_2} \qquad \frac{C_1 \xrightarrow{l} C_1'}{C_1 \,;\, C_2 \xrightarrow{l} C_1' \,;\, C_2}$$

$$\frac{C_1 \xrightarrow{l} \checkmark}{C_1 \,||\, C_2 \xrightarrow{l} C_2} \qquad \frac{C_1 \xrightarrow{l} C_1'}{C_1 \,||\, C_2 \xrightarrow{l} C_1' \,||\, C_2} \qquad \frac{C_2 \xrightarrow{l} \checkmark}{C_1 \,||\, C_2 \xrightarrow{l} C_1} \qquad \frac{C_2 \xrightarrow{l} C_2'}{C_1 \,||\, C_2 \xrightarrow{l} C_1 \,||\, C_2'}$$

$$\frac{C_1 \xrightarrow{l} \checkmark}{C_1 \text{ or } C_2 \xrightarrow{l} \checkmark} \qquad \frac{C_1 \xrightarrow{l} C_1'}{C_1 \text{ or } C_2 \xrightarrow{l} C_1'} \qquad \frac{C_2 \xrightarrow{l} \checkmark}{C_1 \text{ or } C_2 \xrightarrow{l} \checkmark} \qquad \frac{C_2 \xrightarrow{l} C_2'}{C_1 \text{ or } C_2 \xrightarrow{l} C_2'}$$

Figure 19: Rules of the small-step operational semantics

reasoning as to the small-step operational semantics, the big-step semantics is still defined as the relation $\twoheadrightarrow \subseteq Com \times L^+ \times (Com \cup \{\checkmark\})$, obeying the rules in Figure 17.

We now design an example that combines all the operators in this section's language.

Example 4.1.1. Consider the program $(a \,;\, b)$ or $(c \,||\, d)$. Intuitively we can either execute the sequential composition $a \,;\, b$, in which we first execute a and then b, or execute the parallel composition $c \,||\, d$, in which we can either first perform c and then d or we can first execute d and then c.

Figure 20 shows an execution tree in which nodes represent commands and the edges connecting nodes are labeled by the action that triggered the computation.

With the support of Figure 20 together with the above explanation, we can straightforwardly deduce the words that can be formed by the big-step semantics: a, c, d, ab, cd, and dc.

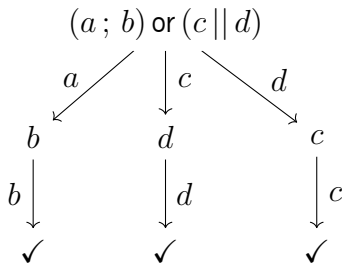
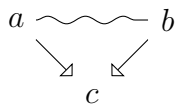


Figure 20: Labeled transition system of $(a \,;\, b)$ or $(c \,||\, d)$

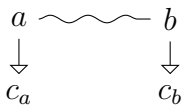
4.2 Constructions on Event Structures

In order to furnish the language with a denotational semantics based on event structures we need to define sequential composition, parallel composition, and non-deterministic choice in the realm of event structures.

Let us begin with sequential composition. Consider $a\,;\,b$ to be the sequential combination of two actions, a and b. According to the rules in Figure 19 we execute b after a has been executed, which with an event structure view means that b causally depends on a. As a first attempt to define sequential composition of two event structures, $\mathsf{E}_1\,;\,\mathsf{E}_2$, one might try to connect every event of E_1 with every event of E_2. However, this approach fails to interpret programs like $(a \text{ or } b)\,;\,c$, as shown in Figure 21a. This failure arises because two ways exist to reach event a, which come from conflicting events. According to the definition of event structures, the conflict relation is hereditary, and an event is not in conflict with itself. Thus, we would end up with an invalid event structure. To address this issue, we introduce a "copy" for each event of E_2 regarding the different ways it can be reached. For example, in the aforementioned program, we create two copies of c: one indicating it was reached by executing event a, and another indicating it was reached by executing event b, as can be seen in Figure 21b.



(a) Unwanted sequential composition



(b) Good sequential composition

We define the sequential composition of event structures as follows:

Definition 4.2.1 (Sequential Composition)**.** Let $\mathsf{E}_1 = (E_1, \leq_1, \#_1)$ and $\mathsf{E}_2 = (E_2, \leq_2, \#_2)$ be event structures. Define $\mathsf{E}_1\,;\,\mathsf{E}_2 = (E, \leq, \#)$ as:

$$
\begin{aligned}
E &= E_1 \uplus (E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)) \\
\leq &= \{e_1 \leq e_1' \mid e_1 \leq_1 e_1'\} \cup \{(e_2, x) \leq (e_2', x) \mid\ e_2 \leq_2 e_2'\} \cup \{e_1 \leq (e_2, x) \mid e_1 \in x\} \\
\# &= \{e \# e' \mid \exists (e_1 \leq e, e_1' \leq e')\ .\ e_1 \#_1 e_1'\} \cup \{(e_2, x) \# (e_2', x) \mid e_2 \#_2 e_2'\}
\end{aligned}
$$

where $E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1) = \{(e, x) \mid e \in E_2,\ x \in \mathcal{C}_{\max}(\mathsf{E}_1)\}$ and $\uplus$ denotes the disjoint union.

Note that we pair events in E_2 with the maximal configurations of E_1, since maximal configurations represent finished computations. Furthermore, the proper definition of the disjoint union is $A \uplus B = \{(a, 0) | a \in A\} \cup \{(b, 1) | b \in B\}$. For $R, S \in A \times B$, the disjoint union extends to a relation as

$(e, i)R \uplus S(e', i')$ whenever $i = 0 = i'$ and eRe' or $i = 1 = i'$ and eSe'. To keep the notations readable, we will keep the 0s and 1s implicit.

The conflict relation in Definition 4.2.1 has a tricky part in $\{e\#e' \mid \exists(e_1 \leq e, e_1' \leq e') \,.\, e_1\#_1 e_1'\}$. Intuitively, this part assures us two things: The first is that the conflict relation of E_1 is preserved. The second is that the conflict relation of events in E_1 is propagated to events in E_2 that causally depend on the conflicting ones. To better see this, consider the events c_a and c_b in Figure 21b. We have $a \leq c_a$, for the former, and $b \leq c_b$, for the latter, with $a\#_1 b$. Consequently, $c_a\#c_b$. In other words, the conflict relation between a and b is propagated to c_a and c_b.

Lemma 4.2.2. Let E_1 and E_2 be event structures. $\mathsf{E}_1 \,;\, \mathsf{E}_2$ is an event structure.

To define the concurrent composition of event structures we adapted the definition in [118]. Specifically, the absence of communication in the language considered simplifies Definition 4.2.3, when compared to [118], since we do not need a synchronization mechanism. In our case, we simply place "side-by-side" the two event structures.

Definition 4.2.3 (Parallel Composition)**.** Let $\mathsf{E}_1 = (E_1, \leq_1, \#_1)$ and $\mathsf{E}_2 = (E_2, \leq_2, \#_2)$ be event structures. Define $\mathsf{E}_1 \,||\, \mathsf{E}_2 = (E, \leq, \#)$ as:

$$E = E_1 \uplus E_2$$
$$\leq = \leq_1 \uplus \leq_2$$
$$\# = \#_1 \uplus \#_2$$

Lemma 4.2.4. Let E_1 and E_2 be event structures. $\mathsf{E}_1 \,||\, \mathsf{E}_2$ is an event structure.

Consider the following example to help to build the intuition of Definition 4.2.3.

Example 4.2.5. Before presenting Definition 4.2.3, we mentioned that composing event structures in parallel is simply placing them "side-by-side", similar to the intuition given at the beginning of Section 4.1. Here we will see that it holds. Consider the two following event structures:



Figure 22: Event structure E_1



Figure 23: Event structure E_2

Their parallel composition is given, as we previously said, by putting the two event structures "side-by-side" as follows:

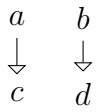


Figure 24: Parallel composition of event structures E_1 and E_2

At last, we have the non-deterministic choice of event structures, adapted from [83]. Let us use the rules in Figure 19 to give the intuition behind the definition. Consider the command a or b. According to the operational semantics, if we execute a, then we cannot execute b; if we execute b, we cannot execute a. If we abstract ourselves and instead of a or b we consider C_1 or C_2, we notice that if we execute an action from C_1, then it is no longer possible to execute any action of C_2 and vice-versa. To capture this behavior in event structures, we must put all the events corresponding to C_1 in conflict with all the events corresponding to C_2. Formally:

Definition 4.2.6 (Non-deterministic composition)**.** Let $\mathsf{E}_1 = (E_1, \leq_1, \#_1)$ and $\mathsf{E}_2 = (E_2, \leq_2, \#_2)$ be event structures. Define E_1 or $\mathsf{E}_2 = (E, \leq, \#)$ as:

$$
\begin{aligned}
E &= E_1 \uplus E_2 \\
\leq &= \leq_1 \uplus \leq_2 \\
\# &= \#_1 \uplus \#_2 \cup \{e_1 \# e_2 \mid e_1 \in E_1,\, e_2 \in E_2\} \cup \{e_2 \# e_1 \mid e_1 \in E_1,\, e_2 \in E_2\}
\end{aligned}
$$

Equivalently, we can define the partial order in Definition 4.2.6 as follows:

$$
e \leq e' = \begin{cases} e \leq_1 e' & \text{if } e, e' \in E_1 \\ e \leq_2 e' & \text{if } e, e' \in E_2 \end{cases}
$$

Lemma 4.2.7. Let E_1 and E_2 be event structures. E_1 or E_2 is an event structure.

The non-deterministic and concurrent composition definitions, respectively given in Definition 4.2.6 and Definition 4.2.3, are very similar. However, they differ in a crucial aspect: how the conflict relation is defined. In the non-deterministic composition, all events from both event structures must be in conflict. To see this distinction, we revisit the event structures E_1 and E_2 from Example 4.2.5 and non-deterministically compose them.

Example 4.2.8. Recall the event structures E_1 (Figure 25) and E_2 (Figure 26) from Example 4.2.5.

Their non-deterministic composition is given by putting the two event structures "side-by-side" and putting in conflict all the events of E_1 with all the events of E_2 (see Figure 27):

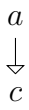


Figure 25: Event structure E_1



Figure 26: Event structure E_2

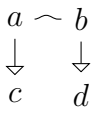


Figure 27: Non-deterministic composition of event structures E_1 and E_2

For this case, it suffices to indicate that a and b are in minimal conflict. Furthermore, since the conflict relation is hereditary, it follows that a is in conflict with d, that c is in conflict with b, and that c is in conflict with d.

We now have everything we need to interpret the language presented in Section 4.1 using event structures.

Definition 4.2.9. We interpret commands as follows:

$$[\![skip]\!] = (\{sk\}, \{sk \leq sk\}, \emptyset)$$
$$[\![a]\!] = (\{a\}, \{a \leq a\}, \emptyset)$$
$$[\![C_1 \,;\, C_2]\!] = [\![C_1]\!] \,;\, [\![C_2]\!]$$
$$[\![C_1 \,||\, C_2]\!] = [\![C_1]\!] \,||\, [\![C_2]\!]$$
$$[\![C_1 \text{ or } C_2]\!] = [\![C_1]\!] \text{ or } [\![C_2]\!]$$

To see the denotational semantics at work, let us develop an example.

Example 4.2.10. In Section 4.1, we explored an example of a worker who listens to music while deciding whether to turn on the computer to work or not to turn on the computer and rest. We can represent this behavior using the language presented in Section 4.1 as follows:

$$Music \,||\, ((PCOn \,;\, Work) \text{ or } (PCOff \,;\, Rest))$$

The drawing of the event structure generated by this command, in Figure 28 closely resembles the box drawings in Section 4.1, being the only difference the absence of boxes here:

Since $Music$ is in parallel with $(PCOn \,;\, Work)$ or $(PCOff \,;\, Rest)$ their respective event structures are placed side-by-side. Moreover, $PCOn$ and $PCOff$ are disjoint choices, *i.e.* executing one

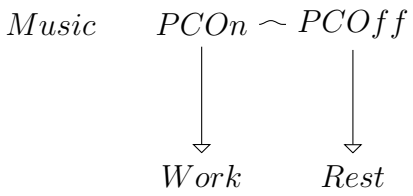


Figure 28: Event structure of $Music \,||\, ((PCOn\,;\, Work)$ or $(PCOff\,;\, Rest))$

prevents executing the other. Hence, they are in conflict, specifically in minimal conflict. At last, the conflict between $Work$ and $Rest$ is inherited from the conflict between $PCOn$ and $PCOff$.

We now focus on the equivalence between the operational and denotational semantics. To do this, we shall use the fact that the big-step operational semantics produces a word of size n and that covering chains are a finite sequence of events. The key intuition here is that a word can be seen as a covering chain, and conversely, a covering chain can be viewed as a word.

Recall that the big-step semantics is defined based on the small-step semantics. Thus it is natural to also have a result that relates the small-step semantics to event structures. Concretely, we noticed that when a transition occurs under the small-step semantics ($C \xrightarrow{l} C'$), the resulting event structure is similar to the original one, with the event that triggered the transition removed ($[\![C']\!] \equiv [\![C]\!] \backslash l$). This leads to the need of defining the notion of "similarity" between event structures, as well as the removal of an initial event from an event structure (an initial event is defined in Definition 4.2.18). The remainder of this section will be dedicated to that.

We start by defining what it means to be a *sub-similar event structure*. For that, we must first be explicit about the form of the events we consider. Events can either be pairs or not. If they are not pairs we call them *plain events*. If they are pairs we call them *composite events*. We shall denote both plain and composite events by $a, \ldots, e$. Thus, examples of composite events are: (e, x_1), $((e, x_1), x_2)$, $(e, 1)$, $((e, x), 1)$. Examples of plain events are: e, e_1, e^1. Composite events have an underlying plain event. To extract a plain event from a composite event, we recursively extract the left side until it is not a pair. Formally, $\pi(a, b) = \pi(a)$, recursively. An equivalent, explicit definition of π is as follows:

$$\pi : E \to E$$

$$e \mapsto \begin{cases} e & \text{if } e \text{ is a plain event} \\ \pi(e') & \text{if } e = (e', x) \text{ for some } e' \end{cases}$$

For example,

$$\pi(a) = a, \qquad \pi(a, x_1) = a, \qquad \pi((a, x_1), x_2) = a.$$

Returning to the notion of sub-similar event structure. Intuitively, an event structure E_1 is a sub-similar event structure of E_2 if plain events, the causal order and the conflict relation of E_1 are preserved in E_2. Note that the idea of ignoring the "copies" created by sequential composition is captured by the condition on plain events. We say that two event structures E_1 and E_2 are *similar* if each is a sub-similar event structure of the other.

A sub-similar event structure is formally defined as follows:

Definition 4.2.11. Let $\mathsf{E}_1 = (E_1, \leq_1, \#_1)$ and $\mathsf{E}_2 = (E_2, \leq_2, \#_2)$ be event structures. Say $\mathsf{E}_1 \sqsubseteq \mathsf{E}_2$ whenever exists an injective function $f : E_1 \to E_2$ such that $\forall e, e' \in E_1$:

$$\pi(f(e)) = \pi(e)$$
$$e \leq_1 e' \Leftrightarrow f(e) \leq_2 f(e')$$
$$e \#_1 e' \Leftrightarrow f(e) \#_2 f(e')$$

We say that two event structures $\mathsf{E}_1, \mathsf{E}_2$ are similar, denoted $\mathsf{E}_1 \equiv \mathsf{E}_2$, iff $\mathsf{E}_1 \sqsubseteq \mathsf{E}_2$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}_1$.

As mentioned before presenting Definition 4.2.11, we ignore the "copies" of events when preserving plain events. This decision comes from the sequential composition definition (Definition 4.2.1), where we create "copies" of the same event to distinguish how an event can be reached. However, these "copies" still refer to the same event. Thus, we want to focus on the event itself and forget how it can be reached. If we have not done this, sequential composition would not be monotone, *i.e.* even if $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$ one could still have $\mathsf{E}_1 \,;\, \mathsf{E}_2 \not\sqsubseteq \mathsf{E}'_1 \,;\, \mathsf{E}'_2$. Consequently, auxiliary results (Lemma 4.2.17 and Lemma 4.4.1) used to show soundness (Theorem 4.4.2) would not hold.

Event structures already have an ordering notion [118, Definition 4.1]. This ordering, formalized in Definition 4.3.3, intuitively states that E_1 is a sub-event structure of E_2, written $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$, if the drawing of E_2 is an extension of that of E_1. In this notion, copies are not ignored, and therefore sequential composition fails to be monotone.

The following example illustrates two commands that are sequentially monotone when we ignore the copies of events (Definition 4.2.11), but fail to be sequentially monotone when we account for the copies (Definition 4.3.3).

Example 4.2.12. Consider the following event structures, where E_1, E'_1, and E_2 are depicted in Figure 29, Figure 30, and Figure 31, respectively:

It is straightforward to see that $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ since the plain events, the causal relation, and the conflict relation of E_1 are preserved by E'_1. Now let us draw the event structures corresponding to $\mathsf{E}_1 \,;\, \mathsf{E}_2$ and $\mathsf{E}'_1 \,;\, \mathsf{E}_2$.



Figure 29: Event structure E_1

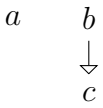


Figure 30: Event structure E'_1



Figure 31: Event structure E_2

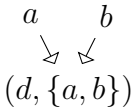


Figure 32: Event structure $\mathsf{E}_1 \,;\, \mathsf{E}_2$

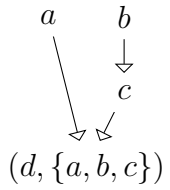


Figure 33: Event structure $\mathsf{E}'_1 \,;\, \mathsf{E}_2$

From Figure 32 and Figure 33 we can see that the event d from E_2 has two different names: in $\mathsf{E}_1 \,;\, \mathsf{E}_2$ its name is $(d, \{a, b\})$ while in $\mathsf{E}'_1 \,;\, \mathsf{E}_2$ its name is $(d, \{a, b, c\})$. Furthermore, note that the underlying plain event is the same (d). Hence, under the notion of sub-similar event structures (Definition 4.2.11), we have that $\mathsf{E}_1 \,;\, \mathsf{E}_2 \sqsubseteq \mathsf{E}'_1 \,;\, \mathsf{E}_2$. However, under the notion of sub-event structures (Definition 4.3.3), we observe that $\mathsf{E}'_1 \,;\, \mathsf{E}_2$ (Figure 33) is not a drawing extension of $\mathsf{E}_1 \,;\, \mathsf{E}_2$ (Figure 32). Thus, although E'_1 (Figure 30) is a drawing extension of E_1 (Figure 29), and consequently $\mathsf{E}_1 \trianglelefteq \mathsf{E}'_1$, we do not have $\mathsf{E}_1 \,;\, \mathsf{E}_2 \trianglelefteq \mathsf{E}'_1 \,;\, \mathsf{E}_2$.

Furthermore, it is clearer that when two event structures are equal, then they are also similar, *i.e.* if $\mathsf{E}_1 = \mathsf{E}_2$ then $\mathsf{E}_1 \equiv \mathsf{E}_2$.

Now we study some properties of Definition 4.2.11 regarding the sequential, concurrent, and non-deterministic commands (the proofs can be found in Appendix A.1, in order to not interrupt the reading with routine proofs). For that, recall that in Section 4.1 we added a special symbol, ✓, to indicate that a computation has finished. In terms of event structures, this is represented by the empty event structure, *i.e.* $[\![\checkmark]\!] = \emptyset = (\emptyset, \emptyset, \emptyset)$. It is straightforward to see that the following holds for an event structure E:

1. $\emptyset \,;\, \mathsf{E} \equiv \mathsf{E} \equiv \mathsf{E} \,;\, \emptyset$
2. $\emptyset \,||\, \mathsf{E} \equiv \mathsf{E} \equiv \mathsf{E} \,||\, \emptyset$

It is also straightforward to prove the following result:

Lemma 4.2.13. Let $\mathsf{E}_1, \mathsf{E}_2$ be event structures. Then $\mathsf{E}_1 \,||\, \mathsf{E}_2 = \mathsf{E}_2 \,||\, \mathsf{E}_1$.

The following lemmas show that the sequential, parallel, and non-deterministic compositions are monotonic with respect to Definition 4.2.11.

Lemma 4.2.14. Let $\mathsf{E}_1, \mathsf{E}_1', \mathsf{E}_2, \mathsf{E}_2'$ be event structures. If $\mathsf{E}_1 \sqsubseteq \mathsf{E}_1'$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}_2'$ then $\mathsf{E}_1 \,;\, \mathsf{E}_2 \sqsubseteq \mathsf{E}_1' \,;\, \mathsf{E}_2'$.

Lemma 4.2.15. Let $\mathsf{E}_1, \mathsf{E}_1', \mathsf{E}_2, \mathsf{E}_2'$ be event structures. If $\mathsf{E}_1 \sqsubseteq \mathsf{E}_1'$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}_2'$ then $\mathsf{E}_1 \,||\, \mathsf{E}_2 \sqsubseteq \mathsf{E}_1' \,||\, \mathsf{E}_2'$.

Lemma 4.2.16. Let $\mathsf{E}_1, \mathsf{E}_1', \mathsf{E}_2, \mathsf{E}_2'$ be event structures. If $\mathsf{E}_1 \sqsubseteq \mathsf{E}_1'$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}_2'$ then E_1 or $\mathsf{E}_2 \sqsubseteq \mathsf{E}_1'$ or E_2'.

Now, Lemma 4.2.17 tells us that whenever we have similar event structures, *i.e.* $\mathsf{E}_1 \equiv \mathsf{E}_1'$ and $\mathsf{E}_2 \equiv \mathsf{E}_2'$, they will still be similar when composed of sequentially, in parallel, and non-deterministically.

Lemma 4.2.17. Let $op \in \{;\ ,\ ||,\ \text{or}\}$ and E_1, E_2, E_1' and E_2' be event structures. If $\mathsf{E}_1 \equiv \mathsf{E}_1'$ and $\mathsf{E}_2 \equiv \mathsf{E}_2'$ then $\mathsf{E}_1 \, op \, \mathsf{E}_2 \equiv \mathsf{E}_1' \, op \, \mathsf{E}_2'$.

Lastly, it lacks to define how to remove an initial event from an event structure. As shall be seen later on, in Section 4.4, this definition is useful when showing the equivalence between the operational and denotational semantics. Intuitively, an initial event is an event that causally depends only on itself. Formally:

Definition 4.2.18 (Set of initial events)**.** Let $\mathsf{E} = (E, \leq, \#)$ be an event structure. Define the set of initial events as follows:

$$\mathcal{I}(\mathsf{E}) = \{e' \mid \nexists e \in E \,.\, e \leq e' \wedge e \neq e'\}$$

When removing an initial event from an event structure, we will eliminate not only the event itself but also all events that are in conflict with it. This decision aims to mimic what happens in a small-step transition. In small-step semantics, once an action triggers a transition, that same action cannot be executed again. Furthermore, if the transition occurs within a non-deterministic program, only the program associated with the triggering action continues, while the others are discarded.

Definition 4.2.19 (Removal of an initial event)**.** Let $\mathsf{E} = (E, \leq, \#)$ be an event structure and $a \in \mathcal{I}(\mathsf{E})$. Define $\mathsf{E}\backslash a = (E', \leq', \#')$ as

$$\begin{aligned} E' &= \{e \in E \mid \neg(e \# a),\ e \neq a\} \\ \leq' &= \{e \leq e' \mid e, e' \in E'\} \\ \#' &= \{e \# e' \mid e, e' \in E'\} \end{aligned}$$

Lemma 4.2.20. Let E be an event structure and $a \in \mathcal{I}(\mathsf{E})$. Then $\mathsf{E}\backslash a$ is an event structure.

To familiarize the reader with Definition 4.2.19, let us analyze the following example:

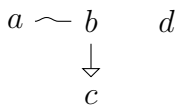


Figure 34: An event structure

Example 4.2.21. Consider the event structure in Figure 34.

It is composed of the events a, b, c, and d where d is concurrent with every event. Furthermore a and b are in conflict, and c causally depends on b (consequently, a and c are also in conflict, since the conflict relation is inherited).

The set of initial events is $\{a, b, d\}$, since there is no other event they causally depend on unless themselves.

Now let us remove the event d, whose event structure is pictured in Figure 35.

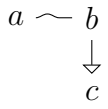


Figure 35: Event structure resultant from removing d

Since we do not have any conflicting event with d, removing d from the original event structure is simply removing it from the drawing. Now let us see what happens if we remove the event b, pictured in Figure 36.



Figure 36: Event structure resultant from removing b

After removing the event b, we see that the event a also disappears, because both events were in conflict. Hence what remains is the event c, which causally depended on b, and the event d, which was concurrent to b.

We now show how the removal of an event interacts with the operations associated with sequential composition, concurrent composition, and non-deterministic composition (the proofs can be found in Appendix A.1). Starting with sequential composition. If we have an event structure like $\mathsf{E}_1 \,;\, \mathsf{E}_2$, then removing the initial event of $\mathsf{E}_1 \,;\, \mathsf{E}_2$ is the same as removing the initial event of E_1 and then sequentially compose it with E_2. This is because the initial events of $\mathsf{E}_1 \,;\, \mathsf{E}_2$ are the same as E_1.

Lemma 4.2.22. Let $\mathsf{E}_1, \mathsf{E}_2$ be event structures. Consider $\mathsf{E}_1 \,;\, \mathsf{E}_2$ such that $l \in \mathcal{I}(\mathsf{E}_1 \,;\, \mathsf{E}_2)$. Then $(\mathsf{E}_1 \,;\, \mathsf{E}_2)\backslash l \equiv (\mathsf{E}_1 \backslash l) \,;\, \mathsf{E}_2$.

In the concurrent composition of E_1 and E_2, we place the event structures side-by-side. Therefore the initial event of $\mathsf{E}_1 \,||\, \mathsf{E}_2$ can be either from E_1 or from E_2. We can then infer that if the initial event l is in E_1 then $(\mathsf{E}_1 \,||\, \mathsf{E}_2)\backslash l \equiv (\mathsf{E}_1\backslash l) \,||\, \mathsf{E}_2$ and similarly whenever l is in E_2.

Lemma 4.2.23. Let $\mathsf{E}_1, \mathsf{E}_2$ be event structures. Consider $\mathsf{E}_1 \,||\, \mathsf{E}_2$ and an event $l \in \mathcal{I}(\mathsf{E}_1 \,||\, \mathsf{E}_2)$. Then

$$(\mathsf{E}_1 \,||\, \mathsf{E}_2)\backslash l \equiv \begin{cases} (\mathsf{E}_1\backslash l) \,||\, \mathsf{E}_2 & \text{if } l \in \mathcal{I}(\mathsf{E}_1) \\ \mathsf{E}_1 \,||\, (\mathsf{E}_2\backslash l) & \text{if } l \in \mathcal{I}(\mathsf{E}_2) \end{cases}$$

Regarding the non-deterministic composition of event structures, recall that the intuition behind it is to place two event structures E_1 and E_2 side-by-side, while simultaneously putting all events of E_1 in conflict with those of E_2. Hence, the initial event of E_1 or E_2 can either be from E_1 or from E_2. If the initial event is from E_1 we remove E_2 from the drawing, since the initial event is in conflict with all the events from E_2, and we stay with $\mathsf{E}_1\backslash l$. A similar reasoning is applied when the initial event is from E_2.

Lemma 4.2.24. Let $\mathsf{E}_1, \mathsf{E}_2$ be event structures. Consider E_1 or E_2 such that $l \in \mathcal{I}(\mathsf{E}_1 \text{ or } \mathsf{E}_2)$. Then

$$(\mathsf{E}_1 \text{ or } \mathsf{E}_2)\backslash l \equiv \begin{cases} \mathsf{E}_1\backslash l & \text{if } l \in \mathcal{I}(\mathsf{E}_1) \\ \mathsf{E}_2\backslash l & \text{if } l \in \mathcal{I}(\mathsf{E}_2) \end{cases}$$

4.3 Introducing cyclic behavior

This section introduces cyclic behavior to the language presented in Section 4.1. Recall that in Section 4.1, the small-step operational semantics did not include the notion of a state. Instead, transitions were labeled by the actions that triggered the computations. To keep this approach of labeling transitions by the actions that triggered the computations, we opt to introduce cyclic behavior through a recursive operator rather than a while loop. A while loop requires checking a condition to determine whether the loop continues or terminates. As far as our knowledge goes, this demands associating a memory with commands in the operational semantics, so that one can verify whether the condition is satisfied. However, the operational semantics developed in Section 4.1 (see Figure 80) does not use any form of memory. Consequently, adopting while loops would require modifying the definition of the operational semantics. On the other hand, the recursive operator executes the specified instructions indefinitely, without the need to check any condition. This allows us to keep the operational semantics as originally defined in Section 4.1.

We need to consider that with the introduction of cyclic behavior, we open the door to infinite computations. Hence, we need to decide how to handle such infinite computations. We begin by focusing on

the big-step semantics, where we raise the following question: "given an infinite word, ω^∞, from C can we return a command C', *i.e.* $C \xrightarrow{\omega^\infty}\!\!\!\!\!\rightarrow C'$?". The answer is no and the reason why is quite simple: if the word is infinite, it never ends, meaning there will always be an action triggering another computation. Consequently, we would have an infinite computation and would be unable to return a command C'. Since we expect a final result from any computation, this leads us to consider only finite words.

Regarding event structures, the notion of infinite configurations is odd. Recall that the events in a configuration are guaranteed to occur. However, can we still guarantee that all the events are executed if the configuration is infinite? Answering this question is tricky because we need to define precisely what it means to be an infinite configuration. We focus solely on finite words to avoid problems that infinite configurations might introduce. In the future, exploring the concept of infinite configurations would be interesting, but for now, limiting ourselves to finite words allows us to maintain the reasoning behind Theorem 4.4.2 and Theorem 4.4.4.

We are now ready to reason about the syntax of the recursive command. First, we introduce a set of variables Var such that $X \subseteq Var$. Next we impose some restrictions when forming programs, since we want to avoid commands like $\mu X.X \,;\, a$ and $\mu X.a \,;\, X \,;\, b$. Specifically, we only allow the recursive behavior to appear at the rightmost place in a sequential composition, ensuring that all the instructions on the left side of the sequential composition are captured. This restriction will have future implications, as shall be seen in the monotonicity (Lemma 4.2.14) and continuous (Lemma 4.3.14) properties.

We now update the syntax given in Section 4.1:

$$C ::= skip \mid a \in Act \mid C \,;\, C \mid C \,||\, C \mid C \,\mathsf{or}\, C \mid \mu X.C \mid X$$

where $\mu X.C$ is the recursive command and $X \in Var$. Furthermore, we only consider closed commands, *i.e.* commands in which every variable X is bound by a recursion μX and, as previously mentioned, recursion is allowed to occur only at the rightmost place in sequential composition.

To understand what a closed command is, we define the set of free variables and bound variables in Figure 37.

To capture the intended behavior of recursion in programs sequentially composed, we restrict the sequential composition to those whose free variables and bound variables on the left are empty, *i.e.* $C_1 \,;\, C_2$ if $FV(C_1) = \emptyset = BV(C_1)$. With this restriction we forbid programs like $\mu X.X \,;\, a$ and $\mu X.a \,;\, X \,;\, b$ (with the condition $FV(C_1) = \emptyset$) and programs like $(\mu X.a \,;\, X) \,;\, b$ (with the condition $BV(C_1) = \emptyset$). We want to forbid this kind of program in sequential composition, because if C_1 never terminates, then the sequential composition never terminates. This is also a restriction that comes from the fact that covering chains are only defined in finite sequences and that infinite configurations are odd

$FV(skip) = \emptyset$	$BV(skip) = \emptyset$
$FV(a) = \emptyset$	$BV(a) = \emptyset$
$FV(C_1 \,;\, C_2) = FV(C_1) \cup FV(C_2)$	$BV(C_1 \,;\, C_2) = BV(C_1) \cup BV(C_2)$
$FV(C_1 \,\|\|\, C_2) = FV(C_1) \cup FV(C_2)$	$BV(C_1 \,\|\|\, C_2) = BV(C_1) \cup BV(C_2)$
$FV(C_1 \text{ or } C_2) = FV(C_1) \cup FV(C_2)$	$BV(C_1 \text{ or } C_2) = BV(C_1) \cup BV(C_2)$
$FV(X) = \{X\}$	$BV(X) = \emptyset$
$FV(\mu X.C) = FV(C) \backslash \{X\}$	$BV(\mu X.C) = \{X\} \cup BV(C)$

Figure 37: Free and bound variables of commands

in event structures. Note, however, that we allow programs like $\mu X.X \,||\, a$ and $\mu X.X$ or a, since they do not block the computation.

Before adding a rule for the recursion command to Figure 19, we need to define what it means to substitute a variable X by a command C'. Inspired by [50], we define substitution as follows:

Definition 4.3.1. Let $X \in Var$ and C, C' be commands. Define $C[X \leftarrow C']$, where we substitute every free occurrence of X in C by C' (while changing bound variables to avoid clashes) by induction on C as follows:

$$skip[X \leftarrow C'] = skip$$

$$a[X \leftarrow C'] = a$$

$$(C_1 \,;\, C_2)[X \leftarrow C'] = C_1 \,;\, (C_2[X \leftarrow C'])$$

$$(C_1 \,||\, C_2)[X \leftarrow C'] = C_1[X \leftarrow C'] \,||\, C_2[X \leftarrow C']$$

$$(C_1 \text{ or } C_2)[X \leftarrow C'] = C_1[X \leftarrow C'] \text{ or } C_2[X \leftarrow C']$$

$$X[X \leftarrow C] = C$$

$$(\mu X.C)[X \leftarrow C'] = \mu X.C$$

$$(\mu Y.C)[X \leftarrow C'] = \mu Y.C[X \leftarrow C'] \text{ if } X \neq Y \text{ and } Y \notin FV(C')$$

We then add to Figure 19 the following rule for the recursion command, inspired by [69, Fig. 1]:

$$\frac{C \xrightarrow{l} C'}{\mu X.C \xrightarrow{l} C'[X \leftarrow \mu X.C]}$$

This rule tells us that if an action l triggers a transition from C to C', then from $\mu X.C$ we transit with the same label to $C'[X \leftarrow \mu X.C]$, where we substitute in C' all the occurrences of X by $\mu X.C$.

Now we present an example that shows the behavior of a recursive command.

Example 4.3.2. Figure 38 illustrates the behavior of a non-deterministic toss coin, which produces a possibly empty sequence of a's that finishes with sk. To understand this, we observe that the initial program has two possible transitions: (1) we execute sk that terminates the computation; (2) we execute a, and we transit to a command equal to the initial one, in which we have two possible transitions again.

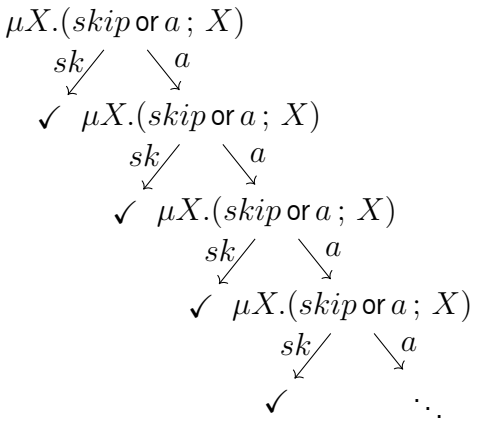


Figure 38: Unrolling the execution of $\mu X.(skip \text{ or } a\,;\, X)$

We now turn our attention to event structures. It is usual to interpret cyclic behavior as a least fixed point and this is what we will do. To this effect, we based ourselves on Knaster-Tarski fixpoint theorem [92, Proposition 2.5], which shows how to build least fixpoints of certain functions. To apply the Knaster-Tarski theorem, we need to guarantee the existence of a partial order with a least element and that the function whose fixpoint we wish to find is continuous.

Let us start by defining the partial order. The intuition behind this partial order is that an event structure E_1 is "inside" another event structure E_2, denoted $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$, whenever the drawing of E_2 is the same as E_1 when we remove the events that are in E_2 but not in E_1. This interpretation considers the copies made during the sequential composition, which differs from Definition 4.2.11. Formally, the order is an adaptation from [118] to our setting and is defined as follows:

Definition 4.3.3. Let $\mathsf{E}_1 = (E_1, \leq_1, \#_1)$ and $\mathsf{E}_2 = (E_2, \leq_2, \#_2)$ be event structures. Say $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ if:

$$E_1 \subseteq E_2$$

$$\forall e, e' \,.\, e \leq_1 e' \Leftrightarrow e, e' \in E_1 \wedge e \leq_2 e'$$

$$\forall e, e' \,.\, e \#_1 e' \Leftrightarrow e, e' \in E_1 \wedge e \#_2 e'$$

At this point, it is worth explaining why we opted to delay the introduction of cyclic behavior until this section. It is known that introducing cyclic behavior requires a notion of ordering. For the case of event

structures, the ordering notion defined in Definition 4.3.3 is too strict. Example 4.2.12 illustrates this, showing that sequential composition is not left-monotone (where we implicitly rely on Definition 4.3.3). This strictness affects how recursion behaves within sequential composition, which in turn justifies the syntactic restrictions imposed on it. Furthermore, this ordering also considers the copies made by the sequential composition. When attempting to establish an equivalence between the operational and denotational semantics, Definition 4.3.3 prevents us from proving certain useful intermediate results (*e.g.* Lemma 4.2.17 and Lemma 4.4.1), which are necessary to obtain soundness and adequacy. Hence, we needed to loosen the ordering notion to ignore the copies created by the sequential composition, which led to the creation of Definition 4.2.11. To sum up, we have two notions of ordering: one for establishing a relation between semantics (Definition 4.2.11) and another for interpreting the recursive command (Definition 4.3.3).

We now start to check the conditions of the Knaster-Tarski theorem. We first verify that Definition 4.3.3 is a partial order with a least element (recall that the proofs can be found in Appendix A.1).

Lemma 4.3.4. $\trianglelefteq$ is a partial order.

Lemma 4.3.5. Define $\bot = (\emptyset, \emptyset, \emptyset)$. $\bot$ is the least element of $\trianglelefteq$.

We then define what it means to be a least upper bound in terms of event structures, demonstrate that it is an event structure, and prove that it is indeed a least upper bound of an ω-chain of event structures.

Definition 4.3.6. Let $\mathsf{E}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}_n \trianglelefteq \ldots$ be a ω-chain. Let $\mathsf{E}^\omega = (E^\omega, \leq^\omega, \#^\omega)$ be its least upper bound where:

- $E^\omega = \cup_{n\in\omega} E_n$
- $\leq^\omega = \cup_{n\in\omega} \leq_n$
- $\#^\omega = \cup_{n\in\omega} \#_n$

Lemma 4.3.7. E^ω is an event structure.

Lemma 4.3.8. Let $\mathsf{E}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}_n \trianglelefteq \ldots$ be a ω-chain. Then E^ω is its least upper bound.

Now we define what it means for an operator in event structures to be monotone and continuous with respect to Definition 4.3.3. Intuitively, an operator is monotone if it preserves the ordering. For example, for the parallel composition we say that it is monotone if $\mathsf{E}_1 \trianglelefteq \mathsf{E}_1'$ and $\mathsf{E}_2 \trianglelefteq \mathsf{E}_2'$ then $\mathsf{E}_1 \,||\, \mathsf{E}_2 \trianglelefteq \mathsf{E}_1' \,||\, \mathsf{E}_2'$. An operator is continuous if the action of the operator on the least upper bounds of event structures is the

same as the least upper bound of the action of the operator on the event structures. For example, for the parallel composition we say that it is continuous if $\bigsqcup_n \mathsf{E}_n \,||\, \bigsqcup_m \mathsf{E}_m = \bigsqcup_{n,m}(\mathsf{E}_n \,||\, \mathsf{E}_m)$. Formally [115, Definition 2.8]:

Definition 4.3.9. Let op be an n-ary operation on the class of event structures $\mathbb{E}$. Say op is *monotonic* iff when for event structures we have

$$\mathsf{E}_1 \trianglelefteq \mathsf{E}'_1, \ldots, \mathsf{E}_n \trianglelefteq \mathsf{E}'_n \text{ then } op(\mathsf{E}_1, \ldots, \mathsf{E}_n) \trianglelefteq op(\mathsf{E}'_1, \ldots, \mathsf{E}'_n)$$

Say op is continuous iff for all countable chains

$$\begin{aligned} &\mathsf{E}_{11} \trianglelefteq \mathsf{E}_{12} \trianglelefteq \cdots \trianglelefteq \mathsf{E}_{1i} \trianglelefteq \ldots \\ &\vdots \\ &\mathsf{E}_{n1} \trianglelefteq \mathsf{E}_{n2} \trianglelefteq \cdots \trianglelefteq \mathsf{E}_{ni} \trianglelefteq \ldots \end{aligned}$$

we have

$$op\left(\bigsqcup_i \mathsf{E}_{1i}, \ldots, \bigsqcup_i \mathsf{E}_{ni}\right) = \bigsqcup_i op(\mathsf{E}_{1i}, \ldots, \mathsf{E}_{ni})$$

where $\bigsqcup$ denotes the least upper bound w.r.t $\trianglelefteq$.

To the previous definition, we add that an operation is continuous iff it is continuous in each argument separately [115].

We now show that the operators of the language are monotone with respect to Definition 4.3.3. Recall that the sequential composition is only right monotone because of the restriction imposed in the syntax, which requires the free variables and bounded variables of the left command to be empty.

Lemma 4.3.10. Let $\mathsf{E}, \mathsf{E}_1, \mathsf{E}_2$ be event structures. If $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ then $\mathsf{E} \,;\, \mathsf{E}_1 \trianglelefteq \mathsf{E} \,;\, \mathsf{E}_2$.

Lemma 4.3.11. Let $\mathsf{E}_1, \mathsf{E}'_1, \mathsf{E}_2, \mathsf{E}'_2$ be event structures. If $\mathsf{E}_1 \trianglelefteq \mathsf{E}'_1$ and $\mathsf{E}_2 \trianglelefteq \mathsf{E}'_2$ then $\mathsf{E}_1 \,||\, \mathsf{E}_2 \trianglelefteq \mathsf{E}'_1 \,||\, \mathsf{E}'_2$.

Lemma 4.3.12. Let $\mathsf{E}_1, \mathsf{E}'_1, \mathsf{E}_2, \mathsf{E}'_2$ be event structures. If $\mathsf{E}_1 \trianglelefteq \mathsf{E}'_1$ and $\mathsf{E}_2 \trianglelefteq \mathsf{E}'_2$ then E_1 or $\mathsf{E}_2 \trianglelefteq \mathsf{E}'_1$ or E'_2.

The following lemma from [115, Lemma 2.9] is helpful to show that the sequential, the concurrent, and the non-deterministic operators are continuous. Intuitively, a function is continuous on event structures if it is monotonic and act continuously on the component sets of events ordered by inclusion.

Lemma 4.3.13. Let op be a unary operation on the class of event structures $\mathbb{E}$. Then op is continuous iff

1. op is monotonic

2. if $\mathsf{E}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}_n \trianglelefteq \ldots$ is an ω-chain then each event of $op(\bigsqcup_n \mathsf{E}_n)$ is an event of $\bigsqcup_n op(\mathsf{E}_n)$.

We proceed to show that the operators are continuous. Note that since the sequential composition is only right-monotone, then it will only be right-continuous.

Lemma 4.3.14. $\bigsqcup_m(\mathsf{E}\,;\,\mathsf{E}_m) = \mathsf{E}\,;\,\bigsqcup_m \mathsf{E}_m$.

Lemma 4.3.15. $\bigsqcup_{n,m}(\mathsf{E}_n \,||\, \mathsf{E}_m) = \bigsqcup_n \mathsf{E}_n \,||\, \bigsqcup_m \mathsf{E}_m$.

Lemma 4.3.16. $\bigsqcup_{n,m}(\mathsf{E}_n \text{ or } \mathsf{E}_m) = \bigsqcup_n \mathsf{E}_n \text{ or } \bigsqcup_m \mathsf{E}_m$.

Lemma 4.3.17 is a version of the Kleene fixed-point theorem for the case of event structures. Intuitively, it tells us how to build a least fixed-point for a continuous operator.

Lemma 4.3.17. Let Γ be a continuous operation on the class of event structures $\mathbb{E}$. Let $\bot = (\emptyset, \emptyset, \emptyset) \in \mathbb{E}$. Define $fix(\Gamma)$ to be the least upper bound of the chain $\bot \trianglelefteq \Gamma(\bot) \trianglelefteq \cdots \trianglelefteq \Gamma^n(\bot) \trianglelefteq \ldots$. Then $\Gamma(fix(\Gamma)) = fix(\Gamma)$.

We now update Definition 4.2.9 to accommodate recursion.

Definition 4.3.18. Define an environment to be a function $\gamma : Var \to \mathbb{E}$ from variables to event structures. For a command C and an environment γ define $[\![C]\!]_\gamma$ [1] as follows:

$$
\begin{aligned}
&[\![skip]\!]_\gamma = (\{sk\}, \{sk \leq sk\}, \emptyset)\\
&[\![a]\!]_\gamma = (\{a\}, \{a \leq a\}, \emptyset)\\
&[\![C_1\,;\,C_2]\!]_\gamma = [\![C_1]\!]_\gamma\,;\,[\![C_2]\!]_\gamma\\
&[\![C_1 \,||\, C_2]\!]_\gamma = [\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma\\
&[\![C_1 \text{ or } C_2]\!]_\gamma = [\![C_1]\!]_\gamma \text{ or } [\![C_2]\!]_\gamma\\
&[\![X]\!]_\gamma = \gamma(X)\\
&[\![\mu X.C]\!]_\gamma = fix(\Gamma^{C,\gamma})
\end{aligned}
$$

where $\Gamma^{C,\gamma} : \mathbb{E} \to \mathbb{E}$ is given by $\Gamma^{C,\gamma}(\mathsf{E}) = [\![C]\!]_{\gamma(X \leftarrow \mathsf{E})}$.

Since $\Gamma^{C,\gamma}$ is a new operator, we need to show that it is continuous. To do this, it is useful to know that fix is continuous [3].

[1] whenever a command is not recursive, for simplicity we may drop the environment γ from $[\![-]\!]_\gamma$

Lemma 4.3.19. $\Gamma^{C,\gamma}$ is continuous.

With the introduction of the recursive command, we need the two following lemmas for Lemma 4.4.1.

Lemma 4.3.20 (Substitution lemma)**.** $[\![C'[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma = [\![C']\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}$

Lemma 4.3.21. $[\![\mu X.C]\!]_\gamma = [\![C]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}$

We present an example where we illustrate a fragment of the event structure that is obtained by interpreting a recursive command.

Example 4.3.22. Let us reuse the command in Example 4.3.2. A fragment of the event structure that is generated by interpreting $[\![\mu X.(skip \operatorname{or} a\,;\, X)]\!]_\gamma$ is given in Figure 39.

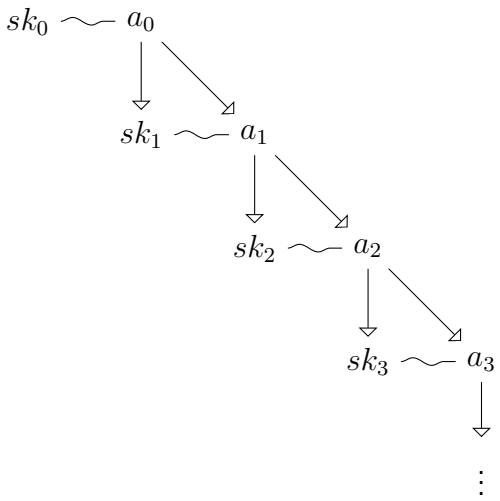


Figure 39: Unrolling the event structure of $[\![\mu X.(skip \operatorname{or} a\,;\, X)]\!]_\gamma$

Note that the events have in subscript a natural number indicating how many times the recursive command has been unfolded.

4.4 Soundness and Adequacy

This section connects the operational and the denotational semantics via a soundness and adequacy theorem. These results are supported by previous ones obtained in Section 4.2 and Section 4.3. The proofs can be found in Appendix A.1.

We now begin to establish a relation between the operational and denotational semantics. We start with soundness, *i.e.* any behavior observed in the operational semantics is captured by the denotational semantics. To do this, we first need a lemma that establishes a relation between the small-step and

denotational semantics. This step is needed because the big-step semantics is defined via the small-step semantics.

The intuition behind Lemma 4.4.1 is that if the action l triggers a transition from C to C', then removing l from $[\![C]\!]_\gamma$ is similar to have $[\![C']\!]_\gamma$. This can be further clarified if we recall the intuition given when defining the removal of an initial event from an event structure (Definition 4.2.19).

Lemma 4.4.1 (Small-soundness)**.** If $C \xrightarrow{l} C'$ then $\forall \gamma,\ [\![C']\!]_\gamma \equiv [\![C]\!]_\gamma \backslash l$.

With a result that establishes a relation between the small-step and denotational semantics, we now focus on the big-step semantics. Recall that the big-step semantics creates words, which are sequences of labels. On the other hand, event structures have the notion of covering chains, which are finite sequences representing the execution of events. We can exploit this similarity to formulate an equivalence between the big-step and denotational semantics.

Theorem 4.4.2 (Soundness)**.** If $C \xRightarrow{\omega} C'$ then $\exists x \in \mathcal{C}([\![C]\!]_\gamma)$ such that $\emptyset \xrightarrow{\omega}\subset x$.

Having achieved soundness, we turn our attention to adequacy. Adequacy is the converse of soundness, *i.e.* if a behavior is observed in the denotational semantics, then the operational semantics should be able to capture it.

Since adequacy is the reverse of soundness, following a similar procedure to prove it makes sense. Hence, we first need to establish a relation between denotational and small-step semantics. To do this, we take advantage once again of the intuition behind removing an initial event from an event structure.

Lemma 4.4.3 (Small-adequacy)**.** Let $l \in \mathcal{I}([\![C]\!]_\gamma)$. Then $\exists C' \in (Com \cup \{\checkmark\})$ s.t. $C \xrightarrow{l} C'$ and $[\![C]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma$.

Remark 1. Lemma 4.4.3 is a weak version of an adequacy lemma because, from an initial event $l \in \mathcal{I}([\![C]\!]_\gamma)$, we have information about the small-step operational semantics and denotational semantics. A stronger adequacy lemma would state something like: "If $[\![C]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma$, for $l \in \mathcal{I}(\mathsf{E})$, then $C \xrightarrow{l} C'$". However, this version is too strong because we may have $[\![C]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma$ and not $C \xrightarrow{l} C'$. For example consider the event structure corresponding to the command $C = l\,;\,(a \,\mathsf{or}\, b)\,;\,c$, depicted in Figure 40.

It is straightforward to see that the initial event is l. Hence we can remove it to obtain the following event structure, in Figure 41.

It is straightforward to see that the resultant event structure, in Figure 41, corresponds to both commands $(a\,;\,c)$ or $(b\,;\,c)$ and $(a \,\mathsf{or}\, b)\,;\,c$. Now, if we apply the rules of the small-step semantics

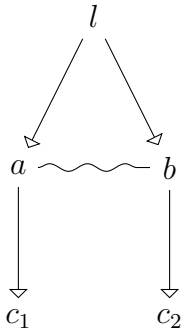


Figure 40: Event structure of $[\![l \,;\, (a \text{ or } b) \,;\, c]\!]$

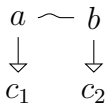


Figure 41: Event structure of $[\![l \,;\, (a \text{ or } b) \,;\, c]\!] \backslash l$

to $l \,;\, (a \text{ or } b) \,;\, c$ we obtain the following: $l \,;\, (a \text{ or } b) \,;\, c \xrightarrow{l} (a \text{ or } b) \,;\, c$. As it is possible to see, we have that $(a \text{ or } b) \,;\, c \neq (a \,;\, c) \text{ or } (b \,;\, c)$. Hence, the lemma would fail.

With Lemma 4.4.3, we can turn our efforts to relate the denotational semantics and the big-step semantics. To formulate the adequacy theorem, we reverse the reasoning used for Theorem 4.4.2. However, we need to be careful when selecting the configurations. Specifically, we avoid the empty configuration because the big-step semantics does not allow empty words. The adequacy theorem is then formulated as follows:

Theorem 4.4.4 (Adequacy)**.** If $\emptyset \neq x \in \mathcal{C}([\![C]\!]_\gamma)$ and $\emptyset \xrightarrow{\omega} x$ then $\exists C'$ s.t. $C \overset{\omega}{\twoheadrightarrow} C'$.

Summing up both theorems: soundness (Theorem 4.4.2) states that every word ω derived from the big-step semantics corresponds to a covering chain, and consequently to a configuration. Conversely, adequacy (Theorem 4.4.4) indicates that if we have a non-empty covering chain ω, then there exists a command C' reachable from C by executing ω.

We now delve into two examples to show soundness and adequacy in practice.

Example 4.4.5. The command in Example 4.1.1, $(a \,;\, b) \text{ or } (c \,||\, d)$, is depicted as an event structure in Figure 11. From Example 4.1.1, we have the words cd and dc, and from Example 3.2.9 we have the covering chains $\emptyset \xrightarrow{d} \{d\} \xrightarrow{c} \{d, c\}$ and $\emptyset \xrightarrow{c} \{c\} \xrightarrow{d} \{d, c\}$. It is then straightforward to see that each word in the operational semantics corresponds to a covering chain (soundness), and that each covering chain corresponds to a word (adequacy).

Example 4.4.6. Figure 42 shows the event structure corresponding to the interpretation of $[\![(a\,;\,b)\,||\,c]\!]$. The set of configurations is $\{\emptyset, \{a\}, \{c\}, \{a,b\}, \{a,c\}, \{a,b,c\}\}$, where we note that in the presence of concurrent events, a configuration has more than one possible covering chain.

To illustrate the equivalence between the operational and the denotational semantics, recall the words that can be formed by the big-step semantics in Example 4.1.1: a, c, ab, ac, ca, abc, acb, and cab.

Each word corresponds to a covering chain, which represents a configuration. For instance, the words ac and ca correspond to the covering chains $\emptyset \xrightarrow{a}\subset \{a\} \xrightarrow{c}\subset \{a,c\}$ and $\emptyset \xrightarrow{c}\subset \{c\} \xrightarrow{a}\subset \{a,c\}$, respectively. These covering chains correspond to the configuration $\{a,c\}$. Conversely, for each covering chain, a corresponding word exists.

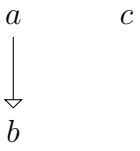


Figure 42: Event structure of $[\![(a\,;\,b)\,||\,c]\!]$

To conclude this section, let us develop an example that shows Theorem 4.4.2 and Theorem 4.4.4 in practice for a recursive command.

Example 4.4.7. Recall the command used in Example 4.3.2 and Example 4.3.22, $\mu X.(skip \text{ or } a\,;\,X)$. From the former, we know that the words formed by the big-step semantics are a possibly empty sequence of a's that finishes with sk. Consider the word $\omega = a\ a\ sk$, which gives $\mu X.(skip \text{ or } a\,;\,X) \xrightarrow{\omega}\!\!\!\!\rightarrow \checkmark$. From Example 4.3.22, we can deduce that we have a configuration

$$x = \{a_0, a_1, sk_2\} \in \mathcal{C}([\![\mu X.(skip \text{ or } a\,;\,X)]\!]_\gamma)$$

such that $\emptyset \xrightarrow{\omega}\subset x$. On the other hand, from

$$\emptyset \xrightarrow{a_0}\subset \{a_0\} \xrightarrow{a_1}\subset \{a_0, a_1\} \xrightarrow{sk_2}\subset \{a_0, a_1, sk_2\}$$

i.e. $\emptyset \xrightarrow{\omega}\subset \{a_0, a_1, sk_2\}$, we have that $\mu X.(skip \text{ or } a\,;\,X) \xrightarrow{\omega}\!\!\!\!\rightarrow \checkmark$.

Furthermore, we can extend this argument as follows. Let $\omega = a^n\ sk$, with n being a finite number and a^n denoting a sequence of n computations triggered by the action a. We have that

$$\mu X.(skip \text{ or } a\,;\,X) \xrightarrow{\omega}\!\!\!\!\rightarrow \checkmark$$

On the event structure side, we have a configuration

$$x = \cup_{i=0}^{n}\{a_i\} \cup \{sk_{n+1}\} \in \mathcal{C}([\![\mu X.(skip \text{ or } a\,;\,X)]\!]_\gamma)$$

with $\emptyset \xrightarrow{a_0}\subset \ldots \xrightarrow{sk_{n+1}}\subset \cup_{i=0}^{n}\{a_i\} \cup \{sk_{n+1}\}$.

Chapter 5

Adding probabilistic choice

5.1 Language

We continue using event structures to give denotational semantics to different languages, depending on the command added to the core language in Section 3.5. In this chapter, we focus on probabilistic behavior. We shall thus use probabilistic event structures to furnish a probabilistic language with a denotational semantics. Following the methodology of Chapter 4, we first introduce the language without cyclic behavior, along with the respective definitions of its operators in terms of probabilistic event structures and the equivalence between semantics. Later, in Section 5.3, we extend the language to include cyclic behavior.

We extend the BNF grammar of the core language, in Section 3.5, with probabilistic choice, as follows:

$$C ::= skip \mid a \in Act \mid C\,;C \mid C \,||\, C \mid C \,+_p\, C$$

$C \,+_p\, C$ is the probabilistic command with $p \in]0,1[$. The reason for requiring $p \in]0,1[$ arises when adapting the removal of an initial event (Definition 4.2.19) to the probabilistic setting (Definition 5.2.16). As shall be seen, removing the initial event from a probabilistic event structure also requires some adjustments in the probabilities of the resultant probabilistic event structure.

With the introduction of probabilities, the operational semantics in Section 3.5 needs to be properly adjusted to the new setting. For that, we extend the set of labels from Section 3.5 with a new label τ, *i.e.* $L' = L \cup \{\tau\}$ and let it be ranged by l'. The label τ represents an invisible transition, concretely one triggered by the probabilistic command. Given a function $\psi : X \to [0,1]$, we denote by $sup(\psi) = \{x \in X \mid \psi(x) > 0\}$ the support of ψ, *i.e.* the set of elements $x \in X$ whose probability $\psi(x)$ is greater than zero. Then we let $\mathsf{V}_1(X) = \{\psi : X \to [0,1] \mid sup(\psi)$ finite$, \sum_{x\in X} \psi(x) = 1\}$ be the set of probabilistic distributions with finite support. At last, we define the small-step semantics (as a labeled Segala automaton [95]), $\longrightarrow \subseteq Com \times \mathsf{V}_1(L' \times (Com \cup \{\checkmark\}))$, obeying the rules in Figure 43:

$$skip \longrightarrow 1 \cdot (sk, \checkmark) \qquad a \longrightarrow 1 \cdot (a, \checkmark) \qquad C_1 +_p C_2 \longrightarrow p \cdot (\tau, C_1) + (1-p) \cdot (\tau, C_2)$$

$$\frac{C_1 \longrightarrow 1 \cdot (l, \checkmark)}{C_1 \,;\, C_2 \longrightarrow 1 \cdot (l, C_2)} \qquad \frac{C_1 \longrightarrow 1 \cdot (l, C_1')}{C_1 \,;\, C_2 \longrightarrow 1 \cdot (l, C_1' \,;\, C_2)} \qquad \frac{C_1 \longrightarrow \sum_i p_i \cdot (\tau, C_i)}{C_1 \,;\, C_2 \longrightarrow \sum_i p_i \cdot (\tau, C_i \,;\, C_2)}$$

$$\frac{C_1 \longrightarrow 1 \cdot (l, \checkmark)}{C_1 \,||\, C_2 \longrightarrow 1 \cdot (l, C_2)} \qquad \frac{C_1 \longrightarrow 1 \cdot (l, C_1')}{C_1 \,||\, C_2 \longrightarrow 1 \cdot (l, C_1' \,||\, C_2)} \qquad \frac{C_1 \longrightarrow \sum_i p_i \cdot (\tau, C_i)}{C_1 \,||\, C_2 \longrightarrow \sum_i p_i \cdot (\tau, C_i \,||\, C_2)}$$

$$\frac{C_2 \longrightarrow 1 \cdot (l, \checkmark)}{C_1 \,||\, C_2 \longrightarrow 1 \cdot (l, C_1)} \qquad \frac{C_2 \longrightarrow 1 \cdot (l, C_2')}{C_1 \,||\, C_2 \longrightarrow 1 \cdot (l, C_1 \,||\, C_2')} \qquad \frac{C_2 \longrightarrow \sum_j p_j \cdot (\tau, C_j)}{C_1 \,||\, C_2 \longrightarrow \sum_j p_j \cdot (\tau, C_1 \,||\, C_j)}$$

Figure 43: Rules of the probabilistic small-step operational semantics

In the rules in Figure 43, the arrow no longer carries the action's label that triggered the transition. Instead, we represent transitions as pairs consisting of the label and any remaining computation, if that is the case. This leads probability distributions to be of the form $\sum_i p_i \cdot (l_i, C_i)$, where p_i indicates the probability of the pair (l_i, C_i). We interpret the transition $C \longrightarrow \sum_i p_i \cdot (l_i, C_i)$ as follows: from C we transition to C_i with probability p_i by performing the action labeled by l_i, for all (l_i, C_i) in $sup(\sum_i p_i \cdot (l_i, C_i))$. Now, let us delve into the rules. The rules for $skip$ and a are straightforward, since they transition with probability 1 to the terminal symbol. Regarding the probabilistic command, the transition is labeled by τ and it goes to a probabilistic distribution where the following command to be executed is either C_1 with probability p or C_2 with probability $1 - p$. The transition of the probabilistic command was defined in this way to match the essence of the quantum measurement command in Section 6.1. This influences how sequential and parallel commands behave: if an atomic action triggers the transition, we transit with probability 1 to a pair formed by the action's label and the command that remains to be evaluated; otherwise, if a probabilistic command triggers the transition, then we transit to a sum where each element is a pair composed of the label τ and a command to be executed, each with a certain probability of occurring.

Similar to Section 4.1, we also equip the language with a big-step semantics, with the words now formed by the labels in L'. Formally, the big-step semantics is defined as the relation $\twoheadrightarrow \subseteq Com \times \mathsf{V}_1((L')^+ \times (Com \cup \{\checkmark\}))$ obeying the rules in Figure 44.

Akin to what was done in Section 4.1, we define the big-step semantics by resorting to the small-step

$$\frac{C \longrightarrow \sum_i p_i \cdot (l', C_i)}{C \twoheadrightarrow \sum_i p_i \cdot (l', C_i)} \qquad \frac{C \longrightarrow \sum_i p_i \cdot (l', C_i) \qquad \forall i\, C_i \twoheadrightarrow \sum_j p_j \cdot (\omega_{ij}, C_{ij})}{C \twoheadrightarrow \sum_i p_i \cdot \Big(\sum_j p_j \cdot (l' : \omega_{ij}, C_{ij})\Big)}$$

Figure 44: Rules of the big-step operational semantics

semantics. Similarly, the big-step semantics can be viewed as the transitive closure of the small-step semantics: the left rule represents a single computation step. In contrast, the right rule captures the transitive aspect, allowing multiple steps. The rule on the left states that if the word has size 1, the big-step semantics coincide with the small-step semantics. The rule on the right is applied whenever the word has size greater than 1. Specifically, if C transitions to $\sum_i p_i \cdot (l', C_i)$ and each C_i further transitions to $\sum_j p_j \cdot (\omega_{ij}, C_{ij})$, then by appending l' to each ω_{ij}, we can transition from C to $\sum_i p_i \cdot \Big(\sum_j p_j \cdot (l' : \omega_{ij}, C_{ij})\Big)$. Note that in the rule's conclusion, for each i, the probabilities from the small-step transition are multiplied by those from the big-step transition.

The operational semantics developed in this section need to consider two types of transition: probabilistic and non-deterministic. The former arises from the probabilistic command. The latter comes from the concurrent command. Hence, when drawing the execution tree of a command, it is necessary to distinguish between these two transitions. To illustrate this, we present an example that incorporates all language operators.

Example 5.1.1. In Figure 45, the execution tree uses straight arrows to denote a transition from a command to a distribution, which we denote by $\bullet$, labeled by the triggering action, and wiggly arrows to represent a transition from a distribution to a command labeled by the associated probability.

Thus from $(a\,;\,b)\ +_p\ (c\,||\,d)$ we transit with τ to the distribution $p \cdot (a\,;\,b) + (1-p) \cdot (c\,||\,d)$, which transits with probability p to $a\,;\,b$ and with probability $1-p$ to $c\,||\,d$. For the former, we reach the end of the computation by executing first a and then b. For the latter, since it is a concurrent program, to finish the computation, we can either execute first c and then d, or we can execute first d and then c.

Based on Figure 45 and following the rules in Figure 44, we can deduce that with probability p the word τab leads to a final computation and the same behavior is captured with probability $1-p$ with the words τcd and τdc.

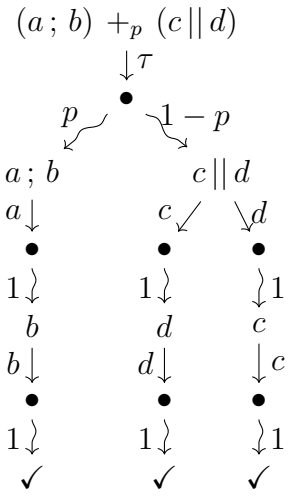


Figure 45: Segala Automaton of $(a\,;\,b)\ +_p\ (c\,||\,d)$

5.2 Constructions on Probabilistic Event Structures

The constructions on probabilistic event structures are an extension of the ones defined in Section 4.2. Hence, the sequential and parallel composition explanation will be focused on the valuation, and we will provide more details about the probabilistic choice.

Let P_1 and P_2 be two probabilistic event structures. For the valuation of the sequential composition, we note the following: either the configuration belongs to $\mathcal{C}(\mathsf{P}_1)$ and in that case the valuation of the sequential composition equals the valuation of P_1, or the configuration has elements of both probabilistic event structures. In that case, we multiply the valuation of a maximal configuration in P_1 with the valuation of a configuration in P_2 whose events are reached by the maximal configuration of P_1.

Definition 5.2.1 (Sequential Composition)**.** Let $\mathsf{P}_1 = (E_1, \leq_1, \#_1, v_1)$ and $\mathsf{P}_2 = (E_2, \leq_2, \#_2, v_2)$ be probabilistic event structures. Define $\mathsf{P}_1\,;\,\mathsf{P}_2 = (E, \leq, \#, v)$ as:

$$
\begin{aligned}
E &= E_1 \uplus (E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{P}_1)) \\
\leq &= \{e_1 \leq e_1' \mid e \leq_1 e'\} \cup \{(e_2, x) \leq (e_2', x) \mid\ e_2 \leq_2 e_2'\} \cup \{e_1 \leq (e_2, x) \mid e_1 \in x\} \\
\# &= \{e \# e' \mid \exists (e_1 \leq e, e_1' \leq e')\ .\ e_1 \#_1 e_1'\} \cup \{(e_2, x) \# (e_2', x) \mid e_2 \#_2 e_2'\} \\
\forall x \in \mathcal{C}(\mathsf{P}_1\,;\,\mathsf{P}_2)\,.\, v(x) &= \begin{cases} v_1(x) & \text{if } x \in \mathcal{C}(\mathsf{P}_1) \\ v_1(x_1) \cdot v_2(x_2) & \text{if } x = x_1 \cup (x_2 \times \{x_1\}) \\ & \qquad \text{with } x_1 \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}_1),\ x_2 \in \mathcal{C}(\mathsf{P}_2) \end{cases}
\end{aligned}
$$

where $E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{P}_1) = \{(e, x) \mid e \in E_2,\ x \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}_1)\}$ and $x_2 \times \{x_1\} = \{(e_2, x_1) \mid e_2 \in x_2\}$.

Lemma 5.2.2. Let P_1 and P_2 be probabilistic event structures. $\mathsf{P}_1 \,;\, \mathsf{P}_2$ is a probabilistic event structure.

For the parallel composition, and by considering the intuition that the parallel composition places "side-by-side" the two event structures, the valuation is determined by multiplying the valuations obtained from projecting the configuration in $\mathsf{P}_1 \,||\, \mathsf{P}_2$ onto the corresponding configurations of P_1 and P_2.

Definition 5.2.3 (Parallel Composition)**.** Let $\mathsf{P}_1 = (E_1, \leq_1, \#_1, v_1)$ and $\mathsf{P}_2 = (E_2, \leq_2, \#_2, v_2)$ be probabilistic event structures. Define $\mathsf{P}_1 \,||\, \mathsf{P}_2 = (E, \leq, \#, v)$ as:

$$
\begin{aligned}
E &= E_1 \uplus E_2 \\
\leq &= \leq_1 \uplus \leq_2 \\
\# &= \#_1 \uplus \#_2 \\
\forall x \in \mathcal{C}(\mathsf{P}_1 \,||\, \mathsf{P}_2) \,.\, v(x) &= v_1(x \cap E_1) \cdot v_2(x \cap E_2)
\end{aligned}
$$

Lemma 5.2.4. Let P_1 and P_2 be probabilistic event structures. $\mathsf{P}_1 \,||\, \mathsf{P}_2$ is a probabilistic event structure.

We now turn our attention to the probabilistic choice. As it can be deduced, the probabilistic choice of two probabilistic event structures is going to be very similar to the non-deterministic choice of event structures in Definition 4.2.6. Indeed when composing two probabilistic event structures P_1 and P_2 under the probabilistic operator, the events of P_1 will all be in conflict with the events of P_2 (recall that the meaning of $C_1 +_p C_2$ is to execute C_1 with probability p or execute C_2 with probability $1-p$). Furthermore, each event of P_1 and P_2 will causally depend on an event representing the invisible action τ. In other words, the invisible action τ should be the initial event. Regarding the valuations, if the configuration obtained by removing τ belongs to $\mathcal{C}(\mathsf{P}_1)$, then we multiply by p the valuation in P_1. If the configuration obtained by removing τ belongs to $\mathcal{C}(\mathsf{P}_2)$, then we multiply by $(1-p)$. Otherwise, the value of the valuation is 1. Formally:

Definition 5.2.5 (Probabilistic choice)**.** Let $\mathsf{P}_1 = (E_1, \leq_1, \#_1, v_1)$ and $\mathsf{P}_2 = (E_2, \leq_2, \#_2, v_2)$ be probabilistic event structures. Define $\mathsf{P}_1 +_p \mathsf{P}_2 = (E, \leq, \#, v)$ as:

$$
\begin{aligned}
E &= \{\tau\} \uplus (E_1 \uplus E_2) \\
\leq &= \{\tau \leq e \mid e \in E\} \cup \leq_1 \uplus \leq_2 \\
\# &= \#_1 \uplus \#_2 \cup \{e_1 \# e_2 \mid e_1 \in E_1,\, e_2 \in E_2\} \cup \{e_2 \# e_1 \mid e_1 \in E_1,\, e_2 \in E_2\} \\
\forall x \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2) \,.\, v(x) &= \begin{cases} p \cdot v_1(x \backslash \tau) & \text{if } x \backslash \tau \in \mathcal{C}(\mathsf{P}_1) \\ (1-p) \cdot v_2(x \backslash \tau) & \text{if } x \backslash \tau \in \mathcal{C}(\mathsf{P}_2) \\ 1 & \text{if } x = \{\tau\} \vee x = \emptyset \end{cases}
\end{aligned}
$$

Lemma 5.2.6. Let P_1 and P_2 be probabilistic event structures. $\mathsf{P}_1 +_p \mathsf{P}_2$ is a probabilistic event structure.

Note that, in the valuation of Definition 5.2.5, the configurations considered for P_1 and P_2 are disjoint. This follows from the fact that, in $\mathsf{P}_1 +_p \mathsf{P}_2$, every event in P_1 is in conflict with every event in P_2. Since configurations are conflict-free by definition, no configuration $x \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2)$ can contain events from both P_1 and P_2. Hence, the configurations $x \backslash \tau$ used in the valuation for P_1 and P_2 are disjoint.

Furthermore, from Definition 5.2.5, it is also possible to notice that $\mathsf{P} +_p \mathsf{P} \neq \mathsf{P}$. This is a consequence of introducing the τ event in $\mathsf{P} +_p \mathsf{P}$.

The next example shows how to compose probabilistic event structures using the probabilistic operator.

Example 5.2.7. Recall the event structures E_1 and E_2 used in Example 4.2.8. Let us first adapt them to the probabilistic setting and rename them:

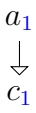
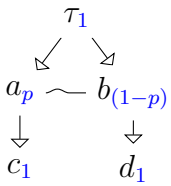


Figure 46: Probabilistic event structure P_1



Figure 47: Probabilistic event structure P_2

We use the same notation as in Example 3.3.2, *i.e.* we subscript each event with its respective probability. Note that for P_1 and P_2 the valuation of any configuration is always 1. Figure 48 shows the result of composing both probabilistic event structures under the probabilistic operator (Definition 5.2.5).



$$v(x) = \begin{cases} p & \text{if } a \in x \\ (1-p) & \text{if } b \in x \\ 1 & \text{otherwise} \end{cases}$$

Figure 48: Probabilistic event structure $\mathsf{P}_1 +_p \mathsf{P}_2$

From Figure 48 we notice that: τ is the initial event, which captures the essence that in a probabilistic command the first action to be executed is the invisible one; all the events of P_1 are in conflict with the events of P_2, similarly to what happens in the non-deterministic composition of event structures; and the events a and b have associated to them the probabilities p and $1-p$, respectively, because they are the first events to occur after τ. Furthermore, any configuration containing a has a valuation of p, while any configuration with b in it has a valuation of $1-p$. This intuition is confirmed by the valuation on the right side of Figure 48.

Although in Definition 5.2.5 we use the notation $\mathsf{P}_1 +_p \mathsf{P}_2$ to represent the probabilistic composition of probabilistic event structures, we may sometimes put the probabilities explicit on both sides, *i.e.* $p \cdot \mathsf{P}_1 + (1-p) \cdot \mathsf{P}_2$. That leaves us with $\mathsf{P}_1 +_p \mathsf{P}_2 = p \cdot \mathsf{P}_1 + (1-p) \cdot \mathsf{P}_2$.

Remark 2. Definition 5.2.5 can be generalized to allow the probabilistic composition of n probabilistic event structures. Consider we have a finite number n of probabilistic event structures. Let $1 \leq i \leq n$ and define $\sum_i p_i \cdot \mathsf{P}_i$, with $\sum_i p_i = 1$, as follows:

$$
\begin{aligned}
E &= \{\tau\} \uplus \biguplus_i E_i \\
\leq &= \{\tau \leq e \mid e \in E\} \cup \biguplus_i \leq_i \\
\# &= \biguplus_i \#_i \cup \{e_i \# e_j \mid e_i \in E_i,\, e_j \in E_j, 1 \leq i \neq j \leq n\} \\
&\qquad \cup \{e_j \# e_i \mid e_i \in E_i,\, e_j \in E_j, 1 \leq i \neq j \leq n\} \\
\forall x \in \mathcal{C}\left(\sum_i p_i \cdot \mathsf{P}_i\right).\, v(x) &= \begin{cases} p_i \cdot v_i(x \backslash \tau) & \text{if } x \backslash \tau \in \mathcal{C}(\mathsf{P}_i) \\ 1 & \text{if } x = \{\tau\} \vee x = \emptyset \end{cases}
\end{aligned}
$$

This will be useful when showing the equivalence between the operational and the denotational semantics.

We can now interpret the language shown in Section 5.1 as probabilistic event structures.

Definition 5.2.8. We interpret commands as follows:

$$
\begin{aligned}
&[\![skip]\!] = (\{sk\}, \{sk \leq sk\}, \emptyset, v(\{sk\}) = 1) \\
&[\![a]\!] = (\{a\}, \{a \leq a\}, \emptyset, v(\{a\}) = 1) \\
&[\![C_1 \,;\, C_2]\!] = [\![C_1]\!] \,;\, [\![C_2]\!] \\
&[\![C_1 \,||\, C_2]\!] = [\![C_1]\!] \,||\, [\![C_2]\!] \\
&[\![C_1 +_p C_2]\!] = [\![C_1]\!] +_p [\![C_2]\!]
\end{aligned}
$$

The following example shows how the denotational semantics works.

Example 5.2.9. Recall the command in Example 4.2.10:

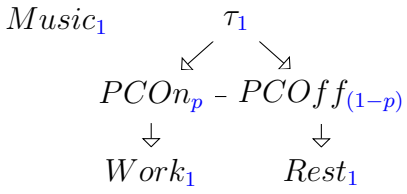
$$Music \,||\, ((PCOn \,;\, Work) \text{ or } (PCOff \,;\, Rest))$$

Now let us replace the non-deterministic operator with the probabilistic operator:

$$Music \,||\, ((PCOn\,;\, Work)\ +_p\ (PCOff\,;\, Rest))$$

In this new setting, at the same time the worker listens to music, he now behaves probabilistically: he can either turn on the computer and work with probability p or let the computer off and rest with probability $1-p$.

The respective probabilistic event structure is:



$$v(x) = \begin{cases} p & \text{if } PCOn \in x \\ 1-p & \text{if } PCOff \in x \\ 1 & \text{otherwise} \end{cases}$$

Figure 49: Probabilistic event structure of $Music \,||\, ((PCOn\,;\, Work)\ +_p\ (PCOff\,;\, Rest))$

From Figure 49, we can infer that whenever the event of turning on the computer appears in a configuration, the valuation of that configuration will be p. Similarly, for configurations where the event of turning off the computer occurs, the valuation will be $1-p$.

We focus on the equivalence between the operational and the denotational semantics. The intuition between the equivalence of both semantics is similar to the one given in the non-deterministic case. Concretely, the big-step semantics returns a probability distribution where we have an associated probability for each word. Hence, when we pick a word from the probability distribution along with its respective probability, we expect to find a covering chain whose configuration-valuation matches the probability of the word and vice-versa.

From Chapter 4, we can deduce that an extension of the notion of sub-event structures (Definition 4.3.3) to the probabilistic setting is unsuitable for relating the operational and the denotational semantics. Hence, we extend the notion of sub-similar event structures (Definition 4.2.11) to the probabilistic realm. Recall what was said regarding plain and composite events in Section 4.2 above Definition 4.2.11.

Definition 5.2.10. Let $\mathsf{P}_1 = (E_1, \leq_1, \#_1, v_1)$ and $\mathsf{P}_2 = (E_2, \leq_2, \#_2, v_2)$ be probabilistic event structures. Say $\mathsf{P}_1 \sqsubseteq \mathsf{P}_2$ whenever exists an injective function $f : E_1 \to E_2$ such that $\forall e, e' \in E_1$:

$$\pi(f(e)) = \pi(e)$$
$$e \leq_1 e' \implies f(e) \leq_2 f(e')$$

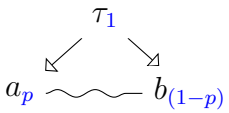
$$e\#_1 e' \implies f(e)\#_2 f(e')$$
$$\forall x \in \mathcal{C}(\mathsf{P}_1), y \in \mathcal{C}(\mathsf{P}_2)\,.\, f[x] \subseteq y \Rightarrow v_1(x) \geq v_2(y)$$

We say that two probabilistic event structures $\mathsf{P}_1, \mathsf{P}_2$ are similar, denoted $\mathsf{P}_1 \equiv \mathsf{P}_2$, iff $\mathsf{P}_1 \sqsubseteq \mathsf{P}_2$ and $\mathsf{P}_2 \sqsubseteq \mathsf{P}_1$.

To define $\sqsubseteq$ in the probabilistic setting, we based ourselves on the fact that, given two configurations x and y such that $x \subseteq y$ the probability of x must be greater or equal to the probability of y, *i.e.* $v(x) \geq v(y)$. This comes from the intuition that the probability tends to decrease as long as the computation continues.

The valuation condition in Definition 5.2.10 uses $v_1(x) \geq v_2(y)$ instead of $v_1(x) \geq v_2(f[x])$, because $f[x]$ may not be a configuration in P_2. For instance, let $\mathsf{P}_1 = a$ and $\mathsf{P}_2 = b \rightarrowtriangle a$. It is straightforward to see that $a \sqsubseteq b \rightarrowtriangle a$, via the identity function. However, $id[\{a\}] \notin \mathcal{C}(\mathsf{P}_2)$, since in P_2 the event a causally depends on b, and therefore cannot occur on its own.

Furthermore, note that in Definition 5.2.10, the conditions on causal and conflict relations use an implication rather than an equivalence (recall that in the classical case, the definition of sub-similar event structures used an equivalence – Definition 4.2.11). This adjustment is necessary because the interaction between probabilistic choice and parallel composition can duplicate events, introducing new causal and conflict dependencies. For example, in Figure 50, the event c is concurrent with all others, while in Figure 51 its copies c' and c'' now causally depend on τ and are each in conflict with events from the opposite branch (*i.e.* c' is in conflict with b and c'' is in conflict with a). Hence, if in Definition 5.2.10 we kept the "equivalence", then Lemma 5.4.2 would not hold and, consequently we would not be able to obtain soundness neither adequacy.



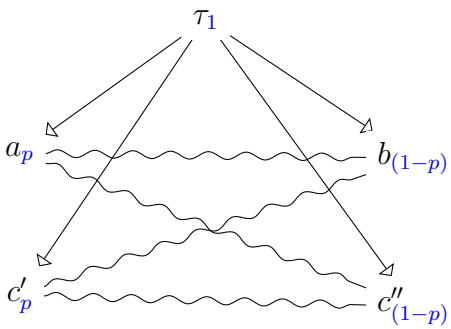
c_1

$$v(x) = \begin{cases} p & \text{if } a \in x \\ 1-p & \text{if } b \in x \\ 1 & \text{otherwise} \end{cases}$$

Figure 50: Probabilistic event structure of $(a +_p b) \,||\, c$

Look at Example 5.2.11 to see how impactful the condition on valuations is.

Example 5.2.11. Consider the three probabilistic event structures in Figure 52, Figure 53, and Figure 54:



$$v(x) = \begin{cases} p & \text{if } a \text{ or } c' \in x, \\ 1-p & \text{if } b \text{ or } c'' \in x, \\ 1 & \text{otherwise.} \end{cases}$$

Figure 51: Probabilistic event structure of $(a \,||\, c) \,+_p\, (b \,||\, c)$

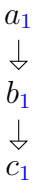


Figure 52: ProbES P_1

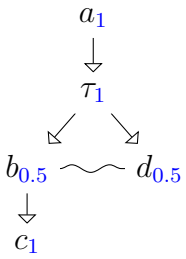


Figure 53: ProbES P_2

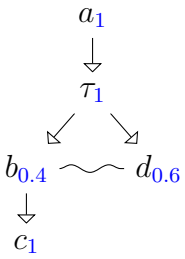


Figure 54: ProbES P_3

Let us first focus on P_1 and P_2. It is straightforward to see that the first three conditions in Definition 5.2.10 are satisfied (we are considering the injective function to be the identity, for all the cases considered in this example). Now we focus on the configurations. For P_1 we have that

$$\mathcal{C}(\mathsf{P}_1) = \{\emptyset_1, \{a\}_1, \{a,b\}_1, \{a,b,c\}_1\}$$

and for P_2 we have

$$\mathcal{C}(\mathsf{P}_2) = \{\emptyset_1, \{a\}_1, \{a,\tau\}_1, \{a,\tau,b\}_{0.5}, \{a,\tau,b,c\}_{0.5}, \{a,\tau,d\}_{0.5}\}$$

where we denote the respective valuation of each configuration in subscript. From this, it is straightforward to check that for every configuration $x_1 \in \mathcal{C}(\mathsf{P}_1)$ that is included in a configuration $x_2 \in \mathcal{C}(\mathsf{P}_2)$ we have $v_1(x_1) \geq v_2(x_2)$. In other words, the valuations of the configurations in $\mathcal{C}(\mathsf{P}_1)$ that are included in configurations of $\mathcal{C}(\mathsf{P}_2)$ are always greater than or equal to those in $\mathcal{C}(\mathsf{P}_2)$. Hence we can conclude that $\mathsf{P}_1 \sqsubseteq \mathsf{P}_2$.

A similar reasoning is applied to verify that $\mathsf{P}_1 \sqsubseteq \mathsf{P}_3$.

Now we focus on P_2 and P_3. These probabilistic event structures differ only in the probabilities associated with the events b and d. Hence, let us write down the set of configurations of P_3:

$$\mathcal{C}(\mathsf{P}_3) = \{\emptyset_1, \{a\}_1, \{a,\tau\}_1, \{a,\tau,b\}_{0.4}, \{a,\tau,b,c\}_{0.4}, \{a,\tau,d\}_{0.6}\}$$

The question to answer now is: does $\mathsf{P}_2 \sqsubseteq \mathsf{P}_3$ or $\mathsf{P}_3 \sqsubseteq \mathsf{P}_2$? The former does not hold because of $v_2(\{a, \tau, d\}) = 0.5 \not\geq 0.6 = v_3(\{a, \tau, d\})$. The latter also fails to hold because $v_3(\{a, \tau, b\}) = 0.4 \not\geq 0.5 = v_2(\{a, \tau, b\})$. Hence we have $\neg(\mathsf{P}_2 \sqsubseteq \mathsf{P}_3)$ and $\neg(\mathsf{P}_3 \sqsubseteq \mathsf{P}_2)$.

From $\neg(\mathsf{P}_2 \sqsubseteq \mathsf{P}_3)$ and $\neg(\mathsf{P}_3 \sqsubseteq \mathsf{P}_2)$ we can conclude that changing the probabilities of some events drastically affects how the probabilistic event structures are related with each other.

Now we study some properties of Definition 5.2.10 regarding the sequential, concurrent, and probabilistic commands (the proofs can be found in Appendix A.2). For this, we interpret the special symbol ✓ as the empty probabilistic event structure whose valuation of the empty configuration is 1, *i.e.* $[\![\checkmark]\!] = (\emptyset, \emptyset, \emptyset, v(\emptyset) = 1)$. It is then straightforward to see that for any probabilistic event structure P we have:

1. $\emptyset \,;\, \mathsf{P} \equiv \mathsf{P} \equiv \mathsf{P} \,;\, \emptyset$
2. $\emptyset \,||\, \mathsf{P} \equiv \mathsf{P} \equiv \mathsf{P} \,||\, \emptyset$

Furthermore, it is also straightforward to see that:

Lemma 5.2.12. Let $\mathsf{P}_1, \mathsf{P}_2$ be probabilistic event structures. Then $\mathsf{P}_1 \,||\, \mathsf{P}_2 = \mathsf{P}_2 \,||\, \mathsf{P}_1$.

We begin by showing that the sequential, parallel, and probabilistic operators are monotonic with respect to Definition 5.2.10.

Lemma 5.2.13. Let $\mathsf{P}_1, \mathsf{P}'_1, \mathsf{P}_2, \mathsf{P}'_2$ be probabilistic event structures. If $\mathsf{P}_1 \sqsubseteq \mathsf{P}'_1$ and $\mathsf{P}_2 \sqsubseteq \mathsf{P}'_2$ then $\mathsf{P}_1 \,;\, \mathsf{P}_2 \sqsubseteq \mathsf{P}'_1 \,;\, \mathsf{P}'_2$.

Lemma 5.2.14. Let $\mathsf{P}_1, \mathsf{P}'_1, \mathsf{P}_2, \mathsf{P}'_2$ be probabilistic event structures. If $\mathsf{P}_1 \sqsubseteq \mathsf{P}'_1$ and $\mathsf{P}_2 \sqsubseteq \mathsf{P}'_2$ then $\mathsf{P}_1 \,||\, \mathsf{P}_2 \sqsubseteq \mathsf{P}'_1 \,||\, \mathsf{P}'_2$.

Lemma 5.2.15. Let $\mathsf{P}_1, \mathsf{P}'_1, \mathsf{P}_2, \mathsf{P}'_2$ be probabilistic event structures. If $\mathsf{P}_1 \sqsubseteq \mathsf{P}'_1$ and $\mathsf{P}_2 \sqsubseteq \mathsf{P}'_2$ then $\mathsf{P}_1 \;+_p\; \mathsf{P}_2 \sqsubseteq \mathsf{P}'_1 \;+_p\; \mathsf{P}'_2$.

We now extend the concept of removing an initial event to the probabilistic setting. To do so, we need to ensure that an initial event with probability 0 cannot be removed. This restriction is necessary because removing an event with probability zero would lead to a division by zero. This is why we have $p \in]0, 1[$ in the probabilistic operator added to the core language.

Definition 5.2.16 (Remove initial event)**.** Let $\mathsf{P} = (E, \leq, \#, v)$ be a probabilistic event structure and $a \in \mathcal{I}(\mathsf{P})$, s.t. $v(\{a\}) \neq 0$. Define $\mathsf{P}\backslash a = (E', \leq', \#', v')$ as

$$E' = \{e \in E \mid \neg(e \# a),\, e \neq a\}$$

$$\leq' = \{e \leq e' \mid e, e' \in E'\}$$
$$\#' = \{e \# e' \mid e, e' \in E'\}$$
$$\forall x \in \mathcal{C}(\mathsf{P}\backslash a)\,.\, v'(x) = \frac{v(x \cup \{a\})}{v(\{a\})}$$

Lemma 5.2.17. Let P be a probabilistic event structure. $\mathsf{P}\backslash a$ is a probabilistic event structure.

We now show how probabilistic event structures composed of the sequential and concurrent operators interact with the removal of initial events.

Lemma 5.2.18. Let P_1 and P_2 be probabilistic event structures. Consider $\mathsf{P}_1 \,;\, \mathsf{P}_2$ such that $l \in \mathcal{I}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$. Then $(\mathsf{P}_1 \,;\, \mathsf{P}_2)\backslash l \equiv (\mathsf{P}_1\backslash l) \,;\, \mathsf{P}_2$.

Lemma 5.2.19. Let P_1 and P_2 be probabilistic event structures. Consider $\mathsf{P}_1 \,||\, \mathsf{P}_2$ such that $l \in \mathcal{I}(\mathsf{P}_1 \,||\, \mathsf{P}_2)$. Then

$$(\mathsf{P}_1 \,||\, \mathsf{P}_2)\backslash l \equiv \begin{cases} (\mathsf{P}_1\backslash l) \,||\, \mathsf{P}_2 & \text{if } l \in \mathcal{I}(\mathsf{P}_1) \\ \mathsf{P}_1 \,||\, (\mathsf{P}_2\backslash l) & \text{if } l \in \mathcal{I}(\mathsf{P}_2) \end{cases}$$

From Definition 4.2.18, we can infer that, in a probabilistic event structure interpreting a command, any configuration consisting solely of an initial event has valuation 1. The following lemma confirms this intuition.

Lemma 5.2.20. Let C be a command and $l \in \mathcal{I}([\![C]\!])$. Then $v(\{l\}) = 1$.

5.3 Introducing cyclic behavior

Similarly to Section 4.3, here we also opt to use recursion instead of while loops to introduce cyclic behavior. We also recall that we do not want to allow commands like: $\mu X.X \,;\, a$ and $\mu X.a \,;\, X \,;\, b$, as justified in Section 4.3.

We now extend the language's syntax in Section 5.1 to include the recursion command. Recall that $X \in Var$, with Var a set of variables. The syntax is now given by:

$$C ::= skip \mid a \in Act \mid C \,;\, C \mid C \,||\, C \mid C \,+_p\, C \mid \mu X.C \mid X$$

Similarly to Section 4.3, we only consider closed commands, *i.e.* commands in which every variable X is bound by μX, and in sequential composition we only allow recursion to occur at right.

Regarding the sets of free variables and bound variables in Section 4.3 (see Figure 37), non-deterministic choice is replaced by probabilistic choice and we are done. This is possible because those sets are defined

similarly for the non-deterministic and probabilistic commands. With the same rationale, the definition of substitution (Definition 4.3.1) is straightforwardly adopted to the probabilistic command.

We then add to Figure 43 the following rules for recursion:

$$\frac{C \longrightarrow 1 \cdot (l, C')}{\mu X.C \longrightarrow 1 \cdot (l, C'[X \leftarrow \mu X.C])} \qquad \frac{C \longrightarrow \sum_i p_i \cdot (\tau, C_i)}{\mu X.C \longrightarrow \sum_i p_i \cdot (\tau, C_i[X \leftarrow \mu X.C])}$$

Similarly to what was done in Figure 43, here we also distinguish if the action that triggered the transition was an atomic or invisible action. Consequently, we define a rule for each case. Both rules follow the same reasoning as the one explained in the non-deterministic case. However, in the case where the transition is triggered by τ, we need to substitute the occurrence of X by $\mu X.C$ in each C_i that is in $\sum_i p_i \cdot (\tau, C_i)$.

Now we develop an illustrative example of recursion in the probabilistic setting.

Example 5.3.1. Figure 55 illustrates a probabilistic coin toss scenario where each time we toss a coin, we execute with probability p the command $skip$ or continue tossing with probability $1 - p$. To understand this behavior, we focus on the initial command $\mu X.(skip +_p X)$. From there, we transit to a distribution formed by the commands $skip$ and $\mu X.(X +_p skip)$, where the latter is the same as the initial command. From this distribution we transit to $skip$ with probability p or to $\mu X.(X +_p skip)$ with probability $1 - p$, enabling us to repeat the process.

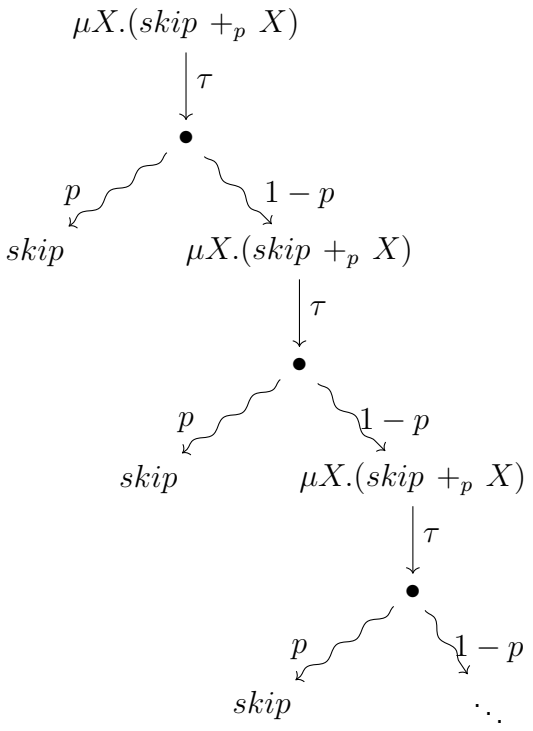


Figure 55: Fragment of the execution of $\mu X.(skip +_p X)$

Note that in Figure 55, in the branches where we have the $skip$ command, we omit the transition from $skip$ to ✓, in order to not overburden the figure with notation.

From Figure 55 we can deduce an interesting property regarding the probabilities of the words that we can form: if we have a word that ends with sk, its associated probability is $p \cdot (1-p)^{n-1}$, where n is the number of times the recursive command is unfolded. Note that by Lemma 5.4.5 we have the guarantee that at least one unfolding of the recursive command exists. Furthermore, if the word is only formed by τ's, its associated probability is $(1-p)^n$, reflecting the successive transitions triggered by τ with probability $1-p$ each time.

We now focus on probabilistic event structures, following the same approach as in Section 4.3. We begin by defining a partial order on probabilistic event structures. To achieve this, we extend the notion of sub-event structures (Definition 4.3.3) to the probabilistic setting by imposing the equality of the valuations. Formally:

Definition 5.3.2. Let $\mathsf{P}_1 = (E_1, \leq_1, \#_1, v_1)$ and $\mathsf{P}_2 = (E_2, \leq_2, \#_2, v_2)$ be probabilistic event structures. Say $\mathsf{P}_1 \trianglelefteq \mathsf{P}_2$ if:

$$
\begin{aligned}
&E_1 \subseteq E_2 \\
&\forall e, e' \,.\, e \leq_1 e' \Leftrightarrow e, e' \in E_1 \wedge e \leq_2 e' \\
&\forall e, e' \,.\, e \#_1 e' \Leftrightarrow e, e' \in E_1 \wedge e \#_2 e' \\
&\forall x \in \mathcal{C}(\mathsf{P}_1) \,.\, v_1(x) = v_2(x)
\end{aligned}
$$

Remark 3. Recall the probabilistic event structures in Example 5.2.11. There we had $\mathsf{P}_1 \sqsubseteq \mathsf{P}_2$ and $\mathsf{P}_1 \sqsubseteq \mathsf{P}_3$. However, according to Definition 5.3.2 we do not have $\mathsf{P}_1 \trianglelefteq \mathsf{P}_2$ and $\mathsf{P}_1 \trianglelefteq \mathsf{P}_3$. That is because P_2 and P_3 are not a "drawing" extension of P_1. Furthermore, in Example 5.2.11 we also showed that $\neg(\mathsf{P}_2 \sqsubseteq \mathsf{P}_3)$ and $\neg(\mathsf{P}_3 \sqsubseteq \mathsf{P}_2)$. This still holds according to Definition 5.3.2, since the valuation of the configuration $\{a, \tau, b\}$ in P_2 is not equal to the valuation in P_3.

The following lemmas confirm that Definition 5.3.2 is a partial order and that it possesses a least element.

Lemma 5.3.3. $\trianglelefteq$ is a partial order.

Lemma 5.3.4. Define $\bot = (\emptyset, \emptyset, \emptyset, v_\bot(\emptyset) = 1)$. $\bot$ is the least element of $\trianglelefteq$.

We extend Definition 4.3.6 to the probabilistic realm, *i.e.* we define what it means for a probabilistic event structure to be a least upper bound in a chain of probabilistic event structures.

Definition 5.3.5. Let $\mathsf{P}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{P}_n \trianglelefteq \ldots$ be an ω-chain. Let $\mathsf{P}^\omega = (E^\omega, \leq^\omega, \#^\omega, v^\omega)$ be its least upper bound where:

- $E^\omega = \cup_{n\in\omega} E_n$
- $\leq^\omega = \cup_{n\in\omega} \leq_n$
- $\#^\omega = \cup_{n\in\omega} \#_n$
- $\forall x \in \mathcal{C}(\mathsf{P}^\omega)\,,\ \exists n \in \omega\,.\, x \in \mathcal{C}(\mathsf{P}_n)\,.\, v^\omega(x) = v_n(x)$

We then show that the structure in Definition 5.3.5 is in fact a probabilistic event structure and a least upper bound in a chain of probabilistic event structures.

Lemma 5.3.6. P^ω is a probabilistic event structure.

Lemma 5.3.7. Let $\mathsf{P}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{P}_n \trianglelefteq \ldots$ be a ω-chain. Then P^ω is its least upper bound.

We have that sequential and concurrent composition, as well as probabilistic choice are monotone and continuous with respect to Definition 5.3.2. Recall that monotonicity and continuity definitions are given by Definition 4.3.9.

Lemma 5.3.8. Let $\mathsf{P}, \mathsf{P}_1, \mathsf{P}_2$ be probabilistic event structures. If $\mathsf{P}_1 \trianglelefteq \mathsf{P}_2$ then $\mathsf{P}\,;\,\mathsf{P}_1 \trianglelefteq \mathsf{P}\,;\,\mathsf{P}_2$.

Lemma 5.3.9. Let $\mathsf{P}_1, \mathsf{P}'_1, \mathsf{P}_2, \mathsf{P}'_2$ be probabilistic event structures. If $\mathsf{P}_1 \trianglelefteq \mathsf{P}'_1$ and $\mathsf{P}_2 \trianglelefteq \mathsf{P}'_2$ then $\mathsf{P}_1 \,||\, \mathsf{P}_2 \trianglelefteq \mathsf{P}'_1 \,||\, \mathsf{P}'_2$.

Lemma 5.3.10. Let $\mathsf{P}_1, \mathsf{P}'_1, \mathsf{P}_2, \mathsf{P}'_2$ be probabilistic event structures. If $\mathsf{P}_1 \trianglelefteq \mathsf{P}'_1$ and $\mathsf{P}_2 \trianglelefteq \mathsf{P}'_2$ then $\mathsf{P}_1 \,+_p\, \mathsf{P}_2 \trianglelefteq \mathsf{P}'_1 \,+_p\, \mathsf{P}'_2$.

Lemma 5.3.11. $\bigsqcup_m (\mathsf{P}\,;\,\mathsf{P}_m) = \mathsf{P}\,;\,\bigsqcup_m \mathsf{P}_m$.

Lemma 5.3.12. $\bigsqcup_{n,m} (\mathsf{P}_n \,||\, \mathsf{P}_m) = \bigsqcup_n \mathsf{P}_n \,||\, \bigsqcup_m \mathsf{P}_m$.

Lemma 5.3.13. $\bigsqcup_{n,m} (\mathsf{P}_n \,+_p\, \mathsf{P}_m) = \bigsqcup_n \mathsf{P}_n \,+_p\, \bigsqcup_m \mathsf{P}_m$.

When adapting Lemma 4.3.13 (helps to show that operators defined in event structures are continuous) and Lemma 4.3.17 (a version of the Kleene fixed-point theorem for the case of event structures) to the probabilistic realm, we notice that the proofs we need to do are analogous to the ones done in the non-deterministic case. Hence, we omit their formulation in this section and their proofs in Appendix A.2.

We now extend Definition 5.2.8 with the recursion command and let $\mathbb{P}$ denote the class of probabilistic event structures.

Definition 5.3.14. Define an environment to be a function $\gamma : Var \to \mathbb{P}$ from variables to probabilistic event structures. For a command C and an environment γ define $[\![C]\!]_\gamma$ as follows:

$$
\begin{aligned}
&[\![skip]\!]_\gamma = (\{sk\}, \{sk \leq sk\}, \emptyset, v(\{sk\} = 1)) \\
&[\![a]\!]_\gamma = (\{a\}, \{a \leq a\}, \emptyset, v(\{a\} = 1)) \\
&[\![C_1 \,;\, C_2]\!]_\gamma = [\![C_1]\!]_\gamma \,;\, [\![C_2]\!]_\gamma \\
&[\![C_1 \;+_p\; C_2]\!]_\gamma = [\![C_1]\!]_\gamma \;+_p\; [\![C_2]\!]_\gamma \\
&[\![C_1 \,||\, C_2]\!]_\gamma = [\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma \\
&[\![X]\!]_\gamma = \gamma(X) \\
&[\![\mu X.C]\!]_\gamma = fix(\Gamma^{C,\gamma})
\end{aligned}
$$

where $\Gamma^{C,\gamma} : \mathbb{P} \to \mathbb{P}$ is given by $\Gamma^{C,\gamma}(\mathsf{P}) = [\![C]\!]_{\gamma(X \leftarrow \mathsf{P})}$.

Since $\Gamma^{C,\gamma}$ is a new operator we need to show that it is continuous.

Lemma 5.3.15. $\Gamma^{C,\gamma}$ is continuous.

To prove a lemma that is helpful for showing soundness (Lemma 5.4.6), we need the probabilistic versions of the substitution lemma (Lemma 4.3.20) and Lemma 4.3.21. Fortunately, the formulation of these probabilistic versions is the same as the original ones. However, the proof of the substitution lemma needs to be adjusted to accommodate the probabilistic command. Hence, we show here its formulation (Lemma 5.3.16).

Lemma 5.3.16. $[\![C'[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma = [\![C']\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}$

We conclude this section with an example of a fragment of the probabilistic event structure that arises from command $\mu X.(X \;+_p\; skip)$.

Example 5.3.17. Let us reuse the command in Example 5.3.1. A fragment of the probabilistic event structure generated by interpreting $[\![\mu X.(X \;+_p\; skip)]\!]_\gamma$ is given by Figure 56:

Note that the valuation of configurations composed only of τ's will be $(1-p)^n$, where n is the number of times the recursive command is unrolled. On the other hand, the valuation of a configuration containing sk_n is $p \cdot (1-p)^{n-1}$. Furthermore, these configurations are maximal. This detail will be important when formulating soundness (Theorem 5.4.7) and adequacy (Theorem 5.4.9).

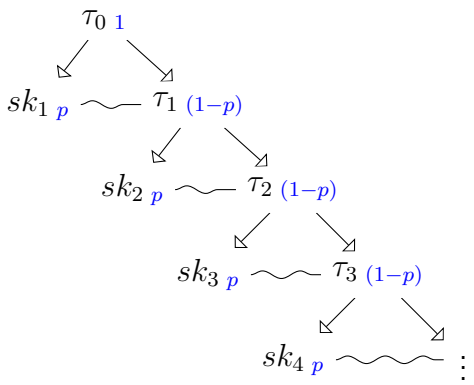


Figure 56: Unrolling the probabilistic event structure $[\![\mu X.(X +_p skip)]\!]$

5.4 Soundness and Adequacy

This section presents the equivalence between the operational and the denotational semantics. These results are supported by previous ones obtained in Section 5.2 and Section 5.3. The proofs can be consulted in Appendix A.2.

We start by presenting two useful lemmas to prove small-soundness (Lemma 5.4.6). The first lemma states that sequentially composing a sum of probabilistic event structures with another probabilistic event structure is equal to first sequentially composing the probabilistic event structures and then using the probabilistic operator.

Lemma 5.4.1. Let $\mathsf{P}, \sum_i p_i \cdot \mathsf{P}_i$ be probabilistic event structures. Then $(\sum_i p_i \cdot \mathsf{P}_i)\ ;\ \mathsf{P} = \sum_i p_i \cdot (\mathsf{P}_i\ ;\ \mathsf{P})$

The second lemma expresses a certain "lax" linearity condition of concurrent composition.

Lemma 5.4.2. Let $\mathsf{P}, \sum_i p_i \cdot \mathsf{P}_i$ be probabilistic event structures.

1. $(\sum_i p_i \cdot \mathsf{P}_i)\ ||\ \mathsf{P} \sqsubseteq \sum_i p_i \cdot (\mathsf{P}_i\ ||\ \mathsf{P})$
2. $x \in \mathcal{C}_{\mathsf{max}}((\sum_i p_i \cdot \mathsf{P}_i)\ ||\ \mathsf{P})$ iff $x \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot (\mathsf{P}_i\ ||\ \mathsf{P}))$

The following lemma helps proving soundness (Theorem 5.4.7) and adequacy (Theorem 5.4.9). Intuitively it states that if the invisible action τ triggers a transition from C, which is sequentially or concurrently composed, to $\sum_i p_i \cdot C_i$, then any maximal configuration of $[\![C]\!]_\gamma$ is a maximal configuration in $\sum_i p_i \cdot [\![C_i]\!]_\gamma$. Furthermore, there exists a command $[\![C_i]\!]_\gamma$ in $\sum_i p_i \cdot [\![C_i]\!]_\gamma$ where the valuation of the maximal configuration matches the valuation of the maximal configuration in $[\![C]\!]_\gamma$.

Lemma 5.4.3. Let $C = C_1 \,;\, C_2$ or $C = C_1 \,||\, C_2$. If $C \longrightarrow \sum_i p_i \cdot (\tau, C_i)$ then $x \in \mathcal{C}_{\max}(\llbracket C \rrbracket_\gamma)$ and $x \in \mathcal{C}_{\max}(\sum_i p_i \cdot \llbracket C_i \rrbracket_\gamma)$ such that $\exists \llbracket C_i \rrbracket_\gamma \,.\, v(x) = v_i(x)$.

To prove soundness (Theorem 5.4.7), we found it helpful to have the following lemma for the recursion command, whose reasoning is similar to that of Lemma 5.4.3.

Lemma 5.4.4. If $\mu X.C \longrightarrow \sum_i p_i \cdot (\tau, C_i[X \leftarrow \mu X.C])$ then $x \in \mathcal{C}_{\max}(\llbracket \mu X.C \rrbracket_\gamma)$ and $x \in \mathcal{C}_{\max}(\sum_i p_i \cdot \llbracket C_i[X \leftarrow \mu X.C] \rrbracket_\gamma)$ such that $\exists \llbracket C_i \rrbracket_\gamma \,.\, v(x) = v_i(x)$.

The following lemma tells us that the big-step semantics always progresses, *i.e.* the system always has at least one valid transition to make.

Lemma 5.4.5. For any C, exists $\sum_i p_i(\omega_i, C_i)$ such that $C \twoheadrightarrow \sum_i p_i(\omega_i, C_i)$.

We can now establish a relation between the small-step and denotational semantics. Recall that in Figure 43, transitions are labeled either by an atomic or invisible action. We can use that to establish a relation between the small-step and denotational semantics. Intuitively, if a transition from C is triggered by an atomic action to C', then we should observe the same behavior as in Lemma 4.4.1. Otherwise, if the invisible action triggers the transition, we transition from C to $\sum_i p_i \cdot C_i$. Having in mind Lemma 5.4.2, we expect $\llbracket C \rrbracket_\gamma$ to be a sub-similar probabilistic event structure of $\sum_i p_i \cdot \llbracket C_i \rrbracket_\gamma$.

Lemma 5.4.6 (Small-soundness)**.**

- If $C \longrightarrow 1 \cdot (l, C')$ then $\forall \gamma, \llbracket C' \rrbracket_\gamma \equiv \llbracket C \rrbracket_\gamma \backslash l$
- If $C \longrightarrow \sum_i p_i \cdot (\tau, C_i)$ then $\forall \gamma, \llbracket C \rrbracket_\gamma \sqsubseteq \sum_i p_i \cdot \llbracket C_i \rrbracket_\gamma$

At this point, we can prove soundness, establishing a relation between the big-step and the denotational semantics. Recall that the big-step semantics of a command C returns a probability distribution of possible outcomes. Each outcome is a pair, formed by a word ω (describing the sequence of events that occurred), and either a command C', with instructions yet to execute, or the special symbol ✓, to indicate termination, *i.e.*

$$C \twoheadrightarrow \sum_i p_i \cdot (\omega_i, \checkmark) + \sum_j p_j \cdot (\omega_j, C_j)$$

where, unlike Figure 44, we separate computations that terminate from those that do not. Having in mind our experience with soundness in the non-deterministic case (Theorem 4.4.2), if a word leads to a terminal computation then there exists a maximal configuration that matches the word. Consequently, the probability associated with that terminal command equals the valuation of the maximal configuration.

Theorem 5.4.7 (Soundness II)**.** If $C \twoheadrightarrow p_0 \cdot (\omega_0, \checkmark) + \sum_k p_k \cdot (\omega_k, C_k)$ then exists $x_0 \in \mathcal{C}_{\max}(\llbracket C \rrbracket_\gamma)$ such that $\emptyset \xrightarrow{\omega_0} \subset x_0$ and $p_0 = v(x_0)$.

We now center our attention to show adequacy. Similar to what was done previously, we check if a relation exists between the denotational and small-step semantics. To do this, we will take advantage of the initial events. If an initial event represents an atomic action, we should be in a similar situation as in the small-adequacy lemma in the non-deterministic case (Lemma 4.4.3). Otherwise, if the initial event represents an invisible action, we deal with a transition triggered by a probabilistic command. In this case, we need to adapt the previous situation to the probabilistic case, considering Lemma 5.4.2 and the similar case in small-soundness (Lemma 5.4.6).

Lemma 5.4.8 (Small-adequacy)**.** Let $l' \in \mathcal{I}(\llbracket C \rrbracket_\gamma)$.

1. If $l' \neq \tau$ then $\exists C' \in (Com \cup \{\checkmark\})\,.\, C \longrightarrow 1 \cdot (l', C')$ and $\llbracket C \rrbracket_\gamma \backslash l' \equiv \llbracket C' \rrbracket_\gamma$.
2. If $l' = \tau$ then $\exists C', C''\,.\, \exists e' \in \mathcal{I}(\llbracket C' \rrbracket_\gamma)\,.\, C \longrightarrow p \cdot (\tau, C') + (1-p) \cdot (\tau, C'')$ and $\llbracket C \rrbracket_\gamma \sqsubseteq p \cdot \llbracket C' \rrbracket_\gamma + (1-p) \cdot \llbracket C'' \rrbracket_\gamma$, with $p = v(\{\tau, e'\})$.

Note that in Lemma 5.4.8, for the case where $l' = \tau$, we only consider the existence of an initial event of a command C'. This is because to know the probability of C'' to occur, we only need to know the probability p and subtract it from 1. Furthermore, the result states $\llbracket C \rrbracket_\gamma \sqsubseteq p \cdot \llbracket C' \rrbracket_\gamma + (1-p) \cdot \llbracket C'' \rrbracket_\gamma$ instead of $\llbracket C \rrbracket_\gamma \equiv p \cdot \llbracket C' \rrbracket_\gamma + (1-p) \cdot \llbracket C'' \rrbracket_\gamma$. The use of sub-similarity instead of similarity arises with a similar argument as the one given by changing from "equivalence" to "implication" in the definition of sub-similar probabilistic event structures (Definition 5.2.10). For instance, consider the following small-step transition $(a +_p b) \,||\, c \longrightarrow p \cdot (\tau, a \,||\, c) + (1-p) \cdot (\tau, b \,||\, c)$. Here, the action c is duplicated on both branches of the probabilistic choice. Now recall Figure 50, corresponding to $\llbracket (a +_p b) \,||\, c \rrbracket$, and Figure 51, corresponding to $p \cdot \llbracket a \,||\, c \rrbracket_\gamma + (1-p) \cdot \llbracket b \,||\, c \rrbracket_\gamma$. In the former figure, c is concurrent with every event, while in the latter figure, the copies of c, *i.e.* c' and c'', causally depend on τ: c' is in conflict with b, and c'' is in conflict with a. These new relations are not present in $\llbracket (a +_p b) \,||\, c \rrbracket$. Because the duplication of events in the probabilistic branches introduces new causal and conflict relations, the probabilistic event structure obtained after the transition cannot be similar to the original one, it can only be sub-similar. Thus, the usage of $\sqsubseteq$ in the second case of Lemma 5.4.8.

We are ready to demonstrate the equivalence between the denotational and big-step semantics. The reasoning is similar to the one presented in Theorem 5.4.7.

Theorem 5.4.9 (Adequacy)**.** For all $x_0 \in \mathcal{C}_{\mathsf{max}}(\llbracket C \rrbracket_\gamma)$, if $\emptyset \xrightarrow{\omega_{x_0}} x_0$ then we have $C \twoheadrightarrow v(x_0) \cdot (\omega_0, \checkmark) + \sum_k p_k \cdot (\omega_k, C_k)$, for some ω_k, p_k, C_k.

In small-soundness (Lemma 5.4.6) and small-adequacy (Lemma 5.4.8), we see the usefulness of

introducing the label τ. It helps us identify the situations where a transition occurred due to the probabilistic command and where it did not.

Summing up both theorems: soundness (Theorem 5.4.7) assures that whenever any program execution leads to a terminal command, we have a maximal configuration that matches the word and the respective probability. Adequacy (Theorem 5.4.9) tells us that for every maximal configuration of a command C and for every covering chain of that configuration, there is an execution of the program leading to a terminal command that matches the covering chain and its respective probability.

The following example shows soundness (Theorem 5.4.7) and adequacy (Theorem 5.4.9) in practice, for commands without any recursive behavior.

Example 5.4.10. The probabilistic event structure in Example 3.3.2 corresponds to the command in Example 5.1.1, $(a\,;\,b)\,+_p\,(c\,||\,d)$. To see how both semantics relate, recall the maximal configurations in Example 3.3.2 ($\{\tau, a, b\}$ and $\{\tau, c, d\}$) and the words that lead to the end of a computation in Example 5.1.1 (τab, τcd, and τdc). Similarly to what was shown in Example 4.4.5, it is straightforward to see that each word corresponds to a covering chain and vice-versa. What is left to verify is the probability. From Example 5.1.1, we know that the word τab has probability p, which is the same probability as that of the corresponding covering chain. Similarly, the words τcd and τdc have probability $1 - p$, which equals the probability of the respective covering chains. Conversely, if we pick a covering chain of a maximal configuration, we quickly notice that its probability and the probability of the respective word are the same.

We now develop an example that shows Theorem 5.4.7 and Theorem 5.4.9 in practice for a recursive command.

Example 5.4.11. Recall the command used in Example 5.3.1 and Example 5.3.17, $\mu X.skip\,+_p\,X$. The former example showed a fragment of its execution tree, while the latter showed the resultant probabilistic event structure after unrolling the recursive command a few times. Let us focus on Theorem 5.4.7 first. According to Example 5.3.1, a word that reaches ✓ must end with the sk label. This indicates that the probability of the word is $p \cdot (1-p)^{n-1}$, where n represents the number of times the recursive command was unfolded. In Example 5.3.17, we said that configurations containing an event sk_n are maximal and that its respective valuation is given by $p \cdot (1-p)^{n-1}$. This confirms what is formulated in Theorem 5.4.7. To verify Theorem 5.4.9, we reverse the reasoning: we start with a maximal configuration containing an event sk_n and a covering chain whose last event to be added is sk_n. Then we refer to Example 5.3.1 to check that the covering chain corresponds to a word that leads to ✓ and, consequently, that the word has probability $p \cdot (1-p)^{n-1}$.

Chapter 6

Adding quantum commands

6.1 Language

We now adapt the core language in Section 3.5 to the quantum realm. The set of atomic actions consists of unitary gates, and the measurement command is added to the core language. Furthermore, we also follow the same approach as in Section 4.1 and Section 5.1, where we postpone the introduction of cyclic behavior to Section 6.3.

We consider at our disposal a finite number of qubits N, whose associated space is $\mathbb{C}^{2^N}$. Each qubit is denoted by a natural number n and we let $\tilde{n}$ be a sequence of numbers, denoting multiple qubits. As stated, the set of atomic actions is composed of unitary gates. To identify the action that triggered a transition, in the operational semantics, we consider the set of labels to be $L = Act \cup \{sk\} \cup \{P_0(n), P_1(n)\}$, where we use l to represent elements of L. Note the presence of $P_0(n)$ and $P_1(n)$, which represent the projection of qubit n onto $|0\rangle$ and $|1\rangle$, respectively. These projections come from the measurement command, which projects the qubit n onto $|0\rangle$ and $|1\rangle$, as will be seen later.

For the quantum setting, the BNF grammar of the core language is altered by considering the application of unitary gates as atomic actions, and by adding the measurement command, which is inspired by [120, Definition 3.1.1]:

$$C ::= skip \mid U(\tilde{n}) \mid C\,;\, C \mid C \,||\, C \mid \mathtt{M}(n, C_1, C_2)$$

where $U(\tilde{n})$ applies the unitary gate U to the qubits presented in $\tilde{n}$ and $\mathtt{M}(n, C_1, C_2)$ represents the measurement of a qubit n such that when applying P_0 we execute C_1, and when applying P_1 we execute C_2. This behavior of $\mathtt{M}(n, C_1, C_2)$ is analogous to a classical "if" command, where a measurement yielding P_0 can be seen as a Boolean condition not holding, and a measurement yielding P_1 corresponds to a Boolean condition holding.

The definition of a unitary event structure (see Definition 3.4.1) entails that quantum operators of con-

current events must commute. This restriction demands some caution regarding the parallel composition we consider in our language. If the parallel composition is unrestricted, it is possible to use the same qubit in two different commands that are composed in parallel, *e.g.* $X(1) \,||\, Z(1)$. Suppose we were to interpret this command in terms of event structures. In that case, the events associated with $X(1)$ and $Z(1)$ are concurrent and, according to Definition 3.4.1, their associated quantum operator should commute. However, $[X, Z] = XZ - ZX \neq 0$, *i.e.* the quantum operators do not commute. Consequently, the interpretation of $X(1) \,||\, Z(1)$ is not a unitary event structure. To ensure that all commands in our language can be interpreted using unitary event structures, we limit the parallel composition of commands to those that are disjoint, *i.e.* do not share any qubit. The trade-off of considering a disjoint parallel operator and ensuring that every command can be interpreted as a unitary event structure is that programs like $X(1) \,||\, Z(1)$ are not valid in our language.

Let us make formal what it means for two commands to be disjoint. We start by defining the set of qubits being used in a command C.

$$
\begin{aligned}
&\mathsf{qVar}(skip) = \emptyset \\
&\mathsf{qVar}(U(\tilde{n})) = \tilde{n} \\
&\mathsf{qVar}(C_1 \,;\, C_2) = \mathsf{qVar}(C_1) \cup \mathsf{qVar}(C_2) \\
&\mathsf{qVar}(C_1 \,||\, C_2) = \mathsf{qVar}(C_1) \cup \mathsf{qVar}(C_2) \\
&\mathsf{qVar}(\mathtt{M}(n, C_1, C_2)) = \{n\} \cup \mathsf{qVar}(C_1) \cup \mathsf{qVar}(C_2)
\end{aligned}
$$

We then say that two programs are disjoint whenever $\mathsf{qVar}(C_1) \cap \mathsf{qVar}(C_2) = \emptyset$. Since we want to restrict the parallel composition to programs that are disjoint we have:

$$C_1 \,||\, C_2 \text{ if } \mathsf{qVar}(C_1) \cap \mathsf{qVar}(C_2) = \emptyset$$

Now, let us focus on the operational semantics. Comparing to the rules in Figure 16, we instantiate the atomic actions to unitary applications and add the corresponding rules for the measurement command. Thus the small-step transition is defined as the smallest relation $\rightarrow \subseteq Com \times L \times (Com \cup \{\checkmark\})$ obeying the rules in Figure 57.

Most rules were already discussed in Section 3.5. Hence, we skip their explanation and focus solely on the rules related to the measurement command. Previously, we said the measurement command could be seen as a classical if statement. Another way of seeing it is by using the non-deterministic operator from Section 4.1, together with the projections P_0 and P_1. Concretely we can view $\mathtt{M}(n, C_1, C_2)$ as $P_0(n) \,;\, C_1$ or $P_1(n) \,;\, C_2$. With this perspective and the knowledge obtained in Section 4.1 regarding

$$skip \xrightarrow{sk} \checkmark \qquad U(\tilde{n}) \xrightarrow{U(\tilde{n})} \checkmark \qquad \mathtt{M}(n, C_1, C_2) \xrightarrow{P_0(n)} C_1 \qquad \mathtt{M}(n, C_1, C_2) \xrightarrow{P_1(n)} C_2$$

$$\frac{C_1 \xrightarrow{l} \checkmark}{C_1 \,;\, C_2 \xrightarrow{l} C_2} \qquad \frac{C_1 \xrightarrow{l} C_1'}{C_1 \,;\, C_2 \xrightarrow{l} C_1' \,;\, C_2}$$

$$\frac{C_1 \xrightarrow{l} \checkmark}{C_1 \,||\, C_2 \xrightarrow{l} C_2} \qquad \frac{C_1 \xrightarrow{l} C_1'}{C_1 \,||\, C_2 \xrightarrow{l} C_1' \,||\, C_2} \qquad \frac{C_2 \xrightarrow{l} \checkmark}{C_1 \,||\, C_2 \xrightarrow{l} C_1} \qquad \frac{C_2 \xrightarrow{l} C_2'}{C_1 \,||\, C_2 \xrightarrow{l} C_1 \,||\, C_2'}$$

Figure 57: Rules of the small-step operational semantics

sequential and non-deterministic operators, we can say that from $\mathtt{M}(n, C_1, C_2)$ we transit to C_1 if the transition was triggered by $P_0(n)$. Conversely, if the transition is triggered by $P_1(n)$, we transit to C_2.

In Section 5.1, we mentioned that the measurement command influenced the semantics of the probabilistic command. This is because, in the quantum setting, probabilities arise from measuring states (recall that our semantics does not contemplate states; consequently, we do not need to deal with probabilities in the semantics). Furthermore, the measurement rules are axioms, in which the transitions are triggered by the projections P_0 or P_1, leading directly to C_1 or C_2, respectively. The similarity between the measurement command and the probabilistic command can also be found in their respective event structure definitions.

The language shown in this section is equipped with a big-step semantics, which is the one defined in Section 3.5, *i.e.* it obeys the rules in Figure 17.

We now develop an example that illustrates the behavior of a measurement command.

Example 6.1.1. In Figure 58, we have the execution tree of $H(1)\,;\,\mathtt{M}(1, Z(1), Id(2)\,||\,X(1))$. This program applies first the Hadamard gate to qubit 1 and then measures it. If the measurement was made by $P_0(1)$, then we apply the Z gate to qubit 1 and we are done. Otherwise we have two possible ways of reaching the terminal command: either we apply first the identity to qubit 2 and then we apply X to qubit 1 or we first apply X to qubit 1 and then we apply the identity to qubit 2. With the help of Figure 58, it is straightforward to see that the words that lead to a terminal command are: $H(1)P_0(1)Z(1)$, $H(1)P_1(1)I(2)X(1)$, and $H(1)P_1(1)X(1)I(2)$.

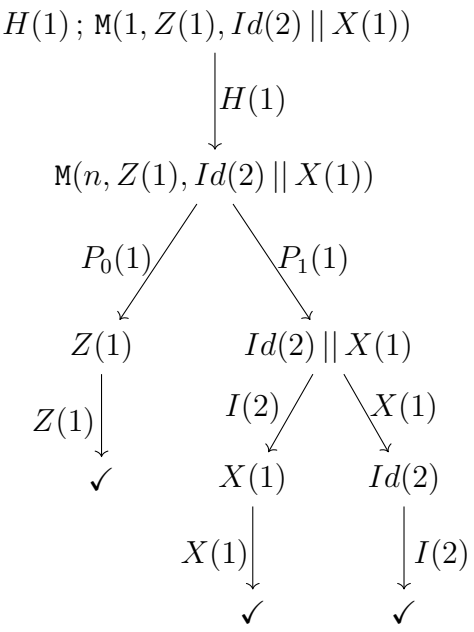


Figure 58: Labeled transition system of $H(1)\,;\,\mathtt{M}(1, Z(1), Id(2) \,||\, X(1))$

6.2 Constructions on Unitary Event Structures

Similarly to what was done in Section 5.2, where we extended the sequential and parallel definitions of event structures in Section 4.2 to the probabilistic case, here we extend these definitions to the quantum case. In sum, we need to include the corresponding mapping of events to unitary or projection operators. Additionally, we define the measurement operation by slightly adjusting the definition of non-deterministic choice provided in Section 4.2, inspired by the definition of probabilistic choice in Section 5.2.

The sequential composition of two unitary event structures is defined as follows:

Definition 6.2.1 (Sequential composition)**.** Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures. Define $\mathsf{U}_1\,;\,\mathsf{U}_2 = (E, \leq, \#, Q)$ as:

$$
\begin{aligned}
E &= E_1 \uplus (E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{U}_1)) \\
\leq &= \{e_1 \leq e_1' \mid e \leq_1 e'\} \cup \{(e_2, x) \leq (e_2', x) \mid \ e_2 \leq_2 e_2'\} \cup \{e_1 \leq (e_2, x) \mid e_1 \in x\} \\
\# \ &= \{e \# e' \mid \exists (e_1 \leq e, e_1' \leq e') \,.\, e_1 \#_1 e_1'\} \cup \{(e_2, x) \#(e_2', x) \mid e_2 \#_2 e_2'\} \\
Q(e) &= \begin{cases} Q_1(e) & \text{if } e \in E_1 \\ Q_2(e_2) & \text{if } e = (e_2, x) \in E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{U}_1) \end{cases}
\end{aligned}
$$

where $E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{U}_1) = \{(e, x) \mid e \in E_2,\ x \in \mathcal{C}_{\mathsf{max}}(\mathsf{U}_1)\}$.

Lemma 6.2.2. Let U_1 and U_2 be unitary event structures. $\mathsf{U}_1\,;\,\mathsf{U}_2$ is a unitary event structure.

When defining the parallel composition, we must consider the restriction in Definition 3.4.1 requiring the operators associated with concurrent events to commute. Since in Definition 4.2.3 every event in E_1 is concurrent with every event of E_2, then when extending this definition to the quantum case, the operators associated to $\mathsf{U}_1 = (\mathsf{E}_1, Q_1)$ and $\mathsf{U}_2 = (\mathsf{E}_2, Q_2)$ must commute for every event, *i.e.* $\forall e_1 \in E_1, e_2 \in E_2.\ [Q_1(e_1), Q_2(e_2)] = 0$.

Definition 6.2.3 (Parallel composition)**.** Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures, such that $\forall e_1 \in E_1, e_2 \in E_2.\ [Q_1(e_1), Q_2(e_2)] = 0$. Define $\mathsf{U}_1 \,||\, \mathsf{U}_2 = (E, \leq, \#, Q)$ as:

$$
\begin{aligned}
E &= E_1 \uplus E_2 \\
\leq &= \leq_1 \uplus \leq_2 \\
\# &= \#_1 \uplus \#_2 \\
Q(e) &= \begin{cases} Q_1(e) & \text{if } e \in E_1 \\ Q_2(e) & \text{if } e \in E_2 \end{cases}
\end{aligned}
$$

Lemma 6.2.4. Let U_1 and U_2 be unitary event structures. $\mathsf{U}_1 \,||\, \mathsf{U}_2$ is a unitary event structure.

When defining the measurement command in terms of unitary event structures, we must consider the restriction in Definition 3.4.1 that requires the sum of operators associated with events in minimal conflict to be the identity.

Definition 6.2.5 (Measurement composition)**.** Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures. Define $\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2) = (E, \leq, \#, Q)$ as:

$$
\begin{aligned}
E &= \{\tau_0^n, \tau_1^n\} \uplus E_1 \uplus E_2 \\
\leq &= \{\tau_0^n \leq e \mid (e = \tau_0^n \vee e \in E_1)\} \cup \{\tau_1^n \leq e \mid (e = \tau_1^n \vee e \in E_2)\} \cup \leq_1 \uplus \leq_2 \\
\# &= \{e \# e' \mid (e = \tau_0^n \vee e \in E_1), (e' = \tau_1^n \vee e' \in E_2)\} \\
&\quad \cup \{e' \# e \mid (e = \tau_0^n \vee e \in E_1), (e' = \tau_1^n \vee e' \in E_2)\} \cup \#_1 \uplus \#_2 \\
Q(e) &= \begin{cases} P_0(n) & \text{if } e = \tau_0^n \\ P_1(n) & \text{if } e = \tau_1^n \\ Q_1(e) & \text{if } e \in E_1 \\ Q_2(e) & \text{if } e \in E_2 \end{cases}
\end{aligned}
$$

Lemma 6.2.6. Let U_1 and U_2 be unitary event structures. $\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2)$ is a unitary event structure.

Note that, in Definition 6.2.5, the fact that $Q(\tau_0^n) = P_0(n)$ and $Q(\tau_1^n) = P_1(n)$, ensures us that $Q(\tau_0^n) + Q(\tau_1^n) = P_0(n) + P_1(n) = Id$.

Remark 4. As previously stated in the operational semantics, the measurement command can be seen as $P_0(n)\,;\, C_1$ or $P_1(n)\,;\, C_2$. To transport such relation to the setting of unitary event structures, we find helpful to write $\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2)$ as $\mathsf{P}_0^n\,;\,\mathsf{U}_1$ or $\mathsf{P}_1^n\,;\,\mathsf{U}_2$, where

$$\mathsf{P}_0^n = (\{\tau_0^n\},\, \{\tau_0^n \leq \tau_0^n\},\, \emptyset,\, Q(\tau_0^n) = P_0(n))$$
$$\mathsf{P}_1^n = (\{\tau_1^n\},\, \{\tau_1^n \leq \tau_1^n\},\, \emptyset,\, Q(\tau_1^n) = P_1(n))$$

Note that by doing this, we are abusing notation, since P_0^n and P_1^n are quantum event structures and not unitary event structures, because the quantum operators associated to the events τ_0^n and τ_1^n are not unitary. However, this notational abuse allows the measurement command to connect to the operational counterpart directly.

In Definition 6.2.5, the events associated with projections include a superscript indicating the qubit on which the projection is performed. Whenever it is clear from the context, we will drop the superscript.

Let us develop an example that shows how a unitary event structure composed through a measurement looks like.

Example 6.2.7. Consider Figure 59 and Figure 60, which are an adaptation to the quantum setting of the event structures E_1 (Figure 25) and E_2 (Figure 26), respectively, used in Example 4.2.8.

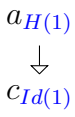


Figure 59: Unitary event structure U_1

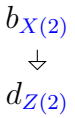


Figure 60: Unitary event structure U_2

Note that we use the same notation as in Example 3.4.4, *i.e.* we subscript to each event the respective quantum operator. Now let us compose both unitary event structures through the measurement operator (Definition 6.2.5).

Figure 61 shows that τ_0 and τ_1 are initial events in minimal conflict. Consequently, the sum of their associated quantum operators is a unitary ($P_0 + P_1 = Id$). Similar to the non-deterministic composition of event structures and the probabilistic composition of probabilistic event structures, the events of U_1 are all in conflict with the events of U_2. This means that if an event from U_1 is performed on a computation, no event of U_2 can be executed, and vice-versa.

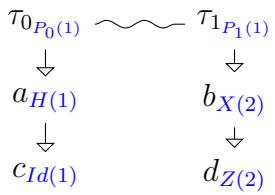


Figure 61: Unitary event structure $\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2)$

We are now ready to interpret commands of the language in Section 6.1 as unitary event structures.

Definition 6.2.8. We interpret commands as follows:

$$[\![skip]\!] = (\{sk\}, \{sk \leq sk\}, \emptyset, Q(sk) = Id)$$

$$[\![U(\tilde{n})]\!] = (\{U_{\tilde{n}}\}, \{U_{\tilde{n}} \leq U_{\tilde{n}}\}, \emptyset, Q(U_{\tilde{n}}) = U(\tilde{n}))$$

$$[\![C_1 \,;\, C_2]\!] = [\![C_1]\!] \,;\, [\![C_2]\!]$$

$$[\![C_1 \,||\, C_2]\!] = [\![C_1]\!] \,||\, [\![C_2]\!]$$

$$[\![\mathtt{M}(n, C_1, C_2)]\!] = \mathsf{P}_0^n \,;\, [\![C_1]\!] \text{ or } \mathsf{P}_1^n \,;\, [\![C_2]\!]$$

We now resort to Example 6.1.1 to see the denotational semantics at work.

Example 6.2.9. Recall the command used in Example 6.1.1, $H(1) \,;\, \mathtt{M}(1, Z(1), Id(2) \,||\, X(1))$. In Figure 62 we represent the unitary event structure associated to $[\![H(1) \,;\, \mathtt{M}(1, Z(1), Id(2) \,||\, X(1))]\!]$.

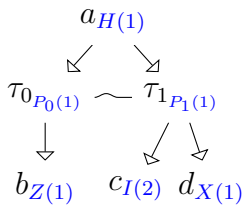


Figure 62: Unitary event structure of $[\![H(1) \,;\, \mathtt{M}(1, Z(1), Id(2) \,||\, X(1))]\!]$

Similarly to what was done in Example 3.4.4, we write the associated quantum operator of an event in subscript. From Figure 62, we can infer that a is an initial event, while τ_0 and τ_1 are the events associated with the projections of the measurement command. Consequently, they are in minimal conflict, and the sum of their associated quantum operators gives a unitary. The events c and d are concurrent, and their associated quantum operators commute. Also from Figure 62 we can deduce the maximal configurations: $\{a_{H(1)}, \tau_{0_{P_0(1)}}, b_{Z(1)}\}$ and $\{a_{H(1)}, \tau_{1_{P_1(1)}}, c_{I(2)}, d_{X(1)}\}$.

We have laid down the foundations of the operational and denotational semantics. Next we begin our journey of showing their equivalence. Since the operational semantics in Section 6.1 is the same as that in Section 4.1, we can reuse the intuition given in the non-deterministic case.

We extend the notion of sub-similar event structures (Definition 4.2.11) to the quantum setting. In this case, we need to ensure that the quantum operator is preserved across unitary event structures, *i.e.* if $\mathsf{U}_1 \sqsubseteq \mathsf{U}_2$ then the quantum operators of the events from U_2 that are also events in U_1 should be the same. Recall what was said regarding plain and composite events in Section 4.2 above Definition 4.2.11.

Definition 6.2.10. Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures. Say $\mathsf{U}_1 \sqsubseteq \mathsf{U}_2$ whenever exists an injective function $f : E_1 \to E_2$ such that $\forall e, e' \in E_1$:

$$\pi(f(e)) = \pi(e)$$
$$e \leq_1 e' \Leftrightarrow f(e) \leq_2 f(e')$$
$$e \#_1 e' \Leftrightarrow f(e) \#_2 f(e')$$
$$Q_1(e) = Q_2(f(e))$$

We say that two event structures $\mathsf{U}_1, \mathsf{U}_2$ are similar, denoted $\mathsf{U}_1 \equiv \mathsf{U}_2$, iff $\mathsf{U}_1 \sqsubseteq \mathsf{U}_2$ and $\mathsf{U}_2 \sqsubseteq \mathsf{U}_1$.

Now we study some properties of Definition 6.2.10 regarding the sequential, concurrent, and measurement commands (the proofs can be found in Appendix A.3). For this section, we interpret ✓ as the empty unitary event structure, *i.e.* $[\![\checkmark]\!] = (\emptyset, \emptyset, \emptyset, ! : \emptyset \to Op(\mathcal{H}))$. It is straightforward to see that for any unitary event structure U we have:

1. $\emptyset \,;\, \mathsf{U} \equiv \mathsf{U} \equiv \mathsf{U} \,;\, \emptyset$
2. $\emptyset \,||\, \mathsf{U} \equiv \mathsf{U} \equiv \mathsf{U} \,||\, \emptyset$

Furthermore it is also straightforward to see that:

Lemma 6.2.11. Let $\mathsf{U}_1, \mathsf{U}_2$ be unitary event structures. Then $\mathsf{U}_1 \,||\, \mathsf{U}_2 = \mathsf{U}_2 \,||\, \mathsf{U}_1$.

We start by showing that the sequential, parallel, and measurement operators are monotonic with respect to Definition 6.2.10.

Lemma 6.2.12. Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures. If $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$ and $\mathsf{U}_2 \sqsubseteq \mathsf{U}'_2$ then $\mathsf{U}_1 \,;\, \mathsf{U}_2 \sqsubseteq \mathsf{U}'_1 \,;\, \mathsf{U}'_2$.

Lemma 6.2.13. Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures. If $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$ and $\mathsf{U}_2 \sqsubseteq \mathsf{U}'_2$ then $\mathsf{U}_1 \,||\, \mathsf{U}_2 \sqsubseteq \mathsf{U}'_1 \,||\, \mathsf{U}'_2$.

Lemma 6.2.14. Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures. If $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$ and $\mathsf{U}_2 \sqsubseteq \mathsf{U}'_2$ then $\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2) \sqsubseteq \mathtt{M}(n, \mathsf{U}'_1, \mathsf{U}'_2)$.

We now extend the removal of an initial event to the quantum setting. Intuitively, we need to preserve the quantum operators of the events that were not removed.

Definition 6.2.15 (Remove initial event)**.** Let $\mathsf{U} = (E, \leq, \#, Q)$ be a unitary event structure and $a \in \mathcal{I}(\mathsf{U})$. Define $\mathsf{U}\backslash a = (E', \leq', \#', Q')$ as

$$
\begin{aligned}
E' &= \{e \in E \mid \neg(e \# a),\, e \neq a\} \\
\leq' &= \{e \leq e' \mid e, e' \in E'\} \\
\#' &= \{e \# e' \mid e, e' \in E'\} \\
Q' &= Q|_{E'}
\end{aligned}
$$

Lemma 6.2.16. Let U be a unitary event structure and $a \in \mathcal{I}(\mathsf{U})$. $\mathsf{U}\backslash a$ is a unitary event structure.

In Definition 6.2.15, we used $Q|_{E'}$. This denotes the restriction of the quantum operator $Q : E \to Op(\mathcal{H})$ to $E' \subseteq E$, *i.e.* $Q|_{E'} : E' \to Op(\mathcal{H})$.

Next, we show how unitary event structures composed of the sequential, concurrent, and measurement operators interact with the removal of initial events.

Lemma 6.2.17. Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures. Consider $\mathsf{U}_1 \,;\, \mathsf{U}_2$ such that $l \in \mathcal{I}(\mathsf{U}_1 \,;\, \mathsf{U}_2)$. Then $(\mathsf{U}_1 \,;\, \mathsf{U}_2)\backslash l \equiv (\mathsf{U}_1\backslash l) \,;\, \mathsf{U}_2$.

Lemma 6.2.18. Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures. Consider $\mathsf{U}_1 \,||\, \mathsf{U}_2$ such that $l \in \mathcal{I}(\mathsf{U}_1 \,||\, \mathsf{U}_2)$. Then

$$
(\mathsf{U}_1 \,||\, \mathsf{U}_2)\backslash l \equiv \begin{cases} (\mathsf{U}_1\backslash l) \,||\, \mathsf{U}_2 & \text{if } l \in \mathcal{I}(\mathsf{U}_1) \\ \mathsf{U}_1 \,||\, (\mathsf{U}_2\backslash l) & \text{if } l \in \mathcal{I}(\mathsf{U}_2) \end{cases}
$$

Note that when composing two unitary event structures U_1 and U_2 through the measurement operator, $\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2)$, the initial events are τ_0^n and τ_1^n. If we remove τ_0^n, then we are left only with U_1. This is because τ_0^n is causally related to every event of U_1 and it is in conflict with all events from U_2. A similar reasoning applies when we remove τ_1^n.

Lemma 6.2.19. Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures. Consider $\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2)$ such that $l \in \mathcal{I}(\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2))$. Then

$$
(\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2))\backslash l \equiv \begin{cases} \mathsf{U}_1 & \text{if } l = \tau_0^n \\ \mathsf{U}_2 & \text{if } l = \tau_1^n \end{cases}
$$

6.3 Introducing cyclic behavior

To introduce cyclic behavior in the language shown in Section 6.1, we resort to while loops. The while loop, influenced by [120, Definition 3.1.1], has some similarities with measurements. Concretely, the projections P_0 and P_1 are associated to the validity of Boolean expressions. For instance, applying P_0 can be associated with a Boolean condition not holding, in which case the while loop terminates. On the other hand, the application of P_1 represents the Boolean condition being true, meaning that the instructions inside the while loop are executed and the loop continues. Since evaluating the Boolean condition is tied to applying the projections P_0 and P_1, we maintain the intuition that the operational semantics is essentially "stamping" the actions performed during a computation onto a tape. Hence, we use a while loop-based command in this section to introduce cyclic behavior.

We now update the syntax given in Section 6.1.

$$C ::= skip \mid U(\tilde{n}) \mid C\,;\,C \mid C \,||\, C \mid \mathtt{M}(n, C_1, C_2) \mid while\ \mathtt{M}(n, C)$$

where $while\ \mathtt{M}(n, C)$ is, as previously discussed, a while loop that stops the computation if the measurement is made by P_0 and continues the computation if the measurement is made by P_1.

We also need to update the set of qubits being used in a command C.

$$\begin{aligned}
&\mathsf{qVar}(skip) = \emptyset \\
&\mathsf{qVar}(U(\tilde{n})) = \tilde{n} \\
&\mathsf{qVar}(C_1\,;\,C_2) = \mathsf{qVar}(C_1) \cup \mathsf{qVar}(C_2) \\
&\mathsf{qVar}(C_1 \,||\, C_2) = \mathsf{qVar}(C_1) \cup \mathsf{qVar}(C_2) \\
&\mathsf{qVar}(\mathtt{M}(n, C_1, C_2)) = \{n\} \cup \mathsf{qVar}(C_1) \cup \mathsf{qVar}(C_2) \\
&\mathsf{qVar}(while\ \mathtt{M}(n, C)) = \{n\} \cup \mathsf{qVar}(C)
\end{aligned}$$

Note that in this section we do not limit the while loop to only appear on the right-hand side of the sequential composition. This is due to the while loop being defined in terms of a measurement, where there is always a possibility to finish the computation in a loop.

At last, we add the following rules to the rules in Figure 57.

$$while\ \mathtt{M}(n, C) \xrightarrow{P_0(n)} \checkmark \qquad while\ \mathtt{M}(n, C) \xrightarrow{P_1(n)} C\,;\,while\ \mathtt{M}(n, C)$$

We now delve into an example that illustrates the behavior of a while command.

Example 6.3.1. Figure 63 illustrates the behavior of a quantum coin toss, which, similarly to Example 4.3.2, produces a possibly empty sequence of $H(n)P_1(n)$ that ends with $P_0(n)$. We observe that the initial program has two possible transitions: (1) transits through $P_0(n)$ and the computation finishes; (2) transits through $P_1(n)$ to $H(n)\,;\, while\ \mathtt{M}(n, H(n))$, which executes $H(n)$ to transit to $while\ \mathtt{M}(n, H(n))$, which is the same command as the initial one.

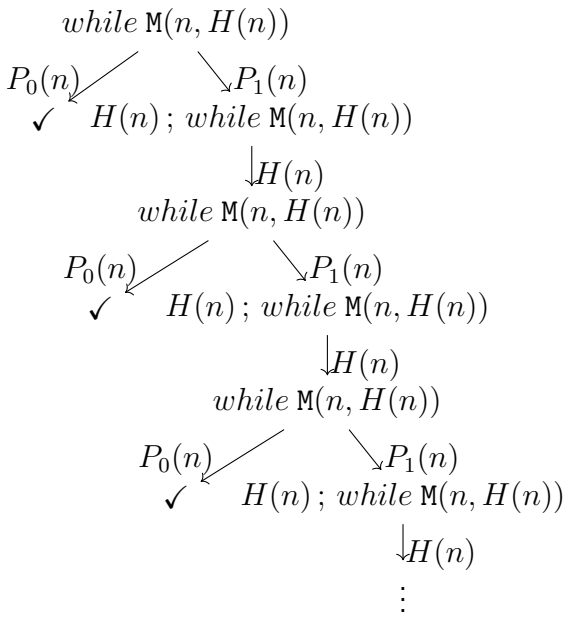


Figure 63: Fragment of the execution of $while\ \mathtt{M}(n, H(n))$

Now, let us focus on unitary event structures. We proceed by unlocking cyclic behavior in unitary event structures, via a suitable extension of Definition 4.3.3 to the quantum case. To do this, we follow a similar approach to the extension of Definition 4.2.11 to the quantum case. Concretely, we need to assure that the quantum operator is preserved throughout unitary event structures, *i.e.* if $\mathsf{U}_1 \trianglelefteq \mathsf{U}_2$ then the quantum operators of the events from U_2 that are also events in U_1 should be the same.

Definition 6.3.2. Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$ and $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$ be unitary event structures. Say $\mathsf{U}_1 \trianglelefteq \mathsf{U}_2$ if:

$$E_1 \subseteq E_2$$

$$\forall e, e' \,.\, e \leq_1 e' \Leftrightarrow e, e' \in E_1 \wedge e \leq_2 e'$$

$$\forall e, e' \,.\, e \#_1 e' \Leftrightarrow e, e' \in E_1 \wedge e \#_2 e'$$

$$\forall e \in E_1 \,.\, Q_1(e) = Q_2(e)$$

The following lemmas confirm that Definition 6.3.2 is a partial order and that it has a least element.

Lemma 6.3.3. $\trianglelefteq$ is a partial order.

Lemma 6.3.4. Define $\bot = (\emptyset, \emptyset, \emptyset, ! : \emptyset \to Op(\mathcal{H}))$. $\bot$ is the least element of $\trianglelefteq$.

We now extend Definition 4.3.6 to the quantum setting.

Definition 6.3.5. Let $\mathsf{U}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{U}_n \trianglelefteq \ldots$ be a ω-chain. Let $\mathsf{U}^\omega = (E^\omega, \leq^\omega, \#^\omega, Q^\omega)$ be its least upper bound where:

- $E^\omega = \cup_{n \in \omega} E_n$
- $\leq^\omega = \cup_{n \in \omega} \leq_n$
- $\#^\omega = \cup_{n \in \omega} \#_n$
- $Q^\omega(e) \Leftrightarrow \exists n \in \omega \,.\, e \in E_n$ and $Q_n(e) = Q^\omega(e)$

We then show that the structure in Definition 6.3.5 is a unitary event structure and a least upper bound in a chain of unitary event structures.

Lemma 6.3.6. U^ω is a unitary event structure.

Lemma 6.3.7. Let $\mathsf{U}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{U}_n \trianglelefteq \ldots$ be an ω-chain. Then U^ω is its least upper bound.

We have that sequential, concurrent, and measurement composition are monotone and continuous with respect to Definition 6.3.2. Recall that the monotone and continuous definitions are given by Definition 4.3.9.

Lemma 6.3.8. Let $\mathsf{U}, \mathsf{U}_1, \mathsf{U}_2$ be unitary event structures. If $\mathsf{U}_1 \trianglelefteq \mathsf{U}_2$ then $\mathsf{U} \,;\, \mathsf{U}_1 \trianglelefteq \mathsf{U} \,;\, \mathsf{U}_2$.

Lemma 6.3.9. Let $\mathsf{U}_1, \mathsf{U}_1', \mathsf{U}_2, \mathsf{U}_2'$ be unitary event structures. If $\mathsf{U}_1 \trianglelefteq \mathsf{U}_1'$ and $\mathsf{U}_2 \trianglelefteq \mathsf{U}_2'$ then $\mathsf{U}_1 \,||\, \mathsf{U}_2 \trianglelefteq \mathsf{U}_1' \,||\, \mathsf{U}_2'$.

Lemma 6.3.10. Let $\mathsf{U}_1, \mathsf{U}_1', \mathsf{U}_2, \mathsf{U}_2'$ be unitary event structures. If $\mathsf{U}_1 \trianglelefteq \mathsf{U}_1'$ and $\mathsf{U}_2 \trianglelefteq \mathsf{U}_2'$ then $\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2) \trianglelefteq \mathtt{M}(n, \mathsf{U}_1', \mathsf{U}_2')$.

Lemma 6.3.11. $\bigsqcup_m (\mathsf{U} \,;\, \mathsf{U}_m) = \mathsf{U} \,;\, \bigsqcup_m \mathsf{U}_m$.

Lemma 6.3.12. $\bigsqcup_{n,m} (\mathsf{U}_n \,||\, \mathsf{U}_m) = \bigsqcup_n \mathsf{U}_n \,||\, \bigsqcup_m \mathsf{U}_m$.

Lemma 6.3.13. $\bigsqcup_{n,m} (\mathtt{M}(q, \mathsf{U}_n, \mathsf{U}_m)) = \mathtt{M}(q, \bigsqcup_n \mathsf{U}_n, \bigsqcup_m \mathsf{U}_m)$.

When adapting Lemma 4.3.13 (helps to show that operators defined in event structures are continuous) and Lemma 4.3.17 (a version of the Kleene fixed-point theorem for the case of event structures) to the quantum realm, we notice that the proofs we need to do are analogous to the ones done in the non-deterministic case. Hence, we defer their formulation, and respective proofs, to Appendix A.3.

We now extend Definition 6.2.8 with the while command and let $\mathbb{U}$ denote the class of unitary event structures.

Definition 6.3.14. We interpret commands as unitary event structures as follows:

$$\begin{aligned}
&[\![skip]\!] = (\{sk\}, \{sk \leq sk\}, \emptyset, Q(sk) = Id)\\
&[\![U(\tilde{n})]\!] = (\{U_{\tilde{n}}\}, \{U_{\tilde{n}} \leq U_{\tilde{n}}\}, \emptyset, Q(U_{\tilde{n}}) = U(\tilde{n}))\\
&[\![\mathtt{M}(n, C_1, C_2)]\!] = \mathsf{P}_0^n \,;\, [\![C_1]\!] \text{ or } \mathsf{P}_1^n \,;\, [\![C_2]\!]\\
&[\![C_1 \,;\, C_2]\!] = [\![C_1]\!] \,;\, [\![C_2]\!]\\
&[\![C_1 \,||\, C_2]\!] = [\![C_1]\!] \,||\, [\![C_2]\!]\\
&[\![while\ \mathtt{M}(n, C)]\!] = fix(\Gamma^n)
\end{aligned}$$

where $\Gamma^n : \mathbb{U} \to \mathbb{U}$ is given by $\Gamma^n(\mathsf{U}) = \mathsf{P}_0^n \text{ or } (\mathsf{P}_1^n \,;\, \mathsf{U})$.

Γ^n is continuous, because it is composed of continuous functions.

We now develop an example where we draw a fragment of the unitary event structure obtained when interpreting a while command.

Example 6.3.15. Let us reuse the command in Example 6.3.1, $while\ \mathtt{M}(n, H(n))$. A fragment of the unitary event structure generated by $[\![while\ \mathtt{M}(n, H(n))]\!]$ is given by Figure 64:

From observing Figure 64, we note that maximal configurations are achieved whenever an event corresponding to $P_0(n)$ is in the configuration. The remaining configurations are composed only of events corresponding to $P_1(n)$ and $H(n)$.

6.4 Unitary Event Structures with initial state

This section shows how the restrictions added to Definition 3.4.1 (minimal conflict being transitive and the sum of the operators in minimal conflict, or an event itself, should be a unitary operator), when compared to Winskel's definition of quantum event structures [119, Definition 2], allow to extend a result obtained by Winskel [119, Theorem 3]. Furthermore, in this section, we show most of the proofs after stating a result, since these are not so evident.

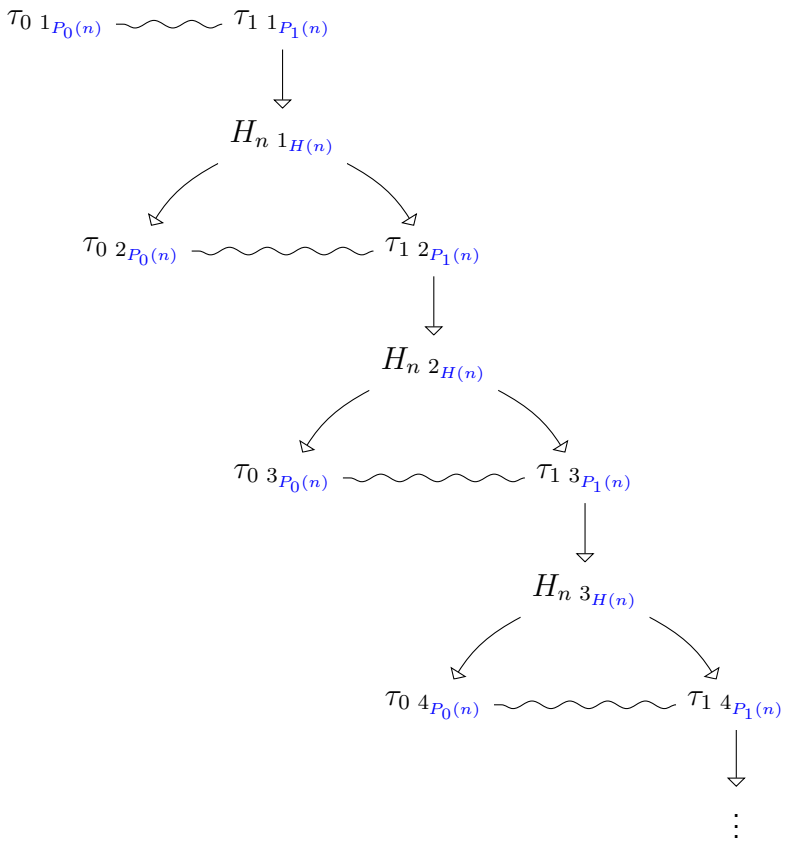


Figure 64: Unrolling the unitary event structure $[\![while\ \mathtt{M}(n, H(n))]\!]$

According to [119, Theorem 3], an elementary quantum event structure, *i.e.* an event structure without conflicting events, paired with an initial state ρ and a valuation function defined as $v(x) = \mathsf{Tr}(A_x^\dagger A_x \rho)$, corresponds to a probabilistic event structure. Recall that by Proposition [119, Proposition 5] it suffices to verify Equation 3.1 for $y \overset{e_1}{\frown} x_1, \ldots, y \overset{e_n}{\frown} x_n$ to check if a structure is a probabilistic event structure or not. Hence, we focus only on the relation between these events. These events are either in conflict or are concurrent. The reason why Winskel considers elementary quantum event structures lies in the fact that in a probabilistic event structure, the sum of the probability of events in conflict is less than or equal to one. For example, consider two conflicting events a and b, such that $v(\{a\}) = 1$ and $v(\{b\}) = 1$. In this case, the sum condition in Definition 3.3.1 fails when taking y as the empty set and x_1, x_2 as $\{a\}$ and $\{b\}$, respectively. Through some calculations, the sum simplifies to $v(\emptyset) - (v(\{a\}) + v(\{b\})) = 1 - (1 + 1) = -1$, which does not meet the criteria of being greater than or equal to 0.

We aim to remove the restriction of considering only elementary quantum event structures from [119, Theorem 3]. Hence, based on the previous example, we must restrict the probabilities associated with events in conflict. To do this, Definition 3.4.1 includes the condition that "$\rightsquigarrow$is transitive". Recall that the intuition behind this condition is that events in minimal conflict correspond to projections that form a

measurement. Since these projections are orthogonal, they are inherently captured by conflicting events. For instance, if we project onto $|0\rangle$, we cannot project onto $|1\rangle$. Furthermore, the probabilities associated with projections are complementary, *i.e.* if the probability of projecting onto $|0\rangle$ is p, then the probability of projecting onto $|1\rangle$ must be $1-p$. By applying this new information to the previous example of the valuation of conflicting events, we have that $v(\emptyset)-(v(\{a\})+v(\{b\}))=1-(p+(1-p))=1-1=0$, which meets the criteria of being greater than or equal to 0. Here, the event a represents the projection to $|0\rangle$ and the event b represents the projection to $|1\rangle$. Furthermore, Definition 3.4.1 also demands the sum of events in minimal conflict or the event itself to be a unitary. This is useful to ensure that we are only considering unitary operations.

To achieve our goal, the difference between our definition and Winskel's quantum event structures lies in the additional restrictions we impose. Therefore, we use as basis the proof outlined in [119, Theorem 3]. This entails that we need only to consider the case in which all the events are mapped to projections such that either all events are in conflict or there are events in conflict. For the former, we have everything. For the latter, we need extra machinery, which we show here.

Recall that according to [119, Proposition 5], to show that a structure is a probabilistic event structure, we only need to show that the condition in Definition 3.3.1 holds for $y \xrightarrow{e_1} x_1, \ldots, y \xrightarrow{e_n} x_n$. We then build a unitary event structure formed by the events of $y \xrightarrow{e_1} x_1, \ldots, y \xrightarrow{e_n} x_n$.

Definition 6.4.1. Let $\mathsf{U} = (E, \leq, \#, Q)$ be a unitary event structure and $y \in \mathcal{C}(\mathsf{U})$. Define $\tilde{\mathsf{U}}_y = (\tilde{E}, \tilde{\leq}, \tilde{\#}, \tilde{Q})$ as follows:

$$\tilde{E} = \{e \mid y \xrightarrow{e} y \cup \{e\}\}$$
$$\tilde{\leq} = \{(e, e) \mid e \in \tilde{E}\}$$
$$\tilde{\#} = \# \cap (\tilde{E} \times \tilde{E})$$
$$\tilde{Q} = Q|_{\tilde{E}}$$

Lemma 6.4.2. $\tilde{\mathsf{U}}_y = (\tilde{E}, \tilde{\leq}, \tilde{\#}, \tilde{Q})$ is a unitary event structure.

Next, we merge the conflicting events from Definition 6.4.1. This gives us an empty conflict relation, hence an elementary unitary event structure (recall that $[e] = \{e' \mid e = e',\ e \sim\!\!\sim e'\}$).

Definition 6.4.3. Let $\tilde{\mathsf{U}}_y$ be a unitary event structure. Define $\hat{\mathsf{U}} = (\hat{E}, \hat{\leq}, \hat{\#}, \hat{Q})$ as follows:

$$\hat{E} = \{[e] \mid e \in \tilde{\mathsf{U}}\}$$
$$\hat{\leq} = \{([e], [e]) \mid [e] \in \hat{E}\}$$

$$\hat{\#} = \emptyset$$

$$\hat{Q}([e]) = \begin{cases} Q(e) & \text{if } |[e]| = 1 \\ \sum_{e' \in [e]} Q(e') & \text{if } |[e]| > 1 \end{cases}$$

Lemma 6.4.4. $\hat{\mathsf{U}} = (\hat{E}, \hat{\leq}, \hat{\#}, \hat{Q})$ is a unitary event structure.

We then define a map of event structures, $proj_{\tilde{\mathsf{E}}}$, between the underlying event structures of $\tilde{\mathsf{U}}_y$ and $\hat{\mathsf{U}}$. Note that $proj_{\tilde{\mathsf{E}}}$ is a total map of event structures. Hence, by recalling Example 3.2.12, for any configuration $x \in \mathcal{C}(\tilde{\mathsf{E}}_y)$ we have $|x| = |proj_{\tilde{\mathsf{E}}}[x]|$. Formally, the definition of the map of event structures is as follows: $proj_{\tilde{\mathsf{E}}} : (\tilde{E}, \tilde{\leq}, \tilde{\#}) \to (\hat{E}, \hat{\leq}, \hat{\#})$, where

$$proj_{\tilde{\mathsf{E}}} : \tilde{E} \to \hat{E}$$
$$e \mapsto [e]$$

Lemma 6.4.5. $proj_{\tilde{\mathsf{E}}} : (\tilde{E}, \tilde{\leq}, \tilde{\#}) \to (\hat{E}, \hat{\leq}, \hat{\#})$ is a map of event structures.

We now prove that our intuition about a feature of $proj_{\tilde{\mathsf{E}}}$ is true, *i.e.* that for a given configuration $x \in \mathcal{C}(\tilde{\mathsf{E}}_y)$ we have $|x| = |proj_{\tilde{\mathsf{E}}}[x]|$.

Lemma 6.4.6. Consider $\tilde{\mathsf{E}}_y$, $\hat{\mathsf{E}}$, and $proj_{\tilde{\mathsf{E}}}$. If $x \in \mathcal{C}(\tilde{\mathsf{E}}_y)$ then $|proj_{\tilde{\mathsf{E}}}[x]| = |x|$.

Proof. Let $x \in \mathcal{C}(\tilde{\mathsf{E}}_y)$. We know that $proj_{\tilde{\mathsf{E}}}$ is total, hence

$$proj_{\tilde{\mathsf{E}}}[x] = \{proj_{\tilde{\mathsf{E}}}(e) \mid e \in x\} = \{[e] \mid e \in x\}$$

Furthermore, $proj_{\tilde{\mathsf{E}}}$ is locally injective. Hence if $e_1, \ldots, e_n \in x$ then

$$proj_{\tilde{\mathsf{E}}}(e_1), \ldots, proj_{\tilde{\mathsf{E}}}(e_n) \in proj_{\tilde{\mathsf{E}}}[x]$$

Thus $|x| = |proj_{\tilde{\mathsf{E}}}[x]|$. □

The next lemma states that for any $\hat{x} \in \mathcal{C}(\hat{\mathsf{E}})$ and $\tilde{x} \in \mathcal{C}(\tilde{\mathsf{E}})$, the set of configurations $\tilde{x}$ satisfying $proj_{\tilde{\mathsf{E}}}[\tilde{x}] = \hat{x}$ consists precisely of those configurations formed by selecting one event from each equivalence class that makes up $\hat{x}$. Furthermore, the equivalence classes in $\hat{x}$ are composed of single events or events in minimal conflict. Consequently, multiple configurations in $\mathcal{C}(\tilde{\mathsf{E}})$ can be mapped to the same configuration in $\mathcal{C}(\hat{\mathsf{E}})$.

Lemma 6.4.7. Consider $\tilde{\mathsf{E}}_y$, $\hat{\mathsf{E}}$. Let $\hat{x} = \{[e_1], \ldots, [e_n]\} \in \mathcal{C}(\hat{\mathsf{E}})$ and $\tilde{x} \in \mathcal{C}(\tilde{\mathsf{E}}_y)$. Then $\{\tilde{x} \mid proj_{\tilde{\mathsf{E}}}[\tilde{x}] = \hat{x}\} = \{\{\tilde{e}_1, \ldots, \tilde{e}_n\} \mid \forall i, \tilde{e}_i \in [e_i]\}$.

Proof. We have two cases:

- $\{\tilde{x} \mid proj_{\tilde{\mathsf{E}}}[\tilde{x}] = \hat{x}\} \subseteq \{\{\tilde{e}_1, \ldots, \tilde{e}_n\} \mid \forall i, \tilde{e}_i \in [e_i]\}$

  Let $\tilde{x} = \{\tilde{e}_1, \ldots, \tilde{e}_n\} \in \mathcal{C}(\tilde{\mathsf{E}}_y)$. By Definition 3.2.10, $proj_{\tilde{\mathsf{E}}}[\tilde{x}] = \hat{x} \in \mathcal{C}(\hat{\mathsf{E}})$. Furthermore, $proj_{\tilde{\mathsf{E}}}[\tilde{x}] = proj_{\tilde{\mathsf{E}}}(e_1), \ldots, proj_{\tilde{\mathsf{E}}}(e_n) = \{[e_1], \ldots, [e_n]\}$. By definition of $[e]$, we know that $\forall i \,.\, \tilde{e}_i \in \tilde{x}$ we have $\tilde{e}_i \in [e_i]$. Hence, we are done.

- $\{\{\tilde{e}_1, \ldots, \tilde{e}_n\} \mid \forall i, \tilde{e}_i \in [e_i]\} \subseteq \{\tilde{x} \mid proj_{\tilde{\mathsf{E}}}[\tilde{x}] = \hat{x}\}$

  Let $\{\tilde{e}_1, \ldots, \tilde{e}_n\} \in \{\{\tilde{e}_1, \ldots, \tilde{e}_n\} \mid \forall i, \tilde{e}_i \in [e_i]\}$. We need to show that $\{\tilde{e}_1, \ldots, \tilde{e}_n\} \in \mathcal{C}(\tilde{\mathsf{E}}_y)$.

  1. $\forall \tilde{e}, \tilde{e}' \in \{\tilde{e}_1, \ldots, \tilde{e}_n\} \,.\, \neg(\tilde{e} \# \tilde{e}')$

     Let $\tilde{e}, \tilde{e}' \in \{\tilde{e}_1, \ldots, \tilde{e}_n\}$. Then we know that $\tilde{e} \in [e]$ and $\tilde{e}' \in [e']$. By definition of $[e]$, we have that $\neg(\tilde{e} \sim\!\!\!\!\sim \tilde{e}')$, which by Definition 6.4.1 means $\neg(\tilde{e} \# \tilde{e}')$.

  2. $\forall \tilde{e}, \tilde{e}' \,.\, \tilde{e}' \tilde{\leq} \tilde{e} \wedge \tilde{e} \in \{\tilde{e}_1, \ldots, \tilde{e}_n\} \Rightarrow \tilde{e}' \in \{\tilde{e}_1, \ldots, \tilde{e}_n\}$

     By Definition 6.4.1, the causal relation is the equality. Hence this condition trivially holds.

  Since $\{\tilde{e}_1, \ldots, \tilde{e}_n\} \in \mathcal{C}(\tilde{\mathsf{E}}_y)$, it lacks to show that $proj_{\tilde{\mathsf{E}}_y}[\{\tilde{e}_1, \ldots, \tilde{e}_n\}] \in \mathcal{C}(\tilde{\mathsf{E}}_y) = \hat{x}$. That comes directly from applying $proj_{\tilde{\mathsf{E}}_y}$ to $\{\tilde{e}_1, \ldots, \tilde{e}_n\} \in \mathcal{C}(\tilde{\mathsf{E}}_y)$, as follows:

$$proj_{\tilde{\mathsf{E}}_y}[\{\tilde{e}_1, \ldots, \tilde{e}_n\}] = \{proj_{\tilde{\mathsf{E}}_y}(\tilde{e}_1), \ldots, proj_{\tilde{\mathsf{E}}_y}(\tilde{e}_n)\} = \{[e_1], \ldots, [e_n]\} = \hat{x}$$

□

To better understand the mapping of event structures, $proj_{\tilde{\mathsf{E}}}$, Lemma 6.4.6, and Lemma 6.4.7 consider the following example.

Example 6.4.8. Let us consider the following unitary event structure (where we omit the associated quantum operators):

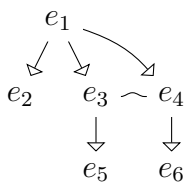


Figure 65: Unitary event structure U

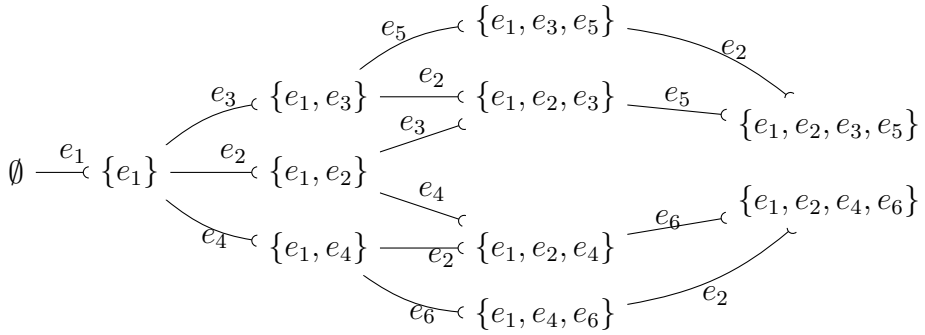


Figure 66: Covering chains of U

From Figure 65 we can deduce all possible covering chains and consequently all possible configurations.

Let us now focus on Figure 67, which is a specific part of Figure 66.

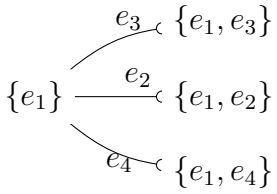


Figure 67: Fragment of the covering chains of U

From Figure 67 we can build the unitary event structure $\tilde{U}$, shown in Figure 68.

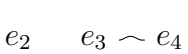


Figure 68: Unitary event structure $\tilde{\mathrm{U}}_y$

By applying the mapping of event structures $proj_{\tilde{\mathrm{E}}}$ we obtain:

$$e_3, e_4 \overset{proj_{\tilde{\mathrm{E}}}}{\mapsto} [e_{3,4}]$$
$$e_2 \overset{proj_{\tilde{\mathrm{E}}}}{\mapsto} [e_2]$$

which gives the unitary event structure in Figure 69.



Figure 69: Unitary event structure $\hat{\mathrm{U}}_y$

Note that all the events of $\tilde{\mathrm{U}}$ were mapped to $\hat{\mathrm{U}}$. Hence the map $proj_{\tilde{\mathrm{E}}}$ is total. Furthermore, the events e_3 and e_4 are mapped to the same event $[e_{3,4}]$.

Now we verify that the size of a configuration $\tilde{x} \in \mathcal{C}(\tilde{\mathsf{E}})$ remains unchanged after applying the map $proj_{\tilde{\mathsf{E}}}$. The set of configurations of $\tilde{\mathsf{E}}$ is $\mathcal{C}(\tilde{\mathsf{E}}) = \{\emptyset, \{e_2\}, \{e_3\}, \{e_4\}, \{e_3, e_2\}, \{e_4, e_2\}\}$ and the set of configurations of $\hat{\mathsf{E}}$ is $\mathcal{C}(\hat{\mathsf{E}}) = \{\emptyset, \{[e_2]\}, \{[e_{3,4}]\}, \{[e_{3,4}], [e_2]\}\}$. By applying $proj_{\tilde{\mathsf{E}}}$ to each configuration of $\mathcal{C}(\tilde{\mathsf{E}})$ we have:

$$\emptyset \overset{proj_{\tilde{\mathsf{E}}}}{\mapsto} \emptyset$$

$$\{e_2\} \overset{proj_{\tilde{\mathsf{E}}}}{\mapsto} [e_2]$$

$$\{e_3\}, \{e_4\} \overset{proj_{\tilde{\mathsf{E}}}}{\mapsto} [e_{3,4}]$$

$$\{e_3, e_2\}, \{e_4, e_2\} \overset{proj_{\tilde{\mathsf{E}}}}{\mapsto} \{[e_{3,4}], [e_2]\}$$

We can then conclude that the map $proj_{\tilde{\mathsf{E}}}$ preserves the size of the configurations in $\mathcal{C}(\tilde{\mathsf{E}})$.

Now it lacks to see Lemma 6.4.7 in action. For this we focus on the case where

$$\{e_3, e_2\}, \{e_4, e_2\} \overset{proj_{\tilde{\mathsf{E}}}}{\mapsto} \{[e_{3,4}], [e_2]\}$$

First we expand the equivalence classes: $\{\{e_3, e_4\}, \{e_2\}\}$. Next, we calculate the set of configurations formed by picking one event from each equivalence class of $\{[e_{3,4}], [e_2]\}$:

$$\{\{\tilde{e}_1, \ldots, \tilde{e}_n\} \mid \forall i, \tilde{e}_i \in [e_i]\} = \{\{e_3, e_2\}, \{e_4, e_2\}\}$$

From this, we can see that the configurations we obtained are precisely the ones that map to $\{[e_{3,4}], [e_2]\}$ after applying $proj_{\tilde{\mathsf{E}}}$.

To prove our claim, we need one more auxiliary lemma. The idea behind the following lemma is: if we can show that $d^{(n)}_{\tilde{v}}[\emptyset\,;\, \{e_1\}, \ldots, \{e_n\}] \geq 0$, where $\tilde{v}(\tilde{x}) = \dfrac{v(y \cup \tilde{x})}{v(y)}$, then it follows directly that $d^{(n)}_{v}[y\,;\, y \cup \{e_1\}, \ldots, y \cup \{e_n\}] \geq 0$. In other words, the condition in Definition 3.3.1 is satisfied.

Lemma 6.4.9. Consider $\tilde{\mathsf{U}}_y$ together with an initial state ρ_y, such that all the events of $\tilde{\mathsf{U}}_y$ are projections. For any $\tilde{x} \in \mathcal{C}(\tilde{\mathsf{U}}_y)$ let $\tilde{v}(\tilde{x}) = \dfrac{v(y \cup \tilde{x})}{v(y)}$. Then $d^{(n)}_{\tilde{v}}[\emptyset\,;\, \{e_1\}, \ldots, \{e_n\}] \geq 0$.

Proof. To show $d^{(n)}_{\tilde{v}}[\emptyset\,;\, \{e_1\}, \ldots, \{e_n\}] \geq 0$, it is helpful to consider $\hat{\mathsf{E}}$ and $proj_{\tilde{\mathsf{E}}}$.

Recall that $\hat{\mathsf{E}}$ does not have events in conflict. Hence, by [119, Corollary 3], we have that $\hat{\mathsf{U}}$ with ρ_y and $\hat{v}(\hat{x}) = Tr(A^{\dagger}_{\hat{x}} A_{\hat{x}} \rho_y)$ is a probabilistic event structure.

We need to show that $\hat{v}(\hat{x}) = Tr(A^{\dagger}_{\hat{x}} A_{\hat{x}} \rho_y) = \sum_{\substack{\tilde{y} \in \mathcal{C}(\tilde{\mathsf{U}}) \\ proj_{\tilde{\mathsf{E}}_y}[\tilde{y}] = \hat{x}}} \tilde{v}(\tilde{y})$.

We note that $\hat{Q}([e]) = U = \sum_{\tilde{e} \in [e]} \tilde{Q}(\tilde{e})$ when $|[e]| > 1$ and that for a configuration $\hat{x} \in \mathcal{C}(\hat{\mathsf{E}})$, the operator $A_{\hat{x}} = \hat{Q}([e_n]) \cdots \hat{Q}([e_1]) = \prod_{[e] \in \hat{x}} \hat{Q}([e])$.

Let us expand $\prod_{[e]\in\hat{x}} \hat{Q}([e])$:

$$\prod_{[e]\in\hat{x}} \hat{Q}([e]) = \prod_{[e]\in\hat{x}} \left(\sum_{\tilde{e}\in[e]} \tilde{Q}(\tilde{e}) \right) = \prod_{i=1}^{n} \left(\sum_{\tilde{e}_i\in[e_i]} \tilde{Q}(\tilde{e}_i) \right) = \sum_{\substack{\tilde{e}_1,\cdots,\tilde{e}_n\in\tilde{E}_y \\ \forall i,\tilde{e}_i\in[e_i]}} \left(\prod_{i=1}^{n} \tilde{Q}(\tilde{e}_i) \right)$$

$$= \sum_{\substack{\tilde{e}_1,\cdots,\tilde{e}_n\in\tilde{E}_y \\ \forall i,\tilde{e}_i\in[e_i]}} \left(\prod_{\tilde{e}\in\{\tilde{e}_1,\cdots,\tilde{e}_n\}} \tilde{Q}(\tilde{e}) \right) \overset{\text{(Lemma 6.4.7)}}{=} \sum_{\substack{\tilde{x}\in\mathcal{C}(\tilde{\mathsf{U}}_y) \\ proj_{\tilde{\mathsf{E}}_y}[\tilde{x}]=\hat{x}}} \left(\prod_{\tilde{e}\in\tilde{x}} \tilde{Q}(\tilde{e}) \right) = \sum_{\substack{\tilde{x}\in\mathcal{C}(\tilde{\mathsf{U}}_y) \\ proj_{\tilde{\mathsf{E}}_y}[\tilde{x}]=\hat{x}}} A_{\tilde{x}}$$

Then it follows directly that:

$$\hat{v}(\hat{x}) = Tr(A_{\hat{x}}^{\dagger} A_{\hat{x}} \rho_y) = Tr\left(\left(\sum_{\substack{\tilde{y}\in\mathcal{C}(\tilde{\mathsf{U}}_y) \\ proj_{\tilde{\mathsf{E}}_y}[\tilde{y}]=\hat{x}}} A_{\tilde{y}}^{\dagger} \right) \left(\sum_{\substack{\tilde{y}'\in\mathcal{C}(\tilde{\mathsf{U}}_y) \\ proj_{\tilde{\mathsf{E}}_y}[\tilde{y}']=\hat{x}}} A_{\tilde{y}'} \right) \rho_y \right)$$

$$= Tr\left(\sum_{\substack{\tilde{y}\in\mathcal{C}(\tilde{\mathsf{U}}) \\ proj_{\tilde{\mathsf{E}}_y}[\tilde{y}]=\hat{x}}} \sum_{\substack{\tilde{y}'\in\mathcal{C}(\tilde{\mathsf{U}}) \\ proj_{\tilde{\mathsf{E}}_y}[\tilde{y}']=\hat{x}}} A_{\tilde{y}}^{\dagger} A_{\tilde{y}'} \rho_y \right) \overset{(\star)}{=} Tr\left(\sum_{\substack{\tilde{y}\in\mathcal{C}(\tilde{\mathsf{U}}) \\ proj_{\tilde{\mathsf{E}}_y}[\tilde{y}]=\hat{x}}} A_{\tilde{y}}^{\dagger} A_{\tilde{y}} \rho_y \right)$$

$$= \sum_{\substack{\tilde{y}\in\mathcal{C}(\tilde{\mathsf{U}}) \\ proj_{\tilde{\mathsf{E}}_y}[\tilde{y}]=\hat{x}}} \Big(Tr(A_{\tilde{y}}^{\dagger} A_{\tilde{y}} \rho_y) \Big) = \sum_{\substack{\tilde{y}\in\mathcal{C}(\tilde{\mathsf{U}}) \\ proj_{\tilde{\mathsf{E}}_y}[\tilde{y}]=\hat{x}}} \tilde{v}(\tilde{y})$$

Where in step $(\star)$ we note that if $\tilde{y} \neq \tilde{y}'$ then $A_{\tilde{y}}^{\dagger} A_{\tilde{y}'} = 0$. That is straightforward to see because $proj_{\tilde{\mathsf{E}}_y}[\tilde{y}] = \hat{x} = proj_{\tilde{\mathsf{E}}_y}[\tilde{y}']$. Hence it exists $\tilde{e} \in \tilde{y}$ and $\tilde{e}' \in \tilde{y}'$ such that $\tilde{Q}(\tilde{e}) \cdot \tilde{Q}(\tilde{e}') = 0$. In other words, $\tilde{e}$ and $\tilde{e}'$ are in conflict.

Now we are ready to show that $d_{\tilde{v}}^{(n)}[\emptyset\,;\,\{e_1\},\ldots,\{e_n\}] \geq 0$.

By[119, Proposition 1],

$$d_{\tilde{v}}^{(n)}[\emptyset\,;\,\{e_1\},\ldots,\{e_n\}] = \tilde{v}(\emptyset) - \sum_{\emptyset\neq I\subseteq\{1,\ldots,n\}} (-1)^{|I|+1} \tilde{v}\left(\bigcup_{i\in I} \{e_i\} \right)$$

$$= \sum_{I\subseteq\{1,\ldots,n\}} (-1)^{|I|} \tilde{v}\left(\bigcup_{i\in I} \{e_i\} \right)$$

Now we note that it exists events that are in conflict, and since the union of events that are in conflict do not form a configuration, we have that its valuation is zero. We can then remove those terms from the sum.

$$\sum_{I\subseteq\{1,\ldots,n\}} (-1)^{|I|} \tilde{v}\left(\bigcup_{i\in I} \{e_i\} \right)$$

$$= \sum_{\substack{I\subseteq\{1,\ldots,n\} \\ \forall i,j\in I\,.\,e_i \text{ co } e_j}} (-1)^{|I|} \tilde{v}\left(\bigcup_{i\in I} \{e_i\} \right)$$

$$= \sum_{\substack{I\subseteq\{1,\dots,n\}\\ \bigcup_{i\in I}\{e_i\}\in\mathcal{C}(\tilde{\mathsf{U}})}} (-1)^{|I|}\tilde{v}\left(\bigcup_{i\in I}\{e_i\}\right)$$

On the other side, on $\hat{\mathsf{U}}$, we have:

$$\begin{aligned}
d^{(k)}_{\hat{v}}[\emptyset\,;\,\hat{x}_1,\dots,\hat{x}_k] &= \hat{v}(\emptyset) - \sum_{\emptyset\neq J\subseteq\{1,\dots,k\}} (-1)^{|J|+1}\hat{v}(\bigcup_{j\in J}\hat{x}_j)\\
&= \sum_{J\subseteq\{1,\dots,k\}} (-1)^{|J|}\hat{v}(\bigcup_{j\in J}\hat{x}_j)\\
&= \sum_{J\subseteq\{1,\dots,k\}} (-1)^{|J|}\hat{v}(\bigcup_{j\in J}\{\hat{e}_j\})\\
&= \sum_{\substack{J\subseteq\{1,\dots,k\}\\ \tilde{y}\in\mathcal{C}(\tilde{\mathsf{U}}_y)\\ proj_{\tilde{\mathsf{E}}_y}[\tilde{y}]=\bigcup_{j\in J}\{\hat{e}_j\}}} (-1)^{|J|}\tilde{v}(\tilde{y})\\
&= \sum_{\substack{J\subseteq\{1,\dots,k\}\\ I\subseteq\{1,\dots,n\}\\ \bigcup_{i\in I}\{e_i\}\in\mathcal{C}(\tilde{\mathsf{U}}_y)\\ proj_{\tilde{\mathsf{E}}_y}[\bigcup_{i\in I}\{e_i\}]=\bigcup_{j\in J}\{\hat{e}_j\}}} (-1)^{|J|}\tilde{v}(\bigcup_{i\in I}\{e_i\})
\end{aligned}$$

By Lemma 6.4.6 we know that $|proj_{\tilde{\mathsf{E}}_y}[\bigcup_{i\in I}\{e_i\}]| = |\bigcup_{i\in I}\{e_i\}| = |I|$. Furthermore we also know that $|\bigcup_{j\in J}\{e_j\}| = |J|$. Since $proj_{\tilde{\mathsf{E}}_y}[\bigcup_{i\in I}\{e_i\}] = \bigcup_{j\in J}\{\hat{e}_j\}$, again by Lemma 6.4.6 we have $|proj_{\tilde{\mathsf{E}}_y}[\bigcup_{i\in I}\{e_i\}]| = |\bigcup_{j\in J}\{\hat{e}_j\}|$. Thus $|I| = |J|$. Hence

$$\sum_{\substack{J\subseteq\{1,\dots,k\}\\ I\subseteq\{1,\dots,n\}\\ \bigcup_{i\in I}\{e_i\}\in\mathcal{C}(\tilde{\mathsf{U}}_y)\\ proj_{\tilde{\mathsf{E}}_y}[\bigcup_{i\in I}\{e_i\}]=\bigcup_{j\in J}\{\hat{e}_j\}}} (-1)^{|J|}\tilde{v}(\bigcup_{i\in I}\{e_i\}) = \sum_{\substack{J\subseteq\{1,\dots,k\}\\ I\subseteq\{1,\dots,n\}\\ \bigcup_{i\in I}\{e_i\}\in\mathcal{C}(\tilde{\mathsf{U}}_y)\\ proj_{\tilde{\mathsf{E}}_y}[\bigcup_{i\in I}\{e_i\}]=\bigcup_{j\in J}\{\hat{e}_j\}}} (-1)^{|I|}\tilde{v}(\bigcup_{i\in I}\{e_i\})$$

$$= \sum_{\substack{I\subseteq\{1,\dots,n\}\\ \bigcup_{i\in I}\{e_i\}\in\mathcal{C}(\tilde{\mathsf{U}}_y)}} \left(\sum_{\substack{J\subseteq\{1,\dots,k\}\\ proj_{\tilde{\mathsf{E}}_y}[\bigcup_{i\in I}\{e_i\}]=\bigcup_{j\in J}\{\hat{e}_j\}}} (-1)^{|I|}\tilde{v}(\bigcup_{i\in I}\{e_i\})\right) \overset{(\star)}{=} \sum_{\substack{I\subseteq\{1,\dots,n\}\\ \bigcup_{i\in I}\{e_i\}\in\mathcal{C}(\tilde{\mathsf{U}}_y)}} (-1)^{|I|}\tilde{v}(\bigcup_{i\in I}\{e_i\})$$

$$= d^{(n)}_{\tilde{v}}[\emptyset\,;\,\tilde{x}_1,\dots,\tilde{x}_n]$$

In step $(\star)$ the sum no longer depends on J, hence we drop it.

We shown that $d^{(n)}_{\tilde{v}}[\emptyset\,;\,\tilde{x}_1,\dots,\tilde{x}_n] = d^{(k)}_{\hat{v}}[\emptyset\,;\,\hat{x}_1,\dots,\hat{x}_k]$.

Hence $d^{(n)}_{\tilde{v}}[\emptyset\,;\,\tilde{x}_1,\dots,\tilde{x}_n] \geq 0$. □

Now we are ready to extend [119, Theorem 3].

Proposition 6.4.10. Let U $= (E, \leq, \#, Q)$ be a unitary event structure with initial state ρ. For each $x \in \mathcal{C}(\mathsf{U})$ let $v(x) = Tr(\rho_x) = Tr(A_x^\dagger A_x \rho)$. Then U $= (E, \leq, \#, v)$ is a probabilistic event structure.

Proof. From [119, Proposition 5] we need to show $d_v^{(n)}[y\,;\, x_1, \ldots x_n]$ when $y \xrightarrow{e_1,\ldots,e_n} x_1, \ldots, x_n$. We then identify the following cases:

1. $v(\emptyset) = 1$
2. $\exists e_i \in e_1, \ldots, e_n$ such that $Q(e_i)$ is a unitary
3. $\forall e_i \in e_1, \ldots, e_n$ we have $Q(e_i)$ is a projection
   (a) all the events are concurrent
   (b) all events are in conflict
   (c) there are events in conflict

The proof of 1, 2, and 3.(a) can be found in [119, Theorem 3]. Thus, we only show the proof of 3.(b) and 3.(c).

3.(b) Case every event is in conflict we know that the sum of the associated quantum operators is a unitary. Hence we are in case 2. and consequently $d_v^{(n)}[y\,;\, x_1, \ldots x_n] = 0$.

3.(c) Case there are events in conflict :

$$
\begin{aligned}
& d_v^{(n)}[y\,;\, x_1, \ldots x_n] \\
=& v(y) - \sum_{\emptyset \neq I \subseteq \{1,\ldots,n\}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i) \\
=& v(y) - \left(\sum_{\substack{\emptyset \neq I \subseteq \{1,\ldots,n\} \\ |I|=1}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i) + \sum_{\substack{\emptyset \neq I \subseteq \{1,\ldots,n\} \\ |I|>1}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i) \right) \\
=& v(y) - \sum_{i=1}^{n} v(x_i) - \sum_{\substack{\emptyset \neq I \subseteq \{1,\ldots,n\} \\ |I|>1}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i)
\end{aligned}
$$

Focus on

$$
\sum_{\substack{\emptyset \neq I \subseteq \{1,\ldots,n\} \\ |I|>1}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i)
$$

Since $x_i = \{e_i\} \cup y$ then

$$\sum_{\substack{\emptyset \neq I \subseteq \{1,\dots,n\} \\ |I|>1}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i) = \sum_{\substack{\emptyset \neq I \subseteq \{1,\dots,n\} \\ |I|>1}} (-1)^{|I|+1} v(\bigcup_{i \in I} \{e_i\} \cup y)$$

We know that with $|I| > 1$ we are not considering singletons. Hence we are either making the union of events that are concurrent or are in conflict. W.l.o.g consider $v(\{e_j, e_k\} \cup y)$ with $1 \leq j \neq k \leq n$. Case $e_j \# e_k$ then we know that $Q(e_j) \cdot Q(e_k) = 0 = Q(e_k) \cdot Q(e_j)$ and consequently we have $v(\{e_j, e_k\} \cup y) = Tr(A_y^\dagger \cdot (Q(e_j) \cdot Q(e_k))^\dagger \cdot Q(e_j) \cdot Q(e_k) \cdot A_y \rho) = 0$. On the other side, case e_j co e_k then we know that $Q(e_j) \cdot Q(e_k) = Q(e_k) \cdot Q(e_j)$ and consequently $v(\{e_j, e_k\} \cup y) \geq 0$.

When events are in conflict their contribution to the sum is null, hence we can discard them. As a consequence, the sum is composed of elements that are concurrent. Hence we have

$$\begin{aligned}
& v(y) - \sum_{i=1}^{n} v(x_i) - \sum_{\substack{\emptyset \neq I \subseteq \{1,\dots,n\} \\ |I|>1}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i) \\
=& v(y) - \sum_{i=1}^{n} v(x_i) - \sum_{\substack{\emptyset \neq I \subseteq \{1,\dots,n\} \\ |I|>1 \\ \bigcup_{i \in I} x_i \in \mathcal{C}(\mathsf{U})}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i) \\
=& v(y) - \sum_{\substack{\emptyset \neq I \subseteq \{1,\dots,n\} \\ |I|=1}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i) - \sum_{\substack{\emptyset \neq I \subseteq \{1,\dots,n\} \\ |I|>1 \\ \forall (i \neq j) \in I \,.\, e_i \text{ co } e_j}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i) \\
=& v(y) - \sum_{\substack{\emptyset \neq I \subseteq \{1,\dots,n\} \\ \forall (i \neq j) \in I \,.\, e_i \text{ co } e_j}} (-1)^{|I|+1} v(\bigcup_{i \in I} x_i)
\end{aligned}$$

Despite removing the valuations from ill-configurations, we are not in case $3.$(a) since there are still events in conflict. We thus resort to Lemma 6.4.9. Concretely:

$$\begin{aligned}
& d_{\tilde{v}}^{(n)}[\emptyset\,;\, \tilde{x}_1, \dots, \tilde{x}_n] \geq 0 \\
\Leftrightarrow & \tilde{v}(\emptyset) - \sum_{\substack{\emptyset \neq I \subseteq \{1,\dots,n\} \\ \bigcup_{i \in I} \{e_i\} \in \mathcal{C}(\tilde{\mathsf{E}}_y)}} (-1)^{|I|+1} \tilde{v}(\bigcup_{i \in I} \{e_i\}) \geq 0 \\
\Leftrightarrow & \sum_{\substack{I \subseteq \{1,\dots,n\} \\ \bigcup_{i \in I} \{e_i\} \in \mathcal{C}(\tilde{\mathsf{E}}_y)}} (-1)^{|I|} \tilde{v}(\bigcup_{i \in I} \{e_i\}) \geq 0 \\
\Leftrightarrow & \sum_{\substack{I \subseteq \{1,\dots,n\} \\ \bigcup_{i \in I} \{e_i\} \in \mathcal{C}(\tilde{\mathsf{E}}_y)}} (-1)^{|I|} \frac{v(y \cup \bigcup_{i \in I} \{e_i\})}{v(y)} \geq 0
\end{aligned}$$

$$\Leftrightarrow \sum_{\substack{I\subseteq\{1,\ldots,n\}\\ \bigcup_{i\in I}\{e_i\}\in\mathcal{C}(\tilde{\mathsf{E}}_y)}} (-1)^{|I|} v(y \cup \bigcup_{i\in I}\{e_i\}) \geq 0$$

$$\Leftrightarrow d_v^{(n)}[y\,;\, x_1,\ldots,x_n] \geq 0$$

With all cases proved, we have that $\mathsf{U} = (E, \leq, \#, v)$ is a probabilistic event structure. □

Hence we proved that any unitary event structure with an initial state is a probabilistic event structure.

6.5 Soundness and Adequacy (with initial state)

This section presents the equivalence between the operational and the denotational semantics. These results are supported by previous ones obtained in Section 6.2 and Section 6.3. The proofs can be found in Appendix A.3.

In Section 6.4 we saw that given an initial state to a unitary event structure, we have a probabilistic event structure. Later in this section, we shall make use of this fact to show the equivalence between the operational semantics and the denotational semantics when initial states are taken into consideration.

We start by not considering initial states. In this case, the formulation of the following lemmas and theorems is equal to the ones in Section 4.4. We thus suggest that the reader recalls what was said about these lemmas and theorems. It is useful to recall that $\bot = [\![\checkmark]\!]$, and moreover that $[\![while\ \mathtt{M}(n, C)]\!] = \mathtt{M}(n, \checkmark, [\![C\,;\, while\ \mathtt{M}(n, C)]\!]) = \mathsf{P}_0^n$ or $\mathsf{P}_1^n\,;\, [\![C\,;\, while\ \mathtt{M}(n, C)]\!]$.

The formulation of small-soundness (Lemma 6.5.1), soundness (Theorem 6.5.2), small-adequacy (Lemma 6.5.3), and adequacy (Theorem 6.5.4) are depicted below. Note that these formulations do not take into consideration an initial state. We postpone that for later in this section.

Lemma 6.5.1 (Small-soundness)**.** If $C \xrightarrow{l} C'$ then $[\![C']\!] \equiv [\![C]\!] \backslash l$.

Theorem 6.5.2 (Soundness)**.** If $C \xrightarrow{\omega}\!\!\!\!\rightarrow C'$ then $\exists x \in \mathcal{C}([\![C]\!])$ such that $\emptyset \xrightarrow{\omega}\!\!\!\!\subset x$.

Lemma 6.5.3 (Small-adequacy)**.** Let $l \in \mathcal{I}([\![C]\!])$. Then $\exists C' \in (C \cup \{\checkmark\})$ s.t. $C \xrightarrow{l} C'$ and $[\![C]\!] \backslash l \equiv [\![C']\!]$.

Theorem 6.5.4 (Adequacy)**.** If $\emptyset \neq x \in \mathcal{C}([\![C]\!])$ s.t. $\emptyset \xrightarrow{\omega}\!\!\!\!\subset x$ then $\exists C'$ s.t. $C \xrightarrow{\omega}\!\!\!\!\rightarrow C'$.

Throughout Part I, after presenting the main results, we also provided an example to illustrate the results in action. For this case, we will postpone the example for later, when we reformulate Theorem 6.5.2 and Theorem 6.5.4 to account for an initial state.

Now let us focus on establishing a relation between the semantics by considering an initial state, following the spirit of Section 6.4. Knowing that any unitary event structure with an initial state is a probabilistic event structure, we can calculate the probability associated with quantum computations with an initial state. In other words, given a covering chain linked to a configuration and an initial state, we can calculate the probability of that configuration. From Section 6.5, we know that covering chains correspond to words in the operational semantics. We then should be able to extract a probability from a word that acts on an initial state. Recall that the big-step we use for the language in Section 6.1 is the same as in Section 4.1. Recall also that the intuition behind it is to stamp a tape of size n with the n actions executed during a computation. If we associate an initial state with the command being evaluated, the evolution of the state will correspond to the application of the word to the initial state. We define a state to be a partial density operator, *i.e.* a density operator whose trace is less than or equal to one, and denote the set of partial density operators as $\mathcal{D}_{\leq 1}(\mathcal{H})$. We now define how a word is applied to a state:

Definition 6.5.5. Let ω be a word and ρ a partial density operator. Define $\omega(\rho)$ inductively as follows:

$$\omega(\rho) = \begin{cases} a\rho a^\dagger & \text{if } \omega = a \\ \omega'(a\rho a^\dagger) & \text{if } \omega = a : \omega' \end{cases}$$

Note that when applying a word ω to a state ρ, the first action to be applied on ρ is the head of ω.

Following Lemma 6.4.10, we can define a new denotational semantics for quantum event structures that considers initial states.

Definition 6.5.6. We define $[\![-]\!]_ : Com \times \mathcal{D}_{\leq 1}(\mathbb{C}^{2^N}) \to (\mathsf{U},\, \rho,\, v)$, where U is a unitary event structure, as follows:

$$[\![C]\!]_\rho = ([\![C]\!],\, \rho,\, v)$$

At this point, we can establish an equivalence between the semantics considering an initial state. For that we take advantage of the results obtained without an initial state. First, we reformulate Theorem 6.5.2 and Theorem 6.5.4 to consider initial states, and then we explain how we leverage the results obtained in Section 6.5.

Theorem 6.5.7 (Soundness)**.** Let ρ be an initial state. If $C \overset{\omega}{\twoheadrightarrow} C'$ then $\exists x \in \mathcal{C}([\![C]\!]_\rho)$ such that $\emptyset \overset{\omega}{\multimap} x$ and $v(x) = Tr(\omega(\rho))$.

Theorem 6.5.8 (Adequacy)**.** Let ρ be an initial state. If $(x \neq \emptyset) \in \mathcal{C}([\![C]\!]_\rho)$ s.t. $\emptyset \overset{\omega}{\multimap} x$ then $\exists C'$ s.t. $C \overset{\omega}{\twoheadrightarrow} C'$ and $v(x) = Tr(\omega(\rho))$.

From Theorem 6.5.2 and Theorem 6.5.4, we know that a word ω in the operational semantics corresponds to a covering chain $\emptyset \overset{\omega}{\subset} x$ in the denotational semantics and vice-versa. Hence, given an initial state ρ, applying A_x to ρ is the same as applying ω to ρ, because the operations to be applied are the same. Therefore, verifying that $v(x) = Tr(\omega(\rho))$ comes for free.

Now let us delve into an example that takes into consideration an initial state.

Example 6.5.9. Recall the command used in Example 6.1.1 and Example 6.2.9

$$H(1)\,;\,\mathtt{M}(1, Z(1), Id(2)\,||\,X(1))$$

Recall that from the former, the following words lead to a terminal command: $H(1)P_0(1)Z(1)$, $H(1)P_1(1)I(2)X(1)$, and $H(1)P_1(1)X(1)I(2)$. From the latter we have the following maximal configurations:

$$\{a_{H(1)}, \tau 0_{P_0(1)}, b_{Z(1)}\} \text{ and } \{a_{H(1)}, \tau 1_{P_1(1)}, c_{I(2)}, d_{X(1)}\}$$

Let us consider $\rho = |00\rangle\langle 00|$ as the initial state. By applying each maximal word to ρ we obtain the state $0.5|00\rangle\langle 00|$. Since each word corresponds to a covering chain, and consequently to a configuration, we have that for each maximal configuration the operator A_x, when applied to ρ, also gives $0.5|00\rangle\langle 00|$. From here, we can conclude that the probabilities match.

We end this section with an example that shows how both semantics relate to each other in the quantum setting, with and without an initial state.

Example 6.5.10. Recall the program used in Example 6.3.1, $while\ \mathtt{M}(n, H(n))$. From that example, we know that the words formed by the big-step semantics consist of a (possibly empty) sequence of $H(n)P_1(n)$, ending with $P_0(n)$. From Example 6.3.15, we observed that maximal configurations contain zero or more occurrences of events associated with $H(n)$ and $P_1(n)$, and exactly one event associated with $P_0(n)$. The previous examples indicate that the words leading to terminal computations correspond to the maximal configurations.

Now consider the initial state $\rho = |1\rangle\langle 1|$. According to the big-step semantics, we know that $P_0(n)(H(n)P_1(n))^m$, with $m \geq 0$, is a word that leads to a terminal computation. It is then straightforward to see that the probability of reaching a terminal computation depends on m. If $m = 0$, the probability is $\mathrm{Tr}(P_0(n)|1\rangle\langle 1|) = 0$. On the other hand, if $m > 0$, that probability is $\frac{1}{2^m}$. As m increases, the probability of reaching a terminal computation decreases. The probabilistic behavior exhibited in these two cases is reflected in the probabilities calculated through maximal configurations. For instance, when the maximal configuration consists solely of the event associated with $P_0(n)$, its probability

is 0, because we are projecting the state $|1\rangle\langle 1|$ onto $|0\rangle\langle 0|$, which are orthogonal. Conversely, when the maximal configuration contains additional events beyond the one associated with $P_0(n)$, its probability is $\frac{1}{2^m}$.

Chapter 7

Closing the stage

7.1 Related Work

Event structures are a model for concurrency. Most of the work around them either proposes new definitions of event structures, to overcome limitations of the original formulations, or to capture additional computational effects, such as probabilistic or quantum behavior. However, when event structures are employed as denotational semantics for programming languages, the languages considered are usually process-algebraic, often relying on communication channels. In other words, most existing applications of event structures as semantic models are tailored to process algebras rather than to imperative shared-variable languages.

In the classical setting, event structures have different forms: elementary event structures [118, Definition 1.1], prime event structures [118, Definition 2.1] (used in our work, and referred to simply as "event structures"), stable event structures [118, Definition 3.1], family event structures [115, Definition 1.1], bundle event structures [61, Definition 2.10], among others (see [61, Section 2.2] for a comparison among several of these forms and for references to additional variants). These different notions were later used to provide denotational semantics for process algebras. For instance, stable and family event structures were used to give semantics to CCS [118, 115], while bundle event structures were employed for CCSP, a process algebra combining operators from CCS and CSP [61]. Since CCS and CCSP communicate through channels, the definition of parallel composition must include a synchronization mechanism, which complicates its semantics compared to a shared-variable setting, where parallel processes can simply be placed side by side (recall Definition 4.2.3). Furthermore, CCS and CCSP rely on action prefixing instead of sequential composition. Action prefixing can be seen as a special case of sequential composition, but is simpler to define since it fixes a single initial action.

A closer approach to ours is found in Castellan's work [23], where event structures are used to interpret a simple imperative concurrent language within the context of weak memory models. His objective

was to capture execution paths that may be introduced by compilers during code optimization but are missed by interleaving semantics. His definitions of parallel and sum operators (the latter corresponding to our non-deterministic composition of event structures) are similar to ours. Castellan also introduced a concatenation operator to represent sequential composition, but in a relaxed way [23, Definition 5]: given two event structures E_1 and E_2, it is not required that all events of E_2 are causally dependent on all events of E_1. More concretely, the causal dependencies from E_1 to E_2 are those specified by a relation R on labels (Castellan considers a labeling function that assigns a label to each event). Hence, events of E_2 whose labels are not related via R to any label of E_1 are concurrent with events of E_1. This allows such events of E_2 to occur before some events of E_1, capturing the behaviors allowed by weak memory models.

In the probabilistic setting, probabilistic event structures were first introduced by Katoen [61, Definition 9.2] as an extension of bundle event structures, where probabilities were assigned directly to events. Later, Varacca [110, Definition 3.3] proposed an alternative definition based on a particular subclass of prime event structures known as confusion-free event structures (informally, they are event structures in which the minimal conflict relation is transitive and the enabling of events does not create new conflicts). Varacca then defined a global valuation over finite configurations, thereby assigning probabilities to configurations. Since the set of configurations ordered by inclusion forms a domain [117, Theorem 1.1.16], this valuation extends to a continuous valuation, linking probabilistic event structures to domain theory. Following Varacca's approach, Winskel [119, Definition 1] proposed a simplified definition of probabilistic event structures that drops the confusion-free requirement and additional constraints of the global valuation by a drop condition (which, in Definition 3.3.1, we replaced by the sum condition). However, Winskel's definition cannot directly furnish a language combining probabilistic and non-deterministic constructs with a denotational semantics, as discussed in [26]. To address this, Marc de Visme [26] refined Winskel's approach by dividing the conflict relation into internal and external conflicts, allowing him to provide PCCS (a probabilistic variant of CCS with both probabilistic and non-deterministic constructs) with a denotational semantics.

In the quantum setting, Winskel [119, Definition 2] extended prime event structures by associating quantum operators to events. So far, quantum event structures have primarily served as a backbone for game semantics [25] and, more recently, as a basis for defining quantum Petri nets [58].

7.2 Future Work

We identify three possible directions in which the work developed in Part I can be extended.

1. Introducing state-based operational semantics

   The operational semantics developed for the languages considered in Part I does not take into account program states. However, imperative languages are characterized by their ability to modify states rather than merely record actions on a tape. Hence, it would be interesting to modify the operational semantics, both small-step and big-step, to explicitly consider states, in the spirit of Reynolds [92, Sections 6 and 8] and Section 9.1.

   For the language considered in the non-deterministic case (Section 4.1), the resulting state-based operational semantics would resemble that of Section 8.1, where states were added to the core language. For the languages in the probabilistic (Section 5.1) and quantum settings (6.1), introducing states into the operational semantics would yield semantics similar to that in Section 8.2.

   By considering a state-based operational semantics, we would have two forms of operational semantics for the languages in Part I: (i) the operational semantics considered in Part I, where transitions are labeled by triggering actions, and (ii) a state-based approach. It would then be interesting to establish a connection between them. Doing so would create a bridge linking the denotational semantics based on event structures, with an operational semantics that accounts for states. This would strengthen the relationship between the work developed in Part I and Part II.

2. Using mixed event structures [26]

   Following the approach used in Part I, we could use mixed event structures [26] to interpret a language that combines probabilistic and non-deterministic commands, since Winskel's probabilistic event structures [119, Definition 1] are unable to do that. It would also be interesting to repeat the state-based extension discussed above, but now within the framework of mixed event structures. This leads to the natural question: "Is it possible to combine mixed event structures and unitary event structures to define a generic concurrent language that supports quantum commands, as well as probabilistic and non-deterministic ones?". If the answer is positive, then two further questions follow:

   (i) "Do mixed event structures, in the quantum setting, together with an initial state correspond to mixed event structures in the probabilistic realm (Proposition 6.4.10)?"

   (ii) "What is the relation between this new denotational model and the resumptions model used in [80]?"

3. Developing a software tool for event structures

A more practical direction, concerning the work in Part I, would be to develop a tool similar to the one in Part II possibly including a graphical interface. One feature of such a tool could be the ability to draw the event structure corresponding to a given program, allowing users to inspect the configurations associated with it. Another feature could allow users to construct event structures manually and identify which program corresponds to each. Note, however, that this task would not be trivial, since there exist event structures that do not correspond to any command (for instance, an event structure with three events a, b, and c where $a \leq c$, a co b, and $b \# c$).

7.3 Conclusion

In this part, we considered a stateless operational semantics for the core language, *i.e.* an operational semantics that does not take states into account. Instead, the operational semantics behaves as if it were stamping onto a tape the actions performed during a computation. Since states are not considered, event structures naturally arise as a suitable semantic model for the languages presented in Part I. Recall that event structures are defined in three different realms: non-deterministic, probabilistic, and quantum. For each realm, a specific command was introduced, and the operational semantics was suitably adapted. Each resulting language was then equipped with a denotational semantics based on the corresponding notion of event structures. We subsequently proved soundness and adequacy for each case.

Focusing on the quantum realm, we restricted the definition of quantum event structures [119], obtaining what we call unitary event structures. This restriction enabled us to extend a result originally established by Winskel [119, Theorem 3]. Furthermore, it opened the possibility of introducing states into the operational semantics, where an initial state evolves according to the instructions recorded on the tape. Consequently, we were also able to establish soundness and adequacy results for the state-based setting as well.

Part II

A concurrent (quantum) language and its interpreter

Prologue

The development of tools, grounded in theoretical frameworks, to support reasoning about complex programs is of great importance. First, such tools save time by avoiding manual calculations. Second, they facilitate program testing and bug detection. For these reasons, in this part we extend a quantum concurrent language [80] by adding a synchronization primitive. We then develop a Haskell interpreter for the extended language.

We begin by introducing the notion of state into the core language in Section 3.5, obtaining a truly imperative concurrent language. The operational semantics is then updated, both in small-step and big-step forms. Afterwards, we add non-deterministic choice and probabilistic choice to the core language. These additions, together with the inclusion of if-then-else statements and while loops, give us a generic probabilistic concurrent language [80]. This language has an operational semantics defined using schedulers and a denotational semantics based on resumptions. Note that, unlike in Part I, where a specific command is added to the core language depending on the event structure under study (*e.g.* probabilistic choice is added when dealing with probabilistic event structures), in Part II we include all commands in a single language. This is possible because the resumption model used to give a denotational semantics to the generic probabilistic concurrent language [80] is expressive enough to capture these distinct constructs within a unified framework.

The core language extended with states resembles the languages used by Park [88] and Brookes [21], which are equipped with a denotational semantics based on transition traces. Intuitively, transition traces are sequences of state pairs capable of capturing interference in computations. As preliminary work, we furnished the generic probabilistic concurrent language with a denotational semantics based on transition traces. To do so, transition traces were adapted to the probabilistic setting by using the monad $\mathrm{P}_{\mathrm{TM}}\mathrm{V}$ [109, 59, 73], a monad of convex finitely generated sets. However, our approach has limitations: it prevents instantiating the generic language to the quantum setting and complicates the establishment of a correspondence between operational and denotational semantics.

The generic probabilistic concurrent language also admits an instantiation to the quantum setting [80]. To avoid building two separate interpreters (one for the generic probabilistic concurrent language and another for its quantum instantiation), we combine both into a single language. This unified language enables reasoning about hybrid quantum-classical computations with concurrency and probabilistic behavior. Furthermore, the language is extended with a synchronization primitive. The motivation is that, in a concurrent setting, access to variables that are considered critical must be handled carefully, and this

principle should also apply in the quantum setting. Having established the language, and motivated by its potential usefulness as an assistant to reason about programs, we developed a corresponding interpreter.

To test the tool's limits under specific scenarios, we conducted two stress tests. The first concerns state explosion and examines the maximal number of iterations that a random walk can execute. Since the tool offers two forms of output, valuations and histograms, we compared how the executions of the random walk differ across these two forms of output. The second test focuses on qubit scaling. Its goal is to verify how many qubits can simultaneously be in superposition before the computation starts to become infeasible, *i.e.* before the runtime becomes intractable as the number of qubits increases. For this experiment, we compared the results produced by our tool with those obtained using the QIO monad [6], a Haskell library for quantum computation. We also implemented a wide range of examples, covering all effects supported by the language: concurrency, probability, concurrency mixed with probability, quantum computation, and quantum mixed with concurrency.

The main examples used concern the Dining Philosophers problem in three different settings: classical, probabilistic, and quantum. In all versions, the synchronization primitive ensures the correct behavior of the algorithm. In the quantum case, the interpreter helped us show that [5, Theorem 3] does not hold. In sum, that theorem claimed the existence of an efficient deterministic solution to the quantum Dining Philosophers problem. However, calculations performed with the tool demonstrate that the theorem does not hold. This reopens the search for an efficient deterministic solution to the quantum Dining Philosophers problem.

Chapter 8

Setting the stage

8.1 Adding states to the core language

Recall the core language introduced in Section 3.5, which we used when studying event structures in Part I. In that context, the notion of state appeared only in the quantum case, where the goal was to show how a unitary event structure can be viewed as a probabilistic event structure. In this part of the thesis, we explicitly incorporate the notion of state into the operational semantics, following the approach of [92].

We assume a set of states S and a set of atomic actions Act. Each atomic action a is associated with a function $[\![a]\!] : S \to S$, which, given a state s, returns the state obtained by applying a to s. We write this result as $[\![a]\!]s$.

Recall that commands (Com) are built according to the following BNF grammar:

$$C ::= skip \mid a \in Act \mid C\,;\, C \mid C \,||\, C$$

To equip the language with an operational semantics, we resort to the notion of a configuration. Intuitively, a configuration is a pair formed by a command C and a state $s \in S$, *i.e.* $\langle C,\ s\rangle$. We then define the small-step operational semantics as a relation $\longrightarrow \subseteq (C \times S) \times (S + (C \times S))$ obeying the rules in Figure 70 (where we omit the symmetric rule for the parallel operator).

Intuitively, a transition in the small-step semantics starts from a configuration and either produces a final state (*i.e.* the computation finishes), or another configuration (*i.e.* the computation continues).

In Figure 70, the rules for $skip$ and atomic actions are straightforward: $skip$ leaves the state unchanged, while an atomic action transforms the state via $[\![a]\!]$. For sequential composition, if $\langle C_1,\ s\rangle$ evaluates to a final state s', then C_2 begins execution in s'. If instead $\langle C_1,\ s\rangle$ yields another configuration $\langle C_1',\ s'\rangle$, then the remainder of the computation is $\langle C_1'\,;\, C_2,\ s'\rangle$. A similar reasoning applies to parallel composition.

Note that the rationale behind the rules in Figure 70 and Figure 16 is pretty similar: when a transition

$$\langle skip,\ s\rangle \longrightarrow s \qquad \langle a,\ s\rangle \longrightarrow [\![a]\!]s$$

$$\frac{\langle C_1,\ s\rangle \longrightarrow s'}{\langle C_1\,;\,C_2,\ s\rangle \longrightarrow \langle C_2,\ s'\rangle} \qquad \frac{\langle C_1,\ s\rangle \longrightarrow \langle C_1',\ s'\rangle}{\langle C_1\,;\,C_2,\ s\rangle \longrightarrow \langle C_1'\,;\,C_2,\ s'\rangle}$$

$$\frac{\langle C_1,\ s\rangle \longrightarrow s'}{\langle C_1\,||\,C_2,\ s\rangle \longrightarrow \langle C_2,\ s'\rangle} \qquad \frac{\langle C_1,\ s\rangle \longrightarrow \langle C_1',\ s'\rangle}{\langle C_1\,||\,C_2,\ s\rangle \longrightarrow \langle C_1'\,||\,C_2,\ s'\rangle}$$

Figure 70: Rules for the simple concurrent imperative language

reaches the end of a computation, the rules in Figure 70 return a state, while the rules in Figure 16 use a special symbol to denote termination. In both sequential and parallel composition, whenever the computation of C_1 finishes we execute what is left, which is C_2, or if the computation does not finish, we compose what is left to compute, either sequentially or in parallel, with C_2. The key difference is that Figure 16 uses labels to mark the action that triggered the transition, while in Figure 70 the action is directly applied to the state (with $skip$ leaving the state unchanged).

The core language with states is also furnished with a big-step semantics, which is the transitive closure of the small-step semantics. Specifically, the big-step semantics is denoted by $\twoheadrightarrow$ and is defined via the rules in Figure 71:

$$\frac{\langle C,\ s\rangle \longrightarrow s'}{\langle C,\ s\rangle \twoheadrightarrow s'} \qquad \frac{\langle C,\ s\rangle \longrightarrow \langle C',\ s'\rangle \qquad \langle C',\ s'\rangle \twoheadrightarrow s''}{\langle C,\ s\rangle \twoheadrightarrow s''}$$

Figure 71: Rules of the big-step semantics

As previously stated, when dealing with concurrency we need to take into consideration the possibility of the execution of a command being interfered by the execution of another one. Let us illustrate this with an example.

Example 8.1.1. Recall Example 3.5.1 where we have two commands $C_1 = a$, which performs only the atomic action a, and $C_2 = b\,;\,c$, which first executes b and then c. Given s as initial state for both commands, it is straightforward to see that after applying C_1 we obtain as final state $[\![a]\!]s$. On the other hand, after executing C_2 the final state is $[\![c]\!][\![b]\!]s$, *i.e.* we apply $[\![c]\!]$ to the state $[\![b]\!]s$.

Now let us compose both commands using the parallel operator, *i.e.* $C_1\,||\,C_2 \equiv a\,||\,(b\,;\,c)$ and draw

the execution tree of this command when feeding s as the initial state.

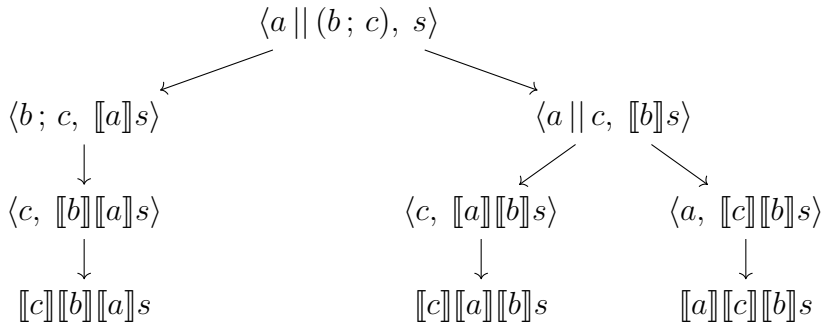


Figure 72: Execution tree of $\langle a \,||\, (b\,;\, c),\ s\rangle$

From Figure 72, we note that the final states obtained from executing $\langle a \,||\, (b\,;\, c),\ s\rangle$ are $[\![c]\!][\![b]\!][\![a]\!]s$, $[\![c]\!][\![a]\!][\![b]\!]s$, and $[\![a]\!][\![c]\!][\![b]\!]s$. To obtain the final states $[\![c]\!][\![b]\!][\![a]\!]s$ and $[\![a]\!][\![c]\!][\![b]\!]s$ we first execute C_1 and then C_2 or the other way around, respectively. In each case, each command is completely executed before the other starts its execution. Hence, the execution of both commands was not interfered. On the other side, to obtain the final state $[\![c]\!][\![a]\!][\![b]\!]s$, the execution of C_2 is interrupted and interfered by the execution of C_1. Let us analyze how this state was achieved. We start executing the atomic action b from C_2, which results in $[\![b]\!]s$. Next, the computation of the remainder of C_2 is interrupted and interfered by the execution of C_1, giving us $[\![a]\!][\![b]\!]s$. At last, we resume the computation of what is left to execute from C_2. From this we obtain the final state $[\![c]\!][\![a]\!][\![b]\!]s$.

From Example 8.1.1, we see that concurrent computations are non-deterministic: executing the program $a \,||\, (b\,;\, c)$ with initial state s may yield any of the final states $[\![c]\!][\![b]\!][\![a]\!]s$, $[\![c]\!][\![a]\!][\![b]\!]s$, or $[\![a]\!][\![c]\!][\![b]\!]s$. In other words, running the program multiple times can lead to different outcomes.

The non-determinism introduced by the concurrent operator can be resolved using *schedulers*. Informally, a scheduler is an entity that, given the history of a computation, decides the next step. For instance, in Example 8.1.1, consider we use a scheduler to decide which action to execute at each transition. If the scheduler always selects an action from the left-hand command, then the final state is $[\![c]\!][\![b]\!][\![a]\!]s$. If the scheduler always selects an action from the right-hand command, then the final state is $[\![a]\!][\![c]\!][\![b]\!]s$. Finally, if the scheduler selects from the right-hand side first and then alternates to the left-hand side consecutively until the end of the computation, the final state returned is $[\![c]\!][\![a]\!][\![b]\!]s$. Under fairness[1], schedulers can often be omitted in big-step semantics, as we did in the big-step semantics of Figure 71. However, when concurrency is combined with probabilities [95, 101], schedulers become essential. This

[1] Intuitively, fairness means that no command is indefinitely denied its turn to execute.

is because such languages have two sources of non-determinism: one from concurrency and another from probabilistic branching. A scheduler is therefore needed to ensure these choices are resolved consistently.

For a detailed overview of this topic, the survey by Sokolova [100] is particularly valuable: it examines different ways concurrency and probability can be combined, and relates them via coalgebraic structures. The formal definition of a scheduler will be given later in Section 8.2.

8.1.1 Towards a denotational semantics with transition traces

The aim of this section is to present transition traces, a model for concurrent languages. We do so because, in Section 8.2.1, as an exercise, we extend transition traces with probabilities in order to provide a denotational semantics for a concurrent language with probabilistic choice.

Transition traces were introduced by David Park [88] to provide concurrent languages with a denotational semantics. Informally, a transition trace is a finite sequence of pairs of states, recording the changes caused by a computation that may be interrupted and/or interfered with. A generic transition trace has the form $(s_0, s'_1)(s_1, s'_2)...(s_{n-1}, s'_n)$, where each change of state from:

- s_i to s'_{i+1} is caused by a computation that was not interrupted nor interfered
- s'_i to s_i indicates that a computation was interrupted, with two possibilities
  - if $s'_i = s_i$ no interference occurred
  - if $s'_i \neq s_i$ the computation was interfered

When all steps in a transition trace satisfy $s'_i = s_i$ for $0 \leq i \leq n$, we say the transition trace is *interference-free*.

Transition traces can also have infinite length, but here we restrict ourselves to the finite case. For simplicity, we will often just write traces when referring to transition traces.

Let us furnish the language in Section 8.1 with a denotational semantics based on transition traces. For that, let S denote the set of states and $\Sigma = S \times S$ be the set of pairs of states. We then let Σ^+ be the set of non-empty sequences of pairs of states. At last, we consider $\mathsf{P}(\Sigma^+)$ to capture the non-determinism introduced by concurrency.

The interpretation of $skip$ and atomic actions is straightforward, so we omit it for now. Instead, we first focus on sequential composition. Given $T_1, T_2 \in \mathsf{P}(\Sigma^+)$, sequential composition is defined by concatenating traces:

$$T_1 \,;\, T_2 = \{\alpha\beta \mid \alpha \in T_1,\ \beta \in T_2\}$$

The parallel operator requires more careful treatment. Park's definition [88] distinguishes between fair and unfair computations, an issue that arises whenever infinite computations are possible. However, since the language in Section 8.1 does not allow infinite behavior, the definition simplifies to one similar to Brookes' transition traces [21, §6]. Intuitively, parallel composition corresponds to the interleaving of traces. To define interleaving, let ε denote the empty trace and $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. For $\alpha, \beta \in \Sigma^*$, define $\alpha \bowtie \beta$ inductively as:

$$\alpha \bowtie \varepsilon = \varepsilon \bowtie \alpha = \{\alpha\}$$
$$\sigma\alpha \bowtie \rho\beta = \{\sigma\gamma \mid \gamma \in (\alpha \bowtie \rho\beta)\} \cup \{\rho\gamma' \mid \gamma' \in (\sigma\alpha \bowtie \beta)\}$$

where $\sigma, \rho \in \Sigma$, $\alpha, \beta \in \Sigma^*$, and ε is the empty trace. Hence, for $T_1, T_2 \in \mathsf{P}(\Sigma^+)$ parallel composition is defined as follows:

$$T_1 \,||\, T_2 = \bigcup \{\alpha \bowtie \beta \mid \alpha \in T_1, \beta \in T_2\}$$

To better understand how the interleaving of traces work, consider the following example.

Example 8.1.2. Let $(s,s)(t,t)$ and (u,u) be traces. For simplicity, let us consider $t_1 = (s,s)$, $t_2 = (t,t)$, and $t_3 = (u,u)$. The interleaving, $t_1 t_2 \bowtie t_3$, is calculated as follows:

$$\begin{aligned} t_1 t_2 \bowtie t_3 &= \{t_1\gamma \mid \gamma \in (t_2 \bowtie t_3)\} \cup \{t_3\gamma' \mid \gamma' \in (t_1 t_2 \bowtie \varepsilon)\} \\ &= \{t_1\gamma \mid \gamma \in \{t_2 t_3, t_3 t_2\}\} \cup \{t_3\gamma' \mid \gamma' \in \{t_1 t_2\}\} \\ &= \{t_1 t_2 t_3, t_1 t_3 t_2\} \cup \{t_3 t_1 t_2\} \\ &= \{t_1 t_2 t_3, t_1 t_3 t_2, t_3 t_1 t_2\} \end{aligned}$$

We have all the necessary tools to interpret the language in Section 8.1 in terms of transition traces.

$$\mathcal{T}_P[\![skip]\!] = \{(s,s) \mid s \in S\}$$
$$\mathcal{T}_P[\![a]\!] = \{(s, [\![a]\!]s) \mid s \in S\}$$
$$\mathcal{T}_P[\![C_1 \,;\, C_2]\!] = \mathcal{T}_P[\![C_1]\!] \,;\, \mathcal{T}_P[\![C_2]\!]$$
$$\mathcal{T}_P[\![C_1 \,||\, C_2]\!] = \mathcal{T}_P[\![C_1]\!] \,||\, \mathcal{T}_P[\![C_2]\!]$$

Consider again the program from Example 8.1.1 and let us interpret it using transition traces.

Example 8.1.3. The program we have in hands is $a \,||\, (b \,;\, c)$. Let us first calculate the interpretation of the individual commands a and $b \,;\, c$, in order to ease the interpretation of the whole program.

$$\mathcal{T}_P[\![a]\!] = \{(s, [\![a]\!]s) \mid s \in S\}$$

$$\begin{aligned}\mathcal{T}_P[\![b\,;\,c]\!] &= \{\alpha\beta \mid \alpha \in \mathcal{T}_P[\![b]\!],\ \beta \in \mathcal{T}_P[\![c]\!]\} \\ &= \{(s, [\![b]\!]s)(s', [\![c]\!]s') \mid s, s' \in S\}\end{aligned}$$

Since the parallel composition of traces corresponds to its interleaving, we just need to interleave the traces from $\mathcal{T}_P[\![a]\!]$ with the traces from $\mathcal{T}_P[\![b\,;\,c]\!]$. The result is as follows:

$$\begin{aligned}\mathcal{T}_P[\![a \,||\, (b\,;\,c)]\!] &= \mathcal{T}_P[\![a]\!] \,||\, \mathcal{T}_P[\![b\,;\,c]\!] \\ &= \bigcup\{\alpha \bowtie \beta \mid \alpha \in \mathcal{T}_P[\![a]\!], \beta \in \mathcal{T}_P[\![b\,;\,c]\!]\} \\ &= \{(v, [\![a]\!]s)(v', [\![b]\!]v')(v'', [\![c]\!]v'') \mid v, v', v'' \in S\} \\ &\cup \{(u, [\![b]\!]u)(u', [\![a]\!]u')(u'', [\![c]\!]u'') \mid u, u', u'' \in S\} \\ &\cup \{(s, [\![b]\!]s)(s', [\![c]\!]s')(s'', [\![a]\!]s'') \mid s, s', s'' \in S\}\end{aligned}$$

Note that the initial states appearing in each pair of states are all different. By consistently instantiating them we can obtain interference-free traces. For example, if we set $v' = [\![a]\!]v$ and $v'' = [\![b]\!][\![a]\!]v$ from $(v, [\![a]\!]v)(v', [\![b]\!]v')(v'', [\![c]\!]v'')$ we obtain the interference-free trace

$$(v, [\![a]\!]v)([\![a]\!]v, [\![b]\!][\![a]\!]v)([\![b]\!][\![a]\!]v, [\![c]\!][\![b]\!][\![a]\!]v)$$

Park's transition traces yield a compositional denotational semantics, but they are too strict regarding program equivalence. For instance $\mathcal{T}_P[\![skip]\!] = \{(s, s) \mid s \in S\}$ and $\mathcal{T}_P[\![skip\,;\,skip]\!] = \{(s, s)(s', s') \mid s, s' \in S\}$. Even though both commands intuitively behave the same (returning the input state unchanged), they produce different traces and are thus distinguishable in this model.

In order to relax program equivalence, Brookes [21] considered the notions of *stuttering* and *mumbling*. Stuttering expresses idling in computations and allows interference, by inserting pairs (s, s) into traces. Mumbling allows skipping intermediate states, *i.e.* from $(s, s_1)(s_1, s_2)$ we may derive (s, s_2). Thus, in Brookes approach, traces live in $\mathsf{P}^{\bullet\dagger}(\Sigma^+)$, *i.e.* traces closed under stuttering and mumbling.

For the sequential and parallel composition of traces we need to consider traces that are closed under stuttering and mumbling and then close the resultant set by stuttering and mumbling. The resultant set is calculated essentially in the same way as previously shown. Concretely, given $T_1, T_2 \in \mathsf{P}^{\bullet\dagger}(\Sigma^+)$, sequential composition and parallel composition are defined as follows:

$$T_1\,;\,T_2 = \{\alpha\beta \mid \alpha \in T_1,\ \beta \in T_2\}^{\bullet\dagger} \qquad T_1 \,||\, T_2 = \left(\bigcup\{\alpha \bowtie \beta \mid \alpha \in T_1, \beta \in T_2\}\right)^{\bullet\dagger}$$

The interpretation of the language in Section 8.1, in terms of transition traces closed by stuttering and mumbling, is as follows:

$$\mathcal{T}_B[\![skip]\!] = \{(s, s) \mid s \in S\}^{\bullet\dagger}$$

$$\mathcal{T}_B[\![a]\!] = \{(s, [\![a]\!]s) \mid s \in S\}^{\dagger}$$
$$\mathcal{T}_B[\![C_1 \,;\, C_2]\!] = \mathcal{T}_B[\![C_1]\!] \,;\, \mathcal{T}_B[\![C_2]\!]$$
$$\mathcal{T}_B[\![C_1 \,||\, C_2]\!] = \mathcal{T}_B[\![C_1]\!] \,||\, \mathcal{T}_B[\![C_2]\!]$$

Reynolds [92, Section 8.8] states that there is a relation between Park's traces and Brookes' traces. For this, given a set of traces $T \in \mathsf{P}(\Sigma^+)$, Reynolds constructs the set

$$T^{\dagger} = \{\hat{\alpha} \mid \exists \alpha \in T \text{ s.t. } \hat{\alpha} \text{ is a stuttering-mumbling variant of } \alpha\}$$

i.e. the set of traces obtained by applying stuttering and mumbling to a trace $\alpha \in T$. We use † in $T^{\dagger}$, to keep the same notation as Reynolds. Note that the dagger used for the transpose conjugate and the dagger used to denote the stuttering-mumbling closure are used in distinct contexts, so their coexistence should not cause confusion here. Using this construction, Reynolds stated that $\mathcal{T}_B[\![C]\!] = \mathcal{T}_P[\![C]\!]^{\dagger}$. He also explained how one can derive the semantic equations for $\mathcal{T}_B[\![-]\!]$ from those of $\mathcal{T}_P[\![-]\!]$. The reader can find more details about these topics in [92, Section 8].

To end this section, we recover the program in Example 8.1.1 and interpret it using the denotational semantics based on Brookes' approach.

Example 8.1.4. Consider again the program $a \,||\, (b \,;\, c)$. The interpretation of the commands are essentially the same as the ones in Example 8.1.3. The only difference is that we now take the stuttering and mumbling closure of the sets. Concretely:

$$\mathcal{T}_B[\![a]\!] = \{(s, [\![a]\!]s) \mid v \in S\}^{\dagger}$$

$$\mathcal{T}_B[\![b \,;\, c]\!] = \{(s, [\![b]\!]s)(s', [\![c]\!]s') \mid s, s' \in S\}^{\dagger}$$

$$\begin{aligned}\mathcal{T}_B[\![a \,||\, (b \,;\, c)]\!] = (&\{(v, [\![a]\!]s)(v', [\![b]\!]v')(v'', [\![c]\!]v'') \mid v, v', v'' \in S\} \\ &\cup \{(u, [\![b]\!]u)(u', [\![a]\!]u')(u'', [\![c]\!]u'') \mid u, u', u'' \in S\} \\ &\cup \{(s, [\![b]\!]s)(s', [\![c]\!]s')(s'', [\![a]\!]s'') \mid s, s', s'' \in S\})^{\dagger}\end{aligned}$$

8.2 Equipping the core language with non-deterministic, probabilistic, and control flow commands

By adding a probabilistic command and a non-deterministic command to the language introduced in Section 8.1, we combine the languages studied in Section 4.1 and Section 5.1. By taking advantage of

dealing with states, which allows us to evaluate Boolean expressions on them, we may introduce if-then-else commands and while loops. Altogether the resulting language is the one defined in [80]:

$$C ::= skip \mid a \in Act \mid C +_p C \mid C \,;\, C \mid C \text{ or } C \mid C \,||\, C \mid$$
$$\texttt{if } b \texttt{ then } \{C\} \texttt{ else } \{C\} \mid \texttt{while } b \texttt{ do } \{C\}$$

where $p \in [0,1] \cap \mathbb{Q}$.

As in Section 8.1, the operational semantics is given in terms of configurations. We consider the set of valuations over a set X, $\mathsf{V}(X) = \{\psi : X \to \mathbb{R}^+ \mid supp(\psi) \text{ is finite}\}$, where $supp(\psi) = \{x \in X | \psi(x) > 0\}$ is the support of ψ, and $\mathbb{R}^+ = \{r \in \mathbb{R} \mid r \geq 0\}$ is the set of non-negative real numbers. The set $\mathsf{V}_1(X)$ denotes the valuations whose weights sum to one and the set $\mathsf{V}_{\leq 1}(X)$ denotes the valuations whose weights are less than or equal to one. It is assumed the existence of a function $[\![a]\!] : S \to \mathsf{V}_1(S)$ that given a state s, applies the atomic action a to s. We also consider a set of Boolean expressions $BExp$ (ranged over by b) and a function $[\![b]\!] : S \to \{tt, ff\}$ that evaluates a Boolean expression in a given state s. At last, the language is equipped with a small-step operational semantics defined as the relation $\longrightarrow \subseteq (Com \times S) \times \mathsf{V}_1(S + (Com \times S))$ obeying the rules in Figure 73.

The rules in Figure 73 are a straightforward extension of the rules in [92, Chapters 6 and 8] to the probabilistic realm. Hence, we omit their explanation. Furthermore, the rules in Figure 73 and Figure 43 for the commands common to both languages share some similarities and some differences. The similarities appear in the rules for the commands $skip$ and atomic actions: in Figure 73 we return a final state, while in Figure 43 we return ✓, indicating that the computation has finished. The differences originate from the rule for the probabilistic choice, which in turn affects the rules for sequential and parallel commands. In Figure 73, the probabilistic choice returns the results of evaluating the commands that compose the probabilistic choice, multiplied by the appropriate constant, *i.e.* $\langle C_1 +_p C_2,\, v\rangle \longrightarrow p \cdot \mu + (1-p) \cdot \nu$, if $\langle C_1,\, v\rangle \to \mu$ and $\langle C_2,\, v\rangle \to \nu$. The way the probabilistic choice is defined allows us to say that a generic transition from the rules in Figure 73 is of the form: $\langle C,\, v\rangle \longrightarrow \sum_i p_i \cdot v_i + \sum_j p_j \cdot \langle C_j,\, v_j\rangle$, where the elements indexed by i corresponds to computations that have terminated, while the elements indexed by j represent computations that have not yet terminated. The generic transition is then used to define the rules for sequential and parallel commands. On the other hand, in Figure 43, the rule for the probabilistic choice does not evaluate the commands that constitute the probabilistic choice. Instead, the rule performs a "silent" transition, labeled by τ, in which the different commands in the probabilistic choice are split and weighted by the appropriate constants, *i.e.* $C_1 +_p C_2 \longrightarrow p \cdot (\tau, C_1) + (1-p) \cdot (\tau, C_2)$. The fact that the probabilistic transition in Figure 43 is labeled by τ allows us to distinguish two types of transitions: those that arise from the probabilistic choice, and those that

$$\langle skip,\ s\rangle \longrightarrow 1 \cdot s \qquad \langle a,\ s\rangle \longrightarrow [\![a]\!]s \qquad \frac{\langle C_1,\ s\rangle \longrightarrow \mu \qquad \langle C_2,\ s\rangle \longrightarrow \nu}{\langle C_1\ +_p\ C_2,\ s\rangle \longrightarrow p \cdot \mu + (1-p) \cdot \nu}$$

$$\frac{\langle C_1,\ s\rangle \longrightarrow \sum_i p_i \cdot s_i + \sum_j p_j \cdot \langle C_j,\ s_j\rangle}{\langle C_1\,;\ C_2,\ s\rangle \longrightarrow \sum_i p_i \cdot \langle C_2,\ s_i\rangle + \sum_j p_j \cdot \langle C_j\,;\ C_2,\ s_j\rangle}$$

$$\frac{\langle C_1,\ s\rangle \longrightarrow \sum_i p_i \cdot s_i + \sum_j p_j \cdot \langle C_j,\ s_j\rangle}{\langle C_1 \,\mathsf{or}\, C_2,\ s\rangle \longrightarrow \sum_i p_i \cdot s_i + \sum_j p_j \cdot \langle C_j,\ s_j\rangle}$$

$$\frac{\langle C_2,\ s\rangle \longrightarrow \sum_i p_i \cdot s_i + \sum_j p_j \cdot \langle C_j,\ s_j\rangle}{\langle C_1 \,\mathsf{or}\, C_2,\ s\rangle \longrightarrow \sum_i p_i \cdot s_i + \sum_j p_j \cdot \langle C_j,\ s_j\rangle}$$

$$\frac{\langle C_1,\ s\rangle \longrightarrow \sum_i p_i \cdot s_i + \sum_j p_j \cdot \langle C_j,\ s_j\rangle}{\langle C_1 \,||\, C_2,\ s\rangle \longrightarrow \sum_i p_i \cdot \langle C_2,\ s_i\rangle + \sum_j p_j \cdot \langle C_j \,||\, C_2,\ s_j\rangle}$$

$$\frac{\langle C_2,\ s\rangle \longrightarrow \sum_i p_i \cdot s_i + \sum_j p_j \cdot \langle C_j,\ s_j\rangle}{\langle C_1 \,||\, C_2,\ s\rangle \longrightarrow \sum_i p_i \cdot \langle C_1,\ s_i\rangle + \sum_j p_j \cdot \langle C_1 \,||\, C_j,\ s_j\rangle}$$

$$\frac{[\![b]\!]s = tt}{\langle \texttt{if}\ b\ \texttt{then}\ \{C_1\}\ \texttt{else}\ \{C_2\},\ s\rangle \longrightarrow 1 \cdot \langle C_1,\ s\rangle}$$

$$\frac{[\![b]\!]s = ff}{\langle \texttt{if}\ b\ \texttt{then}\ \{C_1\}\ \texttt{else}\ \{C_2\},\ s\rangle \longrightarrow 1 \cdot \langle C_2,\ s\rangle}$$

$$\frac{[\![b]\!]s = tt}{\langle \texttt{while}\ b\ \texttt{do}\ \{C\},\ s\rangle \longrightarrow 1 \cdot \langle C\,;\ \texttt{while}\ b\ \texttt{do}\ \{C\},\ s\rangle}$$

$$\frac{[\![b]\!]s = ff}{\langle \texttt{while}\ b\ \texttt{do}\ \{C\},\ s\rangle \longrightarrow 1 \cdot s}$$

Figure 73: Rules of the operational semantics

do not. Consequently, this distinction is reflected in the sequential and concurrent commands, in which the rules vary depending on whether the triggering action is an atomic action or whether the transition is caused by τ.

The language is also equipped with a big-step semantics. The big-step semantics resorts to a scheduler, more precisely a probabilistic scheduler, which solves the non-determinism present in the evaluation of a program by choosing a branch of the computation to be executed. This type of scheduler was explored in [109], and as far as our knowledge goes its origins go back to the work of Segala [95]. Now, the output of the probabilistic scheduler will be a probabilistic distribution. Furthermore, the scheduler has access to the history of the execution and to the configuration in analysis. Let $\langle C,\ s\rangle \longrightarrow$ denote the set $\{\mu \mid \langle C,\ s\rangle \longrightarrow \mu\}$, *i.e.* the set composed of the possible transitions from $\langle C,\ s\rangle$. Formally [80] (to simplify the notation we write $h\langle C,\ s\rangle$ instead of $(h, \langle C,\ s\rangle)$, where h represents the history):

Definition 8.2.1. A probabilistic scheduler $\mathcal{S}$ is a partial function,

$$\mathcal{S} : ((C \times S) \times \mathsf{V}_1(S + (C \times S)))^* \times (C \times S) \rightharpoonup \mathsf{V}_1(S + (C \times S))$$

where whenever $\mathcal{S}(h\langle C,\ s\rangle)$ is well-defined it is a convex combination of valuations in $\langle C,\ s\rangle \longrightarrow$. We call $\mathcal{S}$ deterministic if whenever $\mathcal{S}(h\langle C,\ s\rangle)$ is well-defined it belongs to $\langle C,\ s\rangle \longrightarrow$.

From Definition 8.2.1, we note that when $\mathcal{S}(h\langle C,\ s\rangle)$ is well-defined it looks like $\sum_n q_n \cdot (\sum_i p_i^n \cdot s_i^n + \sum_j p_j^n \cdot \langle C_j^n,\ s_j^n\rangle)$, *i.e.* a convex combination of valuations $\sum_i p_i^n \cdot s_i^n + \sum_j p_j^n \cdot \langle C_j^n,\ s_j^n\rangle$ belonging to $\langle C,\ s\rangle \longrightarrow$. When useful we write μ_n instead of $\sum_i p_i^n \cdot s_i^n + \sum_j p_j^n \cdot \langle C_j^n,\ s_j^n\rangle$. We have all the necessary tools to define the big-step semantics, as the relation

$$\Downarrow^{\mathcal{S},k} \subseteq (((C \times S) \times \mathsf{V}_1(S + (C \times S)))^* \times (C \times S)) \times \mathsf{V}_{\leq 1}(S)$$

where $\mathcal{S}$ is a probabilistic scheduler and k is a natural number, defined by the rules in Figure 74.

$$\frac{}{h\langle C,\ s\rangle \Downarrow^{\mathcal{S},0} 0} \qquad \frac{\mathcal{S}(h\langle C,\ s\rangle) = \sum_n q_n \cdot \mu_n \quad \forall n, j.\ h\langle C,\ s\rangle \mu_n \langle C_j^n,\ s_j^n\rangle \Downarrow^{\mathcal{S},k} \mu_j^n}{h\langle C,\ s\rangle \Downarrow^{\mathcal{S},k+1} \sum_n q_n \cdot (\sum_i p_i^n \cdot s_i^n + \sum_j p_j^n \cdot \mu_j^n)}$$

Figure 74: Big-step operational semantics

Note that in Figure 74, whenever $k = 0$ we return the zero valuation, *i.e.* 0.

8.2.1 Towards a denotational semantics with $\mathsf{P_{TM}V}$

This section presents an attempt to furnish the language from Section 8.2, without while loops, with a denotational semantics based on transition traces. To achieve this, we encapsulate transition traces within a monad that captures both probabilistic and non-deterministic effects.

The idea of using monads to capture computational effects is due to Moggi [77, 78, 79], under the popular "motto": "a program is a function from values to computations". To better understand the meaning of this sentence, let us deconstruct it. A *value* is an element that is pure, *i.e.* it is effect-free, while a *computation* is an element that is subject to some effect. Hence, "a function from values to computations" maps elements that are pure to effectual ones. For example, and considering the non-deterministic effect, the function that given an element s maps it to $\{s\}$, *i.e.* $f : S \to \mathsf{P}(S),\ s \mapsto \{s\}$, lifts the element s, which is pure, to a computation. The previous example can be seen as applying a $skip$ command to a state s which returns $\{s\}$, *i.e.* a set composed of the same input s. Now let us consider a more complex program which first executes an atomic action a and then an atomic action b, *i.e.* $a\,;\,b$. Intuitively, the output of the computation, given an initial state s, should be a set composed of a state indicating that b was applied to the state resultant from the application of a to s, *i.e.* $\{[\![b]\!][\![a]\!]s\}$. Thus the program $a\,;\,b$ is seen as a function $f : S \to \mathsf{P}(S),\ s \mapsto \{[\![b]\!][\![a]\!]s\}$. Furthermore, since a and b are also commands, they are also seen as functions from values to computations. This gives us a clue that f should be equal to the composite of two functions $g : S \to \mathsf{P}(S),\ s \mapsto \{[\![a]\!]s\}$ and $h : S \to \mathsf{P}(S),\ s \mapsto \{[\![b]\!]s\}$. However, when trying to compose g and h we see that the types do not match, because g returns a computation and h needs a value as input. Fortunately, monads are equipped with a tool that allow us to solve this problem. The tool is known as *Kleisli lifting*: it lifts a function f from values to computations to a function from computations to computations. This is done by first evaluating the computation and then applying f to the resulting values. So, what we need to do for the composition of g and h to properly work is to lift $h : S \to \mathsf{P}(S)$ to $h^{\mathsf{Kl}} : \mathsf{P}(S) \to \mathsf{P}(S)$. This way, the composition works as follows: $h^{\mathsf{Kl}} \circ g : S \to \mathsf{P}(S)$ such that $h^{\mathsf{Kl}} \circ g(s) = h^{\mathsf{Kl}}\{[\![a]\!]s\} = \{[\![b]\!][\![a]\!]s\}$.

Definition 8.2.2. A monad [79], also known as a Kleisli triple, over a category $\mathbf{C}$ [2] is a triple $(T, \eta, -^{\mathsf{Kl}})$, where:

- $T : Obj(\mathbf{C}) \to Obj(\mathbf{C})$ is a function between objects of $\mathbf{C}$
- $\eta_A : A \to TA$ is a morphism in $\mathbf{C}$
- we have a morphism in $\mathbf{C}$, $f^{\mathsf{Kl}} : TA \to TB$, for any morphism in $\mathbf{C}$, $f : A \to TB$

such that the following equations hold:

$$\eta_A^{\mathsf{Kl}} = id_{TA} \qquad f^{\mathsf{Kl}} \circ \eta_A = f \qquad g^{\mathsf{Kl}} \circ f^{\mathsf{Kl}} = (g^{\mathsf{Kl}} \circ f)^{\mathsf{Kl}}$$

[2] Informally, a category $\mathbf{C}$ is a structure formed by objects, denoted by $Obj(\mathbf{C})$, and morphisms between the objects [8, 4, 70].

As previously mentioned, monads are used to encapsulate effects in programming languages. For example, to encapsulate non-determinism and/or concurrency we can use the powerset monad. To encapsulate probabilities we can use a probability monad. To model both concurrency and probability simultaneously, we use another monad, namely, $\mathsf{P_{TM}V}$ [106, 73, 109, 111]. The $\mathsf{P_{TM}V}$ monad was first described in the category of semilattices [106, 73], and later reformulated in the category of sets by Varacca [109]. It is this set-theoretic description that we will focus on. Intuitively, the $\mathsf{P_{TM}V}$ monad returns convex sets of valuations, which are finitely generated. From Section 8.2 we already know what valuations are. It lacks to say what a convex set is and what it means for a convex set to be finitely generated. A set X is called convex iff for any $x, y \in X$ and $p \in [0, 1]$ we have $p \cdot x + (1 - p) \cdot y \in X$. A convex set X is finitely generated if there exists a finite set X_0, *i.e.* a finite generator, such that $X = conv(X_0)$, where $conv(X_0)$ is the convex closure of a set defined as:

$$conv(X) = \left\{ \sum_i p_i \cdot x_i \mid \forall i.\, x_i \in X, \sum_i p_i = 1 \right\}$$

At last, the multiplication of a set A by a constant p, is defined as $p \cdot A = \{p \cdot a \mid a \in A\}$, and the sum of sets is the Minkowski sum, *i.e.* for sets A and B, $A + B = \{a + b \mid a \in A, b \in B\}$.

We have everything we need to define the Kleisli triple. For a given set X, let $(\mathsf{P_{TM}V}, \eta, -^{\mathsf{Kl}})$ be as follows:

- $\mathsf{P_{TM}V}(X) = \{Y \subseteq V(X) \mid Y \text{ is convex and finitely generated}\}$
- $\eta_X(x) = \{\delta_x\}$
- $f^{\mathsf{Kl}}(U) = conv\left(\bigcup_{\Psi \in U_0} \sum_{x \in supp(\Psi)} \{\Psi(x) \cdot \psi \mid \psi \in f(x)\}\right)$

where $\mathsf{P_{TM}V}(X)$ is a set of finitely generated convex sets of valuations over X, η_X takes an element x and returns the singleton $\{\delta_x\}$, where δ_x is the Dirac distribution, f^{Kl} takes a convex set $U \in \mathsf{P_{TM}V}(X)$ and, for each $\Psi \in U_0$ (U_0 is a finite generator of U), computes the weighted sum of $\psi \in f(x)$ scaled by $\Psi(x)$, such that the resulting set is then closed under convexity.

To define the functions used to interpret the operators of the language, it is useful to rewrite the Kleisli lifting. Since we are only dealing with finite sets, let $U \in \mathsf{P_{TM}V}(X)$ be of the form $conv\{\sum_i p_i \cdot x_i, \ldots, \sum_j p_j \cdot x_j\}$, where $\{\sum_i p_i \cdot x_i, \ldots, \sum_j p_j \cdot x_j\}$ is a finite generator of U. Then we can rewrite the Kleisli lifting as follows (to simplify we write $\{\Psi_1, \ldots, \Psi_n\}$ instead of $\{\sum_i p_i \cdot x_i, \ldots, \sum_j p_j \cdot x_j\}$):

$$\begin{aligned} & f^{\mathsf{Kl}}(U) \\ & = conv\left(\bigcup_{\Psi \in \{\Psi_1, \ldots, \Psi_n\}} \sum_{x \in supp(\Psi)} \{\Psi(x) \cdot \psi \mid \psi \in f(x)\}\right) \end{aligned}$$

$$= conv\left(\sum_{x\in supp(\Psi_1)}\{\Psi_1(x)\cdot\psi \mid \psi\in f(x)\}\cup\cdots\cup\sum_{x\in supp(\Psi_n)}\{\Psi_n(x)\cdot\psi \mid \psi\in f(x)\}\right)$$
$$= conv\left(\sum_{x\in supp(\Psi_1)}\Psi_1(x)\cdot\{\psi \mid \psi\in f(x)\}\cup\cdots\cup\sum_{x\in supp(\Psi_n)}\Psi_n(x)\cdot\{\psi \mid \psi\in f(x)\}\right)$$
$$= conv\left(\sum_{x\in supp(\Psi_1)}\Psi_1(x)\cdot f(x)\cup\cdots\cup\sum_{x\in supp(\Psi_n)}\Psi_n(x)\cdot f(x)\right)$$

From [109] we know that the Kleisli lifting is independent of the chosen finite generator. This is important because a convex set can have more than one finite generator (*e.g.* the convex set $\{q\cdot a+(1-q)\cdot b \mid q\in[0,1]\}$ has as possible finite generators the sets $\{1\cdot a, 1\cdot b\}$ and $\{1\cdot a, 1\cdot b, 0.5\cdot a+0.5\cdot b\}$). Hence, this assures that the results obtained from the Kleisli lifting are independent from the chosen finite generator.

At the beginning of this section, we said we wanted to develop a denotational semantics, based on transition traces, for the language in Section 8.2. For doing this, we resort to the $\mathsf{P_{TM}V}$ monad, where programs are interpreted as elements of $\mathsf{P_{TM}V}(\Sigma^+)$. Let us unpack $\mathsf{P_{TM}V}(\Sigma^+)$ to understand what is now the general form of the traces. We begin with a finite set S of states. Then we consider pairs of states $S\times S=\Sigma$. Later on we consider non-empty strings of pairs of states Σ^+, which we call *traces*. The generic form of a trace is $(s_0,s_0')(s_1,s_1')\ldots(s_n,s_n')$ and we say that s_0 is the initial state of the trace. Denote such trace by α. We then have $\mathsf{V}(\Sigma^+)=\{\psi:\Sigma^+\to\mathbb{R}^+ \mid supp(\psi) \text{ is finite}\}$. Elements of $\mathsf{V}(\Sigma^+)$ are of the form $\mu=\sum_i p_i\cdot\alpha_i$, with $\alpha_i\in\Sigma^+$, and we call such elements *weighted traces*. At last we have $\mathsf{P_{TM}V}(\Sigma^+)=\{U\subseteq V(\Sigma^+) \mid U \text{ finitely generated and convex}\}$. Elements of $\mathsf{P_{TM}V}(\Sigma^+)$ are of the form $conv\{\sum_i p_i\cdot\alpha_i,\ldots,\sum_j p_j\cdot\alpha_j\}$, whose form is $\sum_n p_n\cdot\mu_n$, where $\sum_n p_n=1$, and we call them *convex traces*. Intuitively, a convex trace is a convex combination of weighted traces.

The way we approach the task of furnishing the language in Section 8.2 with transition traces extended to the probabilistic setting using $\mathsf{P_{TM}V}$, requires us to impose a restriction on the set of sets: it needs to be finite. Recall that when interpreting the command $skip$ with Park's transition traces (Section 8.1.1), we have $\mathcal{T}_P[\![skip]\!]=\{(s,s) \mid s\in S\}$. A direct extension of $\mathcal{T}_P[\![skip]\!]$ to $\mathsf{P_{TM}V}(\Sigma^+)$ gives us, $[\![skip]\!]=conv\{1\cdot(s,s) \mid s\in S\}$. Now, recall that elements of $\mathsf{P_{TM}V}$ are finitely generated convex sets. Thus, the set $conv\{1\cdot(s,s) \mid s\in S\}$ must be finitely generated. This requires the set $\{1\cdot(s,s) \mid s\in S\}$ to be finite, which happens only when S is finite. Thus, to remain within the scope of $\mathsf{P_{TM}V}$, we assume S finite. This also has consequences regarding stuttering and mumbling considered in Brookes'

transition traces. Those closures typically produce infinite sets of traces, which conflicts with the finitely generated requirement of $\mathsf{P_{TM}V}$. Hence, in the $\mathsf{P_{TM}V}(\Sigma^+)$ setting we do not apply stuttering and mumbling. Furthermore, with this approach we cannot interpret a quantum concurrent language. Recall that a qubit in superposition seems like a convex combination of the states $|0\rangle$ and $|1\rangle$, meaning that it exists an infinite number of possible combinations. Despite these restrictions, we found interesting to use the $\mathsf{P_{TM}V}$ monad to see if more limitations arise or not. Note that, if we wanted to avoid all of these problems from the start, we could resort to the CM monad [52], which we do not approach here [3].

Let us now begin to define the functions that shall be used to interpret the operations of the language. To do that, we resort to Kleisli lifting. An advantage of defining these functions using Kleisli lifting is that we can take for granted the following: the resultant set is convex and it is independent of finite generators, *i.e.* if T_0 and T_0' are two finite generators of the convex set T, then applying the Kleisli lifting to the convex closure of T_0 and T_0' is the same as applying the Kleisli lifting to T.

We now proceed to show the definitions, starting with sequential composition. Let $[\![C]\!]$ be the interpretation of a command into $\mathsf{P_{TM}V}(\Sigma^+)$ (the definition will be given later) and denote the sequential composition of $T_1, T_2 \in \mathsf{P_{TM}V}(\Sigma^+)$ by $T_1 \,;\, T_2$. Now let us consider that

$$p \cdot (s, [\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s) \in [\![a \,+_p\, b]\!]$$

and

$$1 \cdot ([\![a]\!]s, [\![c]\!][\![a]\!]s), 1 \cdot ([\![b]\!]s, [\![c]\!][\![b]\!]s) \in [\![c]\!]$$

Our goal is to determine how their sequential composition should be defined.

As an initial attempt, one might try to directly adapt Park's or Brookes' definition of sequential composition (*i.e.* concatenating traces from two sets) to the probabilistic setting. Informally, this would mean picking a convex trace from $[\![a \,+_p\, b]\!]$ and concatenate it with a convex trace from $[\![c]\!]$. Concretely, this first attempt would take the convex trace

$$p \cdot (s, [\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s)$$

and concatenate with

$$1 \cdot ([\![a]\!]s, [\![c]\!][\![a]\!]s)$$

yielding the convex trace

$$p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![c]\!][\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s)([\![a]\!]s, [\![c]\!][\![a]\!]s)$$

[3] Intuitively, the CM monad can be seen as an extension of the $\mathsf{P_{TM}V}$ monad in which the requirement of finite generation is dropped. As a result, the definitions introduced for $\mathsf{P_{TM}V}$ can be straightforwardly adapted to CM. However, since the purpose of this section is to describe an exploratory exercise aimed at extending transition traces to the probabilistic setting, we restrict our presentation to the development carried out under $\mathsf{P_{TM}V}$.

Similarly, we would also concatenate

$$p \cdot (s, [\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s)$$

with

$$1 \cdot ([\![b]\!]s, [\![c]\!][\![b]\!]s)$$

obtaining the convex trace

$$p \cdot (s, [\![a]\!]s)([\![b]\!]s, [\![c]\!][\![b]\!]s) + (1-p) \cdot (s, [\![b]\!]s)([\![b]\!]s, [\![c]\!][\![b]\!]s)$$

At first glance, this may seem like the desired definition. However, examining these traces more closely tells us that none of them are interference-free. For instance, in the convex trace

$$p \cdot (s, [\![a]\!]s)([\![b]\!]s, [\![c]\!][\![b]\!]s) + (1-p) \cdot (s, [\![b]\!]s)([\![b]\!]s, [\![c]\!][\![b]\!]s)$$

the pair $(s, [\![a]\!]s)$ is concatenated with $([\![b]\!]s, [\![c]\!][\![b]\!]s)$, and since $[\![a]\!]s \neq [\![b]\!]s$, the final state of the first pair does not match the initial state of the second. Thus the trace is not interference-free. Such interference can be problematic when attempting to relate operational and denotational semantics. To see why, consider the operational execution of the program

$$(a +_p b)\,;\, c$$

Given an initial state s, this program transits to:

$$\langle (a +_p b)\,;\, c,\ s\rangle \longrightarrow p \cdot \langle c,\ [\![a]\!]s\rangle + (1-p) \cdot \langle c,\ [\![b]\!]s\rangle$$

Then, by abusing notation, it applies the atomic action c to each probabilistic branch:

$$p \cdot \langle c,\ [\![a]\!]s\rangle + (1-p) \cdot \langle c,\ [\![b]\!]s\rangle \longrightarrow p \cdot [\![c]\!][\![a]\!]s + (1-p) \cdot [\![c]\!][\![b]\!]s$$

From these transitions, we infer that the interference-free trace

$$p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![c]\!][\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s)([\![b]\!]s, [\![c]\!][\![b]\!]s)$$

should belong to $[\![(a +_p b)\,;\, c]\!]$. In particular, the operational semantics shows that each trace in the support of

$$p \cdot (s, [\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s)$$

may be sequentially composed with either

$$1 \cdot ([\![a]\!]s, [\![c]\!][\![a]\!]s) \text{ or } 1 \cdot ([\![b]\!]s, [\![c]\!][\![b]\!]s)$$

However, the direct adaptation from Park's or Brooke's definition of the sequential composition does not allow such flexibility. It forces a fixed pairing and consequently fails to generate all traces "produced by" the operational semantics. In fact, what is missing in that definition is what we call *branch independence*. A sequential composition operator with branch independence would allow each branch of the probabilistic choice to be concatenated with any trace of $[\![c]\!]$. This would produce the convex traces previously obtained (with the direct adaptation from Park's or Brooke's definition of the sequential composition):

$$p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![c]\!][\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s)([\![a]\!]s, [\![c]\!][\![a]\!]s)$$
$$p \cdot (s, [\![a]\!]s)([\![b]\!]s, [\![c]\!][\![b]\!]s) + (1-p) \cdot (s, [\![b]\!]s)([\![b]\!]s, [\![c]\!][\![b]\!]s)$$

but also the interference-free trace

$$p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![c]\!][\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s)([\![b]\!]s, [\![c]\!][\![b]\!]s)$$

which the operational semantics exhibits.

Formally, to define the sequential composition we first define a function $seq_U(\alpha)$ that concatenates a trace α with all the weighted traces of the finite generator of U. Then, we use the Kleisli lifting to obtain the sequential composition of two convex sets.

Definition 8.2.3 (Sequential Composition)**.** Let $\alpha \in \Sigma^+$ and $U \in \mathsf{P_{TM}V}(\Sigma^+)$ with

$$U = conv\left\{\sum_i p_i \cdot \alpha_i, \ldots, \sum_i p_j \cdot \alpha_j\right\}$$

Define $seq_U : \Sigma^+ \to \mathsf{P_{TM}V}(\Sigma^+)$ as

$$seq_U(\alpha) = conv\left\{\sum_i p_i \cdot \alpha\alpha_i, \ldots, \sum_j p_j \cdot \alpha\alpha_j\right\}$$

Let $V, U \in \mathsf{P_{TM}V}(\Sigma^+)$. Define $V \,;\, U : \mathsf{P_{TM}V}(\Sigma^+) \times \mathsf{P_{TM}V}(\Sigma^+) \to \mathsf{P_{TM}V}(\Sigma^+)$ as

$$V \,;\, U = seq_U^{\mathsf{Kl}}(V) = conv\left(\sum_i p_i \cdot seq_U(\alpha_i) \cup \cdots \cup \sum_j p_j \cdot seq_U(\alpha_j)\right)$$

where we take $V = conv\left\{\sum_i p_i \cdot \alpha_i, \ldots, \sum_j p_j \cdot \alpha_j\right\}$.

For parallel composition, we follow a similar rationale. We first define a function that acts on a trace and a convex set, and then apply the Kleisli lifting to obtain the desired operation. As in the sequential case, it is necessary to ensure branch independence, but in this case it must be considered for each set individually.

Definition 8.2.4 (Parallel Composition)**.** Let $\sigma \in \Sigma$ and $\alpha, \beta \in \Sigma^+$. Define $\between: \Sigma^+ \times \Sigma^+ \to \mathsf{P}(\Sigma^+)$ as follows

$$\sigma \between \beta = \{\sigma\beta\}$$
$$\sigma\alpha \between \beta = \{\sigma\gamma \mid \gamma \in (\alpha \bowtie \beta)\}$$

Let $U \in \mathsf{P_{TM}V}(\Sigma^+)$ where $U = conv\left\{\sum_i p_i \cdot \alpha_i, \ldots, \sum_j p_j \cdot \alpha_j\right\}$. Define $\between_U: \Sigma^+ \to \mathsf{P_{TM}V}(\Sigma^+)$ as follows:

$$\between_U(\alpha) = conv\left(\sum_i p_i \cdot (\alpha \between \alpha_i) \cup \cdots \cup \sum_j p_j \cdot (\alpha \between \alpha_j)\right)$$

Let $V, U \in \mathsf{P_{TM}V}(\Sigma^+)$. Define $||\ : \mathsf{P_{TM}V}(\Sigma^+) \times \mathsf{P_{TM}V}(\Sigma^+) \to \mathsf{P_{TM}V}(\Sigma^+)$ as follows:

$$\begin{aligned} V \,||\, U &= conv\left(\between_U^{\mathsf{KI}}(V) \cup \between_V^{\mathsf{KI}}(U)\right) \\ &= conv\left(\sum_i p_i \cdot \between_U(\alpha_i) \cup \cdots \cup \sum_j p_j \cdot \between_U(\alpha_j)\right. \\ &\qquad \left. \cup \sum_{i'} p_{i'} \cdot \between_V(\alpha_{i'}) \cup \cdots \cup \sum_{j'} p_{j'} \cdot \between_V(\alpha_{j'})\right) \end{aligned}$$

Note that, when interleaving traces, we use the function $\between$ instead of $\bowtie$, which was introduced in Section 8.1.1. This change is necessary in order to enforce branch independence. Function $\between$ always begins the interleaving with the first pair of the left operand, and then proceeds in the same way as the interleaving operator $\bowtie$ defined earlier. This restriction is important: it prevents the construction of traces whose initial pairs come from different sources. If we were to use $\bowtie$, we would be able to build interference-free traces where the first pair of the trace originates from a different component than the one that begins the corresponding operational computation. Such traces are not operationally feasible and would create problems when attempting to relate operational and denotational semantics. The following example illustrates why we use $\between$ instead of $\bowtie$.

Example 8.2.5. Let us consider the command $(a +_p skip) \,||\, a$, together with an initial state s. Its execution tree is depicted in Figure 75.

Similarly to what we did when giving the intuitions to define sequential composition, from Figure 75 we can infer the following traces:

$$p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s) + (1-p) \cdot (s, s)(s, [\![a]\!]s)$$
$$p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s) + (1-p) \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![a]\!]s)$$

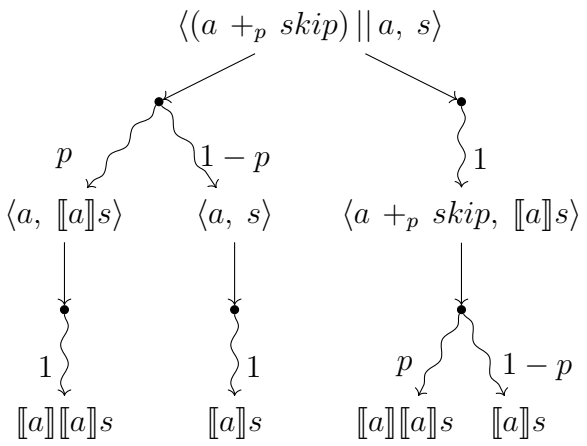


Figure 75: Execution tree of $(a +_p skip) \,||\, a$

Now let us turn our attention to parallel composition (Definition 8.2.4), where we first use $\bowtie$, instead of $\between$. For that, consider the following:

$$p \cdot ([\![a]\!]s, [\![a]\!][\![a]\!]s) + (1-p) \cdot ([\![a]\!]s, [\![a]\!]s) \in [\![a +_p skip]\!]$$

and

$$1 \cdot (s, [\![a]\!]s),\ 1 \cdot ([\![a]\!]s, [\![a]\!][\![a]\!]s) \in [\![a]\!].$$

Having these traces in mind, let us compose them using Definition 8.2.4 with $\bowtie$ instead of $\between$ (some calculation steps are omitted to simplify the presentation):

$$\begin{aligned}
&[\![(a +_p skip) \,||\, a]\!] \\
\supseteq\ & p \cdot ([\![a]\!]s, [\![a]\!][\![a]\!]s) \bowtie (s, [\![a]\!]s) + (1-p) \cdot ([\![a]\!]s, [\![a]\!]s) \bowtie ([\![a]\!]s, [\![a]\!][\![a]\!]s) \\
=\ & p \cdot \{([\![a]\!]s, [\![a]\!][\![a]\!]s)(s, [\![a]\!]s), (s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s)\} \\
& \quad + (1-p) \cdot \{([\![a]\!]s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s), ([\![a]\!]s, [\![a]\!][\![a]\!]s)([\![a]\!]s, [\![a]\!]s)\} \\
\ni\ & p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s) + (1-p) \cdot ([\![a]\!]s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s).
\end{aligned}$$

Let us now focus on the trace that is displayed after the calculation. This trace is interference-free, but it is operationally infeasible. This can be seen in two ways:

1. the trace does not match any of the traces inferred from the operational semantics
2. different initial states appear in

$$p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s) + (1-p) \cdot ([\![a]\!]s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s).$$

Indeed, in $(s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s)$ the initial state is s, while in $([\![a]\!]s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s)$ the initial state is $[\![a]\!]s$. Since this pair represents the beginning of a computation, both initial states should be the same, which is not the case here.

The underlying problem is that $\bowtie$ interleaves all the traces, including the pair that represents the beginning of a computation. However, this initial pair should not be interleaved, because on the operational side the beginning of a computation does not interleave with any of the subsequent computations. For example, consider a parallel program $C_1 \,||\, C_2$. If the first step is taken by C_1, then the interleaving of traces corresponding to this choice should preserve that decision throughout the interleaving. Hence, to capture this behavior we resort to $\between$. It is straightforward to see that, by using $\between$ instead of $\bowtie$ in the previous calculations, the trace

$$p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s) + (1-p) \cdot ([\![a]\!]s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s)$$

ceases to exist.

Furthermore, by considering

$$p \cdot (s, [\![a]\!]s) + (1-p) \cdot (s, s),\ p \cdot ([\![a]\!]s, [\![a]\!][\![a]\!]s) + (1-p) \cdot ([\![a]\!]s, [\![a]\!]s) \in [\![a \ +_p \ skip]\!]$$

and

$$1 \cdot (s, [\![a]\!]s),\ 1 \cdot ([\![a]\!]s, [\![a]\!][\![a]\!]s) \in [\![a]\!],$$

and performing some calculations based on Definition 8.2.4, we obtain the traces

$$p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s) + (1-p) \cdot (s, s)(s, [\![a]\!]s)$$
$$p \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![a]\!][\![a]\!]s) + (1-p) \cdot (s, [\![a]\!]s)([\![a]\!]s, [\![a]\!]s),$$

which exactly correspond to those inferred from Figure 75.

Now we delve into the probabilistic choice. A natural first attempt at its definition is:

$$[\![C_1 \ +_p \ C_2]\!] = p \cdot [\![C_1]\!] + (1-p) \cdot [\![C_2]\!]$$

However, this definition suffers from a problem already illustrated in Example 8.2.5: it allows the first pair of different branches to start with different initial states. For instance, consider the command $a \ +_p \ b$. In terms of the operational semantics, given an initial state s, we have:

$$\langle a \ +_p \ b,\ s \rangle \longrightarrow p \cdot [\![a]\!]s + (1-p) \cdot [\![b]\!]s$$

from which we infer the trace:

$$p \cdot (s, [\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s)$$

But the interpretation of this command, $[\![a \ +_p \ b]\!]$, also allows

$$p \cdot (s, [\![a]\!]s) + (1-p) \cdot (s', [\![b]\!]s')$$

where the trace is interference-free, but the two branches have distinct initial states, contradicting the operational behavior, where all branches must start from the same initial state. This would be problematic when establishing a relation between the operational and denotational semantics.

To overcome such issue, we must find a way to "add" traces only when they have the same initial state. To achieve this, we introduce two auxiliary functions. The first one removes the initial state of a trace, retaining the remainder only if the initial state matches a given state s. Otherwise it returns a trace with weight zero.

Definition 8.2.6. Define $|_s \colon \Sigma^+ \to \mathsf{P_{TM}V}(S + S \times \Sigma^+)$ as follows:

$$|_s \ ((u, u')\alpha) \mapsto \begin{cases} \{1 \cdot u'\alpha\} & \text{if } s = u \\ \{\underline{0}\} & \text{otherwise} \end{cases}$$

While $|_s$ "breaks" traces by removing their initial state, the second auxiliary functions restores this state back to the front of the trace.

Definition 8.2.7. Define $\Delta_s : S + S \times \Sigma^+ \to \mathsf{P_{TM}V}(\Sigma^+)$ as follows:

$$\Delta_s(\alpha) \mapsto \begin{cases} \{1 \cdot (s, s')\} & \text{if } \alpha = s' \\ \{1 \cdot (s, s')\beta\} & \text{if } \alpha = s'\beta \end{cases}$$

We can now present the definition of the probabilistic choice.

Definition 8.2.8 (Probabilistic choice)**.** Let $U, V \in \mathsf{P_{TM}V}(X)$. Define $+_p^s : \mathsf{P_{TM}V}(X) \times \mathsf{P_{TM}V}(X) \to \mathsf{P_{TM}V}(X)$ as

$$U \ +_p^s \ V = conv\left(\cup_{s \in S} \Delta_s^{\mathsf{KI}} \left(|_s^{\mathsf{KI}} \ U +_p \ |_s^{\mathsf{KI}} \ V\right)\right)$$

where $U \ +_p \ V = p \cdot U + (1-p) \cdot V$.

Note that using $|_s$ may produce weighted traces whose sum of the weights may be less than one, as shown next.

Example 8.2.9. Let us recall $[\![a +_p b]\!]$, where we consider the following traces:

$$1 \cdot (s, [\![a]\!]s), 1 \cdot (s', [\![a]\!]s') \in [\![a]\!]$$
$$1 \cdot (s, [\![b]\!]s) \in [\![b]\!]$$

Applying Definition 8.2.8, we obtain the following possible traces:

$$p \cdot (s, [\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s)$$
$$p \cdot (s, [\![a]\!]s)$$
$$(1-p) \cdot (s, [\![b]\!]s)$$
$$0$$

The detailed calculation is as follows:

$$\begin{aligned}
[\![a +_p b]\!] &= conv\left(\cup_{s\in S} \Delta_s^{\mathsf{Kl}}\left(|_s^{\mathsf{Kl}} [\![a]\!] +_p |_s^{\mathsf{Kl}} [\![b]\!]\right)\right) \\
&\supseteq conv\left(\Delta_s^{\mathsf{Kl}}\left(|_s^{\mathsf{Kl}} [\![a]\!] +_p |_s^{\mathsf{Kl}} [\![b]\!]\right) \cup \Delta_{s'}^{\mathsf{Kl}}\left(|_{s'}^{\mathsf{Kl}} [\![a]\!] +_p |_{s'}^{\mathsf{Kl}} [\![b]\!]\right)\right) \\
&= conv\left(\Delta_s^{\mathsf{Kl}}\left(|_s^{\mathsf{Kl}} conv\{1 \cdot (s, [\![a]\!]s), 1 \cdot (s', [\![a]\!]s')\} +_p |_s^{\mathsf{Kl}} \{1 \cdot (s, [\![b]\!]s)\}\right)\right. \\
&\qquad \left.\cup \Delta_{s'}^{\mathsf{Kl}}\left(|_{s'}^{\mathsf{Kl}} conv\{1 \cdot (s, [\![a]\!]s), 1 \cdot (s', [\![a]\!]s')\} +_p |_{s'}^{\mathsf{Kl}} \{1 \cdot (s, [\![b]\!]s)\}\right)\right) \\
&= conv\left(\Delta_s^{\mathsf{Kl}}\left(conv\{1 \cdot [\![a]\!]s, 0\} +_p \{1 \cdot [\![b]\!]s\}\right)\right. \\
&\qquad \left.\cup \Delta_{s'}^{\mathsf{Kl}}\left(conv\{0, 1 \cdot [\![a]\!]s'\} +_p \{0\}\right)\right) \\
&= conv\left(\Delta_s^{\mathsf{Kl}}\left(conv\{p \cdot [\![a]\!]s + (1-p) \cdot [\![b]\!]s, (1-p) \cdot [\![b]\!]s\}\right)\right. \\
&\qquad \left.\cup \Delta_{s'}^{\mathsf{Kl}}\left(conv\{p \cdot [\![a]\!]s', 0\}\right)\right) \\
&= conv\left(conv\{p \cdot (s, [\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s), (1-p) \cdot (s, [\![b]\!]s)\right\} \\
&\qquad \cup conv\{p \cdot (s', [\![a]\!]s'), 0\}) \\
&= conv\{p \cdot (s, [\![a]\!]s) + (1-p) \cdot (s, [\![b]\!]s), (1-p) \cdot (s, [\![b]\!]s), p \cdot (s', [\![a]\!]s'), 0\}
\end{aligned}$$

We now define the atomic action. Its definition resorts to Δ_s and the Kleisli lifting:

Definition 8.2.10 (Atomic action)**.** Let a be an atomic action. Its interpretation is defined as follows:

$$[\![a]\!] = conv(\bigcup_{s\in S}(\Delta_s^{\mathsf{Kl}}\{[\![a]\!]s\}) \cup \{\underline{0}\})$$

Note that the interpretation of the atomic action includes the zero valuation. We delay the explanation of such an addition, since the zero valuation is added to the interpretation of the command $skip$.

At last, we define how a Boolean evaluation is interpreted.

Definition 8.2.11 (Boolean Traces)**.** Let $b \in BExp$ and $s \in S$. Define

$$[\![b]\!] : S \to \mathsf{P}_{\mathsf{TM}}\mathsf{V}(\Sigma^+), s \mapsto \begin{cases} \{1 \cdot (s,s)\} & \text{if } \mathtt{ev}(b)s = tt \\ \{0 \cdot (s,s)\} & \text{otherwise} \end{cases}$$

and

$$[\![\neg b]\!] : S \to \mathsf{P}_{\mathsf{TM}}\mathsf{V}(\Sigma^+), s \mapsto \begin{cases} \{1 \cdot (s,s)\} & \text{if } \mathtt{ev}(b)s = ff \\ \{0 \cdot (s,s)\} & \text{otherwise} \end{cases}$$

This definition, similarly to Definition 8.2.8, also creates weighted traces whose sum of the weights does not equal one.

The interpretation of the commands is given by the following definition.

Definition 8.2.12. Define $[\![-]\!] : C \to \mathsf{P}_{\mathsf{TM}}\mathsf{V}(\Sigma^+)$ as follows:

$$[\![skip]\!] = conv\left(\bigcup_{s \in S} \{1(s,s), \underline{0}\}\right)$$

$$[\![a]\!] = conv\left(\bigcup_{s \in S} (\Delta_s^{\mathsf{KI}}\{[\![a]\!]s\}) \cup \{\underline{0}\}\right)$$

$$[\![C_1 \,;\, C_2]\!] = seq^{\mathsf{KI}}_{[\![C_2]\!]}[\![C_1]\!]$$

$$[\![C_1 \,\mathsf{or}\, C_2]\!] = conv\left([\![C_1]\!] \cup [\![C_2]\!]\right)$$

$$[\![C_1 \;+_p\; C_2]\!] = [\![C_1]\!] \;+^s_p\; [\![C_2]\!]$$

$$[\![C_1 \,||\, C_2]\!] = conv\left(\between^{\mathsf{KI}}_{[\![C_2]\!]}[\![C_1]\!] \cup \between^{\mathsf{KI}}_{[\![C_1]\!]}[\![C_2]\!]\right)$$

$$[\![\mathtt{if}\ \ b\ \ \mathtt{then}\ \{C_1\}\ \ \mathtt{else}\ \{C_2\}]\!] = conv\left(\bigcup_{s \in S} seq^{\mathsf{KI}}_{[\![C_1]\!]}[\![b]\!]s \cup seq^{\mathsf{KI}}_{[\![C_2]\!]}[\![\neg b]\!]s\right)$$

As mentioned when defining the interpretation of atomic actions, the zero valuation is also included in the interpretation of the command $skip$. This ensures that $\underline{0}$ is present in the interpretation of every command, *i.e.* $\underline{0} \in [\![C]\!]$. This is useful, for example, when interpreting while loops, since $\underline{0}$ can be regarded as the least element. There is also another reason for including the zero valuation in the denotation of $skip$ and atomic actions. Recall that the big-step semantics in Figure 74 executes k steps of a computation, where k is a natural number. Whenever $k = 0$, the big-step semantics returns the zero valuation. To capture this behavior in the denotational semantics, we include $\underline{0}$ in the interpretation of $skip$ and atomic actions. Intuitively, the presence of $\underline{0}$ in the interpretation of a command represents the situation where the command is not executed.

Recall that this section is presented as an exercise to furnish the language from Section 8.2, without while loops, with a denotational semantics based on transition traces. Our approach to this problem

was an ad-hoc one: we first attempted to directly extend Park's and Brookes' definitions of sequential and parallel composition to the probabilistic setting. Then, whenever something seemed to not agree with the operational semantics, we try to fix the definitions. As documented throughout this section, this approach comes with some peculiarities: the need for branching independence, the requirement that traces have the same initial state, and the absence of a least element unless the zero valuation is added to the interpretation of $skip$ and atomic actions. These peculiarities, in turn, make the definitions of the operators cumbersome, requiring the introduction of auxiliary functions. A clear example is the probabilistic choice. The natural definition would simply be $[\![C_1 +_p C_2]\!] = p \cdot [\![C_1]\!] + (1-p) \cdot [\![C_2]\!]$. However, the fact that this allows the combination of traces with different initial states makes it unsuitable if one wishes to establish a correspondence between operational and denotational semantics.

The conclusion we draw from this exercise is that this ad-hoc approach is not the most suitable way to furnish the language from Section 8.2 with a denotational based on transition traces. It would therefore be worthwhile to explore other, more "methodical" approaches in the line of [2, 49, 80, 34].

8.3 Haskell in a nutshell

Chapter 10 describes our Haskell interpreter for a concurrent (quantum) language added with synchronization primitives (the one in Section 9.1). We thus give a brief introduction to Haskell.

Haskell [71] is a *functional programming language* that is *lazy* and *statically typed*. These three keywords capture its essence:

- Functional programming language: Programs are expressed in terms of functions and how they transform input data, rather than as step-by-step instructions (the latter being the approach of imperative programming)
- Laziness: Haskell only evaluates expressions when their results are actually needed
- Statically typed: Every expression in Haskell has a type known at compile time. This not only enables type checking – ensuring that only well-typed programs compile – but also provides an additional design perspective, complementary to the functional view.

Here we provide only a brief introduction. For a more detailed exposition, see [68, 84].

Since Haskell is based on functions, let us start with a simple example: the factorial function.

```
factorial :: Int -> Int
factorial 0 = 1
```

```
factorial n = n * factorial (n-1)
```

Here, the function is named `factorial` and its type is `Int -> Int`: it takes an integer and returns an integer. Next, it comes the body of the function, in other words, the function's definition. If the input is `0`, the result is `1`. Otherwise, the result is `n` multiplied by the recursive call `factorial (n-1)`. This way of performing a computation is known as *pattern matching* and it is vastly used in Haskell.

Haskell has several built-in types, such as `Int` for integer numbers, `Char` for characters, and `Double` for floating-point numbers. Haskell also allows one to *rename* existing types. For instance, we can use `Pos` to rename the pair of integers `(Int, Int)`, as follows: `type Pos = (Int, Int)`.

We can also define *data types*, which are used to declare structures composed of value constructors. Value constructors may or may not have arguments. For instance, `data Move = North | South | East | West` defines a data type called `Move` with four value constructors without arguments: `North`, `South`, `East`, and `West`. Similarly, `data Shape = Circle Int Int Int | Square Int` defines a data type called `Shape` with two value constructors, each with different arguments. The first one, `Circle Int Int Int`, represents a circle whose first two arguments denote the coordinates of its center, and whose last argument gives the radius. The second one, `Square Int`, represents a square whose single argument specifies the length of its sides. Haskell also allows us to define recursive data types, such as: `data Tree a = Leaf a | Node (Tree a) (Tree a)`. This data type represents a binary tree: a value can be a `Leaf` containing a value of type `a`, or a `Node` with two subtrees. The parameter `a` is a polymorphic type, so values of types `Tree Int` and `Tree String` are both valid instances. Furthermore, when the structure we want to define consists of a single value constructor with a single argument, we can define it as before, *i.e.* `data Square = Square Int`. However, Haskell also allows us to define such types using `newtype`, as follows: `newtype Square = Square Int`. Note that, although the syntax of `data` and `newtype` is similar, Haskell treats them differently [68, Chapter 11]. For our purposes, it is also useful to note that `newtype` can be written using record syntax [68, Chapter 8]. For instance, `newtype Square = Square { getSize :: Int }` is an equivalent way of defining `newtype Square`, where the function `getSize` is automatically created to retrieve the length of the square's side.

Haskell is a pure language, *i.e.* functions correspond directly to mathematical functions. To model effects (such as state, I/O, nondeterminism, etc.), Haskell uses monads [68, Chapter 12]. In Haskell, a monad `m` comes with two operations:

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

Intuitively, `return` receives a pure value and embeds it into a monad; while >>=, known as *bind*, takes a monadic value and a function, and applies the function to each value encapsulated in the monad. If we recall Definition 8.2.2, one notices that `return` corresponds to η and bind corresponds to Kleisli lifting.

Haskell provides the `do` notation, which is syntactic sugar for chaining computations with >>=. This gives Haskell a more imperative-like feel when needed. To see it, let us write a function that adds `"Hello "` to a string captured from the input given by the user, using the binding operator.

```
sayHello :: IO()
sayHello = getLine >>= (\x -> putStrLn("Hello " ++ x))
```

This function reads user input, through `getLine`, stores it in `x`, and then prints `"Hello "` concatenated with `x`. Using `do` notation, the same function can be rewritten as follows:

```
sayHello :: IO()
sayHello = do
   x <- getLine
   putStrLn ("Hello " ++ x)
```

Although the structure looks imperative, since it looks like it is executing instructions, the program is still purely functional. The do block is simply shorthand for nested uses of bind.

We now consider the monad transformer library, known as *mtl* [45]. For more details and examples, see [67, 60, 81, 93]. Formally,

Definition 8.3.1. A monad transformer [67, Section 7] is any type constructor `t` such that if `m` is a monad, so is `t m`. This can be expressed in Haskell as follows:

```
class (Monad m, Monad (t m)) => MonadT t m where
   lift :: m a -> t m a
```

satisfying the following properties:

- `lift . return = return`
- `lift (m >>= f) = lift m >>= (lift . f)`

Intuitively, a monad transformer can be seen as a monad that is parameterized by an inner monad [51]. The `lift` operation embeds computations from the inner monad `m` into the transformed monad `t m` preserving their behavior. In other words, `lift` embeds and preserves computations made in the inner monad `m`, when those computations are interpreted in the context of the outer one `t m`.

Being a monad transformer parameterized by an inner monad, a transformer gives us a way to combine effects. In fact, monad transformers are useful for building programs that involve multiple interacting effects. Intuitively, we can picture the combination of effects as a tower made of LEGO blocks: the block at the base of the tower corresponds to the inner monad, and the block placed on top corresponds to a monad transformer applied to that monad. If we want to add another effect later, we put another LEGO block on top. The new base is the monad obtained so far (recall that a transformer applied to a monad is a monad), and the new block corresponds to another monad transformer applied on top of it. Thus, each layer builds on the previous one, forming a stack of effects.

At this moment, it is useful to ask: "if the LEGO pieces are colored and they are all different, do we obtain the same tower when we change their order?". The short answer is no, because the ordering of the colors is changed (note that despite the structure still being a tower, what we want to emphasize here is the order in which the colors are stacked). The same phenomenon appears when combining effects using monad transformers. To do this, we rely on the *mtl* [45] library, which provides the monad transformers `StateT` for state and `ExceptT` for exceptions. To model nondeterminism, we use the list monad `[]`. Before combining these effects, it is helpful to recall how the "unwrapping functions" (which we may informally call "deconstructors") of the transformers work. The type constructor of `StateT` is `StateT s m a`, and its unwrapping function is

```
runStateT :: StateT s m a → s → m (a, s)
```

where `s` is a state, `m` is the inner monad, and `a` is a value. Similarly, the type constructor of `ExceptT` is `ExceptT e m a`, with the unwrapping function

```
runExceptT :: ExceptT e m a → m (Either e a)
```

where `e` represents an exception. We are now ready to see how we can combine effects using monad transformers. We have two monad transformers, `StateT` and `ExceptT`, and a monad `[]`. Since `[]` is a monad but not a monad transformer, it cannot take an inner monad (in the LEGO analogy, `[]` must serve as the base block of the tower). We now choose an order for the remaining two effects. Suppose we place the state effect first. This yields the type `StateT s [] a`. By applying `runStateT` we obtain

```
s → [(a, s)]
```

i.e. functions that, given a state `s`, return a list of pairs containing a value `a` and an updated state `s`. Next we place the exception effect on top. This produces the type `ExceptT e (StateT s []) a`. Unwrapping this with `runExceptT` gives

```
StateT s [] (Either e a)
```

and unwrapping again with `runStateT` yields

```
s → [(Either e a, s)]
```

Thus, given a state `s`, we obtain a list of results each consisting of either an exception `e` or a value `a`, paired with a state. Now let us swap the order of the state and exception effects. The base is still the list monad `[]`. Placing `ExceptT` first gives the type `ExceptT e [] a`, which unwraps to

```
[Either e a]
```

i.e. a list whose elements are either exceptions `e` or values `a`. Stacking the state effect on top gives the type `StateT s (ExceptT e []) a`. Unwrapping with `runStateT` yields

```
s → ExceptT e [] (a, s)
```

which further unwraps via `runExceptT` to

```
s → [Either e (a, s)]
```

This means that given a state `s`, we obtain a list whose elements are either exceptions `e` or pairs `(a, s)` containing a value and an updated state. This example shows that although both constructions incorporate the same effects, the order in which we stack the transformers determines the structure of the resulting computations (just as rearranging the colored LEGO blocks produces a tower with a different pattern).

Remark 5. Monads in Haskell are designed to capture the theoretical concept of monads that originates in category theory. However, monad transformers in Haskell differ from their categorical counterparts. Briefly, in terms of category theory, a monad transformer is a structure-preserving endofunctor on monads. In contrast, in Haskell, it is a type constructor that, given a monad, produces a new monad, allowing us to stack effects.

In this work, we are concerned only with monad transformers in the Haskell sense. Therefore, throughout the document, the term “monad transformer” refers exclusively to Definition 8.3.1.

Chapter 9

Adding a mechanism of synchronization

9.1 Extending and instantiating the core language

We now instantiate the language in Section 8.2 to the classical setting and extend it with a synchronization primitive, the *await* command [21, 54], in order to account for critical memory accesses. For convenience, we repeat here some information already presented in Section 8.2.

We start by detailing the aforementioned instantiation. We consider a set of classical variables, $cVar$. We use the letter x, possibly with integers in subscript, to represent classical variables. The set of atomic actions is only composed of deterministic assignments. The latter are of the form $x := e$ where $x \in cVar$ and e is an integer expression. We consider a set of integer expressions and Boolean expressions denoted as Exp and $BExp$, respectively. We use the letter e, possibly with integers in subscript, to denote elements of Exp. Similarly, we use the letter b to talk about elements of $BExp$. We take as integer expressions, Boolean expressions, and commands those formally described by the following BNF grammars.

$$e ::= n \in \mathbb{Z} \mid x \in cVar \mid e\ op_e\ e$$

$$b ::= true \mid false \mid \neg b \mid b \,\&\, b \mid b \mid b \mid e\ op_b\ e$$

$$C ::= skip \mid a \in Act \mid C +_p C \mid C\,;C \mid C \,\mathsf{or}\, C \mid C \,||\, C \mid \texttt{if}\ b\ \texttt{then}\ \{C\}\ \texttt{else}\ \{C\}$$
$$\mid \texttt{while}\ b\ \texttt{do}\ \{C\} \mid \texttt{await}\ b\ \texttt{do}\ \{C\} \mid \texttt{Atom}(C)$$

where $op_e \in \{+, -, \times, /\}$, $op_b \in \{=, \leq, \geq, <, >\}$, & stands for a conjunction, and $\mid$ stands for a disjunction. The operations in op_e are the addition, subtraction, multiplication, and division of integer numbers. The predicates in op_b, also related with integer numbers, are equality, less than or equal, greater than or equal, less than and greater than.

With the addition of the await command, we also added the $\texttt{Atom}(-)$ constructor. In a nutshell, $\texttt{await}\ b\ \texttt{do}\ \{C\}$ will idle as long as b is false, and when it becomes true (if ever), its instructions are

prioritized by encapsulating inside $\texttt{Atom}(-)$. The priority of $\texttt{Atom}(C)$ is achieved using schedulers in the big-step semantics. Intuitively, whenever a scheduler sees a command inside the $\texttt{Atom}(-)$ constructor, it will ignore any other possible execution and focus solely on the instructions inside $\texttt{Atom}(-)$.

We now focus on the operational semantics. In the classical setting, the set of states S is seen as a set of functions $cVar \to \mathbb{Z}$ from classical variables to integers. To evaluate integer expressions we resort to a big-step semantics defined as a partial function $\twoheadrightarrow_e$: $Exp \times S \rightharpoonup \mathbb{Z}$ obeying the rules in Figure 76. Similarly, for Boolean expressions the big-step semantics is defined as $\twoheadrightarrow_b$: $BExp \times S \rightharpoonup \{tt, ff\}$ following the rules in Figure 77. Both semantics are partial to properly handle undefined cases, such as division by zero. If we excluded integer division from op_e, then these semantics could instead be defined as total functions.

$$\langle n,\ s\rangle \twoheadrightarrow_e n \qquad \langle x,\ s\rangle \twoheadrightarrow_e look_x(s)$$

$$\frac{\langle e_1,\ s\rangle \twoheadrightarrow_e n_1 \quad \langle e_2,\ s\rangle \twoheadrightarrow_e n_2}{\langle e_1\ op_e\ e_2,\ s\rangle \twoheadrightarrow_e n} \text{ (if } n1\ op_e\ n2 = n)$$

Figure 76: Rules for integer expressions

Figure 76 shows the rules for integer expressions. The rule for variables uses function $look_x(s)$, which retrieves the value of x from state s. For compound expressions involving op_e, the rule is only valid when the operation $n_1\ op_e\ n_2$ evaluates to an integer n. For example, an expression involving division by zero is undefined, consistent with the partial nature of $\twoheadrightarrow_e$. Figure 77 shows the rules for Boolean expressions. The operators op_b represent relational comparisons over integers, while & and | stand for conjunction and disjunction. Both figures make use of op_e and op_b in order to present the rules concisely, instead of writing one inference rule per operator.

We now turn to the operational semantics of commands. Figure 78 presents the rules for the assignment and await commands. These rules are to be considered in addition to those already given in Figure 73. In particular, note that the function $[\![b]\!]s$ used earlier is now replaced by the evaluation of $\langle b,\ s\rangle$, according to the rules for Boolean expressions given in Figure 77.

For the assignment rule, we use a function $upd_{(x,n)}(s)$ that updates the value of x with n in s. Regarding the await command, its behavior depends on the evaluation of the Boolean condition: if the condition evaluates to false, then the computation idles (*i.e.* no progress is made); otherwise, the associated instructions are executed within an atomic block, denoted by $\texttt{Atom}(-)$. The evaluation of an atomic

$$\langle true,\ s\rangle \twoheadrightarrow_b tt \qquad \langle false,\ s\rangle \twoheadrightarrow_b ff \qquad \frac{\langle b,\ s\rangle \twoheadrightarrow_b tt}{\langle \neg b,\ s\rangle \twoheadrightarrow_b ff} \qquad \frac{\langle b,\ s\rangle \twoheadrightarrow_b ff}{\langle \neg b,\ s\rangle \twoheadrightarrow_b tt}$$

$$\frac{\langle b_1,\ s\rangle \twoheadrightarrow_b bv_1 \quad \langle b_2,\ s\rangle \twoheadrightarrow_b bv_2}{\langle b_1 \,\&\, b_2,\ s\rangle \twoheadrightarrow_b tt} \ (\text{if } bv_1 \,\&\, bv_2 = tt)$$

$$\frac{\langle b_1,\ s\rangle \twoheadrightarrow_b bv_1 \quad \langle b_2,\ s\rangle \twoheadrightarrow_b bv_2}{\langle b_1 \,\&\, b_2,\ s\rangle \twoheadrightarrow_b ff} \ (\text{if } bv_1 \,\&\, bv_2 = ff)$$

$$\frac{\langle b_1,\ s\rangle \twoheadrightarrow_b bv_1 \quad \langle b_2,\ s\rangle \twoheadrightarrow_b bv_2}{\langle b_1 \mid b_2,\ s\rangle \twoheadrightarrow_b tt} \ (\text{if } bv_1 \mid bv_2 = tt)$$

$$\frac{\langle b_1,\ s\rangle \twoheadrightarrow_b bv_1 \quad \langle b_2,\ s\rangle \twoheadrightarrow_b bv_2}{\langle b_1 \mid b_2,\ s\rangle \twoheadrightarrow_b ff} \ (\text{if } bv_1 \mid bv_2 = ff)$$

$$\frac{\langle e_1,\ s\rangle \twoheadrightarrow_e n_1 \quad \langle e_2,\ s\rangle \twoheadrightarrow_e n_2}{\langle e_1\ op_b\ e_2,\ s\rangle \twoheadrightarrow_b tt} \ (\text{if } n_1\ op_b\ n_2 = tt)$$

$$\frac{\langle e_1,\ s\rangle \twoheadrightarrow_e n_1 \quad \langle e_2,\ s\rangle \twoheadrightarrow_e n_2}{\langle e_1\ op_b\ e_2,\ s\rangle \twoheadrightarrow_b ff} \ (\text{if } n_1\ op_b\ n_2 = ff)$$

Figure 77: Rules for Boolean expressions

command $\texttt{Atom}(C)$ depends on how C itself evaluates: if C reaches a final state, then that final state is returned; if C does not yet reach a final state, the atomic wrapper is preserved, and evaluation proceeds with the remaining computation inside it.

9.1.1 Setting up the quantum framework

The instantiation of the language in Section 8.2 to the quantum realm can be done in two ways. The first one, presented in this section and originally introduced in [80], includes both classical and quantum variables. This mixed setting reflects the QRAM architecture [63], which models the current interplay between classical and quantum computers. As we will later see, this language, extended with a synchronization mechanism, serves as the basis for the Haskell interpreter developed in Chapter 10. The second instantiation, which will be presented in Section 9.2, arose from joint work with Manisha Jain, motivated by the

$$\frac{\langle e,\ s\rangle \twoheadrightarrow_e n}{\langle x := e,\ s\rangle \longrightarrow 1 \cdot upd_{(x,n)}(s)}$$

$$\frac{\langle b,\ s\rangle \twoheadrightarrow_b tt}{\langle \texttt{await}\ b\ \texttt{do}\ \{C\},\ s\rangle \longrightarrow 1 \cdot \langle \texttt{Atom}(C),\ s\rangle}$$

$$\frac{\langle b,\ s\rangle \twoheadrightarrow_b ff}{\langle \texttt{await}\ b\ \texttt{do}\ \{C\},\ s\rangle \longrightarrow 1 \cdot \langle \texttt{await}\ b\ \texttt{do}\ \{C\},\ s\rangle}$$

$$\frac{\langle C_1,\ s\rangle \longrightarrow \sum_i p_i \cdot s_i + \sum_j p_j \cdot \langle C_j,\ s_j\rangle}{\langle \texttt{Atom}(C),\ s\rangle \longrightarrow \sum_i p_i \cdot s_i + \sum_j p_j \cdot \langle \texttt{Atom}(C_j),\ s_j\rangle}$$

Figure 78: New rules added to the operational semantics from Figure 73

goal of equipping a concurrent quantum language with a quantum dynamic logic [11, 10, 103]. Because such a logic is formulated solely over quantum variables, this instantiation excludes classical variables. As a result, we are able to focus exclusively on the behavior of the quantum system.

We now describe the first instantiation mentioned. We consider m bits and n qubits at our disposal. The state space, where both bits and qubits will be encoded, is the complex vector space $\mathbb{C}^{(2\times 2)^m} \otimes \mathbb{C}^{(2\times 2)^n}$. The variables x_i ($1 \leq i \leq m$) and q_j ($1 \leq j \leq n$) are used to represent classical variables and qubits, respectively. Furthermore, the notation $\tilde{q}$ represents a list of qubits.

States are represented as classical-quantum states [114], whose general form is

$$\sum_i p_i \cdot |x_i\rangle\langle x_i| \otimes |\psi_i\rangle\langle\psi_i|$$

where x_i represents a classical state, $|\psi_i\rangle \in \mathbb{C}^{2^n}$ is a pure quantum state, and $\sum_i p_i = 1$. Thus, a classical-quantum state is a density operator, in particular, a convex combination of classical and quantum states.

From the postulates of quantum mechanics, the evolution of a quantum system is governed by unitary gates, while information is obtained through measurements. These operations are captured by completely positive trace-preserving maps (CPTP maps), also called channels.

Recall from Chapter 2 that given a unitary U and a density operator ρ, the action of U in ρ is given as $U\rho U^\dagger$, where $U^\dagger$ is the adjoint of U. We also recall the notions of partial trace and measurement, which we reintroduce here following [80].

The partial trace of the j-th position of the classical-quantum state is defined as follows:

$$Tr_j = (\otimes_{k=0}^{j-1} Id) \otimes Tr \otimes (\otimes_{k=j+1}^{n} Id)$$

Intuitively, the partial trace discards the information in the j-th position. The measurement of the j-th position of the classical-quantum state is defined as the "quantum-to-classical" channel [114, Theorem 2.37]

$$\Phi_j \begin{pmatrix} a & b \\ c & d \end{pmatrix} = a \cdot (|0\rangle\langle 0|)^{\otimes 2} + d \cdot (|1\rangle\langle 1|)^{\otimes 2}$$

Intuitively, Φ_j projects the j-th position of the classical-quantum state onto $|0\rangle$ and $|1\rangle$ with probabilities a and d, respectively. The duplication of $|0\rangle\langle 0|$, *i.e.* $(|0\rangle\langle 0|)^{\otimes 2}$, is used to store a copy of the measurement outcome in a classical variable.

We are now ready to instantiate the set of atomic actions, in Section 8.2, to the quantum setting. An atomic action can be a unitary gate, a measurement, or a qubit reset. Unitary gates are denoted as $U(\tilde{q})$, where the unitary U acts on the list of qubits $\tilde{q}$. Resets are written as $\texttt{Reset}(q_j)$, with the j-th qubit set to $|0\rangle$. Measurements are denoted as $\texttt{Meas}(x_i, q_j)$, where the j-th qubit is measured, and the result is stored in the i-th classical variable. Furthermore, the unitary gates we consider are the following:

$Id \rightarrow$ Identity

$X \rightarrow$ Pauli-X (NOT) gate

$Y \rightarrow$ Pauli-Y gate

$Z \rightarrow$ Pauli-Z gate

$SWAP \rightarrow$ SWAP gate

$H \rightarrow$ Hadamard gate

$CNOT \rightarrow$ Controlled-Not gate

$CZ \rightarrow$ Controlled-Z gate

$TOF \rightarrow$ Toffoli gate

$S \rightarrow$ Phase gate (S)

$Ph(\phi) \rightarrow$ Phase shift gate

$CPh(\phi) \rightarrow$ Controlled-Phase shift gate

The atomic action rule in Figure 73 is then instantiated with the rules in Figure 79, where mov_j replaces the j-th qubit with $|0\rangle\langle 0|$ and $mov_{i \leftarrow j}$ moves the measurement outcome of the j-th qubit to the i-th classical variable.

$$\langle U(\tilde{q}),\ |x\rangle\langle x| \otimes |\psi\rangle\langle\psi| \rangle \longrightarrow |x\rangle\langle x| \otimes U|\psi\rangle\langle\psi|U^\dagger$$

$$\langle \texttt{Reset}(q_j),\ |x\rangle\langle x| \otimes |\psi\rangle\langle\psi| \rangle \longrightarrow |x\rangle\langle x| \otimes mov_j(|0\rangle\langle 0| \otimes Tr_j|\psi\rangle\langle\psi|)$$

$$\langle \texttt{Meas}(x_i, q_j),\ |x\rangle\langle x| \otimes |\psi\rangle\langle\psi| \rangle \longrightarrow mov_{i\leftarrow j}\left(Tr_i|x\rangle\langle x| \otimes \Phi_j|\psi\rangle\langle\psi|\right)$$

Figure 79: Rules for quantum atomic actions

Recall that the channel Φ_j introduces probabilities, which are hidden in Figure 79. To make probabilities explicit, let us represent the classical-quantum state as $\rho_c \otimes \rho_q$, where ρ_c is the classical counterpart and ρ_q is the quantum counterpart. Moreover, we further expand ρ_q as $\sum_{k'=1}^{j-1} |\psi_{k'}\rangle\langle\psi_{k'}| \otimes \begin{pmatrix} p_0 & b \\ c & p_1 \end{pmatrix} \otimes \sum_{k''=j+1}^{n} |\psi_{k''}\rangle\langle\psi_{k''}|$ to clearly identify the j-th qubit. We then proceed as follows:

$$\begin{aligned}
&mov_{i\leftarrow j}\left(Tr_i\rho_c \otimes \Phi_j\rho_q\right) \\
&= mov_{i\leftarrow j}\left(Tr_i\rho_c \otimes \Phi_j\left(\sum_{k'=1}^{j-1} |\psi_{k'}\rangle\langle\psi_{k'}| \otimes \begin{pmatrix} p_0 & b \\ c & p_1 \end{pmatrix} \otimes \sum_{k''=j+1}^{n} |\psi_{k''}\rangle\langle\psi_{k''}|\right)\right) \\
&= mov_{i\leftarrow j}\left(Tr_i\rho_c \otimes \left(\sum_{k'=1}^{j-1} |\psi_{k'}\rangle\langle\psi_{k'}| \otimes \Phi_j\begin{pmatrix} p_0 & b \\ c & p_1 \end{pmatrix} \otimes \sum_{k''=j+1}^{n} |\psi_{k''}\rangle\langle\psi_{k''}|\right)\right) \\
&= mov_{i\leftarrow j}\left(Tr_i\rho_c \otimes \left(\sum_{k'=1}^{j-1} |\psi_{k'}\rangle\langle\psi_{k'}| \otimes (p_0 \cdot (|0\rangle\langle 0|)^{\otimes 2}\right.\right. \\
&\qquad\qquad \left.\left. +p_1 \cdot (|1\rangle\langle 1|)^{\otimes 2}) \otimes \sum_{k''=j+1}^{n} |\psi_{k''}\rangle\langle\psi_{k''}|\right)\right) \\
&= mov_{i\leftarrow j}\left(Tr_i\rho_c \otimes \left(\sum_{k'=1}^{j-1} |\psi_{k'}\rangle\langle\psi_{k'}| \otimes (p_0 \cdot (|0\rangle\langle 0|)^{\otimes 2}) \otimes \sum_{k''=j+1}^{n} |\psi_{k''}\rangle\langle\psi_{k''}|\right.\right. \\
&\qquad\qquad \left.\left. + \sum_{k'=1}^{j-1} |\psi_{k'}\rangle\langle\psi_{k'}| \otimes (p_1 \cdot (|1\rangle\langle 1|)^{\otimes 2}) \otimes \sum_{k''=j+1}^{n} |\psi_{k''}\rangle\langle\psi_{k''}|\right)\right) \\
&= mov_{i\leftarrow j}\left(Tr_i\rho_c \otimes \left(p_0 \cdot \sum_{k'=1}^{j-1} |\psi_{k'}\rangle\langle\psi_{k'}| \otimes (|0\rangle\langle 0|)^{\otimes 2} \otimes \sum_{k''=j+1}^{n} |\psi_{k''}\rangle\langle\psi_{k''}|\right.\right. \\
&\qquad\qquad \left.\left. +p_1 \cdot \sum_{k'=1}^{j-1} |\psi_{k'}\rangle\langle\psi_{k'}| \otimes (|1\rangle\langle 1|)^{\otimes 2} \otimes \sum_{k''=j+1}^{n} |\psi_{k''}\rangle\langle\psi_{k''}|\right)\right) \\
&\overset{(\star)}{=} mov_{i\leftarrow j}\left(Tr_i\rho_c \otimes \left(p_0 \cdot \rho_q^{00} + p_1 \cdot \rho_q^{11}\right)\right) \\
&= mov_{i\leftarrow j}\left(Tr_i\rho_c \otimes p_0 \cdot \rho_q^{00} + Tr_i\rho_c \otimes p_1 \cdot \rho_q^{11}\right) \\
&= mov_{i\leftarrow j}\left(p_0 \cdot Tr_i\rho_c \otimes \rho_q^{00} + p_1 \cdot Tr_i\rho_c \otimes \rho_q^{11}\right)
\end{aligned}$$

$$= p_0 \cdot \rho_c^0 \otimes \rho_q^0 + p_1 \cdot \rho_c^1 \otimes \rho_q^1$$

where in step $(\star)$ we use ρ_q^{00} to denote $\sum_{k'=1}^{j-1} |\psi_{k'}\rangle\langle\psi_{k'}| \otimes (|0\rangle\langle 0|)^{\otimes 2} \otimes \sum_{k''=j+1}^{n} |\psi_{k''}\rangle\langle\psi_{k''}|$, and analogously for ρ_q^{11}. Furthermore, p_0 is the probability of measuring $|0\rangle$, ρ_c^0 is the classical state after storing the result 0 in x_i, and ρ_q^0 is the quantum state resulting from applying the projection to $|0\rangle$ and moving the result of the measurement of the j-th qubit to the i-th classical variable. A similar reasoning is applied to p_1, ρ_c^1, and ρ_q^1. Hence we can update the operational semantics of the measurement command, with explicit probabilities, as follows:

$$\langle \texttt{Meas}(x_i, q_j),\ \rho_c \otimes \rho_q \rangle \longrightarrow p_0 \cdot \rho_c^0 \otimes \rho_q^0 + p_1 \cdot \rho_c^1 \otimes \rho_q^1$$

9.2 Fully Quantum Instantiation

This section describes the full quantum instantiation which omits classical variables. This was used in the collaboration with Manisha Jain [54, 55], where a Maude animation of the language was also developed [55]. As discussed in Section 9.1.1, this joint work aimed to equip a concurrent quantum language, extended with a synchronization primitive, with a quantum dynamic logic. Since this logic is formulated only in terms of quantum variables, the language is also restricted to quantum variables. The synchronization primitive adopted in this setting is the $await$ command, inspired by Brookes [21]. Details on how this constructor is adapted to a purely quantum setting will be given later.

The main issue to be considered is the inclusion of this synchronization primitive. Intuitively, the $await$ command gives priority to certain instructions whenever a given condition holds, while idling when the condition does not hold. To the best of our knowledge, this is the first use of the $await$ command in a concurrent quantum language. However, a related idea appears in [122], where Mingsheng Ying introduces a critical-section command that behaves like $await$ whenever the condition is always true.

9.2.1 Language and operational semantics

We begin by explaining how the requirements of the logic influenced the design of the language. At the same time, we compare the language presented in this section with the one introduced in Section 9.1.1. As mentioned earlier, the aim was to develop a concurrent quantum language that could be equipped with a quantum dynamic logic. The logic was inspired by the work of Tsubasa [103], where quantum states are represented as vector states. Baltag also developed a quantum dynamic logic [10, 11]. However, in

Baltag's work quantum states are *rays*, *i.e.* one-dimensional subspaces

$$s = \{\lambda \cdot x \mid \lambda \in \mathbb{C},\ x \in \mathcal{H}\backslash\{0\}\}$$

of a Hilbert space $\mathcal{H}$. By ignoring global phases, rays capture the physical equivalence of quantum states, similarly to density matrices. However, developing an operational semantics based on rays would require more work than the usual approach with vector states or density operators. For instance, when considering vector states, applying the Z gate to the state $|1\rangle$ yields $-|1\rangle$. In contrast, when using rays, the result of the same experiment would be $|1\rangle$, since the global phase is ignored. This discrepancy complicates the formulation of an operational semantics. For this reason, we adopt Tsubasa's logic. Consequently, the design of the language is shaped by the requirements of the logic. This leads to three immediate consequences:

1. **State representation**

   Unlike the approach in Section 9.1.1, which used density operators, we resort here to pure states.

2. **Simplified operational semantics**

   Since the logic does not model probabilities, the operational semantics has to deal only with non-deterministic effects. Consequently, the small-step operational semantics becomes the standard semantics used for classical concurrent languages (recall the semantics used when introducing states in the core language in Section 8.1). Furthermore, we can define a big-step semantics without resorting to schedulers.

3. **Restrictions on language constructs**

   Because the logic does not account for probabilities and does not support cyclic behavior, the language in this section omits the probabilistic choice and the $while$ loop. Nevertheless, since the $await$ command may idle indefinitely, we assume fairness, ensuring that every program in parallel will eventually be executed [1].

We consider at our disposal a finite number of qubits N whose associated space is $\mathbb{C}^{2^N}$, a Hilbert space $\mathcal{H}$ over the complex numbers. Each qubit is identified by a natural number n and we let $\vec{n}$ be a sequence of non-repetitive natural numbers, denoting multiple qubits. We assume a set of unitary gates on $\mathcal{H}$, denoted as $\mathcal{U}(\mathcal{H})$, and use U to represent an element of said set. The syntax of the language

[1] When the condition of an $await$ command is always false, the command will always idle. However, for the command to idle, it must be given a chance to do so, which is ensured by the fairness assumption.

used in [54, 55] is given by the following BNF grammar:

$$C ::= skip \mid U(\vec{n}) \mid \texttt{M}(n, C_1, C_2) \mid C_1 \,;\, C_2 \mid C_1 \,\texttt{or}\, C_2 \mid C_1 \,||\, C_2 \mid \texttt{await}\ \ (n, D)$$
$$D ::= skip \mid U(\vec{n}) \mid \texttt{M}(n, D_1, D_2) \mid D_1 \,;\, D_2$$

where $skip$ is a command that does nothing, $U(\vec{n})$ applies the unitary gate U to the list of qubits $\vec{n}$ (*e.g.* $CX(1,2)$ applies the CNOT-gate to qubits 1 and 2), $\texttt{M}(n, C_1, C_2)$ represents a measurement of qubit n in the computational basis, *i.e.* $\{|0\rangle, |1\rangle\}$, and if the result of the measurement is $|0\rangle$ then we execute C_1, otherwise we execute C_2 (the measurement command can be seen as $\Phi_0(n)? \,;\, C_1$ or $\Phi_1(n)? \,;\, C_2$, where $\Phi_0(n)?$ is the projection of qubit n to $|0\rangle$ and $\Phi_1(n)?$ is the projection of qubit n to $|1\rangle$), $C_1 \,;\, C_2$ is the sequential composition of programs, C_1 or C_2 is the non-deterministic choice of programs, $C_1 \,||\, C_2$ is the parallel composition of programs, and $\texttt{await}\ \ (n, D)$ is the await command, where n is an *await qubit* and D is restricted to a sequence of skips, unitary gates, and/or measurements [2]. An await qubit is an auxiliary qubit, used for the $await$ command, that determines if the instructions D are executed or not. Furthermore, we assume that an await qubit behaves as a bit, *i.e.* its state is either $|0\rangle$ or $|1\rangle$.

The description of the await command is inspired by [21]. Intuitively, the await command is a conditional command that executes instructions with no interruptions when a condition is met. In the classical setting, verifying this condition entails accessing the memory, a process that can occur infinitely many times without collapsing the state. However, in the quantum setting, accessing the memory (*i.e.* measuring a qubit) entails perturbing the state, leading to its collapse. To prevent this collapse, we introduce auxiliary qubits, called *await qubits*, which serve as control qubits for the $await$ condition. These await qubits are assumed not to be in superposition, *i.e.* they behave like classical bits. Moreover, each $await$ command is associated with its own distinct await qubit. For instance, if a program C contains two $await$ commands, then two separate await qubits are introduced.

To furnish the language with an operational semantics we use configurations, *i.e.* a pair formed by a command C and a state $s \in \mathbb{C}^{2^N}$, denoted as $\langle C,\ s\rangle$. We let *Conf* be the set of configurations and we use S to denote $\mathbb{C}^{2^N}$. Intuitively, the small-step semantics represents the execution of a single instruction in a program. Formally, we define the small-step semantics as being the relation $\longrightarrow \subseteq \mathit{Conf} \times (S + \mathit{Conf})$ satisfying the rules in Figure 80. A transition by the small-step semantics can be of two forms: $\langle C,\ s\rangle \longrightarrow$

[2] In Brookes' work [21], instructions inside an $await$ command are sequences of $skip$ and/or assignment commands. Having this in mind, when we develop the language of this section, the inclusion of measurements inside the $await$ command seemed both odd and natural. It seemed odd because measurements give rise to non-deterministic transitions, while Brookes only has deterministic instructions. It seemed natural because measurements play a crucial role in quantum programming, as they allow us to extract information from quantum states. Furthermore, since the result of a measurement depends on the current quantum state, executing them immediately after a sequence of actions without risk of interference is only possible if they are permitted inside the $await$ command.

$\langle C',\ s'\rangle$ or $\langle C,\ s\rangle \longrightarrow s'$. The former tells us that after executing one instruction from C, the state s is altered to s' and the remaining command to be executed is C'. The latter is a special case of the former, saying that no more actions are left to perform, thus returning the final state of the computation s'.

With the introduction of $await$ commands, we also introduced await qubits, which serve as auxiliary control qubits. Hence we let $s = u \otimes u_{\text{await}}$ be the general state associated to a command C, where u is the quantum state corresponding to C (ignoring await qubits) and u_{await} is the state of the auxiliary await qubits. To make the rules of the await command in Figure 80 clear, we assume that we are dealing with m normal qubits (indexed by $1, \ldots, m$) and m' await qubits (indexed by $(m+1) \ldots (m+m')$).

Most of the rules in Figure 80 were already introduced in Section 8.1. Since they are also presented in [92, Chapters 6 and 8], we refer the interested reader there for detailed explanations. The only exceptions are the rules for measurement and await, which we explain here. We start with measurements. Recall that the measurement command can be seen as $\Phi_0(n)?\ ;\ C_1$ or $\Phi_1(n)?\ ;\ C_2$, where $\Phi_0(n)?$ and $\Phi_1(n)?$ are the projections for qubit n onto $|0\rangle$ and $|1\rangle$, respectively. Hence, from $\langle \mathtt{M}(q, C_1, C_2),\ s\rangle$ we can transit to $\langle C_1,\ s_0\rangle$, where s_0 is the resultant state of applying the projection $\Phi_0(n)?$ to s, or we can transit to $\langle C_2,\ s_1\rangle$, where s_1 is the resultant state of applying the projection $\Phi_1(n)?$ to s. In this sense, the measurement behaves analogously to an if-then-else statement: the outcome determines which branch of the computation is executed.

Now consider the rules for the $await$ command. In the rule $(aw1)$, if the await qubit n is in state $|0\rangle$, the computation idles, *i.e.* it transitions back to the same configuration. Intuitively, the condition required to execute D is not yet satisfied. In the rule $(aw2)$, if the await qubit n is in state $|1\rangle$, then we can execute D. Note that, D is executed as an atomic action, since its evaluation is performed using the big-step semantics (Figure 81). After D is executed, the await qubit n is reset to $|0\rangle$. This reset can be understood through an analogy: think of the await qubit as a reserved parking spot. When the qubit is set to $|1\rangle$, the spot is occupied, and no other process can use it. Once the program D finishes, the qubit is flipped back to $|0\rangle$, signaling that the spot is free again and can be reserved by other computations [3].

We also equip the language with a big-step semantics, which equals the one shown in Figure 71 in Section 8.1. Informally, the big-step semantics tells us that executing the command C with state s gives s' as a possible final state. Note that we said "as a possible final state" and not "as the final state". This is due to the presence of non-deterministic behaviors in our language, which are introduced by the non-

[3] When designing the $await$ command (specifically, the mechanism for resetting its control qubit) we also considered how $await$ would behave inside a $while$ loop (a possible future extension of the language, even without quantum dynamic logic). In such a setting, the instructions inside the $await$ command may be executed multiple times. Each time the loop iterates, the guard of the $await$ command must be evaluated again. For this to work correctly, the auxiliary await qubit must be reset after each execution. Otherwise, its previous value could incorrectly influence subsequent iterations.

$$\frac{}{\langle skip,\ s\rangle \longrightarrow s}\ (sk) \qquad \frac{}{\langle U(n),\ s\rangle \longrightarrow U_n s}\ (un)$$

$$\frac{}{\langle \mathtt{M}(n, C_1, C_2),\ s\rangle \longrightarrow \langle C_1,\ s_0\rangle}\ (meas_0) \qquad \frac{}{\langle \mathtt{M}(n, C_1, C_2),\ s\rangle \longrightarrow \langle C_2,\ s_1\rangle}\ (meas_1)$$

$$\frac{\langle C_1,\ s\rangle \longrightarrow s'}{\langle C_1\,;\, C_2,\ s\rangle \longrightarrow \langle C_2,\ s'\rangle}\ (seq1) \qquad \frac{\langle C_1,\ s\rangle \longrightarrow \langle C_1',\ s'\rangle}{\langle C_1\,;\, C_2,\ s\rangle \longrightarrow \langle C_1'\,;\, C_2,\ s'\rangle}\ (seq2)$$

$$\frac{\langle C_1,\ s\rangle \longrightarrow s'}{\langle C_1 \,||\, C_2,\ s\rangle \longrightarrow \langle C_2,\ s'\rangle}\ (parL1) \qquad \frac{\langle C_1,\ s\rangle \longrightarrow \langle C_1',\ s'\rangle}{\langle C_1 \,||\, C_2,\ s\rangle \longrightarrow \langle C_1' \,||\, C_2,\ s'\rangle}\ (parL2)$$

$$\frac{\langle C_2,\ s\rangle \longrightarrow s'}{\langle C_1 \,||\, C_2,\ s\rangle \longrightarrow \langle C_1,\ s'\rangle}\ (parR1) \qquad \frac{\langle C_2,\ s\rangle \longrightarrow \langle C_2',\ s'\rangle}{\langle C_1 \,||\, C_2,\ s\rangle \longrightarrow \langle C_1 \,||\, C_2',\ s'\rangle}\ (parR2)$$

$$\frac{\langle C_1,\ s\rangle \longrightarrow s'}{\langle C_1 \,\mathsf{or}\, C_2,\ s\rangle \longrightarrow s'}\ (ndL1) \qquad \frac{\langle C_1,\ s\rangle \longrightarrow \langle C_1',\ s'\rangle}{\langle C_1 \,\mathsf{or}\, C_2,\ s\rangle \longrightarrow \langle C_1',\ s'\rangle}\ (ndL2)$$

$$\frac{\langle C_2,\ s\rangle \longrightarrow s'}{\langle C_1 \,\mathsf{or}\, C_2,\ s\rangle \longrightarrow s'}\ (ndR1) \qquad \frac{\langle C_2,\ s\rangle \longrightarrow \langle C_2',\ s'\rangle}{\langle C_1 \,\mathsf{or}\, C_2,\ s\rangle \longrightarrow \langle C_2',\ s'\rangle}\ (ndR2)$$

$$\frac{}{\begin{array}{l}\langle \mathtt{await}\ \ (n, D),\ u \otimes |(m+1)\ldots 0\ldots (m+m')\rangle\rangle \longrightarrow \\ \qquad\qquad \langle \mathtt{await}\ \ (n, D),\ u \otimes |(m+1)\ldots 0\ldots (m+m')\rangle\rangle\end{array}}\ (aw1)$$

$$\frac{\langle D,\ u \otimes |(m+1)\ldots 1\ldots (m+m')\rangle\rangle \twoheadrightarrow u' \otimes |(m+1)\ldots n\ldots (m+m')\rangle}{\begin{array}{l}\langle \mathtt{await}\ \ (n, D),\ u \otimes |(m+1)\ldots 1\ldots (m+m')\rangle\rangle \longrightarrow \\ \qquad\qquad u' \otimes |(m+1)\ldots 0\ldots (m+m')\rangle\end{array}}\ (aw2)$$

Figure 80: Rules of the small-step operational semantics

deterministic and parallel operators, as shown in Figure 82. Formally, the big-step semantics is defined as a relation $\twoheadrightarrow \subseteq \mathit{Conf} \times S$ satisfying the rules in Figure 81.

The rules in Figure 81 resemble the transitive closure of the small-step semantics. The $(trans1)$

$$\frac{\langle C,\ s\rangle \longrightarrow s'}{\langle C,\ s\rangle \twoheadrightarrow s'}\ (trans1) \qquad \frac{\langle C,\ s\rangle \longrightarrow \langle C',\ s'\rangle \qquad \langle C',\ s'\rangle \twoheadrightarrow s''}{\langle C,\ s\rangle \twoheadrightarrow s''}\ (trans2)$$

Figure 81: Rules of the big-step semantics

rule tells us that if the small-step semantics evaluates a configuration to a state, then the big-step mimics such an evaluation. Rule $(trans2)$ tells that if the small-step evaluates a configuration $\langle C,\ s\rangle$ to another configuration $\langle C',\ s'\rangle$, and this new configuration evaluates to a state s'' under the big-step semantics, then $\langle C,\ s\rangle$ also evaluates to s''. In other words, when the computation is not yet finished, the big-step semantics keeps following the remaining transitions until a final state is reached.

Example 9.2.1. To better understand how the parallel operator and the await command work, we depict the execution tree of $\langle (X(2)\,;\ \texttt{await}\ \ (2, \texttt{M}(1, skip, skip))) \,||\, H(1),\ |00\rangle\rangle$ in Figure 82. From the initial configuration, we have two possible transitions: either execute $X(2)$ or $H(1)$. By executing the latter, we transit to the configuration $\langle X(2)\,;\ \texttt{await}\ \ (2, \texttt{M}(1, skip, skip)),\ H_1|00\rangle\rangle$, where we write $H_1|00\rangle$, *i.e.* apply the Hadamard gate to qubit 1, instead of $\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$ due to space constraints. Subsequently, applying the X gate to the await qubit permits us to perform the measurement within the await command, yielding the final states $|00\rangle$ or $|10\rangle$. Note that the await qubit was flipped. On the other side, going to $\langle \texttt{await}\ \ (2, \texttt{M}(1, skip, skip)) \,||\, H(1),\ |01\rangle\rangle$ from the initial configuration gives us again two possible transitions: either executing the await command, possible because the await qubit is in state $|1\rangle$, followed by applying the Hadamard gate, resulting in a final state of $H_1|00\rangle$, or first applying the Hadamard gate and then executing the await command, which will measure the state, yielding the final states $|00\rangle$ or $|10\rangle$. Note that the $skip$ command does not alter the state. Hence we omit its action throughout the explanation.

9.2.2 Putting the language to test

Here we present three examples of quantum algorithms encoded by the language described in Section 9.2.1. The first algorithm is a simplification of the quantum version of the Dining Philosophers problem [5], the second algorithm is the quantum teleportation protocol [14], and the third one is the superdense coding [15] together with Grover's algorithm [48].

Example 9.2.2. The Dining Philosophers problem [32] is well known in the concurrency community, and probabilistic [65] and quantum [5] solutions for it have been developed. Since each version will be

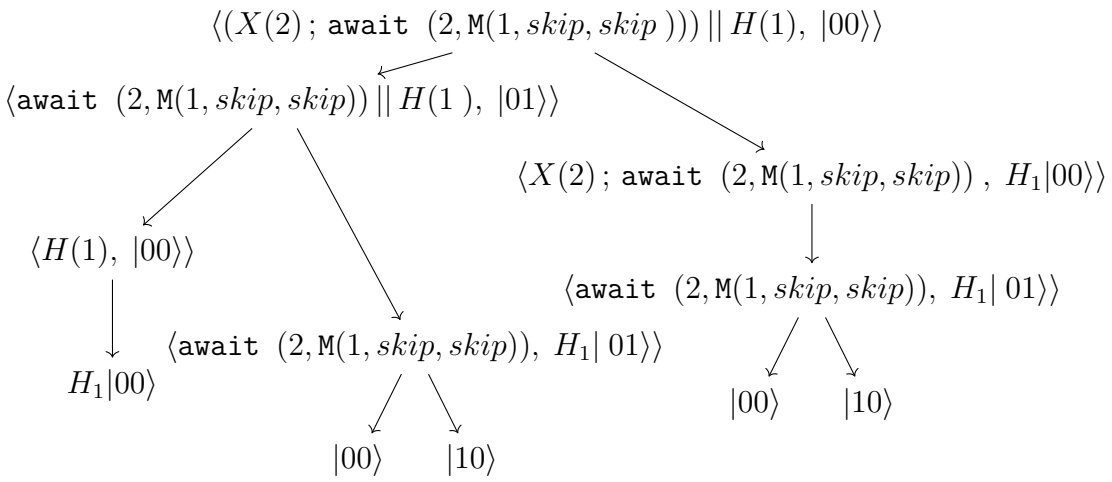


Figure 82: Execution tree of $\langle \big(X(2)\,;\,\texttt{await}\ \ (2,\texttt{M}(1,skip,skip))\big)\,||\,H(1),\ |00\rangle\rangle$

discussed in Section 10.4, we omit its full description here and focus instead on a simplified quantum version. Moreover, because the language introduced in Section 9.2.1 does not support while loops, the protocol is executed only once. To illustrate the need for the $await$ command, we consider only two philosophers.

The scenario is the following. Two philosophers enter a restaurant and request their meals. The restaurant, however, has limited cutlery, so the philosophers must share it. One philosopher must eat first, after which the cutlery is washed and passed to the other. To determine which philosopher eats first, we use a simplified form of [5, Algorithm 2]. Informally, this algorithm takes a symmetric state, such as $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ (both qubits agree), and breaks its symmetry to a state such as $\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$ (the qubits differ), which is then measured. This procedure requires a "magic unitary gate" U_n [104], defined as follows:

$$U_n = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & e^{-i\frac{\pi}{n}} \\ -e^{i\frac{\pi}{n}} & 1 \end{pmatrix}$$

Let us now give an overview of the algorithm in Figure 83, which is encoded in the language of Section 9.2. Philosopher$_i$, for $i \in \{0, 1\}$, has four qubits: three qubits ($4 \times i + 1$, $4 \times i + 2$, and $4 \times i + 3$) used for the symmetry-breaking protocol, and a fourth qubit ($4 \times i + 4$) serving as the guard of an $await$ command. The philosophers first entangle qubits $4 \times i + 1$ and $4 \times i + 2$. They then verify whether qubits $4 \times i + 1$ and $4 \times (i \oplus 1) + 2$ (where $\oplus$ denotes addition modulo 2) are equal, using qubit $4 \times i + 3$ as an ancilla (the ancilla qubit stores their parity). After this, qubit $4 \times i + 3$ is measured. If the outcome is $|0\rangle$, the symmetry has not been broken. Hence, a CNOT gate is performed on $4 \times i + 1$ and $4 \times (i \oplus 1) + 2$, followed by the application of the magic unitary U_n to $4 \times i + 1$. On the other hand,

if the outcome of measuring $4 \times i + 3$ is $|1\rangle$, the symmetry is already broken, and the philosopher that will eat first is already known. Once the identity of the first philosopher to eat is known, qubit $4 \times i + 1$ is measured and the result of such measurement is copied to the await qubit $4 \times i + 4$. If the measurement of $4 \times i + 1$ yields $|0\rangle$, then Philosopher$_i$ will wait for his turn to eat. Otherwise, Philosopher$_i$ is given access to the instructions inside the $await$ command, representing the moment where he gets to eat. When he finishes, he notifies Philosopher$_{i \oplus 1}$ by flipping qubit $4 \times (i \oplus 1) + 1$.

```
H(4 × i + 1);CNOT(4 × i + 1, 4 × i + 2);
CNOT(4 × i + 1, 4 × i + 3);CNOT(4 × (i ⊕ 1) + 2, 4 × i + 3);
M(4 × i + 3,
      CNOT(4 × i + 1, 4 × (i ⊕ 1) + 1); Umag2(4 × i + 1),
      Skip);
M(4 × i + 1, Skip, Skip);
CNOT(4 × i + 1, 4 × i + 4);
await(4 × i + 4, eat);
X(4 × (i ⊕ 1) + 1)
```

Figure 83: Code describing the behavior of Philosopher$_i$

In Figure 83, the `eat` component of the $await$ command can be regarded as a finite sequence of `skip` commands. Note that infinite computations are possible. Suppose Philosopher 1 has just finished eating and is about to notify Philosopher 2, who is waiting for permission to eat. If the program repeatedly attempts to evolve via Philosopher 2, the computation becomes infinite. Consequently, attempting to execute this program in Maude using the big-step semantics yields no result: since the big-step semantics explores all transitions, the presence of a single infinite branch causes the semantics to diverge.

This example therefore illustrates both the motivation for introducing the $await$ command and the limitations of the big-step semantics without schedulers.

Example 9.2.3 (Quantum Teleportation)**.** The quantum teleportation protocol allows us to transfer a quantum state without the direct use of quantum communication channels. Consider Alice and Bob, who share an entangled state and Alice wants to teleport a state she holds to Bob. To achieve this, Alice entangles the state to be transferred with her part of the shared entanglement. She then measures both

qubits and sends the results of these measurements to Bob. Depending on the results, Bob performs one of the following operations:

- If the measurement result is 00, Bob does nothing to his qubit.
- If the result is 01, he applies a Z gate to his qubit.
- If the result is 10, he applies an X gate to his qubit.
- If the result is 11, he applies the X gate and then the Z gate to his qubit.

This procedure can be encoded using the language in Section 9.2.1 as follows (notice the presence of a third person, Charlie, who is responsible to entangle the qubits that are shared by Alice and Bob):

```
Charlie = H(2); CNOT(2,3)
Alice   = CNOT(1,2); H(1)
Bob     = M(1, skip, X(3)); M(2, skip, Z(3))
qTele   = Charlie; Alice; Bob
```

with initial state given as $\psi = (a|0\rangle + b|1\rangle) \otimes |00\rangle$ where $a, b \in \mathbb{C}$ s.t. $|a|^2 + |b|^2 = 1$, with $a|0\rangle + b|1\rangle$ being the state to teleport.

In the protocol description, Bob applies corrections based on measurement results sent by Alice. In our encoding, however it is Bob that performs the measurements. This is because we assume a shared-variable memory where communication occurs by reading/writing in a common memory. Since measurement is the only way to read the memory, placing measurements in Alice's program would require repeating them (or similar ones) in Bob's program. To avoid redundancy, we place them directly in Bob's code.

To verify correctness, one can apply the big-step semantics (Figure 81) and check, step by step, that the protocol behaves as expected. A more practical alternative is to use the Maude implementation [53], which automates this verification.

Before presenting the example that combines superdense coding [15] with Grover's algorithm [48], let us first briefly recall what these algorithms do.

Superdense coding [15] allows two classical bits of information to be transmitted using a single entangled qubit. As in Example 9.2.3, we consider the participation of Alice, Bob, and Charlie, where Charlie prepares the entangled state shared between Alice and Bob. Alice encodes the two bits she wants to send by applying specific unitary gates to her qubit, according to the bit pair $b_1 b_2 \in \{00, 01, 10, 11\}$:

$$\texttt{Alice}_{00}\texttt{= skip}$$

$\texttt{Alice}_{01}$= X(1)

$\texttt{Alice}_{10}$= Z(1)

$\texttt{Alice}_{11}$= X(1); Z(1)

Concretely, if Alice wants to send the bits 00 then she does not apply any operation ($\texttt{Alice}_{00}$). If she wants to send the bits 01, then she applies the gate X to her qubit ($\texttt{Alice}_{01}$). If she wants to send the bits 10, she applies the Z gate to her qubit ($\texttt{Alice}_{10}$). At last, to send the bits 11 she applies first the X gate and then the Z to her qubit ($\texttt{Alice}_{11}$).

Using the language in Section 9.2.1, the specification of the superdense coding is as follows:

Alice = $\texttt{Alice}_{00}$ or $\texttt{Alice}_{01}$ or $\texttt{Alice}_{10}$ or $\texttt{Alice}_{11}$ or

Bob = CNOT(1,2); H(1); M(1,skip,skip); M(2,skip,skip)

Charlie = H(1); CNOT(1,2)

sdc = Charlie; Alice; Bob

where we consider $\psi = |00\rangle$ to be the initial state.

Grover's algorithm [48] performs searches on unsorted databases. It is comprised of three parts:

1. Initialization: apply Hadamard gates to the qubits representing the database elements, creating a uniform superposition, and prepare ancilla qubits.

2. Oracle query: flip the phase of the target element using an oracle.

3. Diffusion operation: amplify the amplitude of the target state.

For a database with N elements, the oracle and diffusion steps are repeated about $\sqrt{N}$ times [82, Chapter 6].

For our purposes we consider a database with four elements (00, 01, 10, and 11), which we can encode using two qubits. Furthermore, to encode the Grover algorithm, we need an extra qubit, an ancilla qubit. The sequential version of Grover's algorithm can be specified as follows (using CCX, the Toffoli gate, in the oracle):

init = H(1); H(2)

anc = X(3); H(3)

$\texttt{oracle}_{ij}$ = $\texttt{X}^{(1-i)}$(1); $\texttt{X}^{(1-j)}$(2); CCX(1,2,3); $\texttt{X}^{(1-i)}$(1); $\texttt{X}^{(1-j)}$(2)

diff = H(1); X(1); H(2); X(2); CZ(1,2); X(1); H(1); X(2); H(2)

$\texttt{grover}_{ij}$ = init; anc; $\texttt{oracle}_{ij}$; diff

where $i, j \in \{0, 1\}$ and we consider $\psi = |000\rangle$ to be the initial state.

Each oracle is defined according to the element being searched. For example, to find the element 11, one uses $\texttt{grover}_{11}$, which entails using $\texttt{oracle}_{11}$, which is simply `CCX(1,2,3)`. Through Maude we were able to confirm that both the superdense coding and Grover's algorithm work as expected. This can be checked in [53].

We are now ready to combine superdense coding with Grover's algorithm.

Example 9.2.4 (Superdense Coding together with Grover's Algorithm)**.** The superdense code protocol allows Alice to send two classical bits using a pair of entangled qubits, while Grover's enables Bob to search for an element in an unstructured database. By combining them, Alice can tell Bob, via superdense coding, which element to search for, and Bob can then use Grover's algorithm to locate it. The interaction can be illustrated as follows: Alice wishes to find an item in her database and asks Bob for help. Using superdense coding, Alice sends Bob the index of the item she wants. Bob then applies Grover's algorithm to search for that item.

In the sequential specification, we first execute the superdense coding protocol and then Grover's algorithm. One modification is needed in the oracle. In the earlier explanation of Grover's algorithm, we introduced $\texttt{oracle}_{ij}$, which depends on the target element (i, j). This requires a different Grover instance for each element. To avoid this duplication, we take advantage of the information sent via superdense coding. Concretely, the choice of the correct oracle is automated by nesting two measurement commands – one for each bit sent by Alice. The resulting specification is (qubits 1 and 2 are reserved for superdense coding, while Grover's qubits are shifted from 1, 2, and 3, respectively to 3, 4, and 5):

```
oracle  = M(1,
            M(2, oracle00, oracle01),
            M(2, oracle10, oracle11)
           )
grover  = init; anc; oracle; diff
sdcGrov = sdc; grover
```

By resorting to the Maude implementation [53], we verified that the algorithm works as intended.

To introduce concurrency in the superdense coding, Grover's algorithm, and in their mix, we targeted operations that commute. This way, concurrency does not alter the final result but the order in which some commands are executed is unknown. The specification of the algorithms is as follows:

```
sdcP     = Charlie; Alice; CNOT(1,2); H(1);
           (M(1, skip, skip) || M(2, skip, skip))
initP    = H(1) || H(2)
diffP    = ((H(1); X(1)) || (H(2); X(2))); CZ(1,2);
           ((X(1); H(1)) || (X(2); H(2)))
sdcGrovP = (sdcP || initP || anc); oracle; diffP
```

Once again, by resorting to the Maude implementation [53], we ran the program and confirmed that the algorithm works as it should. However, a practical issue emerged: performance. Running the concurrent version required waiting several weeks to obtain the result.

Although the details of the Maude implementation are not the focus of this thesis, we can briefly explain the performance issue. Introducing concurrency increases the number of steps that the big-step semantics, as implemented in Maude, must explore in order to compute the final results. The more actions are executed in parallel, the more the number of iterations, and consequently the longer Maude takes to return an answer. In addition, the Maude implementation of the big-step semantics includes a function that removes repeated states to improve the readability of the output. While this "beautification" step makes the results easier to interpret, it also adds extra computational overhead, further increasing the runtime.

Chapter 10

An interpreter for the concurrent (quantum) language

10.1 Brief description

With the theoretical foundations in place, we are now ready to present the tool that was developed. The tool's code is available in a public Git repository [37], where the reader can also consult the full source code. It was developed in collaboration with Inês Dias in her MSc dissertation [31], which was co-supervised by the author of this thesis. The tool implements the language introduced in Section 9.1, having in mind both classical and quantum instantiations, and includes the $await$ command. It is implemented in Haskell and is composed of two main components: a parser and an implementation of the semantics introduced in Section 8.2. A reason to adopt Haskell is that the implementation of the rules in Haskell closely resembles their formal definitions, providing a direct correspondence between theory and practice. Furthermore, Haskell encodes computational effects using monads, an idea that traces back to Moggi [77, 79], making it a natural fit for capturing the effects required by the semantics. Finally, Haskell's strong type system ensures that any incompatibilities in the data being evaluated are detected at compile time, thereby increasing reliability.

Let us give a brief overview of the tool. For that we resort to Figure 84, which aims to show how the tool works. The tool takes as input a file containing one or more programs together with the auxiliary information required to run them. Each program specifies:

- the number of samples and the variables of interest for building a histogram
- the number of steps to be performed by the big-step semantics (recall Figure 73)
- a configuration, consisting of a command, an initial classical state, a linking function, and an initial quantum state

The parser processes the input file and translates its contents into corresponding Haskell data structures. Each configuration, once encoded in Haskell, is then evaluated by the semantics together with the specified

number of steps. Since the big-step semantics in Figure 73 relies on schedulers, a scheduler must also be provided to guide the evaluation. At last, the tool provides two forms of output. First, it can return valuations, which directly correspond to the results of the big-step semantics. Alternatively, it can produce histograms, built from the number of samples and the variables specified in the input file.

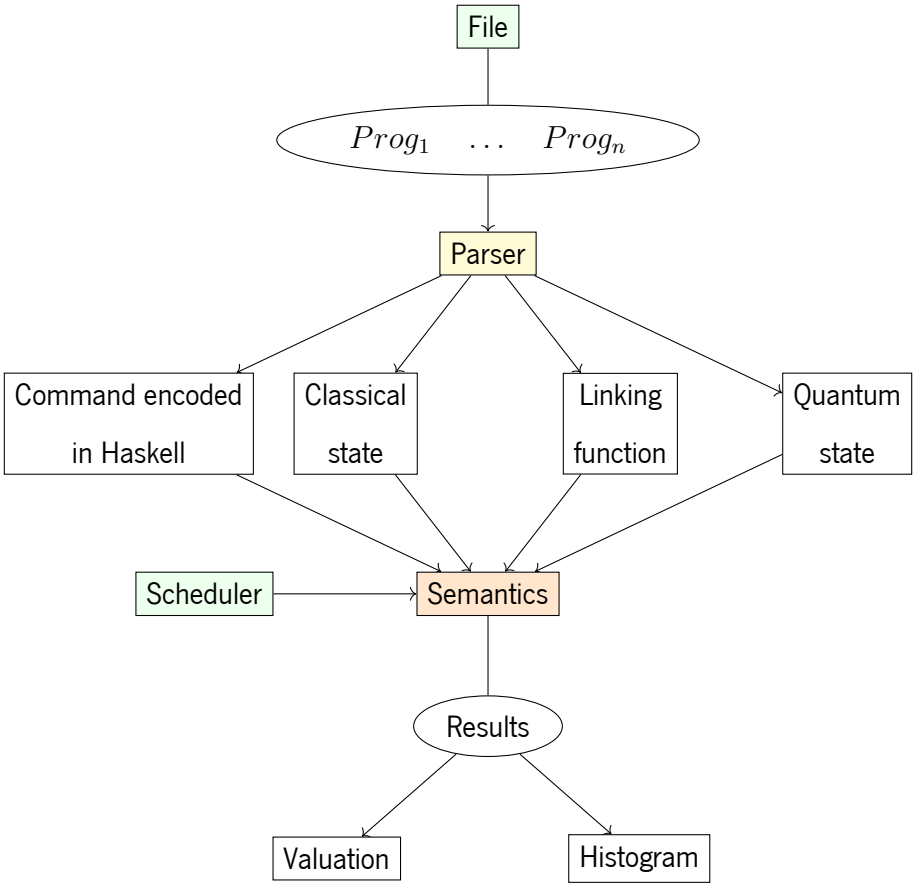


Figure 84: Tool workflow in a nutshell

To complement the information provided by Figure 84, let us illustrate how the tool works by considering a concrete example. Specifically, we present the format of a program in the input file and show the corresponding outputs. For that, let us consider a program that encodes two quantum coin tosses in parallel.

```
---QtCoinTossConc---
hist: (100, [[x1],[x2]])
k: 10
<
  H(q1); Meas(x1,q1) || H(q2); Meas(x2,q2),
  [(x1,0),(x2,0)],
```

```
  [(q1,1),(q2,2)],
  (1.0)|00><00|
>
---QtCoinTossConc---
```

The information associated to each command is between two marks that indicate the start and end of the program's definition. In this case, we have `---QtCoinTossConc---`. After writing the starter marker, the information related to the histogram is given. This is represented by `hist: (_, _)`, which is a pair composed of the number of samples and the variables considered to build the histogram. In this example, we wrote `[[x1],[x2]]`, meaning that two histograms will be produced: one for the variable `x1` and the other for the variable `x2`. We could also write `[x1,x2]`. In this case, only one histogram is output, combining the values of both variables, as illustrated in Figure 87. The next information to be provided is related to the number of steps the big-step semantics is able to perform. This is represented by `k: _`. Then, we specify the configuration, which includes the command to be performed, the initial classical state, the linking function, and the initial quantum state. Note that this information is enclosed between the delimiters < and > to resemble configurations typically handled on the theoretical side. For this case, the command to be performed is as follows: the Hadamard gate is applied to qubit `qi`, which is then measured with the result stored in the classical variable `xi` (`i` $\in \{1, 2\}$). The classical variables are initialized at 0. The linking function [1] assigns to the quantum variables `q1`, `q2` the values 1 and 2, respectively, and the initial quantum state is `(1.0)|00><00|`. After this, we write again `---QtCoinTossConc---` to indicate the ending of the program.

Now that all information concerning a program is given, the program is ready to be parsed. After being parsed, the information is represented as Haskell data types. We are then prepared to evaluate the command using the implemented semantics. Now, depending on the desired output format, we may need to provide a scheduler as input. Specifically, suppose the chosen output is a valuation. In that case, it is also necessary to provide a scheduler, which should be defined by the user. On the other hand, if the output chosen is a histogram, then the user does not need to give a scheduler. This is because a predefined scheduler is used in the creation of the histogram. In Section 10.2, we shall give more details about these two output formats. Let us first show the result obtained when opting for valuations. Recall that we need to provide a scheduler. The scheduler used is a simple one: it always executes the leftmost command. The results obtained are as follows:

[1] The linking function, inspired by Peter Selinger and Benoît Valiron [96], is a bijective function from a set of quantum variables $qVar$ to $\{1, \ldots, n\}$, with n being the number of quantum variables. This provides a handy way to locate a quantum variable within the quantum state.

```
QtCoinTossConc:
0.25·([(x1,0),(x2,0)], (1.0)|00><00|) +
0.25·([(x1,0),(x2,1)], (1.0)|01><01|) +
0.25·([(x1,1),(x2,0)], (1.0)|10><10|) +
0.25·([(x1,1),(x2,1)], (1.0)|11><11|)
```

Recall that a valuation is of the form $\sum_i p_i \cdot x_i$, where p_i is the probability associated with the element x_i. Instantiating to our tool, each x_i is a tuple composed of a classical state and a quantum state. The classical state is a list of pairs, where each pair stores the classical variable and its respective value. The quantum state is a density operator. Thus, the valuation obtained indicates that the possible states of the two coins are: $|00\rangle$, $|01\rangle$, $|10\rangle$, or $|11\rangle$, each with a probability of 0.25.

Now let us show the output obtained if we chose histograms instead. Recall, from the specification of the program, that we are going to collect 100 samples. Figure 85 and Figure 86 show the histograms obtained for the variables `x1` and `x2`, respectively.

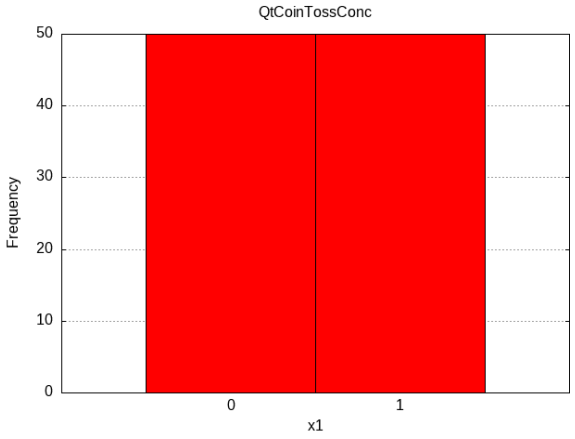


Figure 85: Histogram for the variable `x1`

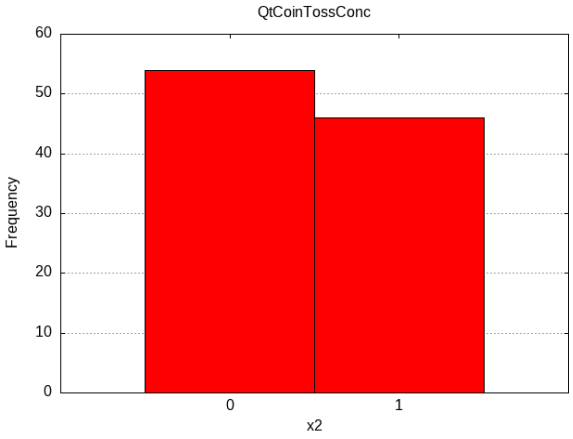


Figure 86: Histogram for the variable `x2`

In Figure 85 and Figure 86, the x-axis – labeled `x1` and `x2`, respectively – displays the states that were obtained. The y-axis, labeled by `Frequency`, indicates how often each state occurred. In Figure 85, the two states, 0 and 1, appear with the same frequency. In contrast, Figure 86 shows that state 0 occurs more often that state 1: the former is recorded slightly more than 50 times, while the latter appears slightly more than 40.

If we wrote `hist: (100, [x1,x2])` instead of `hist: (100, [[x1],[x2]])`, we would obtain a histogram that in the x-axis has the combined values of the variables `x1` and `x2`, as can be seen in Figure 87.

Let us now analyze the architecture of the tool by providing an overview of the modules that compose

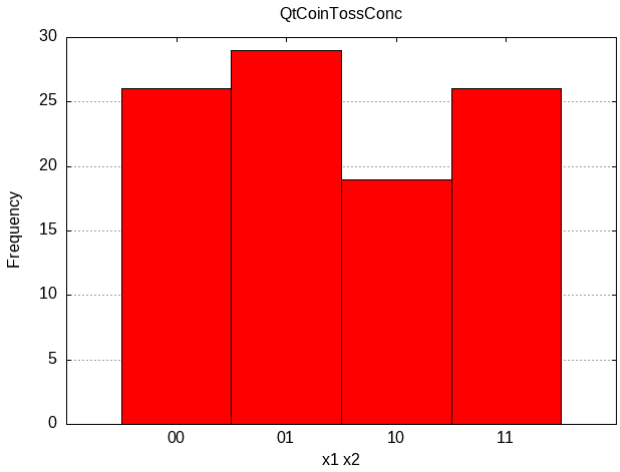


Figure 87: Histogram for the variables `x1` and `x2`

it. These modules can be grouped into five categories: syntax, parsers, semantics, utilities, and runner [2].

The Syntax module

The *Syntax* module contains the information about how states, the linking function, integer and Boolean expressions, and commands are encoded in Haskell. Furthermore, it also tell us which unitary gates can be used.

Recall that a classical state is a set of functions from classical variables to integers, that a quantum state is a density operator, and that the linking function is a bijective function between qubits and integers. In Haskell, classical states are encoded as a list of pairs composed of a classical variable and an integer value, *i.e.* `[(String, Integer)]`. Quantum states are encoded using `Matrix (Complex Double)`, which is a matrix of complex numbers. The linking function is represented as `[(String, Int)]`, *i.e.* a list of pairs formed by a quantum variable and an integer. Note that in the classical memory we used `Integer` and in the linking function we used `Int`. The use of `Integer` for classical variables allows unbounded integers, while `Int` in the linking function is sufficient since we consider n qubits at our disposal (recall that information from Section 9.1.1, where we showed the quantum instantiation made in [80]). Classical states, quantum states, and the linking function are defined as a "type" in Haskell, as shown next:

```
type SC = [(String, Integer)]
type L = [(QVar,Int)]
type SQ = Matrix (Complex Double)
```

[2] We used the name "runner" because the name of the respective module is "Run"

where SC represents classical states, L denotes the linking function, and SQ represents quantum states. In the linking function, we have the presence of QVar which is a string (`type QVar = String`).

It is also in the *Syntax* module that we make two crucial decisions: (1) the form of states and (2) how to encode probabilities.

1. In Section 9.1 we instantiated the language described in Section 8.1 to the classical setting where states are seen as a function from classical variables to integers. Later on, in Section 9.1.1 we instantiated the language to the quantum setting, where the state is now a classical-quantum state given by a density operator. In order to accommodate the classical and quantum instantiation in a unified framework, the classical memory in a classical-quantum state is represented in Haskell as `[(String, Integer)]`. Hence, the memory associated to a program is split in classical and quantum memories, where the quantum memory needs to be accompanied by a linking function.

2. Probabilities associated to the probabilistic command are fractional numbers, while in the quantum setting, the use of irrational numbers is needed in order to encode the Hadamard gate (recall that the Hadamard gate contains the coefficient $\frac{1}{\sqrt{2}}$). This puts us in a place where we need to choose how to represent probabilities in Haskell. In other words, we could adopt `Rational`, used to represent fractional numbers but not able to capture irrational numbers, or we could use `Double`, which captures fractional and irrational numbers with the cost of losing precision. Since with `Double` it is possible to capture both irrational and fractional numbers, we opt to use `Double` to represent probabilities. This means that, whenever we write a probabilistic command, its probability is inputted as a fractional number, which is then converted to a `Double`. In the implementation, probabilities are captured by the "type" signature, *i.e.* `type Prob = Double`.

Parsing modules: ParserAEBE, ParserCom, and ParserFile

The parsing infrastructure is based on the Parsec library [66], a monadic parser combinator framework in Haskell.

- *ParserAEBE*: parses integer and Boolean expressions, as well as auxiliary constructs like whitespace.

- *ParserCom*: parses commands built with the language in Section 9.1.

- *ParserFile*: parses entire input files, combining program metadata (histograms, step bounds, initial configuration) with the parsed commands.

This layered parsing makes programs more user-friendly. For example, we can simply write the command of the `QtCoinTossConc` example as:

```
H(q1); Meas(x1,q1) || H(q2); Meas(x2,q2)
```

instead of its verbose Haskell encoding:

```
Paral (Seq (U H ["q1"]) (Meas ("x1","q1")))
(Seq (U H ["q2"]) (Meas ("x2","q2")))
```

The details of the parser implementation can be found in Appendix B.

Semantics modules: SemBE_Brookes, SmallStep, and KStep

The semantics of the language is captured across three modules:

- *SemBE_Brookes*: encodes the big-step semantics of integer and Boolean expressions (Figures 76 and 77).
- *SmallStep*: implements the small-step semantics of commands (Figure 73). This module depends on:
  - *DistTMonad*, which uses the *mtl* package [45] to encapsulate probabilistic behavior and combine it with other effects.
  - *QuantumCalc*, which provides the core quantum operations: applying unitary gates and measurements to quantum states.
- *KStep*: implements the big-step semantics (Figure 81) based on repeated small-step execution.

Additionally, the *HistogramSem* module builds histograms from big-step evaluations.

Utility modules: Beautify

The raw outputs of the semantics, specifically density matrices, are not easy to read. The *Beautify* module formats them into more accessible forms. For example, quantum states are displayed using Dirac's bra-ket notation rather than raw matrices, making results easier to interpret.

The Run module

All of the modules we previously mention, serve as the backbone for the module that joins everything: the *Run* module. This module has two primary functions: runHist and runSem. The former generates a histogram provided a file path, while the latter returns a valuation from the big-step semantics, given a file path and a scheduler. For example, the outputs shown earlier for the `QtCoinTossConc` program were obtained by running:

```
runSem "examples/qtcoin.txt" initSch
runHist "examples/qtcoin.txt"
```

More details on how to use the tool can be found in [37].

10.2 Implementation of the semantics

We now shift our focus to the implementation of the semantics. Recall the type of the operational semantics $\longrightarrow \subseteq (Com \times S) \times \mathsf{V}_1(S + (Com \times S))$, which can be seen as a function, as follows: $\longrightarrow : (Com \times S) \to \mathsf{P}(\mathsf{V}_1(S + (Com \times S)))$. In the codomain, we can see the following effects being combined: the state effect, the exception effect, the probabilistic effect, and the non-deterministic effect. Following the ideology of Moggi [77], those effects are captured in Haskell resorting to monads. Hence, the state effect is captured by the state monad, the exception effect is captured by the exception monad, the probabilistic effect is captured by the probabilistic monad, and the non-deterministic effect is captured by lists. The probabilistic monad is implemented as a list of pairs, each consisting of an element and its associated probability. To combine all these behaviors in Haskell we resort to monad transformers (recall Remark 5). By convention, a monad transformer is identified by appending "T" to the name of the corresponding effect. For instance, the transformer version of the state effect is written as `StateT`. Recall that in Haskell, defining a `newtype` introduces a constructor that wraps the underlying data. As mentioned in the brief introduction to Haskell (Section 8.3), a `newtype` can be defined either with or without record syntax. When we use record syntax, we get for free a "deconstructor" function [3] to extract such data. In contrast, when record syntax is not used, we must define this deconstructor function. In the *mtl* package, these deconstructors are typically named by prefixing the effect with "run". For example, the deconstructor for `StateT` is `runStateT`.

The order in which effects are stacked is important. To determine whether the chosen order is appropriate, we can inspect the behavior of the corresponding deconstructor. To illustrate this, consider

[3] Recall that in Section 8.3 we mention the use of "deconstructor" to refer to "unwrapping".

`StateT`, `State`, `ExceptT`, and `Either`, which are already defined in the *mtl* package, where `ExceptT` is the monad transformer version of `Either`[4]. Furthermore, let us consider a set of states `S`, an arbitrary set `A`, and a set of exceptions `E`. When writing `StateT S (Either E A)`, we are stacking the exception effect under the state effect. Applying `runStateT` yields:

```
StateT S (Either E A)
        |
        | runStateT
        v
S -> Either E (A, S)
```

This tells us that we receive a state which can either raise an exception or return a value together with a (possibly updated) state. This can also be seen as $S \to E + (A \times S)$. On the other hand, if we change the stacking order to `ExceptT E (State S A)` and apply `runExceptT`, we obtain:

```
ExceptT E (State S A)
        |
        | runExceptT
        v
Either E (State S A)
```

This indicates that we either raise an exception or produce a stateful computation. As we can see, different stacking orders give different behaviors. Therefore, we must be careful when stacking effects to ensure that the resulting semantics captures the intended behavior.

The semantics has a small-step and a big-step version. Let us focus on the small-step first. To do that, we need to stack four effects: state changes, exception handling, probabilistic, and non-deterministic behavior, in exactly this order. In other words, non-determinism lies at the bottom, under probabilities, which in turn lie under exceptions, which are finally placed under the state effect. From the *mtl* package we already have `StateT`, for states, and `ExceptT`, for exceptions. Since non-determinism is at the bottom, we can simply use lists. Now, it lacks to capture the probabilistic effect. But, since the non-deterministic effect is under the probabilistic, we need to define a probabilistic monad transformer. To do it, we first define a new type together with its deconstructor.

```
newtype DistT m a = DistT { runDistT :: m [(a, Double)] }
```

In sum, values of type `DistT m a` are wrapped around the monad `m`, which means their concrete form depends on the choice of `m`. For instance, if `m` is the `IO` monad, then values of type `DistT IO a` are `IO`

[4] Since **`ExceptT`** is the monad transformer version of **`Either`**, following the convention of adding a capital "T", one would expect to have **`EitherT`**. However that does not occur.

computations that produce lists of pairs `(a, Double)`, where each value `a` is associated with a probability. As another example, if `m` is the `Maybe` monad, then values of type `DistT Maybe a` are either a `Just`-wrapped list of such pairs, *e.g.* `DistT (Just [(a, 0.5), (b, 0.5)])`, or `DistT Nothing`, that is, a computation that returns `Nothing`.

Now we define `DistT` as an instance of the Functor class, restricting `m` to be a monad:

Definition 10.2.1.

```
instance (Monad m) => Functor (DistT m) where
  fmap f (DistT mdist) = DistT $ do
    dist <- mdist
    return [(f x, p) | (x, p) <- dist]
```

Proposition 10.2.2. `DistT` obeys the following equations:

- `fmap id = id`
- `fmap (g . h) = fmap g . fmap h`

The next step, is to define it as an instance of the Applicative class.

Definition 10.2.3.

```
instance (Monad m) => Applicative (DistT m) where
  pure x = DistT $ return [(x, 1)]
  (DistT mdistf) <*> (DistT mdist) = DistT $ do
    distf <- mdistf
    dist <- mdist
    return [(f x, pf * px) | (f, pf) <- distf, (x, px) <- dist]
```

Proposition 10.2.4. `DistT` obeys the following equations:

- `pure id <*> x = x`
- `pure f <*> pure x = pure (f x)`
- `x <*> pure y = pure ($ y) <*> x`
- `x <*> (y <*> z) = pure (.) <*> x <*> y <*> z`
- `pure f <*> x = fmap f x`

Next, we define `DistT` as an instance of the Monad class:

Definition 10.2.5.

```
instance (Monad m) => Monad (DistT m) where
  return = pure
  (DistT mdist) >>= f = DistT $ do
    dist <- mdist
    appf <- joinProb f dist
    return (concat appf)
```

where

```
joinProb :: (Monad m) => (a -> DistT m b) -> [(a, Double)]
                                          -> m [[(b, Double)]]
joinProb f dist = forM dist $ \(x, px) -> do
  let DistT mdist = f x
  dist' <- mdist
  return [(y, px * py) | (y, py) <- dist']
```

Proposition 10.2.6. `DistT` obeys the following equations:

- `return x >>= f = f x`
- `m >>= return = m`
- `(m >>= f) >>= g = (m >>= (\x -> fx >>= g))`

Lastly, we define `DistT` as an instance of the MonadTrans class, by providing a `lift` function that lifts computations from the underlying monad:

Definition 10.2.7.

```
instance MonadTrans DistT where
  lift ma = DistT $ liftM (\x -> [(x, 1)]) ma
```

where

```
liftM f m = do
      x <- m
      return (f x)
```

Proposition 10.2.8. `DistT` obeys the following equations:

- `lift(return a) = return a`
- `lift (m >>= f) = lift m >>= (lift . f)`

We are now ready to show the Haskell type that encodes the behavior of the small-step semantics:

```
type StTQC a = StateT LMem (ExceptT LMem (DistT [])) a
```

where `LMem = (SC,L,SQ)`. Let us deconstruct `StTQC` to see what it encodes (Figure 88).

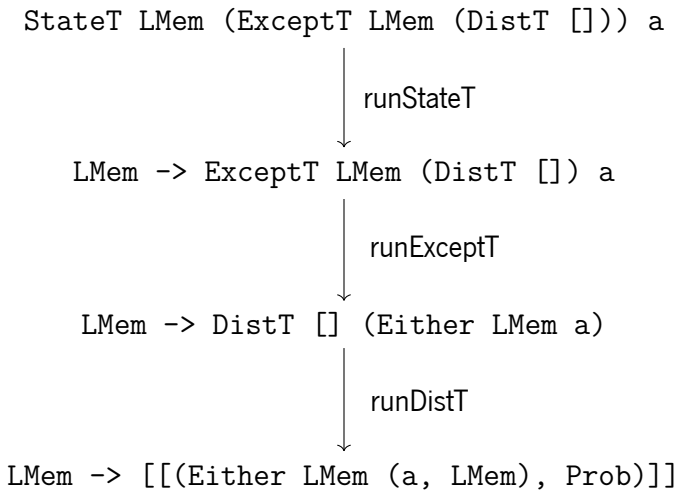


Figure 88: Unwrapping the different constructors from `StTQC a`

Thus when `a` $=$ `C`, the codomain of this stacking of effects captures the codomain of the type of the small-step operational semantics.

We now show the definition of the small-step semantics in Haskell, piece by piece. We begin with its type

```
small :: C -> StTQC C
```

Since elements of `StTQC C` are of the form

```
LMem -> [[(Either LMem (C, LMem), Prob)]]
```

we can say that elements of `small :: C -> StTQC C` are of the form

```
C -> LMem -> [[(Either LMem (C, LMem), Prob)]]
```

which, by uncurrying, gives us

```
(C, LMem) -> [[(Either LMem (C, LMem), Prob)]]
```

This type strongly resembles that of the small-step semantics presented in Section 9.1. Throughout the remainder of this document, we will use the term state to denote an element of `LMem`.

Next, we explain how we encoded the rules of the operational semantics (Figure 73, Figure 78, and Figure 79) in Haskell, beginning with the $skip$ command.

```
small Skip = StateT $ \s -> throwE s
```

Intuitively, given any state `s`, the $skip$ command leaves the state unchanged. Note that we use `throwE` (from `ExceptT`), instead of `StateT $ \s -> ExceptT $ DistT [Left (s, 1.0)]`, since both are equivalent.

Next in line, we have the assignment.

```
small (Asg var e) =
   StateT $ \(sc,l,sq) ->
      throwE $ (changeSt var (bigStepExp e sc) sc, l, sq)
```

Given a state `(sc,l,sq)`, we keep intact the linking function `l` and the quantum state `sq` and just manipulate the classical state `sc`. For such we use two functions, whose definition can be found in [37, SemBE_Brookes.hs]. The first one is `bigStepExp` which, as suggested by the name, evaluates the integer expression `e` under the classical state `sc`. The second one is `changeSt`, which updates the value of the variable `var` with the result of evaluating `e`.

Next, we have the reset.

```
small (Reset q) =
   StateT $ \(sc,l,sq) ->
      throwE $ (sc, l, resetOpDen (qNumsAux q l) sq)
```

Given a state `(sc, l, sq)`, we modify only the quantum state. This is done using the function `resetOpDen`, which locates qubit `q` in the quantum state `sq` with the help of the linking function `l`, and resets it to the state $|0\rangle$. The definition of `resetOpDen` can be found in [37, QuantumCalc.hs].

We now proceed to the application of unitary gates.

```
small (U g qvar) =
   StateT $ \(sc,l,sq) ->
      throwE $ (sc, l, appGateOpDen g (qNums qvar l) sq)
```

Similarly to the reset command, given a state `(sc, l, sq)`, we modify only the quantum state. This is achieved by using the function `appGateOpDen`, which relies on `qNums` to locate the qubits in `qvar` within the quantum state `sq`, with the help of the linking function `l`. The purpose of this procedure is to apply the unitary gate `g` to each qubit in `qvar`. The definition of `appGateOpDen` can be found in [37, QuantumCalc.hs].

It now follows the measurement command.

```
small (Meas (x,q)) = do
  (sc,l,sqt) <- get
  let sq = zeroIfSmallS sqt
      p0 = probOpDen 0 ((qNumsAux q l)) sq
      p1 = probOpDen 1 ((qNumsAux q l)) sq
      sq0 = stateMeasOpDen 0 ((qNumsAux q l)) sq
      sq1 = stateMeasOpDen 1 ((qNumsAux q l)) sq
      sc0 = changeSt x 0 sc
      sc1 = changeSt x 1 sc
  if (p0==0.0)
    then StateT $ \_ -> ExceptT $ DistT $ [[(Left (sc1,l,sq1),p1)]]
    else if (p1==0.0)
    then StateT $ \_ -> ExceptT $ DistT $ [[(Left (sc0,l,sq0),p0)]]
    else StateT $ \_ -> ExceptT $ DistT $
                  [[(Left (sc0,l,sq0),p0), (Left (sc1,l,sq1),p1)]]
```

The definition of the measurement command is more complex than the previous ones, despite that it begins in a similar way, *i.e.* by extracting the current state `(sc, l, sqt)`. Before performing any calculation, each amplitude in the quantum state is examined, and any real or imaginary part whose absolute value is below a certain threshold is set to zero. This allows us to avoid unexpected quantum states arising from the amplification of the limited precision of `Double` values. For instance, if we run the command `H(q1);CNOT(q1,q2);H(q2);X(q2);H(q2);Meas(x1,q1);Meas(x2,q2)` without applying the threshold and with initial state `|00><00|`, we obtain:

```
0.4999999999999998·([(a,0),(b,0)],(1.0)|000><000|) +
1.874699728327321e-33·([(a,0),(b,1)],(1.0)|010><010|) +
0.4999999999999998·([(a,1),(b,1)],(1.0-6.162975822039156e-33i)|110><110|)
```

However, when doing the calculations by hand, we note that the state |010><010| should not appear. This is caused by the inaccuracy of using Double values.

After applying the threshold mechanism, the unwanted term disappears, and the resulting state becomes:

```
0.4999999999999998·([(a,0),(b,0)], (1.0)|000><000|) +
0.4999999999999998·([(a,1),(b,1)], (1.0)|110><110|)
```

The threshold mechanism is performed by `sq = zeroIfSmallS sqt`. After applying the threshold mechanism, we proceed with the calculations. We start by calculating the probabilities of measuring $|0\rangle$ and $|1\rangle$. For that, we resort to `probOpDen`. In terms of inputs, it is similar to the `resetOpDen`. However, it also receives the basis on which to project the qubit to be measured. The probabilities are stored in `p0` and `p1`, where the former is the probability of measuring 0 and the latter the probability of measuring 1. Next, we calculate the quantum state using the function `stateMeasOpDen`, which in terms of inputs is similar to `probOpDen`. The resultant states are stored in `sq0` and `sq1`, with a notation similar to that of the probabilities. At last, we update the classical states, with a familiar function `changeSt`. Using a similar notation as previously, the classical states are stored in `sc0` and `sc1`. At this point, if the probability of measuring 0 is zero, then we return the result of measuring 1. A similar behavior occurs when the probability of measuring 1 is also zero. In the case where both probabilities are not zero, we return the results obtained by measuring 0 and 1. Note the use of `Left` to indicate the return of a state and not of a computation.

The next in line is probabilistic choice.

```
small (P prob c1 c2) = do
  s <- get
  let p = fromRational prob
      cc1 = runDistT $ runExceptT $ runStateT (small c1) s
      cc2 = runDistT $ runExceptT $ runStateT (small c2) s
      pc1 = [[(c,p*p')| (c,p') <- lcc1] | lcc1 <- cc1]
      pc2 = [[(c,(1-p)*p')| (c,p') <- lcc2] | lcc2 <- cc2]
      pc1c2 = concat [[ec1++ec2 | ec2 <- pc2] | ec1 <- pc1]
  StateT $ \_ -> ExceptT $ DistT $ pc1c2
```

As usual, we first receive a state `s`. Now, recall that in our implementation probabilities are represented as values of type `Double`, while the argument `prob` has type `Rational`. To convert between these types,

we use the function `fromRational`, and store the result in `p`. Next, `cc1` and `cc2` store the small-step evaluations of the commands `c1` and `c2`, respectively, both under the input state `s`. To access the resulting distributions, we successively remove the monadic constructors using the corresponding deconstructors: `runStateT`, `runExceptT`, and `runDistT`. After this, `cc1` and `cc2` become lists of distributions, where each distribution is itself a list of pairs of the form `(command_or_state, probability)`. We then scale the probabilities in each distribution. In `pc1`, each probability is multiplied by `p`, while in `pc2`, each probability is multiplied by `1-p`. This corresponds to weighting the outcomes of each branch according to the probabilistic choice. Next, for each distribution in `pc1`, we concatenate it with each distribution in `pc2` using the operator `++`. This operation produces a three-layered list (a list of lists of lists), because `++` joins inner lists while preserving the outer structure. To flatten this result and remove the additional nesting, we apply `concat`, storing the final result in `pc1c2`. Since the type of a result produced by `small` is `StTQC C`, we must re-encapsulate `pc1c2` within all the monadic constructors that compose `StTQC C`, namely `StateT`, `ExceptT`, and `DistT`.

We now focus on the sequential composition.

```
small (Seq c1 c2) = do
    s <- get
    let cp = runDistT $ runExceptT $ runStateT (small c1) s
        seqC = [compSeq dist c2| dist <- cp]
    StateT $ \_ -> ExceptT $ DistT $ seqC
```

Again, we start by receiving a state `s`. Next, we evaluate `c1` using the small-step operational semantics and remove the monadic constructors, storing the result in `cp`. The next step is to either proceed directly with `c2`, if there is some computation of `c1` that has finished, or to sequentially compose what is left to execute from `c1` with `c2`. This behavior is implemented by the auxiliary function `compSeq`, and the result is stored in `seqC`. Finally, we re-encapsulate `seqC` using the appropriate constructors.

Next in line is non-deterministic choice.

```
small (Or c1 c2) = do
    s <- get
    let cp1 = runDistT $ runExceptT $ runStateT (small c1) s
        cp2 = runDistT $ runExceptT $ runStateT (small c2) s
    StateT $ \_ -> ExceptT $ DistT $ (cp1++cp2)
```

After obtaining the initial state `s`, we evaluate both `c1` and `c2`, through the small-step operational

semantics, and store the result of its evaluation in cp1 and cp2, respectively. We then concatenate cp1 and cp2 using ++ and encapsulate the result within the necessary constructors.

Now, we go to the parallel composition.

```
small (Paral c1 c2) = do
    s <- get
    let cp1 = runDistT $ runExceptT $ runStateT (small c1) s
        cp2 = runDistT $ runExceptT $ runStateT (small c2) s
        par1 = [compParR dist c2| dist <- cp1]
        par2 = [compParL dist c1| dist <- cp2]
    StateT $ \_ -> ExceptT $ DistT $ par1++par2
```

After obtaining the state s, the rationale behind it is similar to that of sequential composition. However, in this case, we need to evaluate c1 and c2 separately. Then, we need to compose the results of the evaluation of c1 with c2, and vice-versa. For this last step, we use compParR and compParL. The result is then stored in par1 and par2. Finally, we merge par1 and par2 and wrap the result with the corresponding constructors.

Next we have the if-then-else command.

```
small (IfC bExp c1 c2) = do
  (sc,l,sq) <- get
  let b = bigStepBExp bExp sc
  if b == True
    then StateT $ \_ -> ExceptT $ DistT $ [[(Right (c1,(sc,l,sq)),1)]]
    else StateT $ \_ -> ExceptT $ DistT $ [[(Right (c2,(sc,l,sq)),1)]]
```

After obtaining the initial state (sc,l,sq), we evaluate the Boolean expression bExp under sc and store the result in b. If b evaluates to True, the computation proceeds with c1 with probability 1. Otherwise, it proceeds with c2, also with probability 1.

The next in line is the while command.

```
small (Whl bExp c) = do
  (sc,l,sq) <- get
  let b = bigStepBExp bExp sc
  if b == True
    then StateT $ \_ -> ExceptT $ DistT $
```

```
                    [[(Right (Seq c (Whl bExp c),(sc,l,sq)),1)]]
    else StateT $ \_ -> ExceptT $ DistT $ [[(Left (sc,l,sq),1)]]
```

It behaves similarly to the if-then-else command. However, when the Boolean expression `b` evaluates to `True`, we sequentially compose `c` with the while command. On the other hand, when `b` evaluates to `False`, the loop terminates, and we return the input state.

Now we go to the await command.

```
small (Await bExp c) = do
  (sc,l,sq) <- get
  let b = bigStepBExp bExp sc
  if b == False
    then StateT $ \_ -> ExceptT $ DistT $
                    [[(Right (Await bExp c, (sc,l,sq)),1)]]
    else StateT $ \_ -> ExceptT $ DistT $
                    [[(Right (Atom c, (sc,l,sq)), 1)]]
```

After obtaining the initial state `(sc,q,sq)`, the Boolean expression `bExp` is evaluated and its result is stored in `b`. If the value of `b` is `False`, then we idle, *i.e.* we remain in the same configuration. Otherwise, we encapsulate the instructions inside the await command in the atom constructor.

At last, we have the atom constructor.

```
small (Atom c) = do
  s <- get
  let cp = runDistT $ runExceptT $ runStateT (small c) s
      atom = [inAtom dist | dist <- cp]
  StateT $ \_ -> ExceptT $ DistT $ atom
```

After obtaining the initial state `s`, we store the result of evaluating `c` in `cp`. Note that this evaluation removes the atom constructor. Hence, we need to reintroduce it for the computations that did not finish. For that, we use the function `inAtom` and store the result in `atom`. Finally, we re-encapsulate `atom` with the appropriate constructors.

Now, we move on to the implementation of the big-step semantics. As described in Section 8.2, the big-step semantics relies on a scheduler (Definition 8.2.1). This scheduler resolves the non-determinism present in the system. Consequently, the output produced by the big-step semantics is a valuation whose weights sum to a value less than or equal to one. To capture this behavior in Haskell, we could employ

the transformer `DistT` together with the identity monad, which is already defined in the *mtl* package. Although we have experience in manipulating stacked effects and using the `lift` function, we chose instead to represent the probabilistic effect using only a single monad. For this purpose, we defined the `Dist` monad, which corresponds to `DistT` combined with the identity monad. This way, we avoid the use of the `lift` function.

The scheduler used in the big-step semantics has the following Haskell type:

```
type Sch = ProbPath ->
     Maybe [([(Either LMem (C, LMem), Double)], Double)]
```

where `ProbPath` is as follows:

```
type ProbPath = ( [((C, LMem), [(Either LMem (C, LMem), Double)])],
                  (C, LMem) )
```

We now show the codification (slightly line-wrapped) of the big-step semantics in Haskell.

```
kStepSch :: (Sch, ProbPath, Int) -> Dist LMem
kStepSch (_, path, 0) = Dist [(snd $ snd path, 0)]
kStepSch (sch, l@(path, (c, s)), k) =
  case (sch l) of
    Nothing -> error "Scheduler undefined"
    Just convDist -> do
      let ppL = [
                 [ (s, p*q) | (s, p) <- (projL dist)]
                 | (dist, q) <- convDist
                ]
          next_eval = [
                       [((sch, (path ++ [((c, s), dist)], cs), k-1), p*q)
                       | (cs, p) <- (projR dist)]
                      | (dist, q) <- convDist]
          transStep = (Dist $ concat next_eval) >>= kStepSch
      addDist transStep (Dist $ concat ppL)
```

The inputs of `kStepSch` are a scheduler (Definition 8.2.1), a probabilistic path, and an integer `k`, which indicates the maximum number of steps the computation can perform. According to this last parameter, there are two cases: if the number of steps is zero, we return the zero valuation; otherwise,

further computation is required. In the latter case, the evaluation depends on the scheduler. Recall that a scheduler is a partial function; therefore, if it is undefined, the big-step semantics cannot proceed. Consequently, an error is raised stating that the scheduler is undefined. If, however, the scheduler is defined, it returns a convex combination of distributions, denoted by `convDist`. From this convex combination, we need to separate the computations that have already finished from those that are still ongoing. To do this, we use Haskell comprehension lists. We start by extracting the completed computations and storing them in `ppL`. Since `convDist` is a convex combination of distributions, it is represented as a list of pairs, each composed of a distribution and its corresponding convex coefficient. Using a list comprehension, we take each pair `(dist, q)` in `convDist`. For each `dist`, we project the completed computations by applying `projL`, which returns pairs of the form `(s, p)`, representing a final state and its associated probability. We then multiply the probability `p` by the convex coefficient `q`, yielding the pair `(s, p*q)`. For the ongoing computations, we proceed similarly. From `convDist` we again consider each pair `(dist, q)`. By applying `projR` to `dist`, we extract the computations that continue. However, we cannot simply return a pair consisting of the next command and its probability (given by `p*q`). We must also include additional information so that the big-step semantics can continue executing. Thus, each tuple formed in `next_eval` includes the scheduler `sch`, the probabilistic path extended with the chosen distribution and configuration to be evaluated, `(path ++ [((c, s), dist)], cs)`, and the remaining number of steps, `k-1`. The transitive step of the big-step semantics is represented using the bind operator `>>=`, and its result is stored in `transStep`. Note the use of `concat` in `next_eval`, which flattens a list of lists into a single list, matching the structure expected by `Dist`. Finally, the results obtained from the transitive step are added to the computations that were completed during the first iteration. This is accomplished using `addDist`.

With the introduction of the $await$ command and the atom constructor, we mentioned that the scheduler is responsible for prioritizing the execution of the instructions inside the atom constructor. To achieve this, we rely on two functions, whose definitions can be found in [37, KStep.hs]: `nextAtom` and `evalAtom`. The former is a function that determines whether the next command to be executed is encapsulated within an atom constructor. The latter is a function that uses both `nextAtom` and `small` (recall that `small` is the Haskell implementation of the small-step operational semantics). In sum, `evalAtom` modifies the behavior of `small` so that commands inside an atom constructor are given priority. In the situation where no command is encapsulated within an atom constructor, the function `evalAtom` behaves exactly like `small`.

It now remains to explain how histograms are built. To construct a histogram, we first need to collect

samples. To do this, it is necessary to select an element from a valuation. This task relies on Haskell's capabilities, which will be discussed later on.

Let us now delve into how we produce histograms. To collect data for the histogram, we base our approach on the big-step semantics (Figure 74). We say "based" because several significant modifications were made. The first major change concerns the scheduler. Instead of allowing the user to define and provide their own scheduler, we fix a scheduler that assigns equal probability to all possible transitions. Moreover, this scheduler is memoryless, *i.e.* it does not store the choices made during previous computation steps. Although restricting the scheduler to assign equal probabilities limits the user's control over the scheduling policy, making it memoryless has two key advantages: it reduces memory usage (since we do not store the steps performed by the computation) and it improves execution speed (since there is no need to update the probabilistic path). The second major change is that, after the scheduler acts, we first randomly select one of the distributions (recall that the scheduler produces a convex combination of distributions), and then, based on the probabilities associated with that distribution, we randomly select one element from its support. This selected element may be either a final state or a configuration. The process is repeated until either the program terminates or the maximum number of evaluation steps, specified by the user, is reached. When the process ends, two outcomes are possible. If the number of steps is insufficient to reach a final state, we obtain the zero valuation, which we discard because its probability is zero. Otherwise, we reach a final state, which corresponds to a valid sample for the histogram.

Now that the intuitions have been explained, let us move on to some implementation details (the code regarding the implementation of the histogram can be found in [37, HistogramSem.hs]). The function `sample` codifies the modifications we perform to the big-step.

```
sample :: ((C,LMem),Int) -> DistT IO LMem
sample ((c,s),0) = DistT $ return [(s, 0)]
sample ((c,s),n) = do
  nextDist <- lift $ memorylessSch (c,s)
  nextConf <- lift $ eventNextStep nextDist
  case nextConf of
    (Left lmemi, pi) -> DistT $ return [(lmemi, pi)]
    (Right (ci,si), pi) -> do
      let ppR = [(((ci,si), n-1), pi)]
      (DistT $ return ppR) >>= sample
```

The choice of a distribution, after the action of the scheduler, is made by `memorylessSch`. Intuitively,

this function uses `evalAtom` to continue giving priority to commands inside the atom constructor and to gather all the possible transitions. Next, it chooses a distribution based on a uniform distribution, *i.e.* each element has the same probability of being selected. This is done using the function `randomR` provided by the Haskell package `System.Random`. The function `randomR` takes a tuple specifying the lower and upper bounds of the range, *i.e.* `(lo, hi)`, together with a seed (*i.e.* a pseudo-random number generator). It then returns a pseudo-random number within the closed interval `[lo, hi]`, along with an updated seed. The minimum value of the range represents the index zero of the distribution, while the maximum value of the range represents the length of the distribution (recall that we represent a distribution in Haskell through a list). Thus, `randomR` returns an index identifying the selected distribution. The result of this step is stored in `nextDist`. Furthermore, note the usage of `lift`, to lift the `IO` computation to the `DistT` monad. Next, from the chosen distribution, we select either a configuration or a final value. This is done by resorting to `eventNextStep`. Intuitively, the function `eventNextStep` selects an element from the support of the distribution according to the probabilities of its elements. This functionality is implemented using the functions `makeEventProb` and `enact` from [22]. The former takes a list of tuples composed of a value and its associated probability and creates an element of type `eventM a`, where `a` is the type of the value. The latter executes (or enacts) this event and produces a single result according to the probabilities defined in the `eventM a` parameter. The result of this selection is then stored in `nextConf`. If the value in `nextConf` is a final state, we return it. Otherwise, if it is a configuration, we continue the computation using the bind operator >>=, just as in the implementation of the big-step semantics.

Now that we know how to obtain a sample, we need to collect a given number of them. For that, we create the function `sampleCollect`.

```
sampleCollect :: Int -> C -> LMem -> Int -> IO [Mem]
sampleCollect 0 c s n = return []
sampleCollect k c s n = do
    ((sc,sq),p) <- runSample c s n
    case p of
      0 -> sampleCollect (k-1) c s n
      otherwise -> do
        as <- sampleCollect (k-1) c s n
        return ((sc, limitPrecisionS 5 sq):as)
```

If there are no samples to collect, we simply return the empty list. Otherwise, when samples remain to be collected, we store the current sample in `((sc, sq), p)` and proceed according to the value of

p: if p equals zero, we discard the sample and continue the collection; otherwise, we keep the sample and continue collecting the remaining ones. Note the use of `runSample`. This function essentially executes `sample` and performs two operations: first, it removes the `DistT` constructor from the collected sample; second, it extracts the sample from the list returned by `sample`, yielding the final result directly. Furthermore, observe that the result returned by `sampleCollect` is a list of memories wrapped in the `IO` monad. In other words, we drop the linking function and retain only the classical and quantum memories. These memories will later be processed to improve the presentation of the histogram.

The function responsible for manipulating the collected samples is `processData`. However, this function relies on `processDataAux` to perform the actual work. We focus on the latter:

```
processDataAux :: String -> [String] -> [Mem] -> IO ExitCode
processDataAux name lvar listMem = do
  let listSC = map fst listMem
      nameHist = name
      xlabel = unwords lvar
      lvarSC = [concat [filter (\(var',val) ->  if var'==var
                                                then True
                                                else False) sc
                        | var <- lvar]
                | sc <- listSC]
      valVarSC = [ (map snd varSC) | varSC <- lvarSC]
      sortValVarSC = sort valVarSC
      strSorted = [concat $ (map show) l | l <- sortValVarSC]
  buildHist (confIntoDouble strSorted) nameHist xlabel
```

This function receives three arguments: the program name (`name`), the list of variables to be used in building the histogram (`lvar`), and a list of memories corresponding to the collected samples (`listMem`). We start by extracting the classical states from `listMem`, storing the result in `listSC`. This step is necessary because the outcome of a measurement is stored in classical variables; therefore, the histogram is built solely from classical states. Next, we prepare the title of the histogram's x-axis. The variable `xlabel` stores the result of merging all variable names in `lvar` into a single string. We then filter each classical state, keeping only the variables that appear in `lvar`. The result of this filtering step is stored in `lvarSC`. Afterward, we extract the values associated with these variables and store them in `valVarSC`, sort these values (resulting in `sortValVarSC`), and finally convert the sorted values to strings, stored

in `strSorted`. These string representations will later be used as the labels for the x-axis. All the data manipulation performed so far prepares the inputs for the `buildHist` function, which, as the name suggests, constructs the histogram. Before this, the values in `strSorted` are processed by the function `confIntoDouble`, which assigns an x-coordinate to each distinct value in `strSorted`. For example:

```
ghci> confIntoDouble ["a","b","a"]
[(1.0,"a"),(2.0,"b"),(1.0,"a")]
```

This tells us that the bin corresponding to the value `"a"` will be the first to appear in the histogram, while the bin for `"b"` will appear second. In other words, `confIntoDouble` determines the x-axis position of the histogram values.

We now focus on the definition of `buildHist`.

```
buildHist :: [(Double, String)] -> String -> String -> IO ExitCode
buildHist [] title _ = error "Empty input."
buildHist dataSet title xlabel = plotAdv "" options hist
    where newDataSet = [(d-1,st) | (d,st) <- dataSet]
          doubles =  [d | (d,st) <- newDataSet]
          max = round (maximum doubles) :: Int
          hist = histogramBinSize 1 doubles
          options = Opt.title title $
                    Opt.xLabel xlabel $
                    Opt.xRange2d (-1,max+1) $
                    Opt.xTicks2d (xTicksData newDataSet) $
                    Opt.yLabel "Frequency" $
                    Opt.add (Option.yRange "bounds") ["[0:]"] $
                    Opt.gridYTicks True $
                    defOpts hist
```

The function `buildHist` receives as input the dataset to be plotted (`dataSet`), the histogram title (`title`), and the x-axis label (`xlabel`). To build the histogram, the dataset must be non-empty; otherwise, an error is raised. When `dataSet` is not empty, we begin by shifting the x-coordinates of `dataSet` one unit to the left, so that the bins align correctly with their corresponding x-axis labels. The shifted list is stored in `newDataSet`. Next, we extract the x-coordinates and store them in `doubles`. The variable `max` stores the number of bins the histogram will have, and `hist` stores the histogram generated from the

values in `doubles`, using a bin size of 1. Finally, we define several options related to how the histogram is plotted. Notably, `Opt.xTicks2d (xTicksData newDataSet)` ensures that the bins are aligned with their corresponding labels, and `Opt.add (Option.yRange "bounds") ["[0:]"]` enforces that the y-axis starts at 0.

It only remains to briefly mention the function `showHist`:

```
showHist :: String -> (Int,[[String]]) -> C -> LMem -> Int -> IO ExitCode
showHist name (k, lvars) c s n = do
  listMem <- sampleCollect k c s n
  processData name lvars listMem
```

The function `showHist` receives all the information required to produce the histogram. It first collects the samples using `sampleCollect`, and then calls `processData` to trigger the construction of the histogram.

Throughout this section, we have omitted the definitions of some auxiliary functions. The reader can consult those definitions, and additional implementation details, in [37].

10.3 The limits of the tool

In the previous section, we described the implementation of the semantics. In this section, we evaluate the tool in two scenarios: branch explosion and qubit scaling. For the former, we use a random walk with an increasing number of steps. Here, we compare the execution times obtained from the two output modes provided by the tool (recall Figure 84): valuations, which use a scheduler with memory, and histograms, which use a memoryless scheduler. For qubit scaling, we place an increasing number of qubits into superposition by applying a Hadamard gate to each of them and then performing a measurement. In this case, we compare the results produced by the tool developed in this thesis with those obtained using the QIO monad [6], a Haskell monadic framework for quantum computation. Note that none of the programs used in these benchmarks involve concurrency, even though concurrency is a central feature of our language. This choice arises from the schedulers we implemented and later used in Section 10.4: each scheduler selects a single distribution to evaluate at each step, rather than producing a convex combination of all possible transitions arising from the small-step semantics (recall that the scheduler returns a convex combination of transitions as defined in Definition 8.2.1). Consequently, a concurrent example would primarily compare scheduler behavior, which we already examined in Section 10.4, rather than illustrating state explosion caused by concurrency.

Our goal is to determine when it becomes infeasible to use the tool efficiently. To perform the aforementioned tests, we use the Criterion package. Interested readers can see the package documentation for more information [85]. From all the data provided by Criterion, we focus on the estimated time required for a single execution of a program. To obtain this estimate, Criterion runs the program multiple times, varying the number of iterations. For each number of iterations, it records the total execution time. For instance, it measures the time for one execution, then for two executions, and so on. Criterion then fits a linear regression model using Ordinary Least Squares (OLS) estimation to the collected data, with the number of iterations on the x-axis and the total execution time on the y-axis. Intuitively, Criterion attempts to determine the parameters t_{cost} and C_{overhead} of the linear equation

$$T(n) = t_{\text{cost}} \cdot n + C_{\text{overhead}}$$

where $T(n)$ is the total time, n is the number of iterations, t_{cost} is the estimated time of a single execution (the slope), and C_{overhead} is an offset (the y-intercept) representing the fixed overhead introduced by the measurement loop, timing calls, and other non-scaling factors. Note that these measurements include both the time spent parsing and executing the program.

All execution times reported in this thesis were collected on a 2019 MacBook Pro with the following specifications: 2.4 GHz Quad-Core Intel Core i5 processor, 16 GB RAM (2133 MHz LPDDR3), and Intel Iris Plus Graphics 655 (1536 MB). The benchmarks were compiled and executed using GHC, as required by Criterion.

We begin by testing the tool under the branch explosion scenario. The program used for this test is a random walk:

```
while (i1<it) do {x1:=x1+1 (+)(1%2) x1:=x1-1; i1:=i1+1}
```

where `it` specifies the number of iterations of the walk.

In this scenario, we compare the performance of `runSem` and `runHist`. Recall that `runSem` requires a scheduler as input. Since the program contains no concurrency, we use the scheduler `initSch`, which always selects the head of the list of possible computations. In contrast, `runHist` uses a predefined memoryless scheduler.

What should we expect? Since the scheduler used by `runHist` has no memory, and since it selects a single configuration to execute next, it should be faster than `runSem`. The scheduler in `runSem`, on the other hand, records the entire history of execution and selects a distribution at each step. Two differences therefore distinguish the two approaches:

1. Memory usage

The scheduler in `runSem` stores the sequence of previous steps, while the scheduler in `runHist` does not. As the computation grows, so does the history.

2. Growth of distributions

   At each probabilistic choice, the scheduler in `runSem` returns a distribution whose support doubles in size, thus increasing both memory usage and the cost of subsequent operations. On the other hand, `runHist`, avoids this blow-up by selecting a single configuration rather than a full distribution.

Consequently, `runHist` is expected to be faster. This is confirmed by Figure 89 [5].

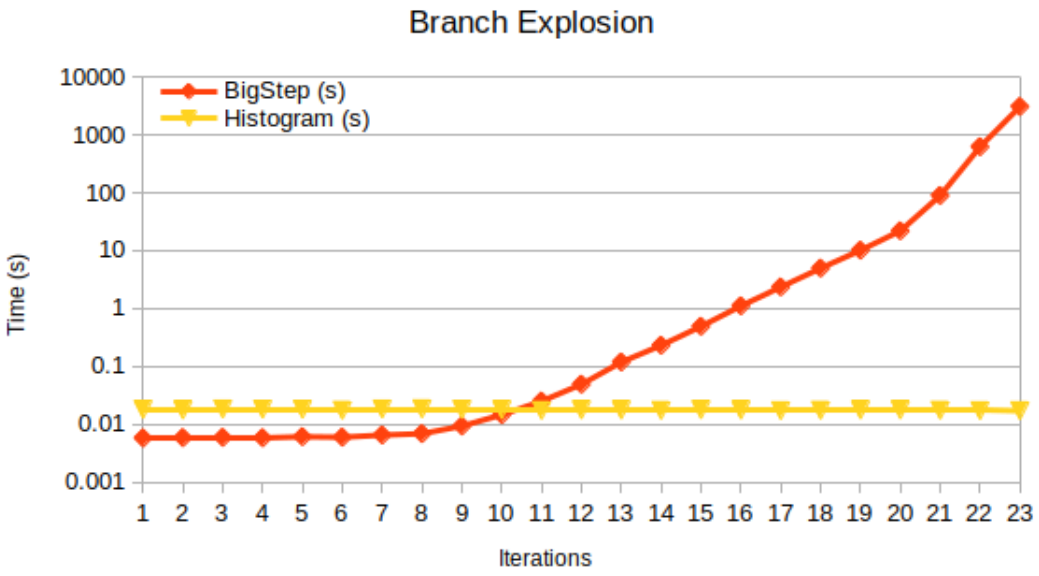


Figure 89: `runSem` vs `runHist`

As shown in Figure 89, the execution time of `runHist` remains almost constant, whereas the time taken by `runSem` grows approximately exponentially. The number of iterations (the value of `it`) goes up to 23, since GHC raises a compiler error when using `runSem` and `it` $= 24$. The y-axis in Figure 89 is displayed on a logarithmic scale, which is justified by the fact that, for `it` $= 15$, the execution time of `runSem` is already about 100 times larger than that of `runHist`. The data used to generate the figure can be found in Appendix D.

We now turn to the qubit-scaling scenario. Here, our aim is to determine the number of qubits for which the execution time becomes so large that it is preferable to use another tool. For comparison, we run the same program using the QIO monad [6].

[5] To obtain these results, we ask the histogram to record only one sample, since the results produced by the big-step semantics correspond to a single execution.

The program used in this test is simple: we apply a Hadamard gate to each qubit, thereby placing all qubits in superposition, and then perform a measurement on each of them. A snippet for three qubits is shown below:

```
H(q1);H(q2);H(q3);
Meas(x1,q1);Meas(x2,q2);Meas(x3,q3)
```

Our goal is to measure how long it takes to execute these instructions. This constitutes a worst-case scenario for both memory usage and computational cost, since all qubits are simultaneously placed in superposition. Recall that our tool represents quantum states using density operators. For i qubits, the density matrix has dimension $2^i \times 2^i$. Moreover, each measurement branch requires several matrix multiplications (*e.g.* $P_0 \rho P_0^\dagger$, with ρ a density operator). As the number of qubits increases, the cost of performing these multiplications grows rapidly. Since matrix multiplication is computationally expensive, an increasing number of such operations results in significantly longer execution times. By contrast, the QIO monad represents quantum states as vectors. Thus, applying a unitary corresponds to multiplying a matrix by a vector, which is considerably cheaper than matrix-matrix multiplication. Consequently, calculations performed in the QIO monad are expected to be much faster than those in our tool. Figure 90 confirms this expectation.

As shown in Figure 90, the execution time increases exponentially with the number of qubits in both tools, but at vastly different scales. In our tool, ten qubits already require several hours of computation, whereas in the QIO monad the runtime has not yet reached one second. For reference, in our tool, nine qubits take approximately 75 minutes to simulate, and ten qubits require more than one day. The data used to generate the figure can be found in Appendix D.

10.4 The interpreter at work

Before proceeding to the examples, we briefly explain how the information inside the input file is processed and then fed into the big-step semantics or used to build histograms. We dedicate the first part of this section to that purpose. The module under consideration here is `Run`. It is composed of four functions: two related to the outputs of the big-step semantics, and two related to the generation of histograms. Starting with the former, we have the function `runSem`.

```
runSem :: String -> Sch -> IO()
runSem path sch = do
```

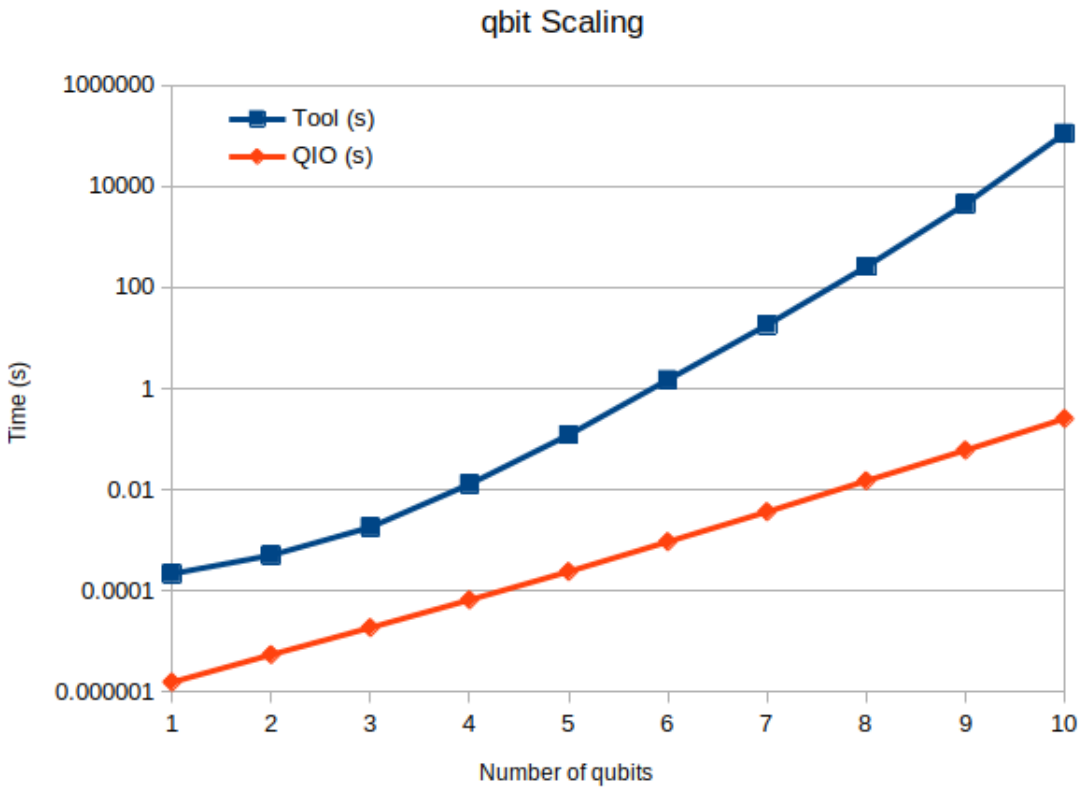


Figure 90: `Tool` vs `QIO`

```
fileContent <- readFile path
case parseRun fileContent of
  Left err -> print err
  Right configs -> runSemAux configs sch
```

The function `runSem` receives the path to the input file, `path`, and a scheduler, `sch`. It reads the content of the file and stores it in `fileContent`. Then, it applies `parseRun` to `fileContent`. If an error occurs during parsing, the error is printed. Otherwise, the variable `configs` contains a list with all the necessary information to produce results using the big-step semantics. This list is then passed to the auxiliary function `runSemAux`, which evaluates each element and prints the corresponding output obtained from the big-step semantics. A similar reasoning applies to the generation of histograms. Thus, we omit its detailed explanation. However, it is worth mentioning that the functions responsible for this task are `runHist` and `runHistAux`. The reader can consult their definitions in [37, Run.hs].

We now illustrate the flexibility of the tool by presenting a plethora of examples.

10.4.1 Dining Philosophers

The Dining Philosophers problem [32] is a classic challenge in the field of concurrency. It was introduced by Edsger Dijkstra in 1965 to illustrate the difficulties of resource allocation in concurrent programming. The problem is described as follows. Suppose n philosophers are sitting around a circular table with a bowl in the center containing an infinite number of pastéis de nata. Each philosopher alternates between three states: thinking, hungry, and eating. Between each pair of philosophers, there is a chopstick. To eat a pastel de nata, a philosopher must hold the two chopsticks adjacent to him – one on his left and one on his right. Importantly, only one philosopher can hold a given chopstick at a time.

The challenge of this problem is to design a protocol or algorithm that: (1) ensures that when a philosopher picks up a fork, no other philosopher picks it up, and (2) guarantees that all philosophers eventually eat. We implemented [6] the solution proposed in [12, Section 7.5], which uses the $await$ command. In sum, the strategy revolves around waiting until both chopsticks are available. Once this condition is met, the philosopher picks up the chopsticks and begins to eat. When the philosopher decides to stop eating, they put down the chopsticks and return to thinking. The crucial step in this strategy is encoding the waiting for both chopsticks as the condition that guards entry to a critical section. When the condition is not satisfied, the philosopher idles. When it is satisfied, the philosopher is guaranteed access to the chopsticks without interference from others.

We encoded this algorithm for three philosophers using the language introduced in Section 9.1. The configuration for this setting is illustrated in Figure 91.

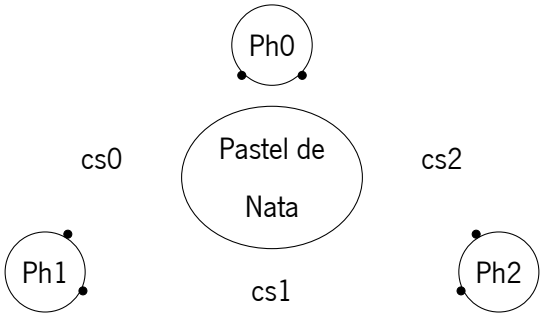


Figure 91: Setting of the three philosophers

The code for each philosopher is shown in Figure 92. Each philosopher alternates between thinking, hungry, and eating states. This behavior is captured by the variable $\texttt{st}_i \in \{0, 1, 2\}$, with $i \in \{0, 1, 2\}$, where:

[6] In [37], under the folder examples/thesis, the reader can find, for this example, a Haskell script named "dinphil.hs". This script prints out a program, following the format requirements described in Section 10.1, for any specified number of philosophers. Furthermore, in the same folder, one can find instantiations for two, three, four, and five philosophers.

- the value 0 indicates that Philosopher$_i$ is eating
- the value 1 indicates that Philosopher$_i$ is thinking
- the value 2 indicates that Philosopher$_i$ is hungry

The chopsticks are represented by the variable $\texttt{cs}_i \in \{0,1\}$, with $i \in \{0,1,2\}$, where the value 0 indicates that chopstick i is taken, and the value 1 indicates that it is free. Since the philosophers execute their instructions inside an infinite loop (see Figure 92) and we do not have a way of printing what each philosopher is doing (*e.g.* printing "Philosopher$_i$ is eating"), we rely on a variable to check whether the philosophers are eating. Thus, each philosopher possesses a variable $\texttt{eat}_i$, with $i \in \{0,1,2\}$, which is incremented every time Philosopher$_i$ puts down the chopsticks. Figure 92 specifies the behavior of Philosopher$_i$, with the convention that the indices are taken modulo 3.

```
while(true) do{
  while(st_i==1) do{st_i:=2 or skip};
  await(cs_i==1 & cs_{i-1}==1) do{cs_i:=cs_i-1; cs_{i-1}:=cs_{i-1}-1};
  st_i:=0;
  while(st_i==0) do{st_i:=1 or skip};
  eat_i:=eat_i+1; cs_i:=cs_i+1; cs_{i-1}:=cs_{i-1}+1
}
```

Figure 92: Code describing the behavior of Philosopher$_i$.

Since each philosopher performs their actions in an infinite loop, the result obtained by executing the big-step semantics is a zero valuation. This is expected, because the big-step semantics assigns the zero valuation to computations that do not terminate (recall Definition 74). Furthermore, the zero valuation is accompanied by the memory state reached before the evaluation could no longer make progress. Thus, we can check the variable $\texttt{eat}_i$ to verify if Philosopher$_i$ has eaten or not.

Now, let us explain what each philosopher does. The while loop controlled by the condition $\texttt{st}_i$==1 represents the stage where the philosopher is thinking. If the philosopher is not hungry, he continues thinking (by executing `skip`). Otherwise, he changes his state to hungry ($\texttt{st}_i$:=2). Then, the philosopher waits for both chopsticks to become available (`await(`$\texttt{cs}_i$==1 & $\texttt{cs}_{i-1}$==1`)`). When both chopsticks

are free, the philosopher picks them up ($\texttt{cs}_i$:=$\texttt{cs}_i$-1; $\texttt{cs}_{i-1}$:=$\texttt{cs}_{i-1}$-1) and begins to eat ($\texttt{st}_i$:=0). The act of eating is represented by the while loop governed by $\texttt{st}_i$==0. In this loop, the philosopher either continues eating (via `skip`) or, when full and ready to think again, changes his state ($\texttt{st}_i$:=1). Finally, the philosopher records that he has eaten ($\texttt{eat}_i$:=$\texttt{eat}_i$+1) and puts down both chopsticks ($\texttt{cs}_i$:=$\texttt{cs}_i$+1; $\texttt{cs}_{i-1}$:=$\texttt{cs}_{i-1}$+1).

Let us put the algorithm to the test using the tool we developed. One important point to keep in mind is that the choice of scheduler affects the results obtained. For this purpose, we use three different schedulers (their definitions can be found in [37, KStep.hs]): `nonBlockSch`, `randNonBlockSchII`, and `roundRobinSch`. To better understand how these schedulers behave, recall that the small-step semantics produces a list of possible computations to perform next. The first scheduler, `nonBlockSch`, selects the head of the list and checks whether it has already occurred. If it has not occurred, the computation proceeds with the head of the list. Otherwise, the scheduler selects the next element in the list and repeats the procedure. The second scheduler, `randNonBlockSchII`, behaves similarly to `nonBlockSch`, with a small modification: the element chosen to be evaluated is selected "randomly". We say "randomly" because the choice depends on the size of the list of possible computations. Specifically, the scheduler computes `mod lenListDist 3`, where `lenListDist` denotes the size of the list of possible computations. Depending on the value obtained from `mod lenListDist 3` we proceed as follows:

- if the result is 0, it picks the head of the list
- if it is 1, it picks an element from the middle of the list
- if it is 2, it picks the last element

The third and final scheduler, `roundRobinSch`, aims to mimic a round-robin scheduling strategy, which is known to be a fair and deterministic scheduler [62, 7]. To achieve this, it relies on the size of the list of possible computations, `lenListDist`, and on the number of commands that have already been executed in the computation, `lenAux`. The next distribution to be evaluated is selected by computing `mod lenAux lenListDist`, similarly to what is done in `randNonBlockSchII`. By taking the modulo with respect to `lenListDist` instead of 3, the scheduler can choose any element of `listDist`, since the result of `mod lenAux lenListDist` always lies between 0 and `lenListDist` $- 1$.

Having said this, let us examine how the algorithm behaves under the three different schedulers. We begin with `nonBlockSch`. Executing `runSem "thesis/dinphil3.txt" nonBlockSch`, with the number of allowed steps set to 100, yields:

```
0.0·([(st0,1),(st1,1),(st2,1),(cs0,1),(cs1,1),(cs2,1),
```

```
      (eat0,7),(eat1,0),(eat2,0)])
```

Under the scheduler `nonBlockSch`, only Philosopher 0 eats, while the others starve (in fact, they do not have the chance to execute any instruction). This happens because the scheduler always gives priority to Philosopher 0: the computation never becomes stuck, as the state of Philosopher 0 is continuously updated. In other words, since the algorithm keeps changing its state, each subsequent step remains different, and Philosopher 0 continues to make progress.

Now, what happens if we try to "randomize" the previous scheduler? The answer is: the first philosopher still eats while the others starve. To see this, we execute `runSem "examples/dinphil.txt" randNonBlockSchII`, with the number of allowed steps again set to 100. The result is as follows:

```
0.0·([(st0,2),(st1,1),(st2,1),(cs0,0),(cs1,1),(cs2,1),
      (eat0,4),(eat1,0),(eat2,0)])
```

Why does this scheduler fail to allow the other philosophers to eat? The explanation is as follows. All philosophers are able to enter their respective while loops (unlike in the previous case, where only the first philosopher managed to do so). However, once this occurs, a pattern emerges in which Philosopher 0 consistently benefits: when Philosopher 1 and Philosopher 2 reach their while loops, they get stuck in the thinking phase. This behavior arises from the calculation `mod lenListDist 3`: the resulting values cause the scheduler to repeatedly select the `skip` command for Philosopher 1 and Philosopher 2, preventing them from progressing. The reason is that, for Philosopher 1, every time he reaches the point where he may either become hungry or continue thinking, `lenListDist` is 4. Since `mod 4 3 = 1`, the scheduler selects the element in the middle of `listDist`, *i.e.* the list of possible computations. This corresponds to the option in which Philosopher 1 keeps thinking (the element with index 2 in `listDist`). For Philosopher 2, whenever he reaches an analogous choice, the last element of `listDist` happens to correspond to the option where he continues thinking. In these situations, `lenListDist` is 5, and `mod 5 3 = 2`, meaning that the scheduler selects the last element of `listDist`. Thus, Philosopher 2 is also forced to keep thinking. As a result, Philosopher 1 and Philosopher 2 never make progress, while Philosopher 0 repeatedly acquires both chopsticks and eats.

The fact that the two previous schedulers fail to let the second and third philosophers eat highlights an important property for verifying this algorithm: fairness. A fair scheduler must give each philosopher a chance to eat. The previous schedulers prioritized non-blocking progress but not fairness. This is evident since `nonBlockSch` always chooses the head of the list of possible computations, while `randNonBlockSchII` selects only among the head, middle, and last elements of that list based on a fixed calculation.

Let us now test `roundRobinSch`, the only scheduler we implemented that is known to be fair. We execute `runSem "thesis/dinphil3.txt" roundRobinSch`, again with 100 allowed steps. The result is:

```
0.0·([(st0,1),(st1,1),(st2,2),(cs0,0),(cs1,0),(cs2,1),
      (eat0,2),(eat1,1),(eat2,1)])
```

Finally, Philosopher 1 and Philosopher 2 are also able to eat. The other two schedulers were limited in their selection strategies: `nonBlockSch` always chose the head of the list of possible computations, and `randNonBlockSchII` restricted itself to choosing among the head, middle, or last elements. As a result, some distributions were never selected for evaluation. On the other hand, `roundRobinSch` imposes no such restriction, *i.e.* it can select any element from the list of possible next computations. This is because the index used to choose an element from `listDist` depends on the size of `listDist`. In particular, the calculation `mod lenAux lenListDist` always yields a value in the range from 0 to `lenListDist` $- 1$, allowing the scheduler to select any position in the list. Furthermore, `lenAux` constantly increases by one as a step of the computation is performed (recall that `lenAux` is the number of commands that have already been executed in the computation). This flexibility allows computations that were previously unreachable to be explored, enabling all philosophers to eat.

Using the criterion package [85], we tested the execution time of the Dining Philosophers problem with three philosophers, as presented in this section. We performed this test for each of the schedulers used: `nonBlockSch`, `randNonBlockSchII`, and `roundRobinSch`. Table 1 shows the results obtained. A more detailed analysis, provided by the criterion package, can be found in [37].

`nonBlockSch`	`randNonBlockSchII`	`roundRobinSch`
171.7 ms	173.2 ms	163.3 ms

Table 1: Execution time of the Dining Philosophers program for each scheduler

Under the hood, the schedulers `nonBlockSch` and `randNonBlockSchII` perform several auxiliary computations to verify whether the next configuration has already occurred. These additional checks explain their longer execution times compared to `roundRobinSch`.

10.4.2 Monty Hall

The Monty Hall problem [46] takes its name from Monty Hall, the host of the popular game show *Let's Make a Deal*, which first aired in the 1960s. In the show a contestant is presented with three doors.

Behind two of them there is no prize, while the remaining one hides the prize. The contestant must choose one of the doors. The rules are quite simple. If the contestant picks the door hiding the prize, they win. Otherwise, they lose. After the contestant makes the choice, the host opens a door that does not contain the prize. At this point, the host asks, "Do you want to change your choice?". Should the contestant stick with the original choice or change? Intuitively, many people decide to stay with their initial choice, assuming that the probability of winning is now $1/2$. However, this reasoning is incorrect because it ignores that one door has already been opened (an event that changes the probability distribution). Once this information is taken into account, switching doors actually doubles the chance of winning. Therefore, the answer to the host's question should be "Yes". This result was explained by Marilyn vos Savant in her column in Parade magazine in 1990 [113].

We can now check that indeed it is better to change, by encoding the Monty Hall with the language in Section 9.1 (see Figure 93).

The first line specifies that the prize is hidden, with equal probability, behind one of the three doors. The next line defines the contestant's initial choice, which is also made uniformly at random. The following series of `if-then-else` statements correspond to the host's action: opening a door that neither hides the prize nor matches the contestant's choice. Afterward, the contestant must decide whether to change the initial choice. If the contestant decides not to change, we do nothing and check if they win. If the contestant decides to change, we update the chosen door according and then verify whether switching was advantageous.

Let us consider a specific case: the contestant chooses door 1 and decides to change. Executing `runSem "examples/monty-hall.txt" initSch` (with the number of allowed steps set to 100; note that the choice of scheduler does not affect the behavior, since we explicitly set `change:=1`) yields the following result:

```
0.33333·([(prize,0),(cont,0),(open_curtain,2),(change,1),(won,1)]) +
0.33333·([(prize,1),(cont,2),(open_curtain,0),(change,1),(won,0)]) +
0.33333·([(prize,2),(cont,2),(open_curtain,0),(change,1),(won,1)])
```

As it is possible to see, the probability of winning when changing is $\frac{2}{3}$ while the probability of losing is $\frac{1}{3}$. The only scenario in which we lose is when the prize is hidden behind the door initially chosen.

Now what happens if the contestant does not change the initial choice (*i.e.* replace `change:=1` with `change:=0` in the code):

```
0.33333·([(prize,0),(cont,1),(open_curtain,2),(change,0),(won,0)]) +
```

```
prize:=0 (+)(1%3) (prize:=1 (+)(1%2) prize:=2);
cont:=0 (+)(1%3) (cont:=1 (+)(1%2) cont:=2);
if (prize==0) then {
  if (cont==0) then {open_curtain:=1 or open_curtain:=2}
  else {if (cont==1) then {open_curtain:=2} else {open_curtain:=1}}
}
else {if (prize==1) then {
  if (cont==0) then {open_curtain:=2}
  else {
    if (cont==1) then {open_curtain:=0 or open_curtain:=2}
    else {open_curtain:=0}
  }
}
else {if (cont==0) then {open_curtain:=1}
  else { if (cont==1) then {open_curtain:=0}
    else {open_curtain:=0 or open_curtain:=1}
  }
}; (change:=1 or change:=0);
if (change==0) then {skip} else {cont:= 3 - cont - open_curtain};
if (cont==prize) then {won:=1} else {won:=0}
```

Figure 93: Monty Hall code

```
0.33333·([(prize,1),(cont,1),(open_curtain,0),(change,0),(won,1)]) +
0.33333·([(prize,2),(cont,1),(open_curtain,0),(change,0),(won,0)])
```

It appears that not changing his initial choice brings "bad luck" to the contestant, since now the only situation in which he wins is when the prize is behind his initial guess.

Next, instead of fixing the contestant's initial choice, let us allow it to be selected uniformly among the three doors. We then collect 100 samples and visualize the results using histograms. However, we must note that when using the command `change:=1 or change:=0`, the scheduler assigns equal probability to both options. This symmetry eliminates the advantage of switching, as illustrated in Figure 95. Therefore, Figure 94 corresponds to the case where `change:=1`. Both histograms were obtained by

running `runHist "examples/monty-hall.txt"`.

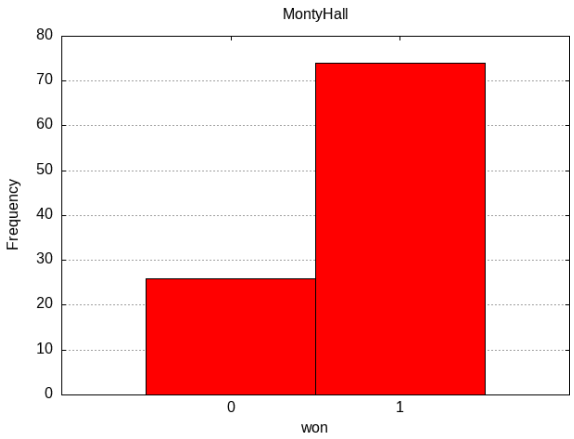


Figure 94: Histogram of the Monty Hall problem where the contestant opts to change his initial guess

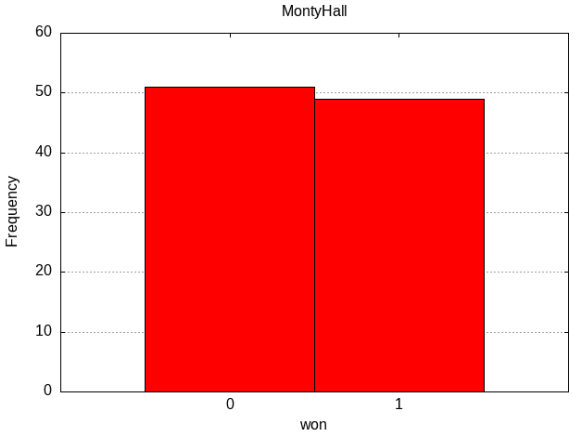


Figure 95: Histogram of the Monty Hall problem where the contestant can choose between changing or not his initial guess

The histogram in Figure 94 visually confirms the expected result: switching doubles the chances of winning. Conversely, Figure 95 confirms our earlier observation about scheduler interference when the contestant can probabilistically choose whether to switch or not. This behavior can be reproduced by setting `change:=1 (+)(1%2) change:=0` and executing `runSem "examples/monty-hall.txt" initSch` (for simplicity, assume the contestant initially chooses door 0):

```
0.16667·([(prize,0),(cont,2),(open_curtain,1),(change,1),(won,0)]) +
0.16667·([(prize,0),(cont,0),(open_curtain,1),(change,0),(won,1)]) +
0.16667·([(prize,1),(cont,1),(open_curtain,2),(change,1),(won,1)]) +
0.16667·([(prize,1),(cont,0),(open_curtain,2),(change,0),(won,0)]) +
0.16667·([(prize,2),(cont,2),(open_curtain,1),(change,1),(won,1)]) +
0.16667·([(prize,2),(cont,0),(open_curtain,1),(change,0),(won,0)])
```

Counting the outcomes shows an equal number of wins and losses, each with the same associated probability. This means that the contestant has an equal chance of winning or losing, demonstrating how probabilistic choices, whether arising from `(+)(1%2)` or from the scheduler, can interfere with the algorithm's expected behavior.

Using the criterion package [85], we also measured the execution time of a version of the Monty Hall problem where the contestant's initial choice is random, but the decision to switch or not is fixed. These examples are available in [37]. Table 2 shows the results.

Change initial guess	Not change initial guess
25.17 s	25.12 s

Table 2: Execution time for the Monty Hall

It comes as no surprise that both times are almost identical, around 25 seconds each. Although this might seem long for such a simple problem, most of this time is spent in the parser, due to the extensive nesting of `if-then-else` statements. Appendix B explains how the parser works. There, one can see that the parser for `if-then-else` statements is heavily affected by backtracking: before the correct parser is reached, several other command parsers are attempted and fail, forcing the parser to restore the consumed input and try again. This process is costly and, consequently, makes the parsing of `if-then-else` statements expensive. Furthermore, because an `if-then-else` statement contains two command blocks, one for the "true" branch and another for the "false" branch, the backtracking problem occurs twice, since each branch is parsed through the same mechanism.

10.4.3 Probabilistic Dining Philosophers

In [65], Lehmann and Rabin proposed a randomized solution to the Dining Philosophers problem. They considered a truly distributed and symmetric setting. The former refers to systems where there is no central memory or central process [7] accessible to all others. In the context of the Dining Philosophers problem, this means that philosophers can only communicate with their immediate neighbors, and that each chopstick is a shared variable accessible only to the two adjacent philosophers. The latter condition, symmetry, means that at the start of the protocol all philosophers are in the same state and all shared variables have the same initial value.

The strategy of Lehmann and Rabin differs from the one shown in Section 10.4.1, particularly in how philosophers attempt to pick up chopsticks. In Section 10.4.1, each philosopher tries to pick up both chopsticks simultaneously, *i.e.* `await(cs`$_i$`==1 & cs`$_{i-1}$`==1)`. If successful, the philosopher eats. Otherwise, they keep trying until they succeed. In contrast, in Lehmann and Rabin's solution, each philosopher first decides randomly, and with equal probability, which chopstick to attempt to pick up first: the left or the right. Let us suppose that a philosopher decides to pick up the left chopstick first. If it is unavailable,

[7] When Lehmann and Rabin exclude systems with a "central process", they are referring to an implemented process that interacts with the philosophers and controls their behavior. An example is a central scheduler that observes the actions of a Philosopher i and then decides that Philosopher j should act next. The distinction between such a central scheduler and the scheduler of Definition 8.2.1, which we use in our implementations, is that the latter is a mathematical abstraction (which may be implemented for semantic evaluation) and does not interact with the philosophers. It merely determines the order in which the global system evolves. A central scheduler, by contrast, communicates with the philosophers and instructs them when they are allowed to move.

the philosopher waits. If it is available, the philosopher picks it up and then checks whether the right chopstick is also available. If it is, the philosopher takes it and begins to eat. Otherwise, the philosopher releases the left chopstick and repeats the procedure.

The philosophers in Section 10.4.1 satisfy the truly distributed and symmetric conditions. However, because their behavior is deterministic, there exists a scheduler that can keep all philosophers in exactly the same state at every step. Such a scheduler preserves the symmetry among the philosophers and, as a consequence, none of them is ever able to eat. This is precisely the argument used by Lehmann and Rabin to show that there is no deterministic solution to the Dining Philosophers problem under a truly distributed and symmetric setting [65, Theorem 1]. To break this symmetry, Lehmann and Rabin allow each philosopher to choose randomly which chopstick to pick up first. This randomization breaks symmetry with probability one and enables Lehmann and Rabin to prove that, under any fair scheduler, every hungry philosopher eventually eats with probability one [65, Theorem 2].

Using the language introduced in Section 9.1, we specify in Figure 96 the behavior of Philosopher$_i$, where indices are taken modulo 3 (we are assuming three philosophers in the protocol). Furthermore, the code in Figure 96 is based on the pseudo-code in [65, Section 5]. Similarly to Section 10.4.1, the variables $\texttt{st}_i$ and $\texttt{eat}_i$, with $i \in \{0, 1, 2\}$, indicate Philosopher$_i$'s state and the number of times Philosopher$_i$ has eaten, respectively. The variables $\texttt{cs}_i$, with $i \in \{0, 1, 2\}$, represent the chopsticks, where 0 indicates that chopstick i is taken, and 1 indicates that it is free. In addition to these, we introduce $\texttt{try}_i$ and $\texttt{chops}_i$, with $i \in \{0, 1, 2\}$. The variable $\texttt{try}_i$ indicates whether Philosopher$_i$ is attempting to pick up a chopstick: if its value is 1, the philosopher is trying; if it is 0, the philosopher is not. The variable $\texttt{chops}_i$ specifies which chopstick philosopher i will try to pick first: if its value is -1, the philosopher tries the left chopstick; if it is 1, the right one.

Now, let us explain what Philosopher$_i$ does. The while loop controlled by $\texttt{st}_i$==1 represents the thinking stage. If the philosopher is not hungry, he continues thinking (via the `skip` command). Otherwise, he becomes hungry by setting $\texttt{st}_i$:=2. Once hungry, the philosopher decides whether to attempt to pick up a chopstick: $\texttt{try}_i$:=1 `or skip`. This decision is made within the loop controlled by $\texttt{try}_i$==0. When $\texttt{try}_i$ becomes 1, the philosopher proceeds to select, with equal probability, which chopstick to try first, left or right, using $\texttt{chops}_i$:=−1 (+)(1%2) $\texttt{chops}_i$:=1. Since the behavior is identical for both choices, let us consider the case where $\texttt{chops}_i$=1 (*i.e.* the philosopher tries to pick up the right chopstick first). If the chopstick is available, the philosopher picks it up ($\texttt{cs}_i$:=$\texttt{cs}_i$−1). Otherwise, he waits until it becomes available, as specified by the `await` command governed by the condition $\texttt{cs}_i$==1. Once the philosopher successfully picks up the right chopstick, he attempts to pick up the left one. If

```
while(true) do{
  while(st_i==1) do{st_i:=2 or skip};
  while(try_i==0) do{try_i:=1 or skip};
  while(try_i==1) do{
    (chops_i:=-1 (+)(1%2) chops_i:=1);
    if(chops_i==1) then{
        await(cs_i==1) do{cs_i:=cs_i-1};
        if(cs_{i-1}==1) then{cs_{i-1}:=cs_{i-1}-1; try_i:=0} else{cs_i:=cs_i+1}
    }
    else{
        await(cs_{i-1}==1) do{cs_{i-1}:=cs_{i-1}-1};
        if(cs_i==1) then{cs_i:=cs_i-1; try_i:=0} else{cs_{i-1}:=cs_{i-1}+1}
    }
  };
  st_i:=0; while(st_i==0) do{st_i:=1 or skip};
  eat_i:=eat_i+1; cs_i:=cs_i+1; cs_{i-1}:=cs_{i-1}+1
}
```

Figure 96: Code describing the behavior of Philosopher$_i$.

the left chopstick is available, he takes it ($\texttt{cs}_{i-1}\texttt{:=cs}_{i-1}\texttt{-1}$) and sets $\texttt{try}_i\texttt{:=0}$, indicating that he is no longer attempting to pick up chopsticks. If the left chopstick is unavailable, he releases the right one ($\texttt{cs}_i\texttt{:=cs}_i\texttt{+1}$) and repeats the process. After successfully picking up both chopsticks, the philosopher sets $\texttt{st}_i\texttt{:=0}$, indicating he is now eating. The loop controlled by $\texttt{st}_i\texttt{==0}$ represents the eating phase. In this loop, the philosopher continues eating (`skip`) or, once full, changes his state to thinking again ($\texttt{st}_i\texttt{:=1}$). Finally, the philosopher increments $\texttt{eat}_i\texttt{:=eat}_i\texttt{+1}$ to record that he has eaten and releases both chopsticks ($\texttt{cs}_i\texttt{:=cs}_i\texttt{+1; cs}_{i-1}\texttt{:=cs}_{i-1}\texttt{+1}$).

Let us now test the algorithm using the same schedulers as in Section 10.4.1.

We first execute `runSem "thesis/probdinphil3.txt" nonBlockSch` with 150 steps:

```
0.0·([(st0,2),(st1,1),(st2,1),(cs0,1),(cs1,1),(cs2,0),
```

```
      (try0,1),(try1,0),(try2,0), (chops0,-1),(chops1,0),(chops2,0),
      (eat0,6),(eat1,0),(eat2,0)]) +
0.0·([(st0,2),(st1,1),(st2,1),(cs0,0),(cs1,1),(cs2,1),
      (try0,1),(try1,0),(try2,0), (chops0,1),(chops1,0),(chops2,0),
      (eat0,6),(eat1,0),(eat2,0)])
```

Only Philosopher 0 manages to eat. This outcome matches the reasoning in Section 10.4.1: by freezing the other philosophers and allowing only the first one to proceed, the system state keeps changing, preventing blocking. Consequently, the scheduler never needs to find a non-blocking alternative.

Next, we test the `randNonBlockSchII` scheduler. Again with 150 allowed steps, the execution of `runSem "thesis/probdinphil3.txt" randNonBlockSchII` yields:

```
0.0·([(st0,2),(st1,1),(st2,1),(cs0,1),(cs1,1),(cs2,1),
      (try0,0),(try1,0),(try2,0),(chops0,-1),(chops1,0),(chops2,0),
      (eat0,4),(eat1,0),(eat2,0)]) +
0.0·([(st0,2),(st1,1),(st2,1),(cs0,1),(cs1,1),(cs2,1),
       (try0,0),(try1,0),(try2,0),(chops0,1),(chops1,0),(chops2,0),
       (eat0,4),(eat1,0),(eat2,0)])
```

Once again, only Philosopher 0 is able to eat. The reasoning mirrors that of the previous scheduler: a pattern emerges that favors the first philosopher, allowing him to eat four times instead of six, while the others starve

At last, we test the `roundRobinSch` scheduler. Following what was done previously, we now run `runSem "thesis/probdinphil3.txt" roundRobinSch` with 150 allowed steps. This execution yields 53 possible states (determined by counting the number of lines in the output, as each line represents one state). For readability, we show only a fragment of the results obtained corresponding to the different values of `eat0`, `eat1`, and `eat2`:

```
[[(eat0,1),(eat1,1),(eat2,1)],
 [(eat0,0),(eat1,1),(eat2,1)],
 [(eat0,0),(eat1,2),(eat2,1)]]
```

As it is possible to verify, under `roundRobinSch` all philosophers get the opportunity to eat, confirming our initial hypothesis. The reasoning is analogous to that discussed in Section 10.4.1.

As before, we now report the results obtained with the criterion package for each scheduler. The results are summarized in Table 3.

nonBlockSch	randNonBlockSchII	roundRobinSch
20.09 s	18.57 s	18.63 s

Table 3: Execution time of the Probabilistic Dining Philosophers for each scheduler

Interestingly, randNonBlockSchII is the fastest scheduler. Why does this happen? Let us first recall how the "no-blocking" mechanism of nonBlockSch and randNonBlockSchII works: if a scheduled computation already appears in the history, then the scheduler ignores it and selects an alternative. With this in mind, when these schedulers are used to execute the probabilistic Dining Philosophers (see Figure 96), we know that the computation never blocks. Consequently, both schedulers always proceed with the computation they choose first (after verifying that it is not present in the history). The scheduler nonBlockSch focuses solely on Philosopher 0, which requires many assignments, arithmetic operations, and Boolean evaluations. We believe these frequent memory accesses and updates slow down execution. Regarding roundRobinSch, it allows every philosopher to act. Although similar computations occur (assignments, arithmetic operations, and Boolean evaluations), some branches involve executing skip, which requires no memory updates and therefore speeds up execution. Finally, randNonBlockSchII causes Philosopher 1 and Philosopher 2 to remain in the loops controlled by $\texttt{try}_1$==0 and $\texttt{try}_2$==0, respectively. As a result, when they act, they repeatedly perform skip, avoiding memory operations entirely. This predominance of very cheap steps explains why randNonBlockSchII achieves the shortest overall execution time.

10.4.4 Quantum Teleportation

Recall the explanation given of the Quantum Teleportation protocol in Section 9.2.2. Here, we adapt the quantum teleport protocol to the language in Section 9.1. The description of the algorithm is as follows:

```
H(q2); CNOT(q2,q3);
CNOT(q1,q2); H(q1); Meas(x2,q2); Meas(x1,q1);
if (x2==0) then {skip} else {X(q3)};
if (x1==0) then {skip} else {Z(q3)}
```

Charlie's contribution to the algorithm remains unchanged. However, the actions performed by Alice and Bob are modified. In the language of Section 9.1, the measurement command no longer behaves like an if-then-else statement (as it did in Section 9.2.1). Instead, it stores the measurement result in a classical variable. Thus, Alice can measure the qubits she holds. Since the measurement outcomes are

now recorded in classical variables, Bob can consult them to apply the appropriate gates to his qubit. This small change in the measurement command allows us to express Alice's and Bob's actions in accordance with the standard description of the quantum teleportation protocol.

To check that quantum teleportation behaves as expected, we evaluate the above program with our interpreter. The state to be teleported is $|0\rangle$. The results obtained are as follows:

```
0.25·([(x1,0),(x2,0),(x3,0)], (1.0)|000><000|) +
0.25·([(x1,1),(x2,0),(x3,0)], (1.0)|100><100|) +
0.25·([(x1,0),(x2,1),(x3,0)], (1.0)|010><010|) +
0.25·([(x1,1),(x2,1),(x3,0)], (1.0)|110><110|)
```

To confirm that the protocol works correctly, we must check whether the third qubit is in the state $|0\rangle$. As shown in the results above, the third qubit is indeed in state $|0\rangle$, meaning that the protocol behaved as expected.

This result was obtained using `runSem` with the scheduler `initSch`. Since the algorithm is entirely sequential, there is no need for a more sophisticated scheduler.

Instead of teleporting only the state $|0\rangle$, we can exploit the fact that the language in Section 9.1 includes non-deterministic choice. In particular, we can use this construct to select among several possible states to be teleported. The modified algorithm is shown in Figure 97.

```
H(q2); CNOT(q2,q3)
(skip; i:=0) or (H(q1); i:=1) or (X(q1); i:=2);
CNOT(q1,q2); H(q1);
Meas(x2,q2); Meas(x1,q1);
if (x2==0) then skip else X(q3);
if (x1==0) then skip else Z(q3);
Meas(x3,q3)
```

Figure 97: Non-deterministic quantum teleportation protocol

In Figure 97, each possible state to teleport is associated with a classical variable `i`. The initial state of the program is $|000\rangle$, so we need a mechanism to identify which state has been selected. This role is played by the classical variable `i`, which records the branch chosen by the non-deterministic choice.

Thus, in this modified version of quantum teleportation, the statement `(skip; i:=0) or (H(q1); i:=1) or (X(q1); i:=2)` defines the different states to teleport and stores the chosen one in `i`. Concretely:

- if `i` = 0, the teleported state is $|0\rangle$
- if `i` = 1, the teleported state is $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$
- if `i` = 2, the teleported state is $|1\rangle$

We ran this algorithm with `runSem` using the various schedulers defined earlier (`nonBlockSch`, `randNonBlockSchII`, `roundRobinSch`) and observed that all of them produced the same results. To break this monotony, we use the interpreter's histogram feature. Recall that histogram generation relies on a predefined scheduler (see Section 10.2). In sum, this scheduler assigns equal probability to all possible next transitions, randomly selects one distribution, and afterwards, based on the probabilities associated with that distribution, randomly selects one element from its support. In this way, a state to be teleported is chosen at random. Note that at the end of the algorithm we measure the qubit q3. This is because histograms are built using only classical variables. Therefore, when `i` = 1 (*i.e.* when the state to be teleported is $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$), the variable `x3` should take the value 0 or 1, each occurring with approximately equal probability.

We now run the algorithm 100 times. The results are shown in Figure 98.

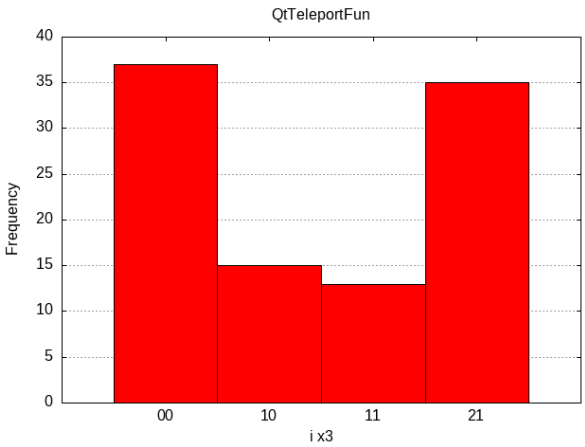


Figure 98: Histogram of the non-deterministic quantum teleportation protocol

How should we interpret the data in Figure 98, particularly the x-axis? The values on the x-axis are formed by concatenating the values of `i` and `x3`. For example, the value `00` occurs about 37 times, meaning that `i` = 0 and the teleported state was $|0\rangle$, which corresponds to `x3` = 0. A similar situation

occurs for 21, which appears 35 times, corresponding to i = 2 (the state $|1\rangle$) and x3 = 1. Finally, the values 10 and 11 both correspond to i = 1, indicating that the teleported state was $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The value of x3 = 0 or 1 indicates the measurement outcome of the teleported qubit. Both 10 and 11 appear with roughly the same frequency, as expected, since the probabilities of measuring 0 or 1 are equal.

According to the results obtained using the criterion package, the execution time of this algorithm was 8.818 ms.

10.4.5 Grover's algorithm

Recall the explanation of Grover's algorithm presented in Section 9.2.2. Here, we adapt the execution of Grover's algorithm to the language introduced in Section 9.1. The description of the algorithm is as follows:

```
i:=0; j:=0;
H(q1); H(q2);X(q3); H(q3);
if (i==0 & j==0) then {X(q1); X(q2); TOF(q1,q2,q3); X(q1); X(q2)}
  else{
    if (i==0 & j==1) then {X(q1); TOF(q1,q2,q3); X(q1)}
    else{
      if (i==1 & j==0) then {X(q2); TOF(q1,q2,q3); X(q2)}
      else {TOF(q1,q2,q3)}
    }
  }
H(q1); X(q1); H(q2); X(q2);CZ(q1,q2);X(q1); H(q1); X(q2); H(q2);
Meas(x1,q1); Meas(x2,q2);Reset(q3)
```

Figure 99: Grover's algorithm code

where the variables i and j encode the location of the variable we want to find. The execution of Reset(q3), at the end of the computation, "discards" the ancilla qubit to ease the interpretation of the resulting state.

Using the tool described in this section, we execute the algorithm and obtain the following result:

```
1.0·([(x1,0),(x2,0),(i,0),(j,0)], (1.0)|000><000|)
```

This indicates that the element 00 was found with probability 1.

We also use this example to illustrate the importance of the threshold mechanism introduced in the implementation of measurement. The result shown above includes the threshold mechanism. However, if we remove it and run the algorithm again, we obtain the following result:

```
1.0·([(x1,0),(x2,0),(i,0),(j,0)], (1.0)|000><000|) +
0.0·([(x1,0),(x2,1),(i,0),(j,0)], (1.0+1.64372i)|010><010|) +
0.0·([(x1,1),(x2,0),(i,0),(j,0)], (1.0-0.41093i)|100><100|) +
0.0·([(x1,1),(x2,1),(i,0),(j,0)], (1.0+3.5708736047928565e15i)|110><110|)
```

As it is possible to see, unwanted states appeared. They appear with probability 0, because when representing the results we round the probabilities to 5 digits. If rounding were disabled, the state `|110><110|` would have a probability of approximately `-6.842277657836021e-49`. Although this value is essentially zero, the mere presence of such states can make heavy computations even more demanding. Therefore, the threshold mechanism not only improves numerical stability but also reduces computational cost.

Both results were obtained by executing `runSem` with the scheduler `initSch`, since the algorithm is sequential.

Similarly to what was done in Section 10.4.4, we can also test Grover's algorithm with different inputs, randomly chosen. The only change required is in the first line of the program. Instead of `i:=0; j:=0`, we write: `(i:=0 or i:=1); (j:=0 or j:=1)`. The histograms produced by executing this modified algorithm 100 times are shown in Figures 100 and 101.

The histogram in Figure 100 shows how often the elements 00, 01, 10, and 11 were picked as the element to be searched for. On the other hand, the histogram in Figure 101 shows how often each element was found. As we can see, the frequency of 00 in Figure 100 matches the frequency of 00 in Figure 101. The same pattern holds for the other values. This confirms that Grover's algorithm successfully identifies the element chosen at the beginning of the computation.

According to the results obtained using the criterion package, the execution time of this algorithm was 785.9 ms.

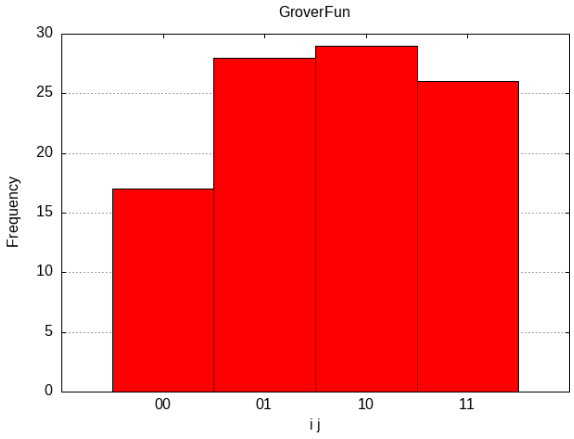


Figure 100: Histogram of Grover's algorithm showing the frequency with which each possible target (i, j) was selected.

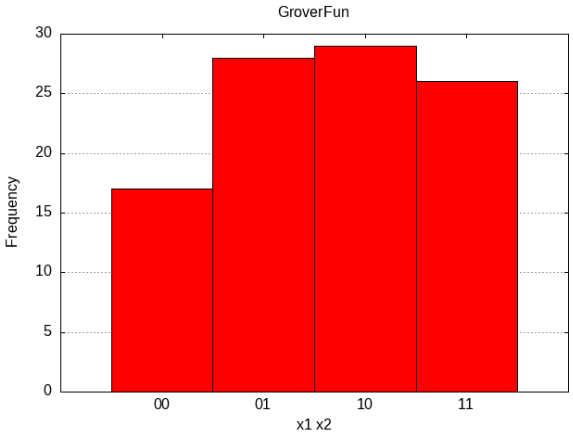


Figure 101: Histogram of Grover's algorithm showing the frequency with which each outcome $(x1, x2)$ was measured.

10.4.6 Superdense Coding with Grover's Algorithm

We conclude the adaptation of the algorithms presented in Section 9.2.2 with a combination of superdense coding and Grover's algorithm. The code is as follows:

Since the language introduced in Section 9.1 handles both classical and quantum variables, we can resort to that fact to reduce the number of qubits required. Thus, we store the results of the superdense coding in the variables `x1` and `x2` (note that the values of `x1` and `x2` are later copied to `i` and `j`, respectively, to specify which item to search for and, consequently, which oracle to apply). Before executing Grover's component, we reset qubits `q1` and `q2`. This operation initializes both qubits to state $|0\rangle$, which is necessary for Grover's algorithm to behave as expected. After running Grover's algorithm, we measure qubits `q1` and `q2` and store the results in `i` and `j`, respectively. The outcome of executing this program 100 times is shown in Figure 103.

As shown in Figure 103, whenever superdense coding selects an element to be searched by Grover's algorithm, that same element is always found. This is visible in the histogram: whenever the superdense coding outputs $\mathtt{x}_1 = 0$ and $\mathtt{x}_2 = 0$, Grover's algorithm returns $\mathtt{i} = 0$ and $\mathtt{j} = 0$. The same behavior is verified for all other possible values.

According to the results obtained using the criterion package, the execution time for this algorithm was 818.4 ms.

```
H(q1); CNOT(q1,q2);
skip or X(q1) or Z(q1) or (X(q1);Z(q1));
CNOT(q1,q2); H(q1);
Meas(x1,q1); Meas(x2,q2);Reset(q1); Reset(q2);
i:=x1; j:=x2; H(q1); H(q2); X(q3); H(q3);
if (i==0 & j==0) then{X(q1); X(q2); TOF(q1,q2,q3); X(q1); X(q2)}
  else{
    if (i==0 & j==1) then{X(q1); TOF(q1,q2,q3); X(q1)}
    else{
      if (i==1 & j==0) then{X(q2); TOF(q1,q2,q3); X(q2)}
      else{TOF(q1,q2,q3)}
    }
  }
H(q1); X(q1); H(q2); X(q2);CZ(q1,q2);X(q1); H(q1); X(q2); H(q2);
Meas(i,q1); Meas(j,q2);Reset(q3)
```

Figure 102: Superdense Coding with Grover's code

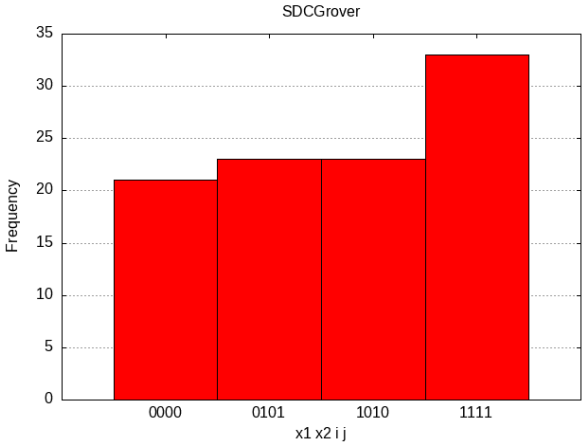


Figure 103: Histogram of the superdense coding combined with Grover's algorithm

10.4.7 Quantum Dining Philosophers

We reach the last example: the quantum version of the Dining Philosophers problem [5]. The setting is the same as in Section 10.4.3, *i.e.* a truly distributed and symmetric setting. In this version, two

solutions are presented: one in which the number of philosophers is known, and another where only an upper bound is known. Here, we tackle the case where the number of philosophers is known, since the solution for the upper bound case uses instructions that our interpreter does not support (*e.g.* the definition of functions and their use within a program). Following the idea of [65], *i.e.* that symmetry must be broken to ensure that all philosophers eventually get to eat, Aharonov *et al.* used quantum phenomena to achieve symmetry breaking. They observed that a modified first phase of the quantum leader election protocol [104] is sufficient to break the system's symmetry. This step divides the philosophers into two groups: the 0-group, which tries to pick up the left chopstick first, and the 1-group, which tries to pick up the right chopstick first. After this step, each hungry philosopher attempts to pick up chopsticks according to the group they belong to. At this point, the strategy differs from the previous versions: whenever a philosopher successfully picks up one chopstick, he keeps it until he can acquire the other. Once the second chopstick is obtained, the philosopher eats and, when finished, releases both chopsticks and resumes thinking.

Let us dive into more details. The symmetry breaking protocol [5, Algorithm 2] is divided in four phases. Initially all philosophers are thinking. At any time, one or more philosophers may become hungry and want to eat. When this happens, all philosophers, including those who do not want to eat, must participate in the symmetry-breaking protocol [8]. Each philosopher requires three qubits: two principal qubits and one ancilla qubit. We denote the principal qubits by $q_{1,i}$ and $q_{2,i}$, and the ancilla qubit by $q_{a,i}$, for philosopher i. All qubits are initially in the state $|0\rangle$ (this means that if the quantum dining philosophers protocol is executed more than once, the qubits must be reset). We now explain the four phases.

- First Phase

  The protocol begins with each philosopher entangling their qubits $q_{1,i}$ and $q_{2,i}$ to create the Bell state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Then $q_{2,i}$ is sent to the right neighbor, along with a wake-up message, after which the philosopher goes to sleep. Note that, since we are in a shared-variable setting, no physical transmission occurs. Instead, the right neighbor is granted permission to access the qubit $q_{2,i}$. Throughout the explanation, we will continue to use the terms sending and receiving for consistency with [5]. Next, a philosopher receives the qubit $q_{2,i-1}$ from their left neighbor. They must now check whether $q_{1,i}$ and $q_{2,i-1}$ are equal. Since directly measuring both qubits would destroy the entangled state and disrupt the rest of the protocol, the philosopher uses their ancilla qubit $q_{a,i}$ to perform this comparison. Specifically, they apply $CNOT(q_{1,i}, q_{a,i})$ followed

[8] All participants become aware that someone is hungry, and consequently that they must participate in the symmetry-breaking protocol, through a message passed between neighbors.

by $CNOT(q_{2,i-1}, q_{a,i})$, and then measure the ancilla qubit. If $q_{1,i}$ and $q_{2,i-1}$ are equal, they store the value 0 in the classical variable x_i. Otherwise, if they differ, x_i is assigned the value 1. Note that if $x_i = 1$, symmetry has been broken.

- Second Phase

  After storing the result of the ancilla measurement in x_i, the philosopher sends this classical value to the left neighbor, along with a wake-up message. The purpose of sending x_i is to indicate whether symmetry has been broken. Note that in the first phase, qubits were sent to the right neighbor, whereas in this phase, classical information is sent to the left. Next, a local counter c_i is initialized to 1. This counter tracks the number of messages received from the right neighbor. The philosopher then enters a loop that continues until either (a) the number of messages equals the total number of philosophers, or (b) one of the messages indicates that symmetry has been broken. If the loop halts for reason (a), the philosopher proceeds to the third phase; otherwise, they proceed directly to the fourth phase. Within the loop, each philosopher first sleeps, then receives a message from the right neighbor, increments the counter, and forwards the message to the left neighbor along with a wake-up signal. If the received message equals 1, symmetry has been broken and the philosopher exits the loop, proceeding to the fourth phase. Otherwise, they continue exchanging messages until the number of messages equals the number of philosophers, indicating that symmetry was not broken, after which all philosophers proceed to phase three.

- Third Phase

  In the third phase, each philosopher applies a unitary operation that performs the following mapping:

  $$|00\rangle \mapsto |00\rangle \qquad |11\rangle \mapsto |10\rangle$$

  Since the mappings of $|01\rangle$ and $|10\rangle$ are irrelevant, we can simply apply a CNOT gate, *i.e.* $CNOT(q_{1,i}, q_{2,i-1})$. At this point, we have two scenarios regarding the number of philosophers:

  - If the number of philosophers is even, the magic unitary U_n (with n being the total number of philosophers) is applied to $q_{1,i}$
  - If the number of philosophers is odd, a $CNOT$ gate is first applied to $q_{1,i}$ and $q_{a,i}$[9], followed by the application of another magic unitary V_n to $q_{1,i}$ and $q_{a,i}$.

[9] In [5, Appendix A], it is stated that the $CNOT$ gate should act on $q_{1,i}$ and a newly initialized qubit in state $|0\rangle$. However, since we are in the third phase, symmetry has not been broken and thus $q_{a,i}$ is already in state $|0\rangle$. Using $q_{a,i}$ therefore reduces the number of qubits required in the protocol.

Both magic unitary gates were developed in [104] and their definition is as follows:

$$U_n = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & e^{-i\frac{\pi}{n}} \\ -e^{i\frac{\pi}{n}} & 1 \end{pmatrix}$$

$$V_n = \frac{1}{\sqrt{1+R_n}} \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \sqrt{R_n} & \frac{e^{i\frac{\pi}{n}}}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\sqrt{R_n}e^{-i\frac{\pi}{n}} & \frac{e^{-i\frac{\pi}{n}}}{\sqrt{2}} \\ \sqrt{R_n} & 0 & \frac{e^{-i\frac{\pi}{2n}}I_n}{i\sqrt{2}R_{2n}} & -\sqrt{R_n} \\ 0 & \sqrt{1+R_n} & 0 & 0 \end{pmatrix}$$

where R_n and I_n correspond to the real and imaginary part of $e^{i\frac{\pi}{n}}$, respectively.

- Fourth Phase

  In the fourth and final phase, each philosopher measures their qubit $q_{1,i}$ and stores the result in the classical variable g_i, which identifies the group to which philosopher i belongs.

Using the language introduced in Section 9.1, we now encode the pseudo-code from [5, Algorithm 1] and instantiate it for three philosophers, as shown in Figure 104 (the operations $\ominus$ and $\oplus$ are modulo 3). Note that the pseudo-code in [5, Algorithm 1] is written resorting to message communications. Here, we implement it using a shared-variable approach, where message passing between adjacent neighbors is replaced by access to shared classical and quantum variables. As in the previous Dining Philosophers examples (Section 10.4.1 and Section 10.4.3), variables $\texttt{st}_i$ and $\texttt{eat}_i$, with $i \in \{0,1,2\}$, indicate philosopher i's state and the number of times philosopher i has eaten, respectively. We also retain variable $\texttt{cs}_i$, with $i \in \{0,1,2\}$, to represent the chopsticks: the value 0 indicates that chopstick i is taken, while 1 means it is free. In addition to these, we introduce the following variables (with $i \in \{0,1,2\}$):

- $\texttt{doneSB}_i$: indicates whether the symmetry-breaking protocol has been completed
- $\texttt{doSB}_i$: a shared variable that signals Philosopher$_i$ to start the symmetry-breaking protocol
- $\texttt{x}_i$: stores the result of the measurement that checks whether two qubits are equal
- $\texttt{isEq}_i$: equals 0 if the qubits differ and 1 otherwise
- $\texttt{wake}_i$: a shared variable that wakes up Philosopher$_i$
- $\texttt{c}_i$: counts the messages received

- $\mathtt{g}_i$: identifies the group to which Philosopher$_i$ belongs
- $\mathtt{n}$: total number of philosophers
- $\mathtt{msg10}$, $\mathtt{msg21}$, $\mathtt{msg02}$: variables shared by philosophers 1–0, 2–1, and 0–2, respectively, to exchange information
- $\mathtt{q}_i$: the principal qubit owned by Philosopher$_i$
- $\mathtt{qa}_i$: the ancilla qubit owned by Philosopher$_i$
- $\mathtt{q20}$, $\mathtt{q12}$, $\mathtt{q01}$: qubits that are entangled with $\mathtt{q}_2$, $\mathtt{q}_1$, and $\mathtt{q}_0$, respectively, to be used by the neighboring philosopher

Let us now explain what Philosopher$_i$ does. The while loop satisfying $\mathtt{st}_i$==1 & $\mathtt{doSB}_i$==0 represents the phase in which the philosopher is thinking. If the philosopher is not hungry but is signaled to perform the symmetry-breaking protocol, he interrupts thinking to execute that protocol. If the symmetry-breaking protocol has not yet been performed, the first philosopher updates the shared variable $\mathtt{doSB}_{i\oplus 1}$:=1; to notify Philosopher$_{i\oplus 1}$ that it is time to begin the symmetry breaking. Next, he waits for the instruction to proceed (await($\mathtt{doSB}_i$==1)). Once that signal is received, he resets all qubits under his control to ensure that the protocol starts with every qubit in the $|0\rangle$ state. The first phase of the symmetry-breaking protocol then begins: the philosopher entangles qubits $\mathtt{q}_i$ and $\mathtt{q}_{i,i\oplus 1}$ and sets $\mathtt{wake}_{i\oplus 1}$:=1 to wake Philosopher$_{i\oplus 1}$. He then idles until Philosopher$_{i\ominus 1}$ sets $\mathtt{wake}_{i\ominus 1}$:=1. Next, Philosopher$_i$ checks whether qubits $\mathtt{q}_i$ and $\mathtt{q}_{i\ominus 1,i}$ have the same value. He uses the auxiliary qubit $\mathtt{qa}_i$ for this purpose, executing the following sequence: CNOT($\mathtt{q}_i$, $\mathtt{qa}_i$); CNOT($\mathtt{q}_{i\ominus 1,\,i}$, $\mathtt{qa}_i$); Meas($\mathtt{x}_i$, $\mathtt{qa}_i$). If the measurement yields 0, $\mathtt{isEq}_i$ is set to 0; otherwise, it is set to 1. The second phase then begins. The shared variable $\mathtt{msg}_{i,\,i\ominus 1}$, used for communication between Philosopher$_i$ and Philosopher$_{i\ominus 1}$, is assigned the value of $\mathtt{isEq}_i$. The counter $\mathtt{c}_i$ is initialized to 1 ($\mathtt{c}_i$:=1), and a wake-up signal is sent to Philosopher$_{i\ominus 1}$ ($\mathtt{wake}_{i\ominus 1}$:=1). Next, Philosopher$_i$ enters a loop satisfying $\mathtt{c}_i$<n & $\mathtt{msg}_{i\oplus 1,\,i}$==0 & $\mathtt{isEq}_i$==0. In words:

- if the number of messages equals the number of philosophers, the loop terminates (the message has circulated through all philosophers); this means symmetry was not broken, and phase three begins
- if the shared variable $\mathtt{msg}_{i\oplus 1,\,i}$ equals 1, symmetry was broken by another philosopher, and the loop ends immediately, moving to phase four

```
while(true) do{
  while(st_i==1 & doSB_i==0) do{ st_i:=2 or skip};
  if(doneSB_i==0) then{
    doSB_{i⊕1}:=1; await(doSB_i==1) do{Reset(q_i); Reset(qa_i); Reset(q_{i,i⊕1})};
    H(q_i); CNOT(q_i, q_{i,i⊕1}); wake_{i⊕1}:=1; while(wake_i==0) do{skip};
    CNOT(q_i, qa_i); CNOT(q_{i⊖1,i}, qa_i); Meas(x_i, qa_i);
    if(x_i==0) then{isEq_i:=0} else{isEq_i:=1};
    msg_{i,i⊖1}:=isEq_i; c_i:=1; wake_{i⊖1}:=1;
    while(c_i<n & msg_{i⊕1,i}==0 & isEq_i==0) do{
      wake_i:=0; msg_{i⊕1,i}:=msg_{i⊕2,i⊕1}; c_i:=c_i+1; msg_{i,i⊖1}:=msg_{i⊕1,i}; wake_{i⊖1}:=1;
    };
    if(c_i==n) then{CNOT(q_i,q_{i⊖1,i}); CNOT(q_i,qa_i); Vmag3(q_i,qa_i)} else{skip};
    Meas(g_i, q_i); doneSB_i:=1; doSB_i:=0;
  } else {skip};
  while(st_i==2) do{ if(g_i==0) then{
      await(cs_{i⊖1}==1) do{cs_{i⊖1}:=cs_{i⊖1}-1}; await(cs_i==1) do{cs_i:=cs_i-1}
    } else{
      await(cs_i==1) do{cs_i:=cs_i-1}; await(cs_{i⊖1}==1) do{cs_{i⊖1}:=cs_{i⊖1}-1}
    }; st_i:=0
  };
  if(st_i==0) then{st_i:=1; eat_i:=eat_i+1; cs_{i⊖1}:=cs_{i⊖1}+1; cs_i:=cs_i+1; doneSB_i:=0}
  else {skip}
}
```

Figure 104: Code describing the behavior of Philosopher$_i$

- if $\texttt{isEq}_i$ equals 1, Philosopher 1 himself broke symmetry, and he also proceeds directly to phase four

Inside the loop, the philosopher first goes to sleep, then updates $\texttt{msg}_{i\oplus 1,\, i}$ with the value in $\texttt{msg}_{i\oplus 2,\, i\oplus 1}$

(corresponding to receiving a message from the right neighbor). The counter $\texttt{c}_i$ is then incremented, $\texttt{msg}_{i,\, i\ominus 1}$ is updated with the new $\texttt{msg}_{i\oplus 1,\, i}$ value (corresponding to sending a message to the left neighbor), and a wake-up signal is sent to Philosopher$_{i\ominus 1}$ ($\texttt{wake}_{i\ominus 1}$:=1). After exiting the loop, two cases arise depending on why it ended:

- If $\texttt{c}_i$ equals the number of philosophers, the philosopher executes the third phase, applying the following sequence of unitary gates: CNOT($\texttt{q}_i$, $\texttt{q}_{i\ominus 1,\, i}$); CNOT($\texttt{q}_i$, $\texttt{qa}_i$); Vmag3($\texttt{q}_i$, $\texttt{qa}_i$)
- Otherwise (if symmetry was broken), he proceeds directly to the fourth phase, where he measures qubit $\texttt{q}_i$ and stores the result in $\texttt{g}_i$, identifying the group to which he belongs

At this point, the symmetry-breaking protocol is complete. If Philosopher$_i$ is not hungry, he skips the remaining computations and resumes thinking. Otherwise, his behavior depends on his group:

- If Philosopher$_i$ belongs to the 0-group, he first attempts to pick up his left chopstick until successful (await($\texttt{cs}_{i\ominus 1}$==1) do{$\texttt{cs}_{i\ominus 1}$:=$\texttt{cs}_{i\ominus 1}$-1};), then does the same for his right chopstick (await($\texttt{cs}_i$==1) do{$\texttt{cs}_i$:=$\texttt{cs}_i$-1})
- If he belongs to the 1-group, the order of these actions is reversed

In both cases, if a chopstick is unavailable, Philosopher$_i$ idles until it becomes free. After obtaining both chopsticks, he sets $\texttt{st}_i$:=0 to indicate that he is eating. Once finished, he updates his state ($\texttt{st}_i$:=1), increments the counter $\texttt{eat}_i$:=$\texttt{eat}_i$+1, releases the chopsticks ($\texttt{cs}_{i\ominus 1}$:=$\texttt{cs}_{i\ominus 1}$+1; $\texttt{cs}_i$:=$\texttt{cs}_i$+1), and resets the variable $\texttt{doneSB}_i$:=0 to prepare for the next round.

Let us now test the algorithm. For this, we shall use the same schedulers as in Section 10.4.1 and Section 10.4.3. However, unlike in the previous sections, we now expect that all schedulers will allow more than one philosopher to eat. This comes from the fact that for Philosopher$_i$ to proceed with his computation, he must wait for a signal from Philosopher$_{i\ominus 1}$. If Philosopher$_{i\ominus 1}$ does not send the signal, the computation blocks, and the schedulers (nonBlockSch and randNonBlockSchII) will try a different distribution to enable the computation to continue.

Having said this, we execute the algorithm with all three schedulers. As in Section 10.4.3 for the roundRobinSch case, we omit here the full computation state. The complete results can be found in [37, ./examples/thesis/results], while we focus only on the variable $\texttt{eat}_i$ (recall that we are considering three philosophers).

We begin by running runSem "thesis/qtdinphil3.txt" nonBlockSch, with 200 being the number of allowed steps. For readability we only show the results obtained for the variable $\texttt{eat}_i$.

```
[[(eat0,1),(eat1,1),(eat2,0)]]
```

As we can see, Philosopher 0 and Philosopher 1 had the opportunity to eat, while Philosopher 2 did not. This confirms our intuition that the scheduler detected a blocked computation and switched to another one.

Next, we run `runSem "thesis/qtdinphil3.txt" randNonBlockSchII`, again with 200 allowed steps. For readability we only show the relevant results obtained for the variable $\texttt{eat}_i$.

```
[[(eat0,1),(eat1,0),(eat2,1)],
 [(eat0,2),(eat1,0),(eat2,1)]]
```

As we can see, Philosopher 0 and Philosopher 2 ate, while Philosopher 1 did not. This suggests that `randNonBlockSchII` produced a pattern in which Philosopher 1 never eats. Moreover, in line with the previous case, this confirms our expectation that the scheduler identified a blocked computation and explored another distribution.

At last, we run `runSem "thesis/qtdinphil3.txt" roundRobinSch`, with 200 steps. For readability we only show the results obtained for the variable $\texttt{eat}_i$.

```
[[(eat0,0),(eat1,0),(eat2,0)],
 [(eat0,1),(eat1,1),(eat2,0)],
 [(eat0,1),(eat1,0),(eat2,1)],
 [(eat0,1),(eat1,1),(eat2,1)]]
```

Not surprisingly, all the philosophers had the chance to eat.

Similarly to Section 10.4.1 and Section 10.4.3, we now show the results obtained from criterion for each of the schedulers used. The results can be found in Table 4.

`nonBlockSch`	`randNonBlockSchII`	`roundRobinSch`
360.1 s	1279 s	1275 s

Table 4: Execution time of the Quantum Dining Philosophers for each scheduler

Once again we observe a surprising result *i.e.* a striking difference in execution times: the scheduler `nonBlockSch` completes one execution of the algorithm in roughly six minutes, while the others take around twenty-one minutes. Why is there such a large difference? The explanation is related to the discussion in Section 10.3. Note that Philosopher 1 only performs an action if Philosopher 0 blocks the computation, and Philosopher 2 acts only when Philosopher 0 and Philosopher 1 block the computation.

From this interaction among the philosophers we can infer that Philosopher 2 will only have the opportunity to execute the symmetry-breaking protocol once Philosopher 0 and Philosopher 1 have already eaten and are again waiting to perform the protocol. Consequently, the quantum state in this scenario is less complex and less memory-consuming. In other words, for each iteration, the size of the quantum state that must be maintained is smaller compared to the cases under the other schedulers. As we can see, this small detail has a significant impact on the time `nonBlockSch` takes to complete one execution of the program. In contrast, the schedulers `randNonBlockSchII` and `roundRobinSch` produce situations in which all three philosophers are actively participating in the symmetry-breaking protocol. This increases the size and complexity of the quantum state, thereby making the associated operations slower and the overall computation time longer.

Before concluding, note that in the results obtained with `roundRobinSch`, we observe a case in which no philosopher eats. Following [5], this outcome is surprising: since the number of philosophers is known, one would expect that after executing the symmetry-breaking protocol, the symmetry would indeed be broken. However, this is not what happens, and we now briefly explain why (a detailed discussion and full calculations can be found in Appendix E).

The issue lies in the symmetry-breaking protocol [5, Algorithm 2], specifically in the use of the magic unitary gate V_n. Consider the case with three philosophers. After applying the protocol, the states of the qubits `q1`, `q2`, and `q3` should not all be equal to $|0\rangle$ nor all equal to $|1\rangle$, since such configurations would mean that all philosophers attempt to pick the same chopstick (left or right), potentially leading to deadlock. However, explicit calculations reveal cases in which the states of `q1`, `q2`, and `q3` are indeed all equal.

To analyse this phenomenon, we use Table 5, following the format of [86, Table 3], to track the amplitudes and identify where the expected cancellations fail. The table is interpreted as follows. For each basis state (rows), we list the amplitudes obtained from the two interfering branches $|000000\rangle$ and $|111111\rangle$. For example, the amplitude of $|000000\rangle$ arising from calculations starting in $|000000\rangle$ is $\frac{1}{2\sqrt{2}}$, while from $|111111\rangle$ it is $-\frac{1}{2\sqrt{2}}$. Consequently, their sum is 0. Proceeding similarly for the remaining entries, we find that six out of nine terms that should cancel do not. Hence, there is a non-zero probability that all philosophers end up in the 0-group. These calculations are fully consistent with the behavior observed in practice.

The reader may now ask what this implies. It implies that the algorithm does not deterministically solve the quantum Dining Philosophers problem efficiently, *i.e.* [5, Theorem 3] does not hold as stated. The proof of that theorem relies on [5, Algorithm 2] successfully breaking the symmetry of the system. But

	$\|000000\rangle$	$\|111111\rangle$	$+$
$\|000000\rangle$	$\frac{1}{2\sqrt{2}}$	$-\frac{1}{2\sqrt{2}}$	0
$\|000001\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{i\frac{\pi}{3}})$
$\|000100\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{i\frac{\pi}{3}})$
$\|000101\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{-i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{-i\frac{\pi}{3}})$
$\|010000\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{i\frac{\pi}{3}})$
$\|010001\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{-i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{-i\frac{\pi}{3}})$
$\|010100\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{-i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{-i\frac{\pi}{3}})$
$\|010101\rangle$	$\frac{1}{2\sqrt{2}}$	$-\frac{1}{2\sqrt{2}}$	0
$\|101010\rangle$	$\frac{1}{2\sqrt{2}}$	$-\frac{1}{2\sqrt{2}}$	0

Table 5: Amplitudes of the states that should cancel out

as Table 5 demonstrates, Algorithm 2 does not always break the symmetry, and therefore cannot serve as a symmetry-breaking procedure. Consequently, the theorem does not hold in this form. Nevertheless, the theorem would hold under a formulation similar to that of [65]. This conjecture is supported by the fact that the probability of all philosophers belonging to the 0-group is extremely small and decreases toward zero as the number of iterations increases.

Chapter 11

Closing the stage

11.1 Related Work

The interpreter presented in Part II processes the language introduced in Section 9.1, which combines three computational paradigms – concurrent, probabilistic, and quantum computation – into a single unified language. The tool enables reasoning about programs belonging to each of these paradigms individually, as well as about programs that integrate them. In what follows, we review related work for each paradigm and contrast it with our own approach.

1. Concurrency

   Reasoning about concurrent systems is crucial, since even simple concurrent programs can quickly become difficult to understand. To support this task, several tools for *modeling* and *simulation* of concurrent systems have been developed. Prominent examples include *mCRL2* [47] and *CADP* [39]. Additional tools can be found in [42, Section 6.4] and on the website [105]. Both mCRL2 and CADP are based on process algebras: mCRL2 builds upon ACP [17], whereas CADP supports the verification of LOTOS [19] and LNT [24] specifications. These tools allow not only the simulation of concurrent systems but also the verification of behavioral properties such as deadlock and livelock freedom. In contrast, our interpreter focuses exclusively on the *execution* of concurrent programs, without providing automated verification capabilities.

2. Probabilities

   Hackage, the central repository for Haskell packages, hosts numerous useful libraries, such as `mtl` and `parsec`, which are used in our implementation. Unsurprisingly, Hackage also includes several libraries devoted to probabilistic programming. In particular, we used the `game-probability` package [22] when collecting samples to build the histogram. Other probabilistic libraries in Hackage, including `monad-bayes` [94], `probability` [35], and `mwc-probability` [107], provide

built-in sampling mechanisms and support the creation of a wide variety of probabilistic distributions. Moreover, `monad-bayes` and `probability` offer plotting facilities to visualize the resulting distributions. However, to implement the small-step operational semantics, none of these packages satisfies our requirement: a list-based probability structure implemented as an instance of the MonadTrans class. For this reason, we develop `DistT` described in Section 10.2. Consequently, features such as conditioning, sampling, and plotting are intentionally omitted from the file that defines `DistT`. It is also worth noting that `mwc-probability` defines a probabilistic monad transformer based on random sampling. While this approach is well suited for sampling concrete outcomes of probabilistic computations, it is not appropriate for modeling a small-step operational semantics based on distributions, since the small-step semantics returns a distribution and not a sample.

3. Concurrency and Probabilities

   Among the tools discussed in the concurrency section, both mCRL2 and CADP also support the analysis of probabilistic processes. While in the former such feature is still an experimental extension, the latter requires an intermediate translation step. In contrast to these two tools, PRISM [64] was developed from the outset with the goal of combining concurrency and probability within a single framework. PRISM is a probabilistic model checker that verifies whether a given property holds with a specified probability. It also supports the simulation of probabilistic systems, similarly to the interpreter described in Part II. However, because PRISM is a model checker, its simulation engine can also be used to perform approximate model checking, a verification technique that relies on simulation and sampling to determine whether a property is satisfied. Moreover, PRISM supports the generation of optimal strategies, *i.e.* the synthesis of schedulers that maximize or minimize the probability of satisfying a given condition.

4. Quantum

   To simulate sequential quantum computations, many programming languages have been developed, such as Qiskit [57], Q# [102], Pennylane [16], Cirq [28], among others. However, they lack an explicit formal foundation, although attempts to formalize Q# are being made [99, 98].

   On the other hand, QWire [89] and Silq [18] are quantum programming languages grounded in the principles of formal methods. Both feature type systems that ensure compliance with the no-cloning theorem and an operational semantics that preserves types throughout computation. In addition to its operational semantics, QWire also provides a denotational semantics defined in terms

of density operators, with both semantics proven to be sound with respect to each other. QWire has been embedded in the Coq proof assistant [91], enabling the use of Coq's formal verification capabilities to reason about quantum circuits. In contrast, Silq focuses on automating the process of uncomputation, *i.e.* automatically discarding ancillary qubits once they are no longer required. Silq is supported by a type checker [1], a proof-of-concept simulator [29], and a development environment [30], which together provide an integrated toolchain for writing and testing quantum programs.

It is also worth mentioning the Quantum IO Monad (QIO) [6], which is implemented in Haskell (it is also available on Hackage, however some minor adjustments are needed to use it). QIO provides a monadic interface for quantum programming in Haskell, embedding quantum computation within a monadic structure that distinguishes between quantum and classical effects. Moreover, QIO allows users to *simulate* and *run* quantum programs. The former produces a probability distribution over quantum states, whereas the latter yields a single measured state for each measurement performed; in other words, when a measurement occurs, the observed outcome, rather than the full probability distribution, is returned.

5. Concurrency and quantum

Most research on quantum concurrency has been theoretical, focusing on the development of quantum process algebras such as CQP [43] and qCCS [121, 36]. Mingsheng Ying also proposed a concurrent quantum imperative language [122], motivated by the definition of a quantum Hoare logic. In his work, Ying developed both a disjoint-variable and a shared-variable concurrent language. For the latter, he introduced an atomic region command (denoted $\langle C \rangle$). This command is equivalent to the $await$ command when the Boolean condition is "true", *i.e.* $(\texttt{await}\ \ true\ \ \texttt{do}\ \ \{C\}) \equiv \langle C \rangle$.

Some examples of practical implementations combining concurrency and quantum computation are those of Hynek Mlnarık [76] and Richard Bornat [20]. Mlnarık developed and implemented an imperative quantum programming language called *LanQ*. LanQ features a C-like syntax and a configuration-based operational semantics, where each configuration consists of a global state and local states, one for each process in parallel. The global state contains the quantum state, represented as a density matrix, while each local state contains classical data. Communication between processes occurs through channels instead of shared variables. Furthermore, Mlnarık also developed an interpreter in Java [75] to simulate the execution of LanQ programs. Richard Bornat, in turn, developed a symbolic simulation tool based on CQP. Unlike numerical simulators,

Bornat's system performs symbolic simulation. For example, when simulating the quantum teleportation protocol, his tool can teleport a generic state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. However, whenever a measurement is performed, symbolic probabilities are converted into numbers. Following the same symbolic-simulation approach, Manisha Jain implemented, in Maude, the concurrent quantum language presented in Section 9.2, developed by the author of this thesis. An advantage of symbolic reasoning is that it mitigates the precision issues that typically arise when using floating-point (`Double`) representations. However, this comes at the cost of slower computations.

11.2 Future Work

Regarding the work developed in Part II, we also foresee some possible extensions for the tool. One of them is to optimize its performance, particularly concerning quantum calculations. As a first approach to this optimization, we could represent quantum states as vectors instead of density matrices. Using density matrices requires matrix-matrix multiplication, which is computationally demanding, whereas using vector states would reduce the computational cost, since it involves matrix-vector multiplication. However, this approach would no longer be fully faithful to the operational semantics in [80].

We also foresee another route of development by incorporating logic into the tool. The work in [80] presents several semi-decidability results concerning the satisfiability of properties expressed in a given logic. This could allow us to answer questions such as: "Given a program C with initial state v, does there exist a scheduler $\mathcal{S}$ such that the final state v' is obtained with at least probability p?" Such an extension would also make it possible to construct optimal schedulers, as in PRISM, to maximize or minimize specific properties.

11.3 Conclusion

In the work presented in Part II, we incorporated states into the core language. We start by looking at a denotational semantics based on transition traces [88, 21], which we then extended to the probabilistic setting through the $\mathsf{P_{TM}V}$ [109, 106, 73]. Within this framework, we observed that non-deterministic and probabilistic commands could be treated together. Thus we extended the core language with those commands, which led to the language described in [80] (apart from the control flow commands). As an explanatory way of extending transition traces to the probabilistic setting, resorting to the $\mathsf{P_{TM}V}$ monad, we found a caveat in our approach: we were not able to capture the quantum instantiation of the language. This limitation arises because $\mathsf{P_{TM}V}$ deals with finitely generated sets, whereas a single qubit can exist in

infinitely many possible states (recall that the superposition of a qubit resembles a convex combination).

Building upon the language in [80], which also includes a quantum instantiation, we extended it with a synchronization primitive. In this context, two lines of work emerged: one considering only quantum states, and another combining classical and quantum states. In the former case, in collaboration with Manisha Jain [54, 55], the language was furnished with a quantum dynamic logic, opening the possibility of verifying the behavior of quantum concurrent programs through logical reasoning. Furthermore, the operational semantics was implemented in Maude [55]. In the case where both classical and quantum variables are supported, we focused on improving the tool originally developed by Inês Dias in her master's thesis [31], which we co-supervised. We improved the parser (removing the need for auxiliary parsing commands), updated the interpreted language, and implemented the semantics with a monadic focus. For this last task, we relied on monad transformers (see Definition 8.3.1) and developed a probabilistic monad transformer.

We subsequently tested the tool and observed that it becomes computationally infeasible when working with eight or more qubits. Afterward, we encoded several well-known examples and verified their behavior using the tool. Notably, we focused on the Dining Philosophers problem, implemented in three variants: concurrent-only, probabilistic, and quantum. In this example, the synchronization primitive proved particularly valuable, ensuring that each chopstick was picked up by only one philosopher at a time. While the concurrent-only and probabilistic versions had already been implemented, the quantum version of the Dining Philosophers presented here appears, to the best of our knowledge, to be the first implementation of such an algorithm. Furthermore, in the quantum version, we showed that an efficient solution to the quantum Dining Philosophers problem is yet to be found ([5, Theorem 3] does not hold). This is because the magic unitary gate, when applied with an odd number of participants, does not completely break the symmetry of the system.

Bibliography

[1] Silq: A high-level quantum programming language. `https://silq.ethz.ch/`, 2025. Accessed: 2025-10-17, developed at ETH Zürich :contentReference[oaicite:0]index=0.

[2] Martin Abadi and Gordon Plotkin. A model of cooperative threads. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–40, 2009.

[3] Samson Abramsky and Achim Jung. Domain theory. 1994.

[4] Samson Abramsky and Nikos Tzevelekos. Introduction to categories and categorical logic. In *New structures for physics*, pages 3–94. Springer, 2010.

[5] Dorit Aharonov, Maor Ganz, and Loick Magnin. Dining philosophers, leader election and ring size problems, in the quantum setting, 2017.

[6] Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 249–258. IEEE Computer Society, 2005.

[7] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[8] S. Awodey. *Category Theory*. Oxford Logic Guides. Ebsco Publishing, 2006.

[9] Christel Baier and Marta Kwiatkowska. Domain equations for probabilistic processes (extended abstract). *Electronic Notes in Theoretical Computer Science*, 7:34–54, 1997. EXPRESS'97.

[10] Alexandru Baltag and Sonja Smets. Lqp: the dynamic logic of quantum information. *Mathematical structures in computer science*, 16(3):491–525, 2006.

[11] Alexandru Baltag and Sonja Smets. Reasoning about quantum information: An overview of quantum dynamic logic. *Applied Sciences*, 12(9):4458, 2022.

[12] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982.

[13] Charles H. Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science*, 560:7–11, 2014. Theoretical Aspects of Quantum Cryptography – celebrating 30 years of BB84.

[14] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Phys. Rev. Lett.*, 70:1895–1899, Mar 1993.

[15] Charles H Bennett and Stephen J Wiesner. Communication via one-and two-particle operators on einstein-podolsky-rosen states. *Physical review letters*, 69(20):2881, 1992.

[16] Ville Bergholm et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations, 2022.

[17] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109 – 137, 1984.

[18] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 286–300, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Tomasso Bolognesi and Hendrik Brinksma. Introduction to the iso specification language lotos. *Computer networks and ISDN systems*, 14(1):25–59, 1987.

[20] Richard Bornat, Jaap Boender, Florian Kammueller, Guillaume Poly, and Rajagopal Nagarajan. Describing and simulating concurrent quantum systems. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 271–277, Cham, 2020. Springer International Publishing.

[21] Stephen Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996.

[22] Neil Brown. game-probability: Simple probability library for dice rolls, card games and similar. Hackage package, 2010. Version 1.1, BSD-3-Clause license, accessed from Hackage.

[23] Simon Castellan. Weak memory models using event structures. In *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, 2016.

[24] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe, and Gideon Smeding. Reference manual of the Int to lotos translator (version 6.7). *Inria/Vasy and Inria/Convecs*, 131, 2017.

[25] Pierre Clairambault, Marc de Visme, and Glynn Winskel. Game semantics for quantum programming. *Proc. ACM Program. Lang.*, 3(POPL):32:1–32:29, 2019.

[26] Marc de Visme. Event structures for mixed choice. In *The 30th International Conference on Concurrency Theory, CONCUR 2019*, 2019.

[27] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

[28] Cirq Developers. *Cirq*. Zenodo, April 2025.

[29] ETH-SRI / Silq development team. silq (github repository). `https://github.com/eth-sri/silq`, 2025. Accessed: 2025-10-17, BSL-1.0 license :contentReference[oaicite:1]index=1.

[30] ETH-SRI / Silq development team. vscode-silq (silq support extension for visual studio code). `https://marketplace.visualstudio.com/items?itemName=eth-sri.vscode-silq`, 2025. Accessed: 2025-10-17, 3,795 installs, syntax highlighting and type checking support.

[31] Maria Inês Machado Correia Brioso Dias. An interpreter for a concurrent quantum language. Master's thesis, University of Minho, 2024.

[32] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[33] Paul Adrien Maurice Dirac. *The principles of quantum mechanics*. Number 27. Oxford university press, 1981.

[34] Yotam Dvir, Ohad Kammar, and Ori Lahav. An algebraic theory for shared-state concurrency. In *Programming Languages and Systems: 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*, pages 3–24. Springer, 2022.

[35] Martin Erwig and Steve Kollmansberger. Probability:probabilistic functional programming. `https://hackage.haskell.org/package/probability`, 2015. BSD-3 license; accessed July 13, 2025; version 0.2.

[36] Yuan Feng, Runyao Duan, and Mingsheng Ying. Bisimulation for quantum processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(4):17, 2012.

[37] Vitor Fernandes and Inês Dias. qProbConc(GitHub Repository), 2025. [Online].

[38] Vítor Fernandes, Marc de Visme, and Benoît Valiron. Non-deterministic, probabilistic, and quantum effects through the lens of event structures. In *Asian Symposium on Programming Languages and Systems*, pages 196–215. Springer, 2024.

[39] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp a protocol validation and verification toolbox. In *International Conference on Computer Aided Verification*, pages 437–440. Springer, 1996.

[40] Richard Feynman. Simulating physics with computers, 1982.

[41] Robert W. Floyd. Assigning meanings to programs. In *J.T. Schwartz: Proceedings of Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[42] Wan Fokkink. *Introduction to Process Algebra*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[43] Simon J Gay and Rajagopal Nagarajan. Types and typechecking for communicating quantum processes. *Mathematical Structures in Computer Science*, 16(3):375–406, 2006.

[44] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003.

[45] Andy Gill. mtl: The monad transformer library. `https://github.com/haskell/mtl`, 2025. Accessed: 2025-08-06.

[46] Richard D. Gill. The monty hall problem is not a probability puzzle* (it's a challenge in mathematical modelling). *Statistica Neerlandica*, 65(1):58–71, January 2011.

[47] J. F. Groote, A. G. Wijs, T. A. C. Willemse, S. Blom, J. J. A. Keiren, B. Lisser, and T. van Dijk. The mcrl2 toolset for analysing concurrent systems. In *Computer Aided Verification (CAV)*, volume 13372 of *Lecture Notes in Computer Science*, pages 502–514. Springer, 2022.

[48] Lov K Grover. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*, 1996.

[49] Matthew Hennessy and Gordon D. Plotkin. Full abstraction for a simple parallel programming language. In Jirí Becvár, editor, *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 1979.

[50] J.R. Hindley and J.P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, 2008.

[51] Brian Huffman. Formal verification of monad transformers, 2012.

[52] Bart Jacobs. Coalgebraic trace semantics for combined possibilitistic and probabilistic systems. *Electronic Notes in Theoretical Computer Science*, 203(5):131–152, 2008.

[53] Manisha Jain and Vitor Fernandes. Maude Implementation for Concurrent Quantum Programming Language. Software, swhId: `swh:1:dir:21be0d264bc1d4ab0a9c3a40e5237a4bb2f1b9ce` (visited on 2025-09-17).

[54] Manisha Jain, Vitor Fernandes, and Alexandre Madeira. Adding concurrency to quantum dynamic logic. In Luis Soares Barbosa Songmao Zhang, editor, *Artificial Intelligence Logic and Applications*. Springer, 2025.

[55] Manisha Jain, Vitor Fernandes, Alexandre Madeira, and Luís S. Barbosa. Exploring a Quantum Programming Language with Concurrency. In Jonathan Edwards, Roly Perera, and Tomas Petricek, editors, *Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025)*, volume 134 of *Open Access Series in Informatics (OASIcs)*, pages 16:1–16:9, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[56] Manisha Jain, Vitor Fernandes, Alexandre Madeira, and Luis S. Barbosa. A quantum programming language with concurrency and its dynamic logic. *Computational and Applied Mathematics*, 2026 (in print).

[57] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, 2024.

[58] Julien Saan Joachim, Marc de Visme, and Stefan Haar. Quantum petri nets with event structures semantics, 2025.

[59] C. Jones and Gordon D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 186–195. IEEE Computer Society, 1989.

[60] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, page 97–136, Berlin, Heidelberg, 1995. Springer-Verlag.

[61] Joost-Pieter Katoen. *Quantitative and Qualitative Extensions of Event Structures*. Phd thesis - research ut, graduation ut, University of Twente, Netherlands, 1996.

[62] Leonard Kleinrock. Analysis of a time-shared processor. *Naval research logistics quarterly*, 11(1):59–73, 1964.

[63] Emmanuel Knill. Conventions for quantum pseudocode. Technical report, Los Alamos National Lab., NM (United States), 1996.

[64] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[65] Daniel Lehmann and Michael O Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–138, 1981.

[66] Daan Leijen, Paolo Martini, and Antoine Latter. parsec: Monadic parser combinators, 2022. [Online].

[67] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 333–343, New York, NY, USA, 1995. Association for Computing Machinery.

[68] Miran Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, 2011.

[69] Natalia López and Manuel Núñez. An overview of probabilistic process algebras and their equivalences. In Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, *Validation of Stochastic Systems - A Guide to Current Research*, volume 2925 of *Lecture Notes in Computer Science*, pages 89–123. Springer, 2004.

[70] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1998.

[71] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[72] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[73] Michael Mislove. Nondeterminism and probabilistic choice: Obeying the laws. In *International Conference on Concurrency Theory*, pages 350–365. Springer, 2000.

[74] Michael W. Mislove, Joël Ouaknine, and James Worrell. Axioms for probability and nondeterminism. In Flavio Corradini and Uwe Nestmann, editors, *Proceedings of the 10th International Workshop on Expressiveness in Concurrency, EXPRESS 2003, Marseille, France, September 2, 2003*, volume 96 of *Electronic Notes in Theoretical Computer Science*, pages 7–28. Elsevier, 2003.

[75] Hynek Mlnarık. Lanq: A quantum imperative programming language. `https://lanq.sourceforge.net/`, 2007. Latest version 0.6.3 (2007-10-04), released under GNU GPL, accessed 2025-10-17.

[76] Hynek Mlnarık. Quantum programming language lanq. *Masaryk University-Faculty of Informatics, Czech Republic*, 2007.

[77] Eugenio Moggi. *Computational lambda-calculus and monads*. University of Edinburgh, Department of Computer Science, Laboratory for ..., 1988.

[78] Eugenio Moggi. *An abstract view of programming languages*. University of Edinburgh. Department of Computer Science. Laboratory for ..., 1990.

[79] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[80] Renato Neves. An adequacy theorem between mixed powerdomains and probabilistic concurrency. In *Combined 32nd International Workshop on Expressiveness in Concurrency and 22nd Workshop on Structural Operational Semantics, Aarhus, Denmark*, EPTCS, page in print, 2025.

[81] Jeff Newbern. All about monads: A comprehensive guide to monadic programming in haskell, version 1.1.0. Web archive: `https://web.archive.org/web/20061211101052/https://www.nomaware.com/monads/html/index.html`, 2003. Accessed: 2025-09-25.

[82] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge, 2000.

[83] Mogens Nielsen and Glynn Winskel. Models for concurrency. In *MFCS*, pages 43–46, 1991.

[84] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, 2008.

[85] Bryan O'Sullivan, maintained by Ryan Scott, and contributors. Criterion: a powerful but simple library for measuring the performance of haskell code. `https://github.com/haskell/criterion`, 2025. Accessed: 2025-10-07, BSD-2-Clause license.

[86] Charlotte Out. Quantum leader election in distributed networks. Bachelor thesis, Vrije Universiteit Amsterdam, June 2019. Supervisor: Prof. Dr. Wan Fokkink, Submitted: 3rd June 2019.

[87] Hugo Paquet. *Probabilistic concurrent game semantics*. PhD thesis, University of Cambridge, UK, 2020.

[88] David Michael Ritchie Park. On the semantics of fair parallelism. In Dines Bjørner, editor, *Abstract Software Specifications, 1979 Copenhagen Winter School, January 22 - February 2, 1979, Proceedings*, volume 86 of *Lecture Notes in Computer Science*, pages 504–526. Springer, 1979.

[89] Jennifer Paykin, Robert Rand, and Steve Zdancewic. Qwire: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 846–858, New York, NY, USA, 2017. Association for Computing Machinery.

[90] Gordon D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[91] Robert Rand, Jennifer Paykin, and Steve Zdancewic. Qwire practice: Formal verification of quantum circuits in coq. *Electronic Proceedings in Theoretical Computer Science*, 266:119–132, February 2018.

[92] John C Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.

[93] Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: what binds them together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 98–113, New York, NY, USA, 2019. Association for Computing Machinery.

[94] Adam M. Şcibior, Dominic Steinitz, Reuben Cohn-Gordon, et al. Monad-bayes: A library for probabilistic programming. `https://monad-bayes.netlify.app/`, 2025. MIT license; accessed July 13, 2025; latest Hackage release ~8 months ago.

[95] Roberto Segala. *Modeling and verification of randomized distributed real-time systems*. PhD thesis, Massachusetts Institute of Technology, 1995.

[96] Peter Selinger, Benoıt Valiron, et al. Quantum lambda calculus. *Semantic techniques in quantum computation*, pages 135–172, 2009.

[97] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[98] Kartik Singhal, Kesha Hietala, Sarah Marshall, and Robert Rand. Q# as a Quantum Algorithmic Language. In *Proceedings of the 19th International Conference on Quantum Physics and Logic (QPL), Oxford, U.K., June 27–July 1, 2022*, June 2022.

[99] Kartik Singhal, Sarah Marshall, Kesha Hietala, and Robert Rand. Toward a type-theoretic interpretation of q# and statically enforcing the no-cloning theorem. In *Informal Proceedings of the Second International Workshop on Programming Languages for Quantum Computing (PLanQC '21)*, June 2021.

[100] Ana Sokolova and Erik P De Vink. Probabilistic automata: system types, parallel composition and comparison. *Validation of Stochastic Systems: A Guide to Current Research*, pages 1–43, 2004.

[101] Mariëlle Stoelinga. An introduction to probabilistic automata. *Bulletin of the EATCS*, 78(176-198):2, 2002.

[102] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q# enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–10, 2018.

[103] Tsubasa Takagi, Canh Minh Do, and Kazuhiro Ogata. Automated quantum program verification in dynamic quantum logic. In *International Workshop on Dynamic Logic*, pages 68–84. Springer, 2023.

[104] Seiichiro Tani, Hirotada Kobayashi, and Keiji Matsumoto. Exact quantum algorithms for the leader election problem, 2007.

[105] Landon Taylor. fmtools.fyi. `https://fmtools.fyi/`, 2025. Accessed: 2025-10-15.

[106] Regina Tix, Klaus Keimel, and Gordon Plotkin. Semantic domains for combining probability and non-determinism. *Electronic Notes in Theoretical Computer Science*, 222:3–99, 2009.

[107] Jared Tobin and Marco Zocca. mwc-probability: Sampling function-based probability distributions. Hackage package, 2020. Version 2.3.1, MIT License, accessed via Hackage (maintainers: Jared Tobin, Marco Zocca).

[108] Robert L Treece. *Resumption models: a denotational semantics for concurrency and synchronization*. PhD thesis, University of Kansas, 1985.

[109] Daniele Varacca. *Probability, nondeterminism and concurrency: two denotational models for probabilistic computation*. BRICS, 2003.

[110] Daniele Varacca, Hagen Völzer, and Glynn Winskel. Probabilistic event structures and domains. *Theor. Comput. Sci.*, 358(2-3):173–199, 2006.

[111] Daniele Varacca and Glynn Winskel. Distributing probability over non-determinism. *Mathematical structures in computer science*, 16(1):87–113, 2006.

[112] John von Neumann and RT Beyer. Mathematical foundations of quantum mechanics. 1955.

[113] Marilyn vos Savant. Ask marilyn. *Parade Magazine*, page 16, September 1990.

[114] John Watrous. *The theory of quantum information*. Cambridge University Press, 2018.

[115] Glynn Winskel. Event structure semantics for ccs and related languages. In *International Colloquium on Automata, Languages, and Programming*, pages 561–576. Springer, 1982.

[116] Glynn Winskel. Synchronization trees. *Theoretical Computer Science*, 34(1-2):33–82, 1984.

[117] Glynn Winskel. Event structures. In *advanced course on Petri nets*, pages 325–392. Springer, 1986.

[118] Glynn Winskel. An introduction to event structures. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 364–397. Springer, 1988.

[119] Glynn Winskel. Probabilistic and quantum event structures. In *Horizons of the Mind. A Tribute to Prakash Panangaden*, pages 476–497. Springer, 2014.

[120] Mingsheng Ying. *Foundations of Quantum Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016.

[121] Mingsheng Ying, Yuan Feng, Runyao Duan, and Zhengfeng Ji. An algebra of quantum processes. *ACM Transactions on Computational Logic (TOCL)*, 10(3):19, 2009.

[122] Mingsheng Ying, Li Zhou, and Yangjia Li. Reasoning about parallel quantum programs. *arXiv preprint arXiv:1810.11334*, 2018.

[123] Noam Zilberstein, Daniele Gorla, and Alexandra Silva. Denotational Semantics for Probabilistic and Concurrent Programs. In Patricia Bouyer and Jaco van de Pol, editors, *36th International Conference on Concurrency Theory (CONCUR 2025)*, volume 348 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:24, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum f”ur Informatik.

Appendix

Chapter A
Proofs of Part I

A.1 Proofs of Chapter 4

Proof of Lemma 4.2.2

Proof. Let

$$\mathsf{E}_1 = (E_1, \leq_1, \#_1)$$
$$\mathsf{E}_2 = (E_2, \leq_2, \#_2)$$
$$\mathsf{E}_1 ; \mathsf{E}_2 = (E, \leq, \#)$$

We need to show that $\leq$ is a partial order and that # is symmetric and irreflexive.

- $\leq$ is a partial order:
  - Reflexivity ($e \leq e$): we have two cases
    1. Case $e \in E_1$. This entails $e \leq_1 e$. Since $\leq_1$ is a partial order we are done.
    2. Case $(e, x) \in E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{E}_1)$. This entails $e \leq_2 e$. Since $\leq_2$ is a partial order we are done.
  - Transitivity ($e \leq e'$ and $e' \leq e''$ then $e \leq e''$): if $e, e', e'' \in E_1$ or $e, e', e'' \in E_2$, we are done since $\leq_1$ and $\leq_2$ are partial orders. We then have two more cases:
    1. Case $e \leq (e', x)$ and $(e', x) \leq (e'', x)$. From $e \leq (e', x)$ we have that $e \in x$. From $(e', x) \leq (e'', x)$ we know that $e' \leq_2 e''$ and since x is the same for both events, ten $e \in x$. Thus $e \leq (e'', x)$.
    2. Case $e \leq e'$ and $e' \leq (e'', x)$. From $e \leq e'$ we know that $e' \in x$. From $e \leq e'$ we know that $e \leq_1 e'$. Since x is a maximal configuration in E_1 and $e \leq_1 e'$ then $e \in x$. Thus $e \leq (e'', x)$
  - Antisymmetry ($e \leq e'$ and $e' \leq e$ then $e' = e$): if $e, e', e'' \in E_1$ or $e, e', e'' \in E_2$, we are done since $\leq_1$ and $\leq_2$ are partial orders. Since it is not possible to have $(e', x) \leq e$ we are done.

  Hence $\leq$ is a partial order.

- # is symmetric and irreflexive:
  - Symmetric (if $e \# e'$ then $e' \# e$): we have two cases:
    1. $(e_2, x) \# (e_2', x)$ then $(e_2', x) \# (e_2, x)$. From $(e_2, x) \# (e_2', x)$ we have that $e_2 \#_2 e_2'$, which is symmetric since $\#_2$ is a partial order. Hence $e_2' \#_e e_2$ and $(e_2', x) \# (e_2, x)$.

2. $e\#e'$ then $e'\#e$. $e\#e'$ entails the existence of $e_1 \leq e$ and $e'_1 \leq e'$ such that $e_1\#_1 e'_1$. Since $\#_1$ is a partial order, it is symmetric. then $e'_1\#_1 e_1$. Furthermore, since $e_1 \leq e$ and $e'_1 \leq e'$ then $e'\#e$.

– Irreflexive ($\neg(e\#e)$): we show this by showing that the elements of # are of the form $e\#e'$ with $e \neq e'$. We then have four cases:

1. Case $e, e' \in E_1$. Then we have $e_1 \leq e$, $e'_1 \leq e'$, and $e_1\#_1 e'_1$. Furthermore, we have $e_1 \leq_1 e$ and $e'_1 \leq_1 e'$. Since $e_1\#_1 e'_1$ and $e_1 \leq_1 e$ we have $e'_1\#_1 e$. Since $e'_1\#_1 e$ and $e'_1 \leq_1 e'$ we have $e\#_1 e'$. By irreflexivity of $\#_1$ we have that $e \neq e'$.
2. Case $(e, x), (e', x') \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$. If $x \neq x'$ then it is clear that $(e, x) \neq (e', x')$. If $x = x'$ then we have $e_1 \leq (e, x)$ and $e'_1 \leq (e', x)$, which entails that $e_1, e'_1 \in x$. However, $e_1 \leq_1 e'_1$, hence $e_1, e'_1 \notin x$. Thus we have a contradiction, and consequently $(e, x) \neq (e', x)$.
3. Case $e \in E_1$ and $e' \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$. They are clearly different.
4. Case $e \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$ and $e' \in E_1$. They are clearly different.

Hence # is symmetric and irreflexive.

We need to show that $\forall e, e', e'' \in E$:

1. $\{e' \mid e' \leq e\}$ is finite

   (a) Case $e \in E_1$ we are done.

   (b) Case $e \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$. Then $e = (e_2, x_1)$ with $e_2 \in E_2$ and $x_1 \in \mathcal{C}_{\max}(\mathsf{E}_1)$. We know that $\{e' \mid e' \leq (e_2, x_1)\} = \{(e'_2, x_1) \mid (e'_2, x_1) \leq (e_2, x_1)\} \cup \{e_1 \mid e_1 \leq (e_2, x_1),\ e_1 \in x_1\}$. Both sets are finite because E_2 is an event structure and x_1 is finite, respectively. Since both sets are finite and the union of finite sets is finite, then $\{e' \mid e' \leq (e_2, x_1)\}$ is finite.

2. $e\#e' \leq e'' \Rightarrow e\#e''$

   (a) Case $e, e', e'' \in E_1$ or $a, a', e, e', e'' \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$ we are done.

   (b) Case $e, e' \in E_1$ and $e'' \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$. We want to show that $e\#e''$. Hence we have to show that $\exists(e_1 \leq e,\ e'_1 \leq e')\ .\ e_1\#_1 e'_1$. Since $e, e' \in E_1$ then $e_1, e'_1 \in E_1$. Let $e_1 = e$ and $e'_1 = e'$. Hence we have $e_1 \leq e \Leftrightarrow e \leq e \Leftrightarrow e \leq_1 e$. By the initial assumption, $e' \leq e'' \Leftrightarrow e'_1 \leq e''$. It lacks to show that $e_1\#_1 e'_1$. That follows directly from the initial assumption $e\#e' \Leftrightarrow e\#_1 e' \Leftrightarrow e_1\#_1 e'_1$.

□

Proof of Lemma 4.2.4

Proof. Let

$$\mathsf{E}_1 = (E_1, \leq_1, \#_1)$$
$$\mathsf{E}_2 = (E_2, \leq_2, \#_2)$$
$$\mathsf{E}_1 \,||\, \mathsf{E}_2 = (E, \leq, \#)$$

We need to show that $\leq$ is a partial order and that # is symmetric and irreflexive.

- $\leq$ is a partial order:

- Reflexivity ($e \leq e$): we have two cases: $e_1 \leq_1 e_1'$ or $e_2 \leq_2 e_2'$. Since $\leq_1$ and $\leq_2$ are partial orders, we are done.
- Transitivity ($e \leq e'$ and $e' \leq e''$ then $e \leq e''$): we have two cases: ($e \leq_1 e'$ and $e' \leq_1 e''$) or ($e \leq_2 e'$ and $e' \leq_2 e''$). Since $\leq_1$ and $\leq_2$ are partial orders, we have $e \leq_1 e''$ or $e \leq_2 e''$. Hence $e \leq e''$.
- Antisymmetry ($e \leq e'$ and $e' \leq e$ then $e' = e$): we have two cases: ($e \leq_1 e'$ and $e' \leq_1 e$) or ($e \leq_2 e'$ and $e' \leq_2 e$). Since $\leq_1$ and $\leq_2$ are partial orders, we have $e = e'$.

Hence $\leq$ is a partial order.

- # is symmetric and irreflexive:
  - Symmetric (if $e\#e'$ then $e'\#e$): we have either $e\#_1 e'$ or $e\#_2 e'$. Since $\#_1$ and $\#_2$ are symmetric then we have $e'\#_1 e$ or $e'\#_2 e$. Thus $e'\#e$.
  - Irreflexive ($\neg(e\#e)$): We have either $\neg(e\#_1 e)$ or $\neg(e\#_2 e)$. Since $\#_1$ and $\#_2$ are irreflexive then $\neg(e\#e)$.

  Hence # is symmetric and irreflexive.

We need to show that $\forall e, e', e'' \in E$:

1. $\{e' \mid e' \leq e\}$ is finite
   (a) Case $e \in E_1$ then $\{e' \mid e' \leq e\} = e' \mid e' \leq_1 e$, which is finite since E_1 is an event structure.
   (b) Case $e \in E_2$ then $\{e' \mid e' \leq e\} = e' \mid e' \leq_2 e$, which is finite since E_2 is an event structure.
2. $e\#e' \leq e'' \Rightarrow e\#e''$

   We have two cases: $e, e', e'' \in E_1$ or $e, e', e'' \in E_2$. In both cases this holds because E_1 and E_2 are event structures.

□

Proof of Lemma 4.2.7

Proof. Let

$$\mathsf{E}_1 = (E_1, \leq_1, \#_1)$$
$$\mathsf{E}_2 = (E_2, \leq_2, \#_2)$$
$$\mathsf{E}_1 \text{ or } \mathsf{E}_2 = (E, \leq, \#)$$

We need to show that $\leq$ is a partial order and that # is symmetric and irreflexive.

- $\leq$ is a partial order:
  - Reflexivity ($e \leq e$): we have two cases: $e_1 \leq_1 e_1'$ or $e_2 \leq_2 e_2'$. Since $\leq_1$ and $\leq_2$ are partial orders, we are done.
  - Transitivity ($e \leq e'$ and $e' \leq e''$ then $e \leq e''$): we have two cases: ($e \leq_1 e'$ and $e' \leq_1 e''$) or ($e \leq_2 e'$ and $e' \leq_2 e''$). Since $\leq_1$ and $\leq_2$ are partial orders, we have $e \leq_1 e''$ or $e \leq_2 e''$. Hence $e \leq e''$.

– Antisymmetry ($e \leq e'$ and $e' \leq e$ then $e' = e$): we have two cases: ($e \leq_1 e'$ and $e' \leq_1 e$) or ($e \leq_2 e'$ and $e' \leq_2 e$). Since $\leq_1$ and $\leq_2$ are partial orders, we have $e = e'$.

Hence $\leq$ is a partial order.

- # is symmetric and irreflexive:
  - Symmetric (if $e\#e'$ then $e'\#e$): we have the following cases
    1. if $e\#_1e'$ or $e\#_2e'$. Since $\#_1$ and $\#_2$ are symmetric then we have $e'\#_1e$ or $e'\#_2e$. Thus $e'\#e$.
    2. if $e \in E_1$ and $e' \in E_2$, or vice-versa, then by Definition 4.2.6 we also have $e' \leq e$.
  - Irreflexive ($\neg(e\#e)$): We have either $\neg(e\#_1e)$ or $\neg(e\#_2e)$. Since $\#_1$ and $\#_2$ are irreflexive then $\neg(e\#e)$. There are no more cases since either $e \in E_1$ or $e \in E_2$.

  Hence # is symmetric and irreflexive.

We need to show that $\forall e, e', e'' \in E$:

1. $\{e' \mid e' \leq e\}$ is finite
   (a) Case $e \in E_1$ then $\{e' \mid e' \leq e\} = e' \mid e' \leq_1 e$, which is finite since E_1 is an event structure.
   (b) Case $e \in E_2$ then $\{e' \mid e' \leq e\} = e' \mid e' \leq_2 e$, which is finite since E_2 is an event structure.
2. $e\#e' \leq e'' \Rightarrow e\#e''$

   We have two cases: $e, e', e'' \in E_1$ or $e, e', e'' \in E_2$. In both cases this holds because E_1 and E_2 are event structures.

□

In this section, we interpret ✓ as the empty event structure, *i.e.* $[\![\checkmark]\!] = (\emptyset, \emptyset, \emptyset) = \emptyset$.

Proof of Lemma 4.2.13

Proof. It follows directly from Definition 4.2.3. □

Proof of Lemma 4.2.14

Proof. Let

$$\begin{aligned}
\mathsf{E}_1 &= (E_1, \leq_1, \#_1)\\
\mathsf{E}'_1 &= (E'_1, \leq'_1, \#'_1)\\
\mathsf{E}_2 &= (E_2, \leq_2, \#_2)\\
\mathsf{E}'_2 &= (E'_2, \leq'_2, \#'_2)\\
\mathsf{E}_1 ; \mathsf{E}_2 &= (E, \leq, \#)\\
\mathsf{E}'_1 ; \mathsf{E}'_2 &= (E', \leq', \#')
\end{aligned}$$

such that $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$.

1. We start by defining $f : E \to E'$ such that

$$f(e) = \begin{cases} f_1(e) & \text{if } e \in E_1 \\ (f_2(e), m_1(x)) & \text{if } e \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1) \end{cases}$$

where $m_1(x)$ is defined as follows: $\forall x \in \mathcal{C}_{\max}(\mathsf{E}_1)$, $m_1(x) \in \mathcal{C}_{\max}(\mathsf{E}'_1)$ such that $f_1[x] \subseteq m_1(x)$. If multiple are possible, $m_1(x)$ is arbitrarily chosen among them.

Now we show that f is injective, *i.e.* if $a \neq b$ then $f(a) \neq f(b)$. We have two cases:

- if $a, b \in E_1$ we are done
- if $a, b \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$
  Let $a = (e, x)$ and $b = (e, y)$. For $(e, x) \neq (e', y)$ we have two cases:
  - $e \neq e'$
    We then have

$$\begin{aligned} f(e,x) \neq f(e',y) &\Leftrightarrow (f_2(e), m_1(x)) \neq (f_2(e'), m_1(y)) \\ &\Leftrightarrow f_2(e) \neq f_2(e') \text{ or } m_1(x) \neq m_1(y) \end{aligned}$$

    Since f_2 is injective we are done.
  - $e = e'$ and $x \neq y$
    We then have

$$\begin{aligned} f(e,x) \neq f(e',y) &\Leftrightarrow (f_2(e), m_1(x)) \neq (f_2(e'), m_1(y)) \\ &\Leftrightarrow f_2(e) \neq f_2(e') \text{ or } m_1(x) \neq m_1(y) \end{aligned}$$

    Since we know that f_2 is an injective function, then we should have $f_2(e) = f_2(e')$. Otherwise we enter in a contradiction. Hence it lacks to see if $m_1(x) \neq m_1(y)$. From definition of $m_1(x)$ and $m_1(y)$ we have, respectively, $f_1[x] \subseteq m_1(x)$ and $f_2[y] \subseteq m_2(y)$. Thus $f_1[x], f_1[y] \in \mathcal{C}(\mathsf{E}'_1)$. Since f_1 is injective and $x \neq y$ then it exists $e \in x$ and $e' \in y$, with $e \neq e'$, such that $f_1(e) \neq f_1(e')$. Thus $f_1[x] \neq f_1[y]$, and consequently $m_1(x) \neq m_1(y)$.

Hence f is injective.

2. $\pi(f(a)) = \pi(a)$

We have two cases:

- $a \in E_1$
  Then $\pi(f(a)) = \pi(f_1(a)) = \pi(a)$ since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$.
- $(a, x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$
  Then $\pi(f(a, x)) = \pi(f_2(a), m_1(x)) = \pi(f_2(a)) = \pi(a)$ since $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$.

3. $a \leq b \Leftrightarrow f(a) \leq' f(b)$

$\Rightarrow$ Assume $a \leq b$.
By Definition 4.2.1 we have:

(a) $a, b \in E_1$.
Hence $a \leq_1 b$. Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$, then $f_1(a) \leq'_1 f_1(b)$. By Definition 4.2.1, $f_1(a) \leq' f_1(b)$.

(b) $(a,x),(b,x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$
Hence $(a,x) \leq (b,x)$ such that $a \leq_2 b$. Since $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$ then $f_2(a) \leq'_2 f_2(b)$. Now let $m_1(x) \in \mathcal{C}_{\max}(\mathsf{E}'_1)$ such that $f_1[x] \subseteq m_1(x)$. Then $(f_2(a), m_1(x)) \leq' (f_2(b), m_1(x))$. By Definition 4.2.1, $f(a,x) \leq' f(b,x)$.

(c) $a \in E_1$ and $(b,x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$ such that $a \in x$
Hence $a \leq (b,x)$. From $(b,x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$ we have $b \in E_2$ and $x \in \mathcal{C}_{\max}(\mathsf{E}_1)$. Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$, then $f_1(a) \in E'_1$ and $f_2(a) \in E'_2$. Now let $m_1(x) \in \mathcal{C}_{\max}(\mathsf{E}'_1)$ such that $f_1[x] \subseteq m_1(x)$. Since $a \in x$ then $f_1(a) \in f_1[x]$. Thus $f_1(a) \in m_1(x)$. Hence $(f_2(b), m_1(x)) = f(b,x) \in E'_2 \times \mathcal{C}_{\max}(\mathsf{E}'_1)$. By Definition 4.2.1, $f(a) \leq' f(b,x)$.

$\Leftarrow$ Assume $f(a) \leq' f(b)$

(a) $f(a) \leq'_1 f(b)$
We have $f(a) = f_1(a), f(b) = f_1(b) \in E'_1$. Hence $f_1(a) \leq'_1 f_1(b)$. Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ then $a \leq_1 b$. By Definition 4.2.1, $a \leq b$.

(b) $f(a,x) \leq' f(b,x)$
We have $f(a,x) = (f_2(a), m_1(x)), f(b,x) = (f_2(b), m_1(x)) \in E'_2 \times \mathcal{C}_{\max}(\mathsf{E}'_1)$. From $m_1(x)$ we have $f_1[x] \subseteq m_1(x)$ such that $x \in \mathcal{C}_{\max}(\mathsf{E}_1)$. From Definition 4.2.1 we have $f_2(a) \leq'_2 f_2(b)$. Since $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$ then $a \leq_2 b$. By Definition 4.2.1 $(a,x) \leq (b,x)$.

(c) $f(a) \leq' f(b,x)$ such that $f(a) \in m_1(x)$.
We have $f(a) = f_1(a) \in E'_1$ and $f(b,x) = (f_2(b), m_1(x)) \in E'_2 \times \mathcal{C}_{\max}(\mathsf{E}'_1)$, which gives $f_2(b) \in E'_2$ and $m_1(x) \in \mathcal{C}_{\max}(\mathsf{E}'_1)$. From $m_1(x)$ we have $f_1[x] \subseteq m_1(x)$ such that $x \in \mathcal{C}_{\max}(\mathsf{E}_1)$. Since $f_1(a) \in f_1[x]$ then $a \in x$. Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$ then $a \in E_1$ and $b \in E_2$. We then have $(b,x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$. By Definition 4.2.1 $a \leq (b,x)$.

4. $a \# b \Leftrightarrow f(a) \#' f(b)$

$\Rightarrow$ Assume $a \# b$.

From Definition 4.2.1 we have two cases:

- $\exists (e_1 \leq a,\ e'_1 \leq b)\ .\ e_1 \#_1 e'_1$
Here we have two subcases:
  - $a, b \in E_1$
Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ we have $f_1(a) \#'_1 f_1(b)$.
  - $a \in E_1$ and $(b,x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$
Then $e_1 \leq a, e'_1 \leq (b,x)$ such that $e_1 \#_1 e'_1$. By Definition 4.2.1, $e_1 \leq_1 a$ and $e'_1 \leq (b,x)$ such that $e'_1 \in x$. Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ we have $f_1(e_1) \leq_1 f_1(a) \Leftrightarrow f(e_1) \leq f(a)$ and $f_1(e_1) \#'_1 f_1(e'_1)$. From $(b,c) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$ we have $b \in E_2$ and $x \in \mathcal{C}_{\max}(\mathsf{E}_1)$. Now let $m_1(x) \in \mathcal{C}_{\max}(\mathsf{E}'_1)$ such that $f_1[x] \subseteq m_1(x)$. Since $e'_1 \in x$ then $f_1(e'_1) \in f_1[x]$. Since $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$ then $f_2(b) \in E'_2$. Thus $(f_2(b), m_1(x)) = f(b,x) \in E'_2 \times \mathcal{C}_{\max}(\mathsf{E}'_1)$. Hence $f(e'_1) \leq f(b,x)$. Thus $f(a) \#' f(b,x)$.
- $(a,x) \#' (b,x)$ such that $a \#_2 b$
From $(a,x),(b,x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$ we have $a, b \in E_2$ and $x \in \mathcal{C}_{\max}(\mathsf{E}_1)$. From $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$ we have $f_2(a), f_2(b) \in E'_2$ and $f_2(a) \#'_2 f_2(b)$. Now let $m_1(x) \in \mathcal{C}_{\max}(\mathsf{E}'_1)$ such that $f_1[x] \subseteq m_1(x)$. Then $(f_2(a), m_1(x)) = f(a,x), (f_2(b), m_1(x)) = f(b,x) \in E'_2 \times \mathcal{C}_{\max}(\mathsf{E}'_1)$. Thus $f(a,x) \#' f(b,x)$.

$\Leftarrow$ Assume $f(a)\#'f(b)$

From Definition 4.2.1 we have two cases:

- $f(a)\#'f(b)$ such that $\exists(f(e_1) \leq f(a), f(e_1') \leq f(b))\ .\ f_1(e_1)\#_1'f_1(e_1')$
  Here we have two subcases:
  - $f(a), f(b) \in E_1'$.
    We are done since $\mathsf{E}_1 \sqsubseteq \mathsf{E}_1'$.
  - $f(a) \in E_1'$ and $f(b,x) \in E_2' \times \mathcal{C}_{\max}(\mathsf{E}_1')$
    Then $f(e_1) \leq f(a) \Leftrightarrow f_1(e_1) \leq f_1(a)$ and $f(e_1') \leq f(b,x) \Leftrightarrow f_1(e_1') \leq (f_2(b), m_1(x))$. By Definition 4.2.1, $f_1(e_1) \leq_1 f_1(a)$ and $f_1(e_1') \leq (f_2(b), m_1(x))$ such that $f_1(e_1') \in m_1(x)$. Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}_1'$, $e_1 \leq_1 a$ and $e_1\#_1e_1'$. From $(f_2(b), m_1(x)) \in E_2' \times \mathcal{C}_{\max}(\mathsf{E}_1')$ we have $f_2(b) \in E_2'$ and $m_1(x) \in \mathcal{C}_{\max}(\mathsf{E}_1')$. Since $\mathsf{E}_2 \sqsubseteq \mathsf{E}_2'$ then $b \in E_2$. Since $f_1(e_1') \in m_1(x)$ and $f_1[x] \subseteq m_1(x)$ then $e_1' \in x$. Thus $(b,x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$. Hence $e_1' \leq (b,x)$. Thus $a\#(b,x)$.
- $f(a,x)\#'f(b,x)$ such that $f_2(a)\#_2'f_2(b)$
  From $f(a,x) = (f_2(a), m_1(x)), f(b,x) = (f_2(b), m_1(x)) \in E_2' \times \mathcal{C}_{\max}(\mathsf{E}_1')$ we have that $f_2(a), f_2(b) \in E_2$ and $m_1(x) \in \mathcal{C}_{\max}(\mathsf{E}_1')$. From $\mathsf{E}_2 \sqsubseteq \mathsf{E}_2'$ we have $a, b \in E_2$ and $a\#_2b$. From $m_1(x)$ we know that $x \in \mathcal{C}_{\max}(\mathsf{E}_1)$. Thus $(a,x)\#(b,x)$.

□

Proof of Lemma 4.2.15

Proof. Let

$$\begin{aligned}
\mathsf{E}_1 &= (E_1, \leq_1, \#_1)\\
\mathsf{E}_1' &= (E_1', \leq_1', \#_1')\\
\mathsf{E}_2 &= (E_2, \leq_2, \#_2)\\
\mathsf{E}_2' &= (E_2', \leq_2', \#_2')\\
\mathsf{E}_1 \,||\, \mathsf{E}_2 &= (E, \leq, \#)\\
\mathsf{E}_1' \,||\, \mathsf{E}_2' &= (E', \leq', \#')
\end{aligned}$$

such that $\mathsf{E}_1 \sqsubseteq \mathsf{E}_1'$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}_2'$.

1. We start by defining a function $f : E \to E'$ such that

$$f(e) = \begin{cases} f_1(e) & \text{if } e \in E_1 \\ f_2(e) & \text{if } e \in E_2 \end{cases}$$

   Clearly f is injective, since if $e \in E_1$ then $f(e) = f_1(e)$ and f_1 is injective and if $e \in E_2$ then $f(e) = f_2(e)$ and f_2 is injective.

2. $\pi(f(a)) = \pi(a)$

   We have two similar cases, $a \in E_1$ or $a \in E_2$. We only show the former.

   $\pi(f(a)) = \pi(f_1(a)) = \pi(a)$, since $\mathsf{E}_1 \sqsubseteq \mathsf{E}_1'$.

3. $a \leq b \Leftrightarrow f(a) \leq' f(b)$

$\Rightarrow$ Assume $a \leq b$.

By Definition 4.2.3, $a \leq_1 b$ or $a \leq_2 b$. Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$, then $f_1(a) \leq'_1 f_1(b)$ and $f_2(a) \leq'_2 f_2(b)$. By Definition 4.2.3 we are done.

$\Leftarrow$ Assume $f(a) \leq' f(b)$.

By Definition 4.2.3 $f(a) \leq'_1 f(b) \Leftrightarrow f_1(a) \leq'_1 f_1(b)$ or $f(a) \leq'_2 f(b) \Leftrightarrow f_2(a) \leq'_2 f_2(b)$. Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$, then $a \leq_1 b$ and $a \leq_2 b$. By Definition 4.2.3 we are done.

4. $a\#b \Leftrightarrow f(a)\#'f(b)$

   Similar to the previous case.

□

Proof of Lemma 4.2.16

Proof. Let

$$\begin{aligned}
\mathsf{E}_1 &= (E_1, \leq_1, \#_1)\\
\mathsf{E}'_1 &= (E'_1, \leq'_1, \#'_1)\\
\mathsf{E}_2 &= (E_2, \leq_2, \#_2)\\
\mathsf{E}'_2 &= (E'_2, \leq'_2, \#'_2)\\
\mathsf{E}_1 \text{ or } \mathsf{E}_2 &= (E, \leq, \#)\\
\mathsf{E}'_1 \text{ or } \mathsf{E}'_2 &= (E', \leq', \#')
\end{aligned}$$

such that $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$.

1. We start by defining a function $f : E \to E'$ such that

$$f(e) = \begin{cases} f_1(e) & \text{if } e \in E_1 \\ f_2(e) & \text{if } e \in E_2 \end{cases}$$

   Clearly f is injective, since if $e \in E_1$ then $f(e) = f_1(e)$ and f_1 is injective and if $e \in E_2$ then $f(e) = f_2(e)$ and f_2 is injective.

2. $\pi(f(a)) = \pi(a)$

   We have two similar cases, $a \in E_1$ or $a \in E_2$. We only show the former.

   $\pi(f(a)) = \pi(f_1(a)) = \pi(a)$, since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$.

3. $a \leq b \Leftrightarrow f(a) \leq' f(b)$

   $\Rightarrow$ Assume $a \leq b$.

   By Definition 4.2.6, $a \leq_1 b$ or $a \leq_2 b$. Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$, then $f_1(a) \leq'_1 f_1(b)$ and $f_2(a) \leq'_2 f_2(b)$. By Definition 4.2.6 we are done.

   $\Leftarrow$ Assume $f(a) \leq' f(b)$.

   By Definition 4.2.6 $f(a) \leq'_1 f(b) \Leftrightarrow f_1(a) \leq'_1 f_1(b)$ or $f(a) \leq'_2 f(b) \Leftrightarrow f_2(a) \leq'_2 f_2(b)$. Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$, then $a \leq_1 b$ and $a \leq_2 b$. By Definition 4.2.6 we are done.

4. $a\#b \Leftrightarrow f(a)\#'f(b)$

   $\Rightarrow$ Assume $a\#b$.

   We have the following cases:

   - If $a\#_1 b$ or $a\#_2 b$, then we are done since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$.
   - If $a \in E_1$ and $b \in E_2$ (or vice-versa).
     Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$, then $f_1(a) \in E'_1$ and $f_2(b) \in E'_2$. By Definition 4.2.6, $f_1(a)\#'f_2(b) \Leftrightarrow f(a)\#'f(b)$.

   $\Leftarrow$ Assume $f(a)\#'f(b)$.

   We have the following cases:

   - If $f(a)\#'_1 f(b) \Leftrightarrow f_1(a)\#'_1 f_1(b)$ or $f(a)\#'_2 f(b) \Leftrightarrow f_2(a)\#'_2 f_2(b)$, we are done since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$.
   - If $f(a) = f_1(a) \in E'_1$ and $f(b) = f_2(b) \in E'_2$ (or vice-versa).
     Since $\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1$ and $\mathsf{E}_2 \sqsubseteq \mathsf{E}'_2$, then $a \in E_1$ and $b \in E_2$. By Definition 4.2.6, $a\#b$.

□

Proof of Lemma 4.2.17

Proof.

$$
\begin{aligned}
&\mathsf{E}_1 \equiv \mathsf{E}'_1 \text{ and } \mathsf{E}_2 \equiv \mathsf{E}'_2 \\
\Rightarrow &\{\text{Definition 4.2.11}\} \\
&\mathsf{E}_1 \sqsubseteq \mathsf{E}'_1 \text{ and } \mathsf{E}'_1 \sqsubseteq \mathsf{E}_1, \mathsf{E}_2 \sqsubseteq \mathsf{E}'_2 \text{ and } \mathsf{E}'_2 \sqsubseteq \mathsf{E}_2 \\
\Rightarrow &\{op \text{ monotone}\} \\
&\mathsf{E}_1\, op\, \mathsf{E}_2 \sqsubseteq \mathsf{E}'_1\, op\, \mathsf{E}'_2 \text{ and } \mathsf{E}'_1\, op\, \mathsf{E}'_2 \sqsubseteq \mathsf{E}_1\, op\, \mathsf{E}_2 \\
\Rightarrow &\{\text{Definition 4.2.11}\} \\
&\mathsf{E}_1\, op\, \mathsf{E}_2 \equiv \mathsf{E}'_1\, op\, \mathsf{E}'_2
\end{aligned}
$$

□

Proof of Lemma 4.2.20

Proof. Let

$$
\begin{aligned}
\mathsf{E} &= (E, \leq, \#) \\
\mathsf{E}\backslash a &= (E', \leq', \#')
\end{aligned}
$$

We need to show that $\leq'$ is a partial order and $\#'$ is symmetric and irreflexive. Since $\leq'$ and $\#'$ respectively equal $\leq$ and $\#$ for the events in E', it follows directly that $\leq'$ is a partial order and $\#'$ is symmetric and irreflexive.

We need to show that $\forall e, e', e'' \in E$:

1. $\{e' \mid e' \leq e\}$ is finite

   Since $\{e' \mid e' \leq e\}$ is finite, then so it is $\{e' \mid e' \leq' e\} = \{e' \mid e' \leq e\}\backslash l$.

2. $e\#e' \leq e'' \Rightarrow e\#e''$

   Let $e, e', e'' \in E'$. Then $e, e', e'' \neq a$ and $\neg(e, e', e''\#a)$. By Definition 4.2.19, $e\#'e'$ entails $e\#e'$ and $e, e' \in E'$ and $e' \leq' e''$ entails $e' \leq e''$ and $e', e'' \in E$. Since E is an event structure, we have $e\#e' \leq e'' \Rightarrow e\#e''$. Thus $e\#'e' \leq' e'' \Rightarrow e\#'e''$ and $e, e', e'' \in E$.

□

Proof of Lemma 4.2.22

Proof. To prove $(\mathsf{E}_1 \,;\, \mathsf{E}_2)\backslash l \equiv (\mathsf{E}_1\backslash l) \,;\, \mathsf{E}_2$, we need to verify that:

- $(\mathsf{E}_1 \,;\, \mathsf{E}_2)\backslash l \sqsubseteq (\mathsf{E}_1\backslash l) \,;\, \mathsf{E}_2$
- $(\mathsf{E}_1\backslash l) \,;\, \mathsf{E}_2 \sqsubseteq (\mathsf{E}_1 \,;\, \mathsf{E}_2)\backslash l$

Let

$$\begin{aligned}
&\mathsf{E}_1 = (E_1, \leq_1, \#_1)\\
&\mathsf{E}_2 = (E_2, \leq_2, \#_2)\\
&\mathsf{E}_1 \,;\, \mathsf{E}_2 = (E_{1;2}, \leq_{1;2}, \#_{1;2})\\
&(\mathsf{E}_1 \,;\, \mathsf{E}_2)\backslash l = (E, \leq, \#)\\
&\mathsf{E}_1\backslash l = (E_1^l, \leq_1^l, \#_1^l)\\
&(\mathsf{E}_1\backslash l) \,;\, \mathsf{E}_2 = (E', \leq', \#')\\
&l \in \mathcal{I}(\mathsf{E}_1 \,;\, \mathsf{E}_2)
\end{aligned}$$

- $(\mathsf{E}_1 \,;\, \mathsf{E}_2)\backslash l \sqsubseteq (\mathsf{E}_1\backslash l) \,;\, \mathsf{E}_2$

  1. We start by defining a function $f : E \to E'$ such that

  $$f(e) = \begin{cases} e & \text{if } e \in E_1 \\ (e, x\backslash l) & \text{if } e \in E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{E}_1) \end{cases}$$

  where $x\backslash l = \{e \mid e \in x \wedge e \neq l\}$.
  First we show that $x\backslash l \in \mathcal{C}_{\mathsf{max}}(\mathsf{E}_1\backslash l)$. It is clear that $(x\backslash l) \cup \{l\} = x \in \mathcal{C}_{\mathsf{max}}(\mathsf{E}_1)$. Furthermore, since $x \in \mathcal{C}_{\mathsf{max}}(\mathsf{E}_1)$ then $\nexists y \in \mathcal{C}(\mathsf{E}_1)$ such that $x \prec\!\!\!\cdot\, y$. Now let $y\backslash l \in \mathcal{C}(\mathsf{E}_1\backslash l)$ such that $x\backslash l \prec\!\!\!\cdot\, y\backslash l$. By adding l to both configurations we would have $x \prec\!\!\!\cdot\, y$, which means that $x \notin \mathcal{C}_{\mathsf{max}}(\mathsf{E}_1)$, hence a contradiction. Thus $x\backslash l \in \mathcal{C}_{\mathsf{max}}(\mathsf{E}_1\backslash l)$.
  Another way of seeing that $x\backslash l \in \mathcal{C}_{\mathsf{max}}(\mathsf{E}_1\backslash l)$ is by noting that $x\backslash l$ removes the initial event from x, which is itself a maximal configuration.
  Now we show that f is injective. We have two cases:
  - $a, b \in E_1$
    Then we are done since the mapping is the identity.
  - $(a, x), (b, y) \in E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{E}_1)$
    From $(a, x) \neq (b, y)$ we have two cases:
    * $a \neq b$
      Trivially holds because the function is the identity
    * $a = b$ and $x \neq y$
      Then $f(a, x) \neq f(b, y) \Leftrightarrow (a, x\backslash l) \neq (b, y\backslash l) \Leftrightarrow a \neq b$ or $x\backslash l \neq y\backslash l$.
      Clearly it cannot be the case $a \neq b$, otherwise it would contradict our assumption that $a = b$, and also the fact that the identity function is injective. Hence it can only be $x\backslash l \neq y\backslash l$. Since $x \neq y$ then $\exists e \in x$ such that $e \notin y$. Since $e \neq l$, then we have $x\backslash l \neq y\backslash l$.
  2. $\pi(f(a)) = \pi(a)$
     We have two cases:

- $a \in E_1$
  It follows directly that $\pi(f(a)) = \pi(a)$, since $f(a) = a$
- $(a, x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$
  Then $\pi(f(a, x)) = \pi(a, x\backslash l) = \pi(a)$

3. $a \leq b \Leftrightarrow f(a) \leq' f(b)$

   $\Rightarrow$ Assume $a \leq b$.
   By Definition 4.2.19, $a, b \in E$ (*i.e.* $a, b \neq l$ and $\neg(a, b\#_{1;2} l)$), and $a \leq_{1;2} b$. We have three cases:

   (i) $a \leq_{1;2} b$ is of the form $a \leq_1 b$.
   We have $a, b \in E_1$. Note that $f(a) = a, f(b) = b \in E'$, hence $a, b \neq l$ and $\neg(a, b\#_{1;2} l)$. By Definition 4.2.19 $a \leq_1^l b$. By Definition 4.2.1 $a \leq' b$.

   (ii) $a \leq_{1;2} b$ is of the form $(a, x) \leq_{1;2} (b, x)$.
   It follows directly that $(a, x\backslash l) = f(a, x) \leq' f(b, x) = (b, x\backslash l)$ since $a, b \in E_2$ and $x\backslash l \in \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$.

   (iii) $a \leq_{1;2} b$ is of the form $a \leq_{1;2} (b, x)$.
   We know that $f(a) = a$ and $f(b, x) = (b, x\backslash l)$ such that $x\backslash l \in \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$ and $a \in x\backslash l$ since $a \neq l$ and $\neg(a\#_{1;2} l)$. By Definition 4.2.1, $a \leq' (b, x\backslash l)$.

   $\Leftarrow$ Assume $f(a) \leq' f(b)$.
   We have three cases:

   (i) $f(a) \leq' f(b)$ comes from $f(a) \leq_1^l f(b)$.
   We know that $f(a) = a$ and $f(b) = b$. Hence we have $a \leq_1^l b$ and, consequently, $a, b \neq l$ and $a, b\#_1 l$. By Definition 4.2.19, $a \leq_1 b$. By Definition 4.2.1 and Definition 4.2.19 that $a \leq b$.

   (ii) $f(a) \leq' f(b)$ is of the form $f(a, x) \leq' f(b, x)$.
   We know that $f(a, x) = (a, x\backslash l)$ and $f(b, x) = (b, x\backslash l)$. Hence we have $(a, x\backslash l) \leq' (b, x\backslash l)$. By Definition 4.2.1 we have $a \leq_2 b$. Again by Definition 4.2.1, we have $(a, x) \leq_{1;2} (b, x)$. Since $x\backslash l \in \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$, we know that none of the events in x are in conflict with l. Hence, $\neg((a, x), (b, x)\#_{1;2} l)$. By Definition 4.2.19, we have $(a, x) \leq (b, x)$.

   (iii) $f(a) \leq' f(b)$ is of the form $f(a) \leq' f(b, x)$.
   We know that $f(a) = a$ and $f(b, x) = (b, x\backslash l)$. Hence we have $a \leq' (b, x\backslash l)$, such that $a \in x\backslash l \in \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$, $a \in E_1^l$, and $b \in E_2$. By Definition 4.2.19, we have $a \in E_1$, such that $a \neq l$ and $\neg(a\#_1 l)$, and $x \in \mathcal{C}_{\max}(\mathsf{E}_1)$. By Definition4.2.1, we have $a \leq_{1;2} (b, x)$, such that $\neg((b, x)\#_{1;2} l)$, since x has no events in conflict with l. By Definition 4.2.19, we have $a \leq (b, x)$.

4. $a\#b \Leftrightarrow f(a)\#' f(b)$

   $\Rightarrow$ Assume $a\#b$.
   By Definition 4.2.19 it entails $a\#_{1;2} b$ and $a, b \in E$ (*i.e.* $a, b \neq l$ and $\neg(a, b\#_{1;2} l)$). By Definition 4.2.1 it entails $\exists(e_1 \leq_{1;2} a, e_1' \leq_{1;2} b)\,.\, e_1\#_1 e_1'$ or $a\#_2 b$. We have three cases:

   (i) $a\#_1 b$.
   Then $e_1 \leq_1 a, e_1' \leq_1 b$. We know that $e_1, e_1' \neq l$ since if $e_1 = l$ then $e_1\#_1 e_1' \leq_1 b \Rightarrow e_1\#_1 b$, which would entail $b \notin E$, a contradiction. Furthermore, $\neg(e_1, e_1'\#_1 l)$ with similar arguments. We know that $f(e) = e$ for $e \in \{e_1, e_1', a, b\}$. By Definition 4.2.19, $a\#_1^l b$, $e_1 \leq_1^l a$, $e_1' \leq_1^l b$, and $e_1\#_1^l e_1'$. By Definition 4.2.1, $a\#' b$.

(ii) $a\#_2 b$.
We then know that $(a,x)\#(b,x)$ and that $a,b \in E_2$. Clearly $(a,x),(b,x) \neq l$ and $\neg((a,x),(b,x)\#_{1;2} l)$. Furthermore, $x\backslash l \in \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$. Thus $(a,x\backslash l) = f(a,x), (b,x\backslash l) = f(b,x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$. It then follows by Definition 4.2.1, $(a,x\backslash l)\#'(b,x\backslash l)$.

(iii) $a\#(b,x)$
Then $e_1 \leq_1 a$, and $e_1' \leq (b,x)$ with $e_1' \in x$. We know that $e_1, e_1' \neq l$ since if $e_1' = l$ then $e_1'\#_1 e_1 \leq_1 a \Rightarrow e_1'\#_1 a$, which would entail $a \notin E$, a contradiction. Furthermore, $\neg(e_1, e_1'\#_1 l)$ with similar arguments. We know that $f(e) = e$ for $e \in \{e_1, e_1', a\}$ and $f(b,x) = (b, x\backslash l)$, since $x\backslash l \in \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$. By Definition 4.2.19, $a\#_1^l b$, $e_1 \leq_1^l a$, $e_1' \leq_1^l b$, and $e_1\#_1^l e_1'$. By Definition 4.2.1, $a\#'(b,x\backslash l)$.

$\Leftarrow$ Assume $f(a)\#'f(b)$.
$f(a)\#'f(b)$ entails $\exists(f(e_1) \leq' f(a), f(e_1') \leq' f(b)) \,.\, f(e_1)\#_1^l f(e_1')$ or $f(a)\#_2 f(b)$ (note that $f(e_1) = e_1$ and $f(e_1') = e_1'$ because $e_1, e_1' \in E_1$). We have three cases:

(i) $f(a)\#_1^l f(b)$.
We know that for $e \in \{e_1, e_1', a, b\}$, we have $f(e) = e$, $e \leq l$, and $\neg(e\#_1 l)$. We then have $a\#_1^l b$. By Definition 4.2.19, $a\#_1 b$, $e_1 \leq_1 a$, $e_1' \leq_1 b$, and $e_1\#_1 e_1'$. By Definition 4.2.1, $a\#_{1;2} b$, $e_1 \leq_{1;2} a$, $e_1' \leq_{1;2} b$, and $e_1\#_{1;2} e_1'$. By Definition 4.2.19, $a\#b$.

(ii) $f(a,x)\#'f(b,x)$.
We know that $f(a,x) = (a,x\backslash l)$ and $f(b,x) = (b,x\backslash l)$. Then $(a,x\backslash l)\#'(b,x\backslash l)$. By Definition 4.2.1, $a\#_2 b$. Furthermore, $(x\backslash l)\cup\{l\} = x \in \mathcal{C}_{\max}(\mathsf{E}_1)$, since $x\backslash l \in \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$. We then have $(a,x),(b,x) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1)$ and $\neg((a,x),(b,x)\#_{1;2} l)$, since x has no events in conflict with l. By Definition 4.2.1, $(a,x)\#(b,x)$.

(iii) $f(a)\#'f(b,x)$.
This entails $f(e_1) \leq' f(a), f(e_1') \leq' f(b,x) \,.\, f(e_1)\#_1^l f(e_1')$. We know that for $e \in \{e_1, e_1', a\}$, we have $f(e) = e$, $e \leq l$, and $\neg(e\#_1 l)$. Furthermore, $f(b,x) = (b,x\backslash l)$. We then have $a\#'(b,x\backslash l)$ and $e_1 \leq' a, e_1' \leq' (b,x\backslash l) \,.\, e_1\#_1^l e_1'$. By Definition 4.2.1, $e_1 \leq_1^l a$ and $e_1' \in x\backslash l \in \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$. By Definition 4.2.19, $e_1 \leq_1 a$ and $e_1\#_1 e_1'$. By Definition 4.2.1, $e_1 \leq_{1;2} a$ and $e_1\#_{1;2} e_1'$. Furthermore, $(x\backslash l)\cup\{l\} = x \in \mathcal{C}_{\max}(\mathsf{E}_1)$, since $x\backslash l \in \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$. Thus $e_1' \in x$ and, consequently, $e_1' \leq_{1;2} (b,x)$. Furthermore, $\neg((b,x)\#_{1;2} l)$ since x has no events in conflict with l. By Definition 4.2.19, $a\#(b,x)$.

- $(\mathsf{E}_1\backslash l)\,;\,\mathsf{E}_2 \sqsubseteq (\mathsf{E}_1\,;\,\mathsf{E}_2)\backslash l$

1. We start by defining $f : E' \to E$ such that

$$f(e) = \begin{cases} e & \text{if } e \in E_1^l \\ (e, x\cup\{l\}) & \text{if } e \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1\backslash l) \end{cases}$$

It is straightforward to see that if $x \in \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$ then $x\cup\{l\} \in \mathcal{C}_{\max}(\mathsf{E}_1)$.

We now show that f is injective. We have two cases:

- $a,b \in E_1^l$
Then we are done since the mapping is the identity.
- $(a,x),(b,y) \in E_2 \times \mathcal{C}_{\max}(\mathsf{E}_1\backslash l)$
From $(a,x) \neq (b,y)$ we have two cases:

* $a \neq b$
  Trivially holds because the function is the identity.
* $a = b$ and $x \neq y$
  Then $f(a,x) \neq f(b,y) \Leftrightarrow (a, x \cup \{l\}) \neq (b, y \cup \{l\}) \Leftrightarrow a \neq b$ or $x \cup \{l\} \neq y \cup \{l\}$.
  Clearly it cannot be the case $a \neq b$, otherwise it would contradict our assumption that $a = b$, and also the fact that the identity function is injective. Hence it can only be $x \cup \{l\} \neq y \cup \{l\}$. Since $x \neq y$ then $\exists e \in x$ such that $e \notin y$. Since $e \neq l$, then we have $x \cup \{l\} \neq y \cup \{l\}$.

- $\pi(f(a)) = \pi(a)$
  Similar to the other case
- $a \leq' b \Leftrightarrow f(a) \leq f(b)$
  Similar to the other case
- $a \#' b \Leftrightarrow f(a) \# f(b)$
  Similar to the other case

□

Proof of Lemma 4.2.23

Proof. It is straightforward to see that if $l \in \mathcal{I}(\mathsf{E}_1 \,||\, \mathsf{E}_2)$ then either $l \in \mathcal{I}(\mathsf{E}_1)$ or $l \in \mathcal{I}(\mathsf{E}_2)$, by Definition 4.2.3. Let us focus in the case where $l \in \mathcal{I}(\mathsf{E}_1)$, since the proof for the other way around is similar.

We need to prove

- $(\mathsf{E}_1 \,||\, \mathsf{E}_2)\backslash l \sqsubseteq (\mathsf{E}_1 \backslash l) \,||\, \mathsf{E}_2$
- $(\mathsf{E}_1 \backslash l) \,||\, \mathsf{E}_2 \sqsubseteq (\mathsf{E}_1 \,||\, \mathsf{E}_2)\backslash l$

Let

$$
\begin{aligned}
&\mathsf{E}_1 = (E_1, \leq_1, \#_1)\\
&\mathsf{E}_2 = (E_2, \leq_2, \#_2)\\
&\mathsf{E}_1 \,||\, \mathsf{E}_2 = (E, \leq, \#)\\
&(\mathsf{E}_1 \,||\, \mathsf{E}_2)\backslash l = (E', \leq', \#')\\
&\mathsf{E}_1 \backslash l = (E_1^l, \leq_1^l, \#_1^l)\\
&\mathsf{E}_2 \backslash l = (E_2^l, \leq_2^l, \#_2^l)\\
&(\mathsf{E}_1 \backslash l) \,||\, \mathsf{E}_2 = (E^l, \leq^l, \#^l)\\
&l \in \mathcal{I}(\mathsf{E}_1 \,||\, \mathsf{E}_2)
\end{aligned}
$$

- $(\mathsf{E}_1 \,||\, \mathsf{E}_2)\backslash l \sqsubseteq (\mathsf{E}_1 \backslash l) \,||\, \mathsf{E}_2$
  1. We start by defining $f : E' \to E^l$ as being the identity, which is injective.
  2. $\pi(f(a)) = \pi(a)$
     It follows directly since f is the identity

3. $e \leq' e' \Leftrightarrow e \leq^l e'$

   $\Rightarrow$ Assume $e \leq' e'$

   By Definition 4.2.19, $e \leq e'$ and $e, e' \in E'$. We have two cases:

   (i) $e \leq e'$ is of the form $e \leq_1 e'$

   Then $e, e' \in E_1$. Since $e, e' \in E'$, then $e, e' \neq l$ and $\neg(e\#l), \neg(e'\#l)$. By Definition 4.2.3 $\neg(e\#_1 l), \neg(e'\#_1 l)$. By Definition 4.2.19, $e, e' \in E_1^l$ and $e \leq_1^l e'$. By Definition 4.2.3, $e \leq^l e'$.

   (ii) $e \leq e'$ is of the form $e \leq_2 e'$

   It follows directly that $e \leq^l e'$, since $l \in E_1$.

   $\Leftarrow$ Assume $e \leq^l e'$

   By Definition 4.2.19 and by Definition 4.2.3 we have two cases:

   (i) $e, e' \in E_1^l$ and $e \leq^l e'$ is of the form $e \leq_1^l e'$

   Since $e, e' \in E_1^l$, by Definition 4.2.19 we have $e, e' \in E_1$, $e, e' \neq l$ and $\neg(e\#_1 l), \neg(e'\#_1 l)$. Hence, by Definition 4.2.3 and Definition 4.2.19 we have $e, e' \in E'$. From $e \leq_1^l e'$, we know that from Definition 4.2.19 we have $e \leq_1 e'$ and $e, e' \in E_1^l$. It then follows by Definition 4.2.3 and Definition 4.2.19 that $e \leq' e'$.

   (ii) $e, e' \in E_2^l$ and $e \leq^l e'$ is of the form $e \leq_2^l e'$

   It follows directly that $e \leq' e'$, since $l \in E_1$.

4. $e\#'e' \Leftrightarrow e\#^l e'$

   The reasoning for this case is similar to the previous one, since the definitions are identical.

- $(\mathsf{E}_1 \backslash l) \,||\, \mathsf{E}_2 \sqsubseteq (\mathsf{E}_1 \,||\, \mathsf{E}_2) \backslash l$

  For this case, we notice that the function $f : E^l \to E'$ is again the identity, which is injective.

  We omit the remaining cases since they are similarly proved.

□

Proof of Lemma 4.2.24

Proof. We need to prove

- $(\mathsf{E}_1 \text{ or } \mathsf{E}_2) \backslash l \sqsubseteq \mathsf{E}_1 \backslash l$, $\mathsf{E}_1 \backslash l \sqsubseteq (\mathsf{E}_1 \text{ or } \mathsf{E}_2) \backslash l$ when $l \in \mathcal{I}(\mathsf{E}_1)$
- $(\mathsf{E}_1 \text{ or } \mathsf{E}_2) \backslash l \sqsubseteq \mathsf{E}_2 \backslash l$, $\mathsf{E}_2 \backslash l \sqsubseteq (\mathsf{E}_1 \text{ or } \mathsf{E}_2) \backslash l$ when $l \in \mathcal{I}(\mathsf{E}_2)$

Since both cases are identical, we focus solely when $l \in \mathcal{I}(\mathsf{E}_1)$.

Let

$$
\begin{aligned}
&\mathsf{E}_1 = (E_1, \leq_1, \#_1)\\
&\mathsf{E}_2 = (E_2, \leq_2, \#_2)\\
&\mathsf{E}_1 \text{ or } \mathsf{E}_2 = (E, \leq, \#)\\
&(\mathsf{E}_1 \text{ or } \mathsf{E}_2) \backslash l = (E', \leq', \#')\\
&\mathsf{E}_1 \backslash l = (E_1^l, \leq_1^l, \#_1^l)\\
&l \in \mathcal{I}(\mathsf{E}_1 \text{ or } \mathsf{E}_2)
\end{aligned}
$$

Consider $l \in \mathcal{I}(\mathsf{E}_1)$.

- $(\mathsf{E}_1 \text{ or } \mathsf{E}_2)\backslash l \sqsubseteq \mathsf{E}_1\backslash l$

  1. We begin by defining the function $f : E' \to E_1^l$ as the identity, which is injective.
  2. $\pi(f(a)) = \pi(a)$
     It follows directly since f is the identity
  3. $e \leq' e' \Leftrightarrow e \leq_1^l e'$

     $\Rightarrow$ Assume $e \leq' e'$.
     By Definition 4.2.19, $e \leq' e'$ entails $e, e' \in E'$. From f we know that $e, e' \notin E_2$. Hence $\neg(e \leq_2 e')$. Thus it only remains that $e \leq_1 e'$ with $e, e' \in E_1$. Again, from f we know that $e, e' \in E_1^l$ since $e, e' \neq l$ and $\neg(e\#l), \neg(e'\#l)$. Thus, by Definition 4.2.19, $e \leq_1^l e'$.

     $\Leftarrow$ Assume $e \leq_1^l e'$.
     By Definition 4.2.19, $e \leq_1^l e'$ entails $e \leq_1 e'$ with $e, e' \in E_1^l$. Hence $e, e' \neq l$, $\neg(e\#_1 l), \neg(e'\#_1 l)$, and $e, e' \in E_1$. By Definition 4.2.6 we have $e, e' \in E$ and $e \leq e'$. Furthermore, we also have $\neg(e\#l), \neg(e'\#l)$ and consequently $e, e' \in E'$. Thus $e \leq' e'$.
  4. $e\#'e' \Leftrightarrow e\#_1^l e'$

     $\Rightarrow$ Assume $e\#'e'$.
     By Definition 4.2.19 we have $e\#e'$ and $e, e' \in E'$ such that $e, e' \neq l$ and $\neg(e\#l), \neg(e'\#l)$. By Definition 4.2.6, $e\#e'$ entails $e\#_1 e'$ or $e\#_2 e'$ or $\{e\#e' \mid e \in E_1, e' \in E_2\}$. Since $l \in \mathcal{I}(\mathsf{E}_1)$, then $l \in E_1$ and consequently $e, e' \notin E_2$. Thus, by exclusion of hypothesis we have $e\#_1 e'$ and consequently $\neg(e\#_1 l), \neg(e'\#_1 l)$. Thus by Definition 4.2.19 we have $e\#_1^l e'$.

     $\Leftarrow$ Assume $e\#_1^l e'$.
     By Definition 4.2.19 we have $e\#_1 e'$ and $e, e' \in E_1^l$, such that $e, e' \neq l$ and $\neg(e\#_1 l), \neg(e'\#_1 l)$. Furthermore $e, e' \in E_1$. By Definition 4.2.6, $e, e' \in E$ and $\neg(e\#l), \neg(e'\#l)$. By Definition 4.2.19 we have $e\#'e'$.

- $\mathsf{E}_1\backslash l \sqsubseteq (\mathsf{E}_1 \text{ or } \mathsf{E}_2)\backslash l$

  For this case, we notice that the function $f : E' \to E$ is again the identity, which is injective.

  We omit the remaining cases since they are similarly proved.

□

Proof of Lemma 4.3.4

Proof. Let E_1, E_2, and E_3 be event structures.

- Reflexivity: $\mathsf{E}_1 \trianglelefteq \mathsf{E}_1$

  It follows directly from Definition 4.3.3.

- Transitivity: $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2, \mathsf{E}_2 \trianglelefteq \mathsf{E}_3 \Rightarrow \mathsf{E}_1 \trianglelefteq \mathsf{E}_3$

  1. $E_1 \subseteq E_3$
     Let $e \in E_1$. Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ then $e \in E_2$. Since $\mathsf{E}_2 \trianglelefteq \mathsf{E}_3$ then $e \in E_3$. Hence $E_1 \subseteq E_3$.
  2. $e \leq_1 e' \Leftrightarrow e, e' \in E_1, e \leq_3 e'$

$\Rightarrow$ Let $e \leq_1 e'$.
Clearly $e, e' \in E_1$. Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ then $e \leq_2 e'$. Furthermore $e, e' \in E_2$. Since $\mathsf{E}_2 \trianglelefteq \mathsf{E}_3$ then $e \leq_3 e'$.

$\Leftarrow$ Let $e, e' \in E_1, e \leq_3 e'$.
Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ then $e, e' \in E_2$. Since $\mathsf{E}_2 \trianglelefteq \mathsf{E}_3$ then $e \leq_2 e'$. Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ $e \leq_1 e'$.

3. $e \#_1 e' \Leftrightarrow e, e' \in E_1, e \#_3 e'$
Similar to $\leq$, *i.e.* previous point.

- Antisymmetry: $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2, \mathsf{E}_2 \trianglelefteq \mathsf{E}_1 \Rightarrow \mathsf{E}_1 = \mathsf{E}_2$

1. $E_1 = E_2$
Let $e \in E_1$. Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ then $e \in E_2$. Let $e \in E_2$. Since $\mathsf{E}_2 \trianglelefteq \mathsf{E}_1$ then $e \in E_1$. Hence $E_1 = E_2$.

2. $(e \leq_1 e' \Leftrightarrow e, e' \in E_1, e \leq_2 e') = (e \leq_2 e' \Leftrightarrow e, e' \in E_2, e \leq_1 e')$

$\Rightarrow$ Let $e \leq_1 e'$. Clearly $e, e' \in \mathsf{E}_1$. From $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$, $e \leq_2 e'$.
Let $e \leq_2 e'$. Clearly $e, e' \in \mathsf{E}_2$. From $\mathsf{E}_2 \trianglelefteq \mathsf{E}_1$, $e \leq_1 e'$.

$\Leftarrow$ Let $e, e' \in E_1, e \leq_2 e'$. Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ then $e \leq_1 e'$. Let $e, e' \in E_2, e \leq_1 e'$. Since $\mathsf{E}_2 \trianglelefteq \mathsf{E}_1$ then $e \leq_2 e'$.

Hence $\leq_1 = \leq_2$

3. $(e \#_1 e' \Leftrightarrow e, e' \in E_1, e \#_2 e') = (e \#_2 e' \Leftrightarrow e, e' \in E_2, e \#_1 e')$
Similar reasoning as previous point.

□

Proof of Lemma 4.3.5

Proof.

- $\bot$ is an event structure

It follows directly that all conditions in Definition 3.2.1 are trivially satisfied because $\bot$ has no events.

- For any event structure $\mathsf{E} = (E, \leq, \#)$ we want to show $\bot \trianglelefteq \mathsf{E}$.

1. $\emptyset \subseteq E$
Trivially holds.

2. $e \leq_\bot e' \Leftrightarrow e, e' \in \emptyset, e \leq e'$
Since $\bot$ has no events and the causal relation is empty, it follows that $e \leq_\bot e'$ and $e, e' \in \emptyset$ are false. Hence the condition trivially holds.

3. $e \#_\bot e' \Leftrightarrow e, e' \in \emptyset, e \# e'$
Similar to previous point.

□

Proof of Lemma 4.3.7

Proof. It follows directly from Definition 4.3.6 that $\leq^\omega$ is a partial order and that $\#^\omega$ is symmetric and irreflexive, since for every case, it exists $n \in \omega$ such that $\leq^\omega = \leq_n$ and $\#^\omega = \#_n$, with $\leq_n$ being a partial order and $\#_n$ being symmetric and irreflexive.

- $\{e' \mid e' \leq^{\omega} e\}$ is finite

  By Definition 4.3.6, $e' \leq^{\omega} e'$ entails $\exists n \in \omega$ such that $e' \leq_n e$. Consequently, $e, e' \in E_n$. Furthermore, E_n is an event structure. Hence $\{e' \mid e' \leq_n e\}$ is finite. Thus $\{e' \mid e' \leq^{\omega} e\}$ is finite.

- $e \#^{\omega} e' \leq^{\omega} e'' \Rightarrow e \#^{\omega} e''$

  By Definition 4.3.6, $e' \leq^{\omega} e'$ entails $\exists n \in \omega$ such that $e \#_n e' \leq_n e''$, where E_n is an event structure. Hence $e \#_n e''$. Thus $e \#^{\omega} e''$.

□

Proof of Lemma 4.3.8

Proof.

- E^{ω} is an upper bound

  $\forall n \in \omega$ we need to have $\mathsf{E}_n \trianglelefteq \mathsf{E}^{\omega}$. It follows directly from Definition 4.3.3 that $\forall n \in \omega\, \mathsf{E}_n \trianglelefteq \mathsf{E}^{\omega}$ since E^{ω} is by definition the union of all E_n.

- E^{ω} is the least upper bound

  Let $\mathsf{E} = (E, \leq, \#)$ be an upper bound of the chain. We need to show that if $\mathsf{E}_n \trianglelefteq \mathsf{E}^{\omega}$ and $\mathsf{E}_n \trianglelefteq \mathsf{E}$ then $\mathsf{E}^{\omega} \trianglelefteq \mathsf{E}$.

  1. $E^{\omega} \subseteq E$

     Let $e \in E^{\omega}$. By Definition 4.3.6, $\exists n \in \omega$ such that $e \in E_n$. By $\mathsf{E}_n \trianglelefteq \mathsf{E}$ we have $e \in E$.

  2. $e \leq^{\omega} e' \Leftrightarrow e, e' \in E^{\omega}$ and $e \leq e'$

     $\Rightarrow$ Let $e \leq^{\omega} e'$.
     By Definition 4.3.6, $\exists n \in \omega$ such that $e \leq_n e'$. It is then clear that $e, e' \in E_n \subseteq E^{\omega}$. Since $\mathsf{E}_n \trianglelefteq \mathsf{E}$ we have $e \leq e'$.

     $\Leftarrow$ $e, e' \in E^{\omega}$ and $e \leq e'$
     By Definition 4.3.6, $\exists n \in \omega$ such that $e, e' \in E_n$. Since $\mathsf{E}_n \trianglelefteq \mathsf{E}$, $e \leq_n e'$. By Definition 4.3.6, $e \leq^{\omega} e'$.

  3. $e \#^{\omega} e' \Leftrightarrow e, e' \in E^{\omega}$ and $e \# e'$

     Similar to previous point.

□

Proof of Lemma 4.3.10

Proof. Let $\mathsf{E} = (E, \leq, \#)$, $\mathsf{E}_1 = (E_1, \leq_1, \#_1)$, $\mathsf{E}_2 = (E_2, \leq_2, \#_2)$, $\mathsf{E}\,;\mathsf{E}_1 = (E^1, \leq^1, \#^1)$, and $\mathsf{E}\,;\mathsf{E}_2 = (E^2, \leq^2, \#^2)$, such that $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$.

1. $E^1 \subseteq E^2 \Leftrightarrow E \uplus (E_1 \times \mathcal{C}_{\mathsf{max}}(\mathsf{E})) \subseteq E \uplus (E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{E}))$

   By Definition 4.2.1 we have two cases:

   (a) $e \in E$

   Then we are done.

   (b) $e \in E_1 \times \mathcal{C}_{\mathsf{max}}(\mathsf{E})$

   We know that e is of the form (e_1, x) where $e_1 \in E_1$ and $x \in \mathcal{C}_{\mathsf{max}}(\mathsf{E})$. Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ then $e_1 \in E_2$ and consequently $e \in \mathcal{C}_{\mathsf{max}}(\mathsf{E})$.

2. $\forall e, e' \,.\, e \leq^1 e' \Leftrightarrow e, e' \in E^1$ and $e \leq^2 e'$

   $\Rightarrow$ Assume $e \leq^1 e$. Clearly $e, e' \in E^1$. By Definition 4.2.1 we have three cases:

   (a) $e \leq^1 e'$ is of the form $e \leq e'$
   Hence $e, e' \in E$. By Definition 4.2.1 $e \leq^2 e'$.

   (b) $e \leq e'$ is of the form $e \leq_1 e'$
   We know that e, e' are of the form $(e, x), (e', x) \in E_1 \times \mathcal{C}_{\mathsf{max}}(\mathsf{E})$, which entails $e, e' \in E_1$ and $x \in \mathcal{C}_{\mathsf{max}}(\mathsf{E})$. Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$, $e, e' \in E_2$ and $e \leq_2 e'$. By Definition 4.2.1 we have $(e, x) \leq^2 (e', x)$.

   (c) $e \leq^1 e'$ is of the form $e \leq^1 (e', x)$
   We know that $e \in E$, $e \in x \in \mathcal{C}_{\mathsf{max}}(\mathsf{E})$, and $(e', x) \in E_1 \times \mathcal{C}_{\mathsf{max}}(esE)$, with the last entailing $e' \in E_1$. Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$, $e' \in E_2$ and consequently $(e', x) \in E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{E})$. By Definition 4.2.1 we have $e \leq^2 (e', x)$.

   $\Leftarrow$ Assume $e, e' \in E^1$ and $e \leq^2 e'$. The cases are distinguished by $\leq^2$.

   (a) $e \leq^2 e'$ is of the form $e \leq e'$
   Hence $e, e' \in E$. By Definition 4.2.1, $e \leq^1 e'$.

   (b) $e \leq^2 e$ is of the form $e \leq_2 e'$
   We know that e, e' are of the form $(e, x), (e', x) \in E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{E})$, which entails $e, e' \in E_2$ and $x \in \mathcal{C}_{\mathsf{max}}(\mathsf{E})$. Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ and $e, e' \in E^1$, which entails for this case that $e, e' \in E_1$, then $e \leq_1 e'$. By Definition 4.2.1 we have $(e, x) \leq^1 (e', x)$.

   (c) $e \leq^2 e'$ is of the form $e \leq^2 (e', x)$
   We know that $e \in E$, $e \in x \in \mathcal{C}_{\mathsf{max}}(\mathsf{E})$, and $(e', x) \in E_2 \times \mathcal{C}_{\mathsf{max}}(esE)$, with the last entailing $e' \in E_2$. Since $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ and $e' \in E^1$, which entails for this case that $e' \in E_1$, then $(e', x) \in E_1 \times \mathcal{C}_{\mathsf{max}}(\mathsf{E})$. By Definition 4.2.1 we have $e \leq^1 (e', x)$.

3. $\forall e, e' \,.\, e\#^1 e' \Leftrightarrow e, e' \in E$ and $e\#^2 e'$

   $\Rightarrow$ Assume $e\#^1 e'$.
   Clearly $e, e' \in E^1$. From Definition 4.2.1 we have that $\exists (a \leq^1 e,\ a' \leq^1 e')$ such that $a\#a'$ or $e\#_1 e'$. For the former we have that $a \leq^1 e,\ a' \leq^1 e'$ entails $a \leq^2 e,\ a' \leq^2 e'$. For the latter we have that $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$, hence $e\#_2 e'$. Thus $e\#^2 e'$.

   $\Leftarrow$ Assume $e, e' \in E^1$ and $e\#^2 e'$.
   By Definition 4.2.1 it exists $a \leq^2 e$ and $a' \leq^2 e'$ such that $a\#a'$ or $e\#_2 e'$. Since $e, e' \in E^1$ and $\mathsf{E}_1 \trianglelefteq \mathsf{E}_2$ we have $a \leq^1 e,\ a' \leq^1 e'$ and $e\#_1 e'$. It then follows directly that $e\#^1 e'$.

□

Proof of Lemma 4.3.11

Proof. It follows directly from Definition 4.2.3. □

Proof of Lemma 4.3.12

Proof. It follows directly from Definition 4.2.6. □

Proof of Lemma 4.3.13

Proof.

- $\Rightarrow$: Assume op is continuous.

  We have $op(\bigsqcup_n \mathsf{E}_n) = \bigsqcup_n op(\mathsf{E}_n)$. Let $\mathsf{E}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}_n \trianglelefteq \ldots$ and $\mathsf{E}'_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}'_n \trianglelefteq \ldots$ be two ω-chains such that $\mathsf{E}_1 \trianglelefteq \mathsf{E}'_1, \ldots, \mathsf{E}_n \trianglelefteq \mathsf{E}'_n$. We want to show that $\mathsf{E}_1 \trianglelefteq \mathsf{E}'_1 \Rightarrow op\mathsf{E}_1 \trianglelefteq op\mathsf{E}'_1, \ldots, \mathsf{E}_n \trianglelefteq \mathsf{E}'_n \Rightarrow op\mathsf{E}_n \trianglelefteq op\mathsf{E}'_n, \ldots$ For that we can make use of the least upper bound, *i.e.* $\bigsqcup_n \mathsf{E}_n \trianglelefteq \bigsqcup_n \mathsf{E}'_n \Rightarrow op(\bigsqcup_n \mathsf{E}_n) \trianglelefteq \bigsqcup_n op(\mathsf{E}'_n)$. Since op is continuous, $op \bigsqcup_n \mathsf{E}_n \trianglelefteq \bigsqcup_n op\mathsf{E}'_n$. Hence op is monotonic. Now it lacks to show that each event of $op(\bigsqcup_n \mathsf{E}_n)$ is an event of $\bigsqcup_n op(\mathsf{E}'_n)$. But that comes freely since $op(\bigsqcup_n \mathsf{E}_n) = \bigsqcup_n op(\mathsf{E}_n)$.

- $\Leftarrow$: Assume $1.$ and $2.$ above.

  We want to show $op(\bigsqcup_n \mathsf{E}_n) = \bigsqcup_n op(\mathsf{E}_n)$. Let $\mathsf{E}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}_n \trianglelefteq \ldots$ be a ω-chain. By $1.$ we know that op is monotonic, hence $\mathsf{E}_n \trianglelefteq \bigsqcup_n \mathsf{E}_n$ entails $op(\mathsf{E}_n) \trianglelefteq op(\bigsqcup_n \mathsf{E}_N)$ that leads to $op(\bigsqcup_n \mathsf{E}_n) \trianglelefteq \bigsqcup_n op(\mathsf{E}_n)$. By $2.$, each event of $op(\bigsqcup_n \mathsf{E}_n)$ is an event of $\bigsqcup_n op(\mathsf{E}_n)$. Hence by Definition 4.3.3, $op(\bigsqcup_n \mathsf{E}_n) = \bigsqcup_n op(\mathsf{E}_n)$.

□

Proof of Lemma 4.3.14

Proof. By Lemma 4.3.10 we know that sequential composition is monotone w.r.t $\trianglelefteq$ at right. It lacks to show that each event of $\mathsf{E} \,;\, \bigsqcup_m \mathsf{E}_m$ is an event of $\bigsqcup_m (\mathsf{E} \,;\, \mathsf{E}_m)$. Let $\mathsf{E}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}_m \trianglelefteq \ldots$ be an ω-chain such that $\bigsqcup_m \mathsf{E}_m$ is its least upper bound and $\mathsf{E} \,;\, \mathsf{E}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E} \,;\, \mathsf{E}_m \trianglelefteq \ldots$ be another ω-chain with $\bigsqcup_m (\mathsf{E} \,;\, \mathsf{E}_m)$ as its least upper bound. Let e be an event of $\mathsf{E} \,;\, \bigsqcup_m \mathsf{E}_m$. By Definition 4.2.1 we have two cases:

1. e is an event of E

   Then we are done, since $\forall m$, e is an event of $\mathsf{E} \,;\, \mathsf{E}_m$. Hence it is an event of $\bigsqcup_m (\mathsf{E} \,;\, \mathsf{E}_m)$.

2. e is an event of $(\bigcup_{m \in \omega} E_m) \times \mathcal{C}_{\mathsf{max}}(\mathsf{E})$

   We know that e is of the form (e_m, x) with e_m an event of $\bigsqcup_m \mathsf{E}_m$ and $x \in \mathcal{C}_{\mathsf{max}}(\mathsf{E})$. The former entails $\exists m$ such that e_m is an event of E_m. By Definition 4.2.1 we have (e_m, x) as an event of $\mathsf{E} \,;\, \mathsf{E}_m$. Consequently (e_m, x) is an event of $\bigsqcup_m (\mathsf{E} \,;\, \mathsf{E}_m)$.

By Lemma 4.3.13 we are done. □

Proof of Lemma 4.3.15

Proof. By Lemma 4.3.11 we know that parallel composition is monotone w.r.t $\trianglelefteq$. It lacks to show that each event of $\bigsqcup_n \mathsf{E}_n \,||\, \bigsqcup_m \mathsf{E}_m$ is an event of $\bigsqcup_{n,m} (\mathsf{E}_n \,||\, \mathsf{E}_m)$.

Let $\mathsf{E}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}_n \trianglelefteq \ldots$ and $\mathsf{E}'_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}'_m \trianglelefteq \ldots$ be ω-chains with least upper bound $\bigsqcup_n \mathsf{E}_n$ and $\bigsqcup_m \mathsf{E}_m$, respectively. Let e be an event of $\bigsqcup_n \mathsf{E}_n \,||\, \bigsqcup_m \mathsf{E}_m$. By Definition 4.2.3 we have two cases:

1. e is an event of $\bigsqcup_n \mathsf{E}_n$

   Then $\exists n \in \omega$ such that e is an event of E_n. By Definition 4.2.3, e is an event of $\mathsf{E}_n \,||\, \mathsf{E}_m$ and consequently is an event of $\bigsqcup_{n,m} (\mathsf{E}_n \,||\, \mathsf{E}_m)$.

2. e is an event of $\bigsqcup_m \mathsf{E}_m$ Similar to previous point.

By Lemma 4.3.13 we are done. □

Proof of Lemma 4.3.16

Proof. By Lemma 4.3.12 we know that non-deterministic composition is monotone w.r.t $\trianglelefteq$. It lacks to show that each event of $\bigsqcup_n \mathsf{E}_n$ or $\bigsqcup_m \mathsf{E}_m$ is an event of $\bigsqcup_{n,m}(\mathsf{E}_n \text{ or } \mathsf{E}_m)$.

Let $\mathsf{E}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}_n \trianglelefteq \ldots$ and $\mathsf{E}'_1 \trianglelefteq \cdots \trianglelefteq \mathsf{E}'_m \trianglelefteq \ldots$ be ω-chains with least upper bound $\bigsqcup_n \mathsf{E}_n$ and $\bigsqcup_m \mathsf{E}_m$, respectively. Let e be an event of $\bigsqcup_n \mathsf{E}_n$ or $\bigsqcup_m \mathsf{E}_m$. By Definition 4.2.6 we have two cases:

1. e is an event of $\bigsqcup_n \mathsf{E}_n$

   Then $\exists n \in \omega$ such that e is an event of E_n. By Definition 4.2.6, e is an event of E_n or E_m and consequently is an event of $\bigsqcup_{n,m}(\mathsf{E}_n \text{ or } \mathsf{E}_m)$.

2. e is an event of $\bigsqcup_m \mathsf{E}_m$ Similar to previous point.

By Lemma 4.3.13 we are done. □

Proof of Lemma 4.3.17

Proof. $\Gamma(fix(\Gamma)) = fix(\Gamma) \Leftrightarrow \Gamma(\bigsqcup_n \Gamma^n(\bot)) = \bigsqcup_n(\Gamma^n(\bot))$. Since Γ is continuous, we have that $\Gamma(\bigsqcup_n \Gamma^n(\bot)) = \bigsqcup_n \Gamma\Gamma^n(\bot) = \bigsqcup_n \Gamma^{n+1}(\bot)$. We note that: $\bot \sqcup \bigsqcup_n \Gamma\Gamma^n(\bot) = \bigsqcup_n \Gamma^n(\bot)$. Since $\bot$ is the 'identity of the least upper bound' we have: $\bot \sqcup \bigsqcup_n \Gamma\Gamma^n(\bot) = \bigsqcup_n \Gamma^n(\bot) \Leftrightarrow \bigsqcup_n \Gamma\Gamma^n(\bot) = \bigsqcup_n \Gamma^n(\bot) \Leftrightarrow \Gamma(\bigsqcup_n \Gamma^n(\bot)) = \bigsqcup_n \Gamma^n(\bot)$.

Now we need to show that $fix(\Gamma)$ is the least fixpoint. Let E be an event structure, $\Gamma(E) \trianglelefteq E$, and $\bot \trianglelefteq E$. By the monotonic property $\Gamma(\bot) \trianglelefteq \Gamma(E)$. Since $\Gamma(E) \trianglelefteq E$ then $\Gamma(\bot) \trianglelefteq E$. By induction $\Gamma^n(E) \trianglelefteq E$. Thus $fix(\Gamma) = \bigsqcup_n \Gamma^n(\bot) \trianglelefteq E$. Hence $fix(\Gamma)$ is the least fixpoint. □

Proof of Lemma 4.3.19

Proof. •

$$
\begin{aligned}
&\Gamma^{C_1\,;C_2,\gamma}(\bigsqcup_n \mathsf{E}_n)\\
=&\{\text{Definition 4.3.18}\}\\
&[\![C_1\,;\,C_2]\!]_{\gamma(X\leftarrow\bigsqcup_n \mathsf{E}_n)}\\
=&\{\text{Definition 4.3.18}\}\\
&[\![C_1]\!]\,;\,[\![C_2]\!]_{\gamma(X\leftarrow\bigsqcup_n \mathsf{E}_n)}\\
=&\{\text{Definition 4.3.18}\}\\
&[\![C_1]\!]\,;\,\bigsqcup_n \Gamma^{C_2,\gamma}(\mathsf{E}_n)\\
=&\{\text{Lemma 4.3.14}\}\\
&=\bigsqcup_n([\![C_1]\!]\,;\,\Gamma^{C_2,\gamma}(\mathsf{E}_n))\\
=&\{\text{Definition 4.3.18}\}\\
&=\bigsqcup_n([\![C_1]\!]\,;\,[\![C_2]\!]_{\gamma(X\leftarrow\mathsf{E}_n)})\\
=&\{\text{Definition 4.3.18}\}\\
&\bigsqcup_n [\![C_1\,;\,C_2]\!]_{\gamma(X\leftarrow\mathsf{E}_n)}
\end{aligned}
$$

- $$\begin{aligned}
&\Gamma^{C_1 \,||\, C_2,\gamma}(\bigsqcup_n \mathsf{E}_n)\\
=&\{\text{Definition 4.3.18}\}\\
&[\![C_1 \,||\, C_2]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{E}_n)}\\
=&\{\text{Definition 4.3.18}\}\\
&[\![C_1]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{E}_n)} \,||\, [\![C_2]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{E}_n)}\\
=&\{\text{Definition 4.3.18}\}\\
&\bigsqcup_n \Gamma^{C_1,\gamma}(\mathsf{E}_n) \,||\, \bigsqcup_n \Gamma^{C_2,\gamma}(\mathsf{E}_n)\\
=&\{\text{Lemma 4.3.15}\}\\
&= \bigsqcup_n (\Gamma^{C_1,\gamma}(\mathsf{E}_n) \,||\, \Gamma^{C_2,\gamma}(\mathsf{E}_n))\\
=&\{\text{Definition 4.3.18}\}\\
&= \bigsqcup_n ([\![C_1]\!]_{\gamma(X \leftarrow \mathsf{E}_n)} \,||\, [\![C_2]\!]_{\gamma(X \leftarrow \mathsf{E}_n)})\\
=&\{\text{Definition 4.3.18}\}\\
&\bigsqcup_n [\![C_1 \,||\, C_2]\!]_{\gamma(X \leftarrow \mathsf{E}_n)}
\end{aligned}$$
- $$\begin{aligned}
&\Gamma^{C_1 \text{ or } C_2,\gamma}(\bigsqcup_n \mathsf{E}_n)\\
=&\{\text{Definition 4.3.18}\}\\
&[\![C_1 \text{ or } C_2]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{E}_n)}\\
=&\{\text{Definition 4.3.18}\}\\
&[\![C_1]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{E}_n)} \text{ or } [\![C_2]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{E}_n)}\\
=&\{\text{Definition 4.3.18}\}\\
&\bigsqcup_n \Gamma^{C_1,\gamma}(\mathsf{E}_n) \text{ or } \bigsqcup_n \Gamma^{C_2,\gamma}(\mathsf{E}_n)\\
=&\{\text{Lemma 4.3.16}\}\\
&= \bigsqcup_n (\Gamma^{C_1,\gamma}(\mathsf{E}_n) \text{ or } \Gamma^{C_2,\gamma}(\mathsf{E}_n))\\
=&\{\text{Definition 4.3.18}\}\\
&= \bigsqcup_n ([\![C_1]\!]_{\gamma(X \leftarrow \mathsf{E}_n)} \text{ or } [\![C_2]\!]_{\gamma(X \leftarrow \mathsf{E}_n)})\\
=&\{\text{Definition 4.3.18}\}\\
&\bigsqcup_n [\![C_1 \text{ or } C_2]\!]_{\gamma(X \leftarrow \mathsf{E}_n)}
\end{aligned}$$
- $$\Gamma^{\mu X.C,\gamma}(\bigsqcup_n \mathsf{E}_n)$$

$$
\begin{aligned}
&= \{\text{Definition 4.3.18}\} \\
&\quad [\![\mu X.C]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{E}_n)} \\
&= \{\text{Definition 4.3.18}\} \\
&\quad fix(\Gamma^{C,\gamma(X \leftarrow \bigsqcup_n \mathsf{E}_n)}) = fix(\mathsf{E} \mapsto \Gamma^{C,\gamma(X \leftarrow \bigsqcup_n \mathsf{E}_n)}(\mathsf{E})) \\
&= \{\text{Definition 4.3.18}\} \\
&\quad fix(\mathsf{E} \mapsto [\![C]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{E}_n, Y \leftarrow E)}) \\
&= \{\text{i.h.}\} \\
&\quad fix(\mathsf{E} \mapsto \bigsqcup_n [\![C]\!]_{\gamma(X \leftarrow \mathsf{E}_n, Y \leftarrow \mathsf{E})}) \\
&= \{\text{[3, Proposition 2.1.18]}\} \\
&\quad fix(\bigsqcup_n (\mathsf{E} \mapsto [\![C]\!]_{\gamma(X \leftarrow \mathsf{E}_n, Y \leftarrow \mathsf{E})})) \\
&= \{fix \text{ continuous}\} \\
&\quad \bigsqcup_n fix(\mathsf{E} \mapsto [\![C]\!]_{\gamma(X \leftarrow \mathsf{E}_n, Y \leftarrow \mathsf{E})}) \\
&= \{\text{Definition 4.3.18}\} \\
&\quad \bigsqcup_n fix(\mathsf{E} \mapsto \Gamma^{C,\gamma(X \leftarrow \mathsf{E}_n)}(\mathsf{E})) = \bigsqcup_n fix(\Gamma^{C,\gamma(X \leftarrow \mathsf{E}_n)}) \\
&= \{\text{Definition 4.3.18}\} \\
&\quad \bigsqcup_n [\![\mu X.C]\!]_{\gamma(X \leftarrow \mathsf{E}_n)} \\
&= \{\text{Definition 4.3.18}\} \\
&\quad \bigsqcup_n \Gamma^{\mu X.C,\gamma}(\mathsf{E}_n)
\end{aligned}
$$

□

Proof of Lemma 4.3.20

Proof.

- $[\![skip[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma$

  It follows directly that $[\![skip]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}$.

- $[\![a[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma$

  It follows directly that $[\![a]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}$.

-

$$
\begin{aligned}
&\quad [\![(C_1 \,;\, C_2)[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma \\
&= \{\text{Definition 4.3.1}\} \\
&\quad [\![C_1 \,;\, (C_2[X \leftarrow [\![\mu X.C]\!]_\gamma])]\!]_\gamma \\
&= \{\text{Definition 4.3.18}\} \\
&\quad [\![C_1]\!] \,;\, ([\![C_2]\!]_\gamma[X \leftarrow [\![\mu X.C]\!]_\gamma]) \\
&= \{\text{i.h.}\} \\
&\quad [\![C_1]\!] \,;\, [\![C_2]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}
\end{aligned}
$$

$$
\begin{array}{l}
\quad =\{\text{Definition 4.3.18}\} \\
\quad\ [\![C_1\ ;\ C_2]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}
\end{array}
$$

- $$
\begin{array}{l}
\quad\ [\![(C_1 \,||\, C_2)[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma \\
=\{\text{Definition 4.3.1}\} \\
\quad\ [\![C_1[X \leftarrow [\![\mu X.C]\!]_\gamma] \,||\, C_2[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma \\
=\{\text{Definition 4.3.18}\} \\
\quad\ [\![C_1]\!][X \leftarrow [\![\mu X.C]\!]_\gamma] \,||\, [\![C_2]\!]_\gamma[X \leftarrow [\![\mu X.C]\!]_\gamma] \\
=\{\text{i.h.}\} \\
\quad\ [\![C_1]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \,||\, [\![C_2]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \\
=\{\text{Definition 4.3.18}\} \\
\quad\ [\![C_1 \,||\, C_2]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}
\end{array}
$$

- $$
\begin{array}{l}
\quad\ [\![(C_1 \text{ or } C_2)[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma \\
=\{\text{Definition 4.3.1}\} \\
\quad\ [\![C_1[X \leftarrow [\![\mu X.C]\!]_\gamma] \text{ or } C_2[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma \\
=\{\text{Definition 4.3.18}\} \\
\quad\ [\![C_1]\!][X \leftarrow [\![\mu X.C]\!]_\gamma] \text{ or } [\![C_2]\!]_\gamma[X \leftarrow [\![\mu X.C]\!]_\gamma] \\
=\{\text{i.h.}\} \\
\quad\ [\![C_1]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \text{ or } [\![C_2]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \\
=\{\text{Definition 4.3.18}\} \\
\quad\ [\![C_1 \text{ or } C_2]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}
\end{array}
$$

- $$
\begin{array}{l}
\quad\ [\![(\mu Y.C')[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma \\
=\{\text{Definition 4.3.1}\} \\
\quad\ [\![\mu Y.(C'[X \leftarrow [\![\mu X.C]\!]_\gamma])]\!]_\gamma \\
=\{\text{Definition 4.3.18}\} \\
\quad\ fix(\Gamma^{C'[X \leftarrow [\![\mu X.C]\!]_\gamma], \gamma}) = fix(\mathsf{E} \mapsto \Gamma^{C'[X \leftarrow [\![\mu X.C]\!]_\gamma], \gamma}(\mathsf{E})) \\
=\{\text{Definition 4.3.18}\} \\
\quad\ fix(\mathsf{E} \mapsto [\![C'[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_{\gamma(Y \leftarrow \mathsf{E})}) \\
=\{\text{i.h.}\} \\
\quad\ fix(\mathsf{E} \mapsto [\![C']\!]_{\gamma(Y \leftarrow \mathsf{E}, X \leftarrow [\![\mu X.C]\!]_\gamma)}) \\
=\{\text{Definition 4.3.18}\} \\
\quad\ fix(\mathsf{E} \mapsto \Gamma^{C', \gamma(Y \leftarrow \mathsf{E}, X \leftarrow [\![\mu X.C]\!]_\gamma)}(\mathsf{E})) = fix(\Gamma^{C', \gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}) \\
=\text{Definition 4.3.18 ;} \\
\quad\ [\![\mu Y.C']\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}
\end{array}
$$

□

Proof of Lemma 4.3.21

Proof.

$$
\begin{aligned}
&[\![\mu X.C]\!]_\gamma \\
=&\text{Definition 4.3.18} \\
&fix(\Gamma^{C,\gamma}) \\
=&\{\text{Lemma 4.3.17}\} \\
&\Gamma^{C,\gamma}(fix(\Gamma^{C,\gamma})) \\
=&\{\text{Definition 4.3.18}\} \\
&\Gamma^{C,\gamma}([\![\mu X.C]\!]_\gamma) \\
=&\{\text{Definition 4.3.18}\} \\
&[\![C]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}
\end{aligned}
$$

□

Proof of Lemma 4.4.1

Proof. Induction over rules in Figure 19.

- $skip \xrightarrow{sk} \checkmark$

  It follows directly that $[\![\checkmark]\!]_\gamma \equiv [\![skip]\!]_\gamma \backslash sk \equiv \emptyset$.

- $a \xrightarrow{a} \checkmark$

  It follows directly that $[\![\checkmark]\!]_\gamma \equiv [\![a]\!]_\gamma \backslash a \equiv \emptyset$.

- $C_1 \,;\, C_2 \xrightarrow{l} C_2$

$$
\begin{aligned}
&C_1 \,;\, C_2 \xrightarrow{l} C_2 \\
\Rightarrow&\{\text{Figure 19 entails}\} \\
&C_1 \xrightarrow{l} \checkmark \\
\Rightarrow&\{\text{i.h.}\} \\
&[\![\checkmark]\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l \\
\Rightarrow&\{\text{Lemma 4.2.17}\} \\
&[\![\checkmark]\!]_\gamma \,;\, [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \backslash l) \,;\, [\![C_2]\!]_\gamma \\
\Rightarrow&\{[\![\checkmark]\!]_\gamma \,;\, [\![C_2]\!]_\gamma \equiv [\![C_2]\!]_\gamma, \text{ Lemma 4.2.22}\} \\
&[\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \,;\, [\![C_2]\!]_\gamma) \backslash l \\
\Rightarrow&\{\text{Definition 4.2.9}\} \\
&[\![C_2]\!]_\gamma \equiv [\![C_1 \,;\, C_2]\!]_\gamma \backslash l
\end{aligned}
$$

- $C_1 \,;\, C_2 \xrightarrow{l} C_1' \,;\, C_2$

$$C_1 \,;\, C_2 \xrightarrow{l} C_1' \,;\, C_2$$

$\Rightarrow$ {Figure 19 entails}
$C_1 \xrightarrow{l} C_1'$
$\Rightarrow$ {i.h.}
$[\![C_1']\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l$
$\Rightarrow$ {Lemma 4.2.17}
$[\![C_1']\!]_\gamma \,;\, [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \backslash l) \,;\, [\![C_2]\!]_\gamma$
$\Rightarrow$ {Lemma 4.2.22}
$[\![C_1']\!]_\gamma \,;\, [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \,;\, [\![C_2]\!]_\gamma) \backslash l$
$\Rightarrow$ {Definition 4.2.9}
$[\![C_1' \,;\, C_2]\!]_\gamma \equiv [\![C_1 \,;\, C_2]\!]_\gamma \backslash l$

- $C_1 \text{ or } C_2 \xrightarrow{l} \checkmark$

$C_1 \text{ or } C_2 \xrightarrow{l} \checkmark$
$\Rightarrow$ {Figure 19 entails}
$C_1 \xrightarrow{l} \checkmark \text{ or } C_2 \xrightarrow{l} \checkmark$
$\Rightarrow$ {i.h.}
$[\![\checkmark]\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l \text{ or } [\![\checkmark]\!]_\gamma \equiv [\![C_2]\!]_\gamma \backslash l$
$\Rightarrow$ {Lemma 4.2.24 for both cases, Definition 4.2.9 }
$[\![\checkmark]\!]_\gamma \equiv [\![C_1 \text{ or } C_2]\!]_\gamma \backslash l$

- $C_1 \text{ or } C_2 \xrightarrow{l} C'$

$C_1 \text{ or } C_2 \xrightarrow{l} C'$
$\Rightarrow$ {Figure 19 entails}
$C_1 \xrightarrow{l} C_1' \text{ or } C_2 \xrightarrow{l} C_2'$
$\Rightarrow$ {i.h.}
$[\![C_1']\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l \text{ or } [\![C_2']\!]_\gamma \equiv [\![C_2]\!]_\gamma \backslash l$
$\Rightarrow$ {Lemma 4.2.24 for both cases, Definition 4.2.9}
$[\![C_1']\!]_\gamma \equiv ([\![C_1 \text{ or } C_2]\!]_\gamma) \backslash l \text{ or } [\![C_2']\!]_\gamma \equiv [\![C_1 \text{ or } C_2]\!]_\gamma \backslash l$

- $C_1 \,||\, C_2 \xrightarrow{l} C_2$

$C_1 \,||\, C_2 \xrightarrow{l} C_2$
$\Rightarrow$ {Figure 19 entails}
$C_1 \xrightarrow{l} \checkmark$
$\Rightarrow$ {i.h.}
$[\![\checkmark]\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l$
$\Rightarrow$ {Lemma 4.2.17}
$[\![\checkmark]\!]_\gamma \,||\, [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \backslash l) \,||\, [\![C_2]\!]_\gamma$

$$
\begin{array}{l}
\Rightarrow\{[\![\checkmark]\!]_\gamma \,||\, [\![C_2]\!]_\gamma \equiv [\![C_2]\!]_\gamma\} \\
\quad [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \backslash l) \,||\, [\![C_2]\!]_\gamma \\
\Rightarrow\{\text{Lemma 4.2.23, Definition 4.2.9}\} \\
\quad [\![C_2]\!]_\gamma \equiv [\![C_1 \,||\, C_2]\!]_\gamma \backslash l
\end{array}
$$

- $C_1 \,||\, C_2 \xrightarrow{l} C_1' \,||\, C_2$

$$
\begin{array}{l}
\quad C_1 \,||\, C_2 \xrightarrow{l} C_1' \,||\, C_2 \\
\Rightarrow\{\text{Figure 19 entails}\} \\
\quad C_1 \xrightarrow{l} C_1' \\
\Rightarrow\{\text{i.h.}\} \\
\quad [\![C_1']\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l \\
\Rightarrow\{\text{Lemma 4.2.17}\} \\
\quad [\![C_1']\!]_\gamma \,||\, [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \backslash l) \,||\, [\![C_2]\!]_\gamma \\
\Rightarrow\{\text{Lemma 4.2.23, Definition 4.2.9}\} \\
\quad [\![C_1' \,||\, C_2]\!]_\gamma \equiv [\![C_1 \,||\, C_2]\!]_\gamma \backslash l
\end{array}
$$

- $C_1 \,||\, C_2 \xrightarrow{l} C_1$

$$
\begin{array}{l}
\quad C_1 \,||\, C_2 \xrightarrow{l} C_1 \\
\Rightarrow\{\text{Figure 19 entails}\} \\
\quad C_2 \xrightarrow{l} \checkmark \\
\Rightarrow\{\text{i.h.}\} \\
\quad [\![\checkmark]\!]_\gamma \equiv [\![C_2]\!]_\gamma \backslash l \\
\Rightarrow\{\text{Lemma 4.2.17}\} \\
\quad [\![C_1]\!]_\gamma \,||\, [\![\checkmark]\!]_\gamma \equiv [\![C_1]\!]_\gamma \,||\, ([\![C_2]\!]_\gamma \backslash l) \\
\Rightarrow\{[\![C_1]\!]_\gamma \,||\, [\![\checkmark]\!]_\gamma \equiv [\![C_1]\!]_\gamma\} \\
\quad [\![C_1]\!]_\gamma \equiv [\![C_1]\!]_\gamma \,||\, ([\![C_2]\!]_\gamma \backslash l) \\
\Rightarrow\{\text{Lemma 4.2.23, Definition 4.2.9}\} \\
\quad [\![C_1]\!]_\gamma \equiv [\![C_1 \,||\, C_2]\!]_\gamma \backslash l
\end{array}
$$

- $C_1 \,||\, C_2 \xrightarrow{l} C_1 \,||\, C_2'$

$$
\begin{array}{l}
\quad C_1 \,||\, C_2 \xrightarrow{l} C_1 \,||\, C_2' \\
\Rightarrow\{\text{Figure 19 entails}\} \\
\quad C_2 \xrightarrow{l} C_2' \\
\Rightarrow\{\text{i.h.}\} \\
\quad [\![C_2']\!]_\gamma \equiv [\![C_2]\!]_\gamma \backslash l
\end{array}
$$

$\Rightarrow$ {Lemma 4.2.17}
$[\![C_1]\!]_\gamma \,||\, [\![C_2']\!]_\gamma \equiv [\![C_1]\!]_\gamma \,||\, ([\![C_2]\!]_\gamma \backslash l)$
$\Rightarrow$ {Lemma 4.2.23, Definition 4.2.9}
$[\![C_1 \,||\, C_2']\!]_\gamma \equiv [\![C_1 \,||\, C_2]\!]_\gamma \backslash l$

- $\mu X.C$
$\Rightarrow$ {Figure 19 entails}
$C \xrightarrow{l} C'$
$\Rightarrow$ {i.h.}
$[\![C']\!]_\gamma \equiv [\![C]\!]_\gamma \backslash l$
$\Rightarrow$ {setting $\gamma = \gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)$}
$[\![C']\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \equiv [\![C]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \backslash l$
$\Rightarrow$ {Lemma 4.3.20, Lemma 4.3.21}
$[\![C'[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma \equiv [\![\mu X.C]\!]_\gamma \backslash l$

□

Proof of Theorem 4.4.2

Proof. Induction over the length of ω, which is denoted by $|\omega|$.

- $|\omega| = 1$
It follows directly that $\exists \{l\} \in \mathcal{C}([\![C]\!]_\gamma) \,.\, \emptyset \xrightarrow{l} \subset \{l\}$
- $|\omega| > 1$
$C \xrightarrow{\omega} \!\!\!\!\rightarrow C'$
$\Rightarrow$ {Definition 17}
$C \xrightarrow{l} C'' \qquad C'' \xrightarrow{\omega'} \!\!\!\!\rightarrow C'$
$\Rightarrow$ {Lemma 4.4.1, i.h.}
$[\![C'']\!]_\gamma \equiv [\![C]\!]_\gamma \backslash l \qquad \exists y \in \mathcal{C}([\![C'']\!]_\gamma) \,.\, \emptyset \xrightarrow{\omega'} \subset y$
$\Rightarrow$ {Definition 4.2.19}
$\{l\} \cup y \in \mathcal{C}([\![C]\!]_\gamma) \,.\, \emptyset \xrightarrow{l} \subset \{l\} \xrightarrow{\omega'} \subset \{l\} \cup y = x$

□

Proof of Lemma 4.4.3

Proof. Induction over the interpretation of commands.

- $sk \in \mathcal{I}([\![skip]\!]_\gamma)$
Let $C' = \checkmark$. It follows directly that $skip \xrightarrow{sk} \checkmark$ and that $[\![skip]\!]_\gamma \backslash sk \equiv [\![\checkmark]\!]_\gamma$.

- $a \in \mathcal{I}([\![a]\!]_\gamma)$

  Let $C' = \checkmark$. It follows directly that $a \xrightarrow{a} \checkmark$ and that $[\![a]\!]_\gamma \backslash a \equiv [\![\checkmark]\!]_\gamma$.

- $l \in \mathcal{I}([\![C_1 \,;\, C_2]\!]_\gamma)$

  By Definition 4.2.1 we have that $l \in \mathcal{I}([\![C_1]\!]_\gamma)$. By i.h., $\exists C'$ such that $C_1 \xrightarrow{l} C'$ and $[\![C_1]\!]_\gamma \backslash l = [\![C']\!]_\gamma$. We have two cases:

  1. $C' = \checkmark$

     We have $C_1 \xrightarrow{l} \checkmark$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![\checkmark]\!]_\gamma$. By the rules in Figure 19, $C_1 \,;\, C_2 \xrightarrow{l} C_2$. By Definition 4.2.1, $([\![C_1]\!]_\gamma \backslash l) \,;\, [\![C_2]\!]_\gamma \equiv [\![\checkmark]\!]_\gamma \,;\, [\![C_2]\!]_\gamma \equiv [\![C_2]\!]_\gamma$.

  2. $C' = C_1'$

     We have $C_1 \xrightarrow{l} C_1'$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![C_1']\!]_\gamma$. By the rules in Figure 19, $C_1 \,;\, C_2 \xrightarrow{l} C_1' \,;\, C_2$. By Definition 4.2.1, $([\![C_1]\!]_\gamma \backslash l) \,;\, [\![C_2]\!]_\gamma \equiv [\![C_1']\!]_\gamma \,;\, [\![C_2]\!]_\gamma$. By Definition 4.2.9, $[\![C_1' \,;\, C_2]\!]_\gamma$.

- $l \in \mathcal{I}([\![C_1 \text{ or } C_2]\!]_\gamma)$

  By Definition 4.2.6 we have two cases:

  1. $l \in \mathcal{I}([\![C_1]\!]_\gamma)$

     By i.h. $\exists C' \,.\, C_1 \xrightarrow{l} C'$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma$. By the rules in Figure 19 we have two cases:

     (a) $C' = \checkmark$

     We have $C_1 \xrightarrow{l} \checkmark$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![\checkmark]\!]_\gamma$. By the rules in Figure 19 we have $C_1 \text{ or } C_2 \xrightarrow{l} \checkmark$. By Lemma 4.2.24 we have $[\![\checkmark]\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l \equiv ([\![C_1]\!]_\gamma \text{ or } [\![C_2]\!]_\gamma) \backslash l$. By Definition 4.2.9, $[\![C_1 \text{ or } C_2]\!]_\gamma \backslash l$.

     (b) $C' = C_1'$

     We have $C_1 \xrightarrow{l} C_1'$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![C_1']\!]_\gamma$. By the rules in Figure 19 we have $C_1 \text{ or } C_2 \xrightarrow{l} C_1'$. By Lemma 4.2.24 we have $[\![C_1']\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l \equiv ([\![C_1]\!]_\gamma \text{ or } [\![C_2]\!]_\gamma) \backslash l$. By Definition 4.2.9, $[\![C_1 \text{ or } C_2]\!]_\gamma \backslash l$.

  2. $l \in \mathcal{I}([\![C_2]\!]_\gamma)$

     By i.h. $\exists C' \,.\, C_2 \xrightarrow{l} C'$ and $[\![C_2]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma$. By the rules in Figure 19 we have two cases:

     (a) $C' = \checkmark$

     We have $C_2 \xrightarrow{l} \checkmark$ and $[\![C_2]\!]_\gamma \backslash l \equiv [\![\checkmark]\!]_\gamma$. By the rules in Figure 19 we have $C_1 \text{ or } C_2 \xrightarrow{l} \checkmark$. By Lemma 4.2.24 we have $[\![\checkmark]\!]_\gamma \equiv [\![C_2]\!]_\gamma \backslash l \equiv ([\![C_1]\!]_\gamma \text{ or } [\![C_2]\!]_\gamma) \backslash l$. By Definition 4.2.9, $[\![C_1 \text{ or } C_2]\!]_\gamma \backslash l$.

     (b) $C' = C_2'$

     We have $C_2 \xrightarrow{l} C_2'$ and $[\![C_2]\!]_\gamma \backslash l \equiv [\![C_2']\!]_\gamma$. By the rules in Figure 19 we have $C_1 \text{ or } C_2 \xrightarrow{l} C_2'$. By Lemma 4.2.24 we have $[\![C_2']\!]_\gamma \equiv [\![C_2]\!]_\gamma \backslash l \equiv ([\![C_1]\!]_\gamma \text{ or } [\![C_2]\!]_\gamma) \backslash l$. By Definition 4.2.9, $[\![C_1 \text{ or } C_2]\!]_\gamma \backslash l$.

- $l \in \mathcal{I}([\![C_1 \,||\, C_2]\!]_\gamma)$

  By Definition 4.2.3 we have two cases:

1. $l \in \mathcal{I}([\![C_1]\!]_\gamma)$

   By i.h. $\exists C' \,.\, C_1 \xrightarrow{l} C'$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma$. By the rules in Figure 19 we have two cases:

   (a) $C' = \checkmark$

   We have $C_1 \xrightarrow{l} \checkmark$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![\checkmark]\!]_\gamma$. By the rules in Figure 19 we have $C_1 \,||\, C_2 \xrightarrow{l} C_2$. By Definition 4.2.3 we have $([\![C_1]\!]_\gamma \backslash l) \,||\, [\![C_2]\!]_\gamma$. By Lemma 4.2.23 we have $([\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma) \backslash l$. By Definition 4.2.9, $[\![C_1 \,||\, C_2]\!]_\gamma \backslash l$.

   (b) $C' = C_1'$

   We have $C_1 \xrightarrow{l} C_1'$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![C_1']\!]_\gamma$. By the rules in Figure 19 we have $C_1 \,||\, C_2 \xrightarrow{l} C_1' \,||\, C_2$. By Definition 4.2.3, $([\![C_1]\!]_\gamma \backslash l) \,||\, [\![C_2]\!]_\gamma$. By Lemma 4.2.23 we have $([\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma) \backslash l$. By Definition 4.2.9, $[\![C_1 \,||\, C_2]\!]_\gamma \backslash l$.

2. $l \in \mathcal{I}([\![C_2]\!]_\gamma)$

   By i.h. $\exists C' \,.\, C_2 \xrightarrow{l} C'$ and $[\![C_2]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma$. By the rules in Figure 19 we have two cases:

   (a) $C' = \checkmark$

   We have $C_2 \xrightarrow{l} \checkmark$ and $[\![C_2]\!]_\gamma \backslash l \equiv [\![\checkmark]\!]_\gamma$. By the rules in Figure 19 we have $C_1 \,||\, C_2 \xrightarrow{l} C_1$. By Definition 4.2.3 we have $[\![C_1]\!]_\gamma \,||\, ([\![C_2]\!]_\gamma \backslash l)$. By Lemma 4.2.23 we have $([\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma) \backslash l$. By Definition 4.2.9, $[\![C_1 \,||\, C_2]\!]_\gamma \backslash l$.

   (b) $C' = C_2'$

   We have $C_2 \xrightarrow{l} C_2'$ and $[\![C_2]\!]_\gamma \backslash l \equiv [\![C_2']\!]_\gamma$. By the rules in Figure 19 we have $C_1 \,||\, C_2 \xrightarrow{l} C_1 \,||\, C_2'$. By Definition 4.2.3, $[\![C_1]\!]_\gamma \,||\, ([\![C_2]\!]_\gamma \backslash l)$. By Lemma 4.2.23 we have $([\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma) \backslash l$. By Definition 4.2.9, $[\![C_1 \,||\, C_2]\!]_\gamma \backslash l$.

- $l \in \mathcal{I}([\![\mu X.C]\!]_\gamma)$

  By Definition 4.3.18 and Definition 4.3.6, $l \in \mathcal{I}([\![C]\!]_\gamma)$. By i.h., $\exists C'$ such that $C \xrightarrow{l} C'$ and $[\![C']\!]_\gamma \equiv [\![C]\!]_\gamma \backslash l$. By the rules in Figure 19 and by setting $\gamma = \gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)$, $\mu X.C \xrightarrow{l} C'[X \leftarrow \mu X.C]$ and $[\![C']\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \equiv [\![C]\!]_{\gamma(X \leftarrow \mu X.C)} \backslash l$

□

Proof of Theorem 4.4.4

Proof. Induction over the length of ω.

- $|\omega| = 1$

  We have $\{l\} \in \mathcal{C}([\![C]\!]_\gamma)$ such that $\emptyset \xrightarrow{l}\subset \{l\}$. Furthermore $l \in \mathcal{I}([\![C]\!])$. By Lemma 4.4.3, $C \xrightarrow{l} C'$ and $[\![C']\!]_\gamma \equiv [\![C]\!]_\gamma \backslash l$. By the rules in Figure 17, $C \xrightarrow{l}\!\!\!\!\rightarrow C'$.

- $|\omega| > 1$

  We have $x \in \mathcal{C}([\![C]\!]_\gamma)$ such that $\emptyset \xrightarrow{\omega}\subset x$. Since $\omega = l_0 l_1 \ldots l_n$, then $\emptyset \xrightarrow{l_0}\subset \{l_0\} \xrightarrow{\omega'}\subset x$. Hence $l_0 \in \mathcal{I}([\![C]\!]_\gamma)$. By Lemma 4.4.3, $C \xrightarrow{l_0} C'$ and $[\![C']\!]_\gamma \equiv [\![C]\!]_\gamma \backslash l_0$. By Definition 4.2.19, $\exists y \in \mathcal{C}([\![C']\!]_\gamma)$ such that $\emptyset \xrightarrow{\omega'}\subset y$. By i.h. $\exists C''$ such that $C' \xrightarrow{\omega'}\!\!\!\!\rightarrow C''$. By the rules in Figure 17, $C \xrightarrow{\omega}\!\!\!\!\rightarrow C''$, where $\omega = l_0 : \omega'$.

□

A.2 Proofs of Chapter 5

Remark 6. Here we may found useful to write the sum in Equation 3.1 in another way. Concretely:

$$v(y) - \sum_{\emptyset \neq I \subseteq \{1,\dots,n\}} (-1)^{|I|+1} v\left(\bigcup_{i \in I} x_i\right) = \sum_{I \subseteq \{1,\dots,n\}} (-1)^{|I|} v\left(y \cup \bigcup_{i \in I} x_i\right)$$

Proof of Lemma 5.2.2

Proof. We begin by showing that $v(x)$ is well-defined, *i.e.* that given $x \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$ such that $x = x_1 \cup (x_2 \times \{x_1\})$ and $x = x_1' \cup (x_2' \times \{x_1'\})$ then $x_1 = x_1'$ and $x_2 = x_2'$.

Let $x \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$, $x_1, x_1' \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}_1)$, $x_2, x_2' \in \mathcal{C}(\mathsf{P}_2)$, and $x = x_1 \cup (x_2 \times \{x_1\}) = x_1' \cup (x_2' \times \{x_1'\})$.

We start by showing that $x_1 = x_1'$. Since $x_1, x_1' \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}_1)$, by Lemma 3.2.8, distinct maximal configurations of P_1 have events in conflict:

$$x_1 \neq x_1' \Rightarrow \exists e1 \in x_1, e_1' \in x_1' \text{ such that } e_1 \#_1 e_1'$$

By contradiction, assume that $x_1 \neq x_1'$. Since maximal configurations have events in conflict, then there exist e_1, e_1' such that $e_1 \#_1 e_1'$. Furthermore, we have that $x_1, x_1' \subseteq x$. Hence $e_1, e_1' \in x$. However, by Definition 5.2.1, the conflict relation # restricts to $\#_1$ on events from E_1. Thus $e_1 \# e_1'$. Since x contains events in conflict, it contradicts that x is a configuration (configurations are conflict-free by definition). Therefore, $x_1 = x_1'$.

We now show that $x_2 = x_2'$. We already know that $x_1 = x_1'$. Hence we can rewrite $x = x_1 \cup (x_2 \times \{x_1\}) = x_1 \cup (x_2' \times \{x_1\})$. Now, we only need to show that $x_2 \times \{x_1\} = x_2' \times \{x_1\}$. For that, consider the map $f_{x_1} : \mathcal{C}(\mathsf{P}_2) \to \mathsf{P}(\mathsf{E}_2 \times \{x_1\})$, $x_2 \mapsto x_2 \times \{x_1\}$. Such map is injective (since we pair each configuration x_2 with the maximal configuration x_1). Hence, it follows from injectivity that $x_2 = x_2'$.

Let

$$\mathsf{P}_1 = (\mathsf{E}_1,\, v_1)$$
$$\mathsf{P}_2 = (\mathsf{E}_2,\, v_2)$$
$$\mathsf{P}_1 \,;\, \mathsf{P}_2 = (\mathsf{E},\, v)$$

By Lemma 4.2.2 we know that E is an event structure. Hence we focus solely on the valuation part.

1. $v(\emptyset) = 1$

   Since $\emptyset \in \mathcal{C}(\mathsf{P}_1)$ then $v(\emptyset) = v_1(\emptyset) = 1$.

2. $d_v^{(n)}[y \,;\, x_1, \dots, x_n] \geq 0$ for all $n \geq 1$ and $y, x_1, \dots, x_n \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$ with $y \subseteq x_1, \dots, x_n$

   We have two cases based on n:

   (a) $n = 0$

   We have $d_v^{(0)}[y \,;\,] = v(y)$. By Definition 5.2.1 we have two cases:

   i. $y \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$ such that $y \in \mathcal{C}(\mathsf{P}_1)$
   It follows directly that $v(y) = v_1(y) \geq 0$.

   ii. $y \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$ such that $y = y_1 \cup (y_2 \times \{y_1\})$ with $y_1 \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}_1), y_2 \in \mathcal{C}(\mathsf{P}_2)$.
   It follows directly that $v(y) = v_1(y_1) \cdot v_2(y_2) \geq 0$, since v_1, v_2 are valuations.

(b) $n > 0$

By [119, Proposition 5] we only need to check the condition for $y \multimap\!\!\subset x_1, \ldots, x_n$. We have three cases:

i. $y \in \mathcal{C}(\mathsf{P}_1)$ but $y \notin \mathcal{C}_{\mathsf{max}}(\mathsf{P}_1)$
By [119, Proposition 5] we know that $x_1, \ldots, x_n \in \mathcal{C}(\mathsf{P}_1)$, since $\multimap\!\!\subset$ is a "single-step" relation. We have $y, x_1, \ldots, x_n \in \mathcal{C}(\mathsf{P}_1)$. It follows directly that $d_v^{(n)}[y\,;\, x_1, \ldots, x_n] = d_{v_1}^{(n)}[y\,;\, x_1, \ldots, x_n] \geq 0$.

ii. $y \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}_1)$
By [119, Proposition 5] we know that $x_1, \ldots, x_n \in \mathcal{C}(\mathsf{P}_1\,;\,\mathsf{P}_2)$ such that $x_1, \ldots, x_n \notin \mathcal{C}(\mathsf{P}_1)$, since $\multimap\!\!\subset$ is a "single-step" relation. Hence exists $x'_1, \ldots, x'_n \in \mathcal{C}(\mathsf{P}_2)$ such that $x_1 = y \cup (x'_1 \times \{y\}), \ldots, x_n = y \cup (x'_n \times \{y\})$. Furthermore $\bigcup_{i\in I} x_i = y \cup \bigcup_{i\in I}(x'_i \times \{y\})$ and let $I \subseteq \{1, \ldots, n\}$.

$$
\begin{aligned}
d_v^{(n)}[y\,;\, x_1, \ldots, x_n] &= \sum_I (-1)^{|I|} v\left(y \cup \bigcup_{i\in I} x_i\right) \\
&= \sum_I (-1)^{|I|} v\left(y \cup \bigcup_{i\in I} (y \cup (x'_i \times \{y\}))\right) \\
&= \sum_I (-1)^{|I|} v\left(y \cup \bigcup_{i\in I} (x'_i \times \{y\})\right) \\
&= \sum_I (-1)^{|I|} v_1(y) \cdot v_2\left(\bigcup_{i\in I} x'_i\right) \\
&= v_1(y) \cdot \sum_I (-1)^{|I|} v_2\left(\bigcup_{i\in I} x'_i\right) \\
&= v_1(y) \cdot d_{v_2}^{(n)}[\emptyset\,;\, x'_1, \ldots, x'_n]
\end{aligned}
$$

Since $v_1(y) \geq 0$ and $d_{v_2}^{(n)}[\emptyset\,;\, x'_1, \ldots, x'_n] \geq 0$, then $v_1(y) \cdot d_{v_2}^{(n)}[\emptyset\,;\, x'_1, \ldots, x'_n] \geq 0$.

iii. $y \in \mathcal{C}(\mathsf{P}_1\,;\,\mathsf{P}_2)$ but $y \notin \mathcal{C}(\mathsf{P}_1)$.
By [119, Proposition 5] we know that $\exists y_1 \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}_1)$ and $y', x'_1, \ldots x'_n \in \mathcal{C}(\mathsf{P}_2)$ such that $y = y_1 \cup (y' \times \{y_1\}), x_1 = y_1 \cup (x'_1 \times \{y_1\}), \ldots, x_n = y_1 \cup (x'_n \times \{y_1\})$, since $\multimap\!\!\subset$ is a "single-step" relation. Furthermore $\bigcup_{i\in I} x_i = \bigcup_{i\in I}(y_1 \cup (x'_i \times \{y_1\})) = y_1 \cup \bigcup_{i\in I}(x'_i \times \{y_1\})$ and let $I \subseteq \{1, \ldots, n\}$.

$$
\begin{aligned}
d_v^{(n)}[y\,;\, x_1, \ldots, x_n] &= \sum_I (-1)^{|I|} v\left(y \cup \bigcup_{i\in I} x_i\right) \\
&= \sum_I (-1)^{|I|} v\left((y_1 \cup (y' \times \{y_1\})) \cup \bigcup_{i\in I} (y_1 \cup (x'_i \times \{y_1\}))\right) \\
&= \sum_I (-1)^{|I|} v\left(y_1 \cup \left((y' \times \{y_1\}) \cup \bigcup_{i\in I} (x'_i \times \{y_1\})\right)\right) \\
&= \sum_I (-1)^{|I|} v_1(y_1) \cdot v_2\left(y' \cup \bigcup_{i\in I} x'_i\right)
\end{aligned}
$$

$$= v_1(y) \cdot \sum_I (-1)^{|I|} v_2 \left(y' \cup \bigcup_{i \in I} x'_i \right)$$
$$= v_1(y) \cdot d_{v_2}^{(n)}[y' \,;\, x'_1, \ldots, x'_n]$$

Since $v_1(y) \geq 0$ and $d_{v_2}^{(n)}[y' \,;\, x'_1, \ldots, x'_n] \geq 0$, then $v_1(y) \cdot d_{v_2}^{(n)}[y' \,;\, x'_1, \ldots, x'_n] \geq 0$.

□

Proof of Lemma 5.2.4

Proof. Let

$$\mathsf{P}_1 = (\mathsf{E}_1,\, v_1)$$
$$\mathsf{P}_2 = (\mathsf{E}_2,\, v_2)$$
$$\mathsf{P}_1 \,||\, \mathsf{P}_2 = (\mathsf{E},\, v)$$

By Lemma 4.2.4 we know that E is an event structure. Hence we focus solely on the valuation part.

1. $v(\emptyset) = 1$

$$v(\emptyset) = v_1(\emptyset \cap E_1) \cdot v_2(\emptyset \cap E_2) = v_1(\emptyset) \cdot v_2(\emptyset) = 1 \cdot 1 = 1$$

2. $d_v^{(n)}[y \,;\, x_1, \ldots, x_n] \geq 0$ for all $n \geq 1$ and $y, x_1, \ldots, x_n \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2)$ with $y \subseteq x_1, \ldots, x_n$

   By [119, Proposition 5] we only need to check the condition for $y \mathrel{—\!\subset} x_1, \ldots, x_n$, *i.e.* $y \overset{e_1,\ldots,e_n}{\longrightarrow\!\!\!\!\!\subset} x_1, \ldots, x_n$. We want to show that $d_v^{(n)}[y \,;\, x_1, \ldots, x_n] = d_{v_1}^{(n_1)}[y \cap E_1 \,;\, x_1 \cap E_1, \ldots, x_n \cap E_1] \cdot d_{v_2}^{(n_2)}[y \cap E_2 \,;\, x_1 \cap E_2, \ldots, x_n \cap E_2]$

   Let $I_1 \subseteq \{1, \ldots, n_1\}$, $I_2 \subseteq \{1, \ldots, n_2\}$, and $I = I_1 \uplus I_2$.

$$d_{v_1}^{(n_1)}[y \cap E_1 \,;\, x_1 \cap E_1, \ldots, x_n \cap E_1] \cdot d_{v_2}^{(n_2)}[y \cap E_2 \,;\, x_1 \cap E_2, \ldots, x_n \cap E_2]$$
$$= \sum_{I_1} (-1)^{|I_1|} v_1 \left((y \cap E_1) \cup \left(\bigcup_{i \in I_1} x_i \cap E_1 \right) \right) \cdot \sum_{I_2} (-1)^{|I_2|} v_2 \left((y \cap E_2) \cup \left(\bigcup_{j \in I_2} x_j \cap E_2 \right) \right)$$
$$= \sum_{I_1} (-1)^{|I_1|} v_1 \left(\left(y \cup \bigcup_{i \in I_1} x_i \right) \cap E_1 \right) \cdot \sum_{I_2} (-1)^{|I_2|} v_2 \left(\left(y \cup \bigcup_{j \in I_2} x_j \right) \cap E_2 \right)$$
$$= \sum_{I_1} \sum_{I_2} (-1)^{|I_1|+|I_2|} v_1 \left(\left(y \cup \bigcup_{i \in I_1} x_i \right) \cap E_1 \right) v_2 \left(\left(y \cup \bigcup_{j \in I_2} x_j \right) \cap E_2 \right)$$
$$= \sum_{I_1, I_2} (-1)^{|I_1|+|I_2|} v_1 \left(\left(y \cup \bigcup_{i \in I_1} x_i \right) \cap E_1 \right) v_2 \left(\left(y \cup \bigcup_{j \in I_2} x_j \right) \cap E_2 \right)$$
$$= \sum_{I_1, I_2} (-1)^{|I_1|+|I_2|} v \left(y \cup \bigcup_{i \in (I_1 \uplus I_2)} x_i \right)$$
$$= \sum_{I} (-1)^{|I|} v \left(y \cup \bigcup_{i \in I} x_i \right)$$
$$= d_v^{(n)}[y \,;\, x_1, \ldots, x_n] \geq 0$$

□

Proof of Lemma 5.2.6

Proof. Let

$$\begin{aligned}\mathsf{P}_1 &= (E_1, \leq_1, \#_1, v_1)\\ \mathsf{P}_2 &= (E_2, \leq_2, \#_2, v_2)\\ \mathsf{P}_1 +_p \mathsf{P}_2 &= (E, \leq, \#, v)\end{aligned}$$

We need to show that $\leq$ is a partial order and that # is symmetric and irreflexive.

- $\leq$ is a partial order:
  - Reflexivity ($e \leq e$): we have the following cases:
    1. Case $e = \tau$. Then we are done since $\tau \leq \tau$.
    2. Case $e_1 \leq_1 e_1'$ or $e_2 \leq_2 e_2'$. Since $\leq_1$ and $\leq_2$ are partial orders, we are done.
  - Transitivity ($e \leq e'$ and $e' \leq e''$ then $e \leq e''$): we have the following cases:
    1. Case $e = \tau$ and ($e', e'' \in E_1$ or $e', e'' \in E_2$). Then $\tau \leq e'$ and $e' \leq e''$. Since $e'' \in E$ the $\tau \leq e''$.
    2. Case ($e \leq_1 e'$ and $e' \leq_1 e''$) or ($e \leq_2 e'$ and $e' \leq_2 e''$). Since $\leq_1$ and $\leq_2$ are partial orders, we have $e \leq_1 e''$ or $e \leq_2 e''$. Hence $e \leq e''$.
  - Antisymmetry ($e \leq e'$ and $e' \leq e$ then $e' = e$): we have the following cases:
    1. Case ($e \leq_1 e'$ and $e' \leq_1 e$) or ($e \leq_2 e'$ and $e' \leq_2 e$). Since $\leq_1$ and $\leq_2$ are partial orders, we have $e = e'$.
    2. Case $e = \tau$ and ($e' \in E_1$ or $e' \in E_2$). We are done since $e' \leq \tau$ is not possible.

  Hence $\leq$ is a partial order.

- # is symmetric and irreflexive:
  - Symmetric (if $e\#e'$ then $e'\#e$): we have the following cases
    1. if $e\#_1 e'$ or $e\#_2 e'$. Since $\#_1$ and $\#_2$ are symmetric then we have $e'\#_1 e$ or $e'\#_2 e$. Thus $e'\#e$.
    2. if $e \in E_1$ and $e' \in E_2$, or vice-versa, then by Definition 5.2.5 we also have $e' \leq e$.
  - Irreflexive ($\neg(e\#e)$): We have either $\neg(e\#_1 e)$ or $\neg(e\#_2 e)$. Since $\#_1$ and $\#_2$ are irreflexive then $\neg(e\#e)$. There are no more cases since either $e \in E_1$ or $e \in E_2$.

  Hence # is symmetric and irreflexive.

Let $e, e', e'' \in E$. We have four conditions to check:

1. $\{e' \mid e' \leq e\}$ is finite

   We have three cases:

   (a) $e = \tau$

   It follows directly that $\{e' \mid e' \leq \tau\} = \{\tau\}$ since $\tau \in \mathcal{I}(\mathsf{P}_1 +_p \mathsf{P}_2)$.

   (b) $e \in E_1$

   We have that $\{e' \mid e' \leq e\} = \{\tau\} \cup \{e' \mid e' \leq_1 e\}$. Since P_1 is a probabilistic event structure, then we know that $\{e' \mid e' \leq_1 e\}$ is finite. Hence $\{\tau\} \cup \{e' \mid e' \leq_1 e\}$ is finite.

(c) $e \in E_2$

We have that $\{e' \mid e' \leq e\} = \{\tau\} \cup \{e' \mid e' \leq_2 e\}$. Since P_2 is a probabilistic event structure, then we know that $\{e' \mid e' \leq_2 e\}$ is finite. Hence $\{\tau\} \cup \{e' \mid e' \leq_2 e\}$ is finite.

2. $e\#e' \leq e'' \Rightarrow e\#e''$

Since τ is not in conflict with any event, this condition trivially holds because we either have $e, e', e'' \in E_1$ or $e, e', e'' \in E_2$ and $\mathsf{P}_1, \mathsf{P}_2$ are probabilistic event structures.

3. $v(\emptyset) = 1$

It follows directly from the definition.

4. $d_v^{(n)}[y\,;\, x_1, \ldots, x_n] \geq 0$ for all $n \geq 1$ and $y, x_1, \ldots, x_n \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2)$ with $y \subseteq x_1, \ldots, x_n$

By [119, Proposition 5] we only need to check the condition for $y \multimap\!\subset x_1, \ldots, x_n$, *i.e.* $y \xrightarrow{e_1,\ldots,e_n}\!\!\subset x_1, \ldots, x_n$. We have three cases:

(a) $y = \emptyset$

We then have $\emptyset \xrightarrow{\tau}\!\!\subset \{\tau\}$.

It follows that $d_v^{(1)}[\emptyset\,;\, \{\tau\}] = v(\emptyset) - v(\{\tau\}) = 1 - 1 = 0$

(b) $y\backslash\tau \in \mathcal{C}(\mathsf{P}_1)$ We have three cases (let $1 \leq i \leq n$):

i. $\forall e_i \in E_1$

We have $x_1\backslash\tau, \ldots, x_n\backslash\tau \in \mathcal{C}(\mathsf{P}_1)$. Let $I \subseteq \{1, \ldots, n\}$.

$$\begin{aligned} d_v^{(n)}[y\,;\, x_1, \ldots, x_n] &= \sum_I (-1)^{|I|} v(y \cup \bigcup_{i \in I} x_i) \\ &= p \cdot \sum_I (-1)^{|I|} v_1((y\backslash\tau) \bigcup_{i \in I} (x_i\backslash\tau)) \\ &= p \cdot d_{v_1}^{(n)}[y\,;\, x_1, \ldots, x_n] \end{aligned}$$

Since $p \in]0, 1[$ and $d_{v_1}^{(n)}[y\,;\, x_1, \ldots, x_n]$, because P_1 is a probabilistic event structure, then $p \cdot d_{v_1}^{(n)}[y\,;\, x_1, \ldots, x_n] \geq 0$.

ii. $\forall e_i \in E_2$

Since $y\backslash\tau \in \mathcal{C}(\mathsf{P}_1)$, $x_i = \{e_i\} \cup y$, and for all $1 \leq i \leq n$ we have $e_i\#e \in y\backslash\tau \in \mathcal{C}(\mathsf{P}_1)$, then $v(x_i) = 0$. Hence

$$\begin{aligned} d_v^{(n)}[y\,;\, x_1, \ldots, x_n] &= \sum_I (-1)^{|I|} v(y \cup \bigcup_{i \in I} x_i) \\ &= v(y) \\ &= p \cdot v_1(y\backslash\tau) \end{aligned}$$

Since $p \in]0, 1[$ and $v_1(y\backslash\tau) \geq 0$, because P_1 is a probabilistic event structure, we have $p \cdot v_1(y\backslash\tau) \geq 0$.

iii. $\exists e_i \in E_2$

Since $y\backslash\tau \in \mathcal{C}(\mathsf{P}_1)$, $x_i = \{e_i\} \cup y$, and for all $1 \leq i \leq n$ such that $e_i\#e \in y\backslash\tau \in \mathcal{C}(\mathsf{P}_1)$, we have $v(x_i) = 0$. Hence

$$d_v^{(n)}[y\,;\, x_1, \ldots, x_n] = \sum_I (-1)^{|I|} v(y \cup \bigcup_{i \in I} x_i)$$

$$
\begin{aligned}
&= \sum_{I'} (-1)^{|I'|} v(y \cup \bigcup_{i \in I'} x_i) \\
&= p \cdot \sum_{I'} (-1)^{|I'|} v_1((y \backslash \tau) \bigcup_{i \in I'} (x_i \backslash \tau)) \\
&= p \cdot d_{v_1}^{(m)}[y\,;\, x_1, \ldots, x_m]
\end{aligned}
$$

where $I' = \{1, \ldots, m\}$, *i.e.* I' is I without those $e_i \in E_2$.
Since $p \in]0, 1[$ and $d_{v_1}^{(m)}[y\,;\, x_1, \ldots, x_m]$, because P_1 is a probabilistic event structure, then $p \cdot d_{v_1}^{(m)}[y\,;\, x_1, \ldots, x_m] \geq 0$.

(c) $y \backslash \tau \in \mathcal{C}(\mathsf{P}_2)$
Similar to previous case.

□

Proof of Lemma 5.2.12

Proof. It follows directly from Definition 5.2.3. □

Proof of Lemma 5.2.13

Proof. Let

$$
\begin{aligned}
\mathsf{P}_1 &= (\mathsf{E}_1,\, v_1) \\
\mathsf{P}'_1 &= (\mathsf{E}'_1,\, v'_1) \\
\mathsf{P}_2 &= (\mathsf{E}_2,\, v_2) \\
\mathsf{P}'_2 &= (\mathsf{E}'_2,\, v'_2) \\
\mathsf{P}_1 \,;\, \mathsf{P}_2 &= (\mathsf{E},\, v) \\
\mathsf{P}'_1 \,;\, \mathsf{P}'_2 &= (\mathsf{E}',\, v')
\end{aligned}
$$

Due to Lemma 4.2.14, we only need to show $\forall x \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2), y \in \mathcal{C}(\mathsf{P}'_1 \,;\, \mathsf{P}'_2)\,.\, f[x] \subseteq y \Rightarrow v(x) \geq v'(y)$.

Let $x \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$ and $y \in \mathcal{C}(\mathsf{P}'_1 \,;\, \mathsf{P}'_2)$ such that $f[x] \subseteq y$. We have three cases:

1. $x \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$ such that $x \in \mathcal{C}(\mathsf{P}_1)$ and $y \in \mathcal{C}(\mathsf{P}'_1 \,;\, \mathsf{P}'_2)$ such that $y \in \mathcal{C}(\mathsf{P}'_1)$

   It follows directly that $v(x) \geq v'(y) \Leftrightarrow v_1(x) \geq v'_1(y)$, since $v(x) = v'_1(x)$, $v'(y) = v'_1(y)$ and $\mathsf{P}_1 \sqsubseteq \mathsf{P}'_1$.

2. $x \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$ such that $x \in \mathcal{C}(\mathsf{P}_1)$ and $y \in \mathcal{C}(\mathsf{P}'_1 \,;\, \mathsf{P}'_2)$ such that $\exists y_1 \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}'_1), y_2 \in \mathcal{C}(\mathsf{P}'_2)$ such that $y = y_1 \cup (y_2 \times \{y_1\})$.

   We know that $v(x) = v_1(x)$ and that $v'(y) = v'_1(y_1) \cdot v'_2(y_2)$. Since $\mathsf{P}_1 \sqsubseteq \mathsf{P}'_1$ then $f_1[x] \subseteq y_1$ and $v_1(x) \geq v'_1(y_1)$. It then follows directly that $v(x) = v_1(x) \geq v'_1(y_1) \geq v'_1(y_1) \cdot v'_2(y_2) = v'(y)$.

3. $x \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$ such that $\exists x_1 \in \mathcal{C}(\mathsf{P}_1), x_2 \in \mathcal{C}(\mathsf{P}_2)$ such that $x = x_1 \cup (x_2 \times \{x_1\})$ and $y \in \mathcal{C}(\mathsf{P}'_1 \,;\, \mathsf{P}'_2)$ such that $\exists y_1 \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}'_1), y_2 \in \mathcal{C}(\mathsf{P}'_2)$ such that $y = y_1 \cup (y_2 \times \{y_1\})$

   We know that $v(x) = v_1(x_1) \cdot v_2(x_2)$ and $v'(y) = v'_1(y_1) \cdot v'_2(y_2)$. Since $\mathsf{P}_1 \sqsubseteq \mathsf{P}'_1$ then $f_1[x_1] \subseteq y_1$ and $v_1(x_1) \geq v'_1(y_1)$, and $\mathsf{P}_2 \sqsubseteq \mathsf{P}'_2$ then $f_2[x_2] \subseteq y_2$ and $v_2(x_2) \geq v'_2(y_2)$.

Furthermore,

$$\begin{aligned} v_1(x_1) \geq v_1'(y_1) &\Leftrightarrow v_1'(y_1) \leq v_1(x_1) \\ &\Leftrightarrow \frac{v_1'(y_1)}{v_1(x_1)} \leq 1 \\ &\Leftrightarrow \left(\frac{v_1'(y_1)}{v_1(x_1)} = 1\right) \text{ or } \left(\frac{v_1'(y_1)}{v_1(x_1)} < 1\right) \end{aligned}$$

Now we show that $v(x) \geq v'(y)$.

$$\begin{aligned} v(x) \geq v'(y) &\Leftrightarrow v_1(x_1) \cdot v_2(x_2) \geq v_1'(y_1) \cdot v_2'(y_2) \\ &\Leftrightarrow v_2(x_2) \geq \frac{v_1'(y_1)}{v_1(x_1)} \cdot v_2'(y_2) \end{aligned}$$

We have two cases:

1. $\frac{v_1'(y_1)}{v_1(x_1)} = 1$

   $v_2(x_2) \geq \frac{v_1'(y_1)}{v_1(x_1)} \cdot v_2'(y_2) \Leftrightarrow v_2(x_2) \geq v_2'(y_2)$ and we are done.

2. $\frac{v_1'(y_1)}{v_1(x_1)} < 1$

   Since $v_2(x_2) \geq v_2'(y_2)$ and $v_2'(y_2) \geq \frac{v_1'(y_1)}{v_1(x_1)} \cdot v_2'(y_2)$ it follows that $v_2(x_2) \geq v_2'(y_2) \geq \frac{v_1'(y_1)}{v_1(x_1)} \cdot v_2'(y_2)$

□

Proof of Lemma 5.2.14

Proof. Let

$$\begin{aligned} \mathsf{P}_1 &= (\mathsf{E}_1, v_1) \\ \mathsf{P}_1' &= (\mathsf{E}_1', v_1') \\ \mathsf{P}_2 &= (\mathsf{E}_2, v_2) \\ \mathsf{P}_2' &= (\mathsf{E}_2', v_2') \\ \mathsf{P}_1 \,||\, \mathsf{P}_2 &= (\mathsf{E}, v) \\ \mathsf{P}_1' \,||\, \mathsf{P}_2' &= (\mathsf{E}', v') \end{aligned}$$

Due to Lemma 4.2.15, we only need to show $\forall x \in \mathcal{C}(\mathsf{P}_1 \,||\, \mathsf{P}_2), y \in \mathcal{C}(\mathsf{P}_1' \,||\, \mathsf{P}_2') \,.\, f[x] \subseteq y \Rightarrow v(x) \geq v'(y)$.

Let $x \in \mathcal{C}(\mathsf{P}_1 \,||\, \mathsf{P}_2)$ and $y \in \mathcal{C}(\mathsf{P}_1' \,||\, \mathsf{P}_2')$ such that $f[x] \subseteq y$. Since $\mathsf{P}_1 \sqsubseteq \mathsf{P}_1'$ then $\forall x_1 \in \mathcal{C}(\mathsf{P}_1), y_1 \in \mathcal{C}(\mathsf{P}_1')$ such that $f_1[x_1] \subseteq y_1$ we have $v_1(x_1) \geq v_1'(y_1)$ and that $\mathsf{P}_2 \sqsubseteq \mathsf{P}_2'$ entails $\forall x_2 \in \mathcal{C}(\mathsf{P}_2), y_2 \in \mathcal{C}(\mathsf{P}_2')$ such that $f_2[x_2] \subseteq y_2$ we have $v_2(x_2) \geq v_2'(y_2)$. By Definition 5.2.3, $v(x) = v_1(x_1) \cdot v_2(x_2)$ and $v'(y) = v_1'(y_1) \cdot v_2'(y_2)$, where $x_1 = x \cap E_1$, $x_2 = x \cap E_1$, $y_1 = y \cap E_1'$, and $y_2 = y \cap E_2'$. We then have:

$$v(x) = v_1(x_1) \cdot v_2(x_2) \geq v_1'(y_1) \cdot v_2'(y_2) = v'(y)$$

□

Proof of Lemma 5.2.15

Proof. Let

$$\begin{aligned}
&\mathsf{P}_1 = (\mathsf{E}_1, v_1)\\
&\mathsf{P}'_1(\mathsf{E}'_1, v'_1)\\
&\mathsf{P}_2 = (\mathsf{E}_2, v_2)\\
&\mathsf{P}'_2 = (\mathsf{E}'_2, v'_2)\\
&\mathsf{P}_1 +_p \mathsf{P}_2 = (\mathsf{E}, v)\\
&\mathsf{P}'_1 +_p \mathsf{P}'_2 = (\mathsf{E}', v')
\end{aligned}$$

1. We start by defining $f : E \to E'$ such that

$$f(e) = \begin{cases} e & \text{if } e = \tau \in \mathcal{I}(\mathsf{P}_1 +_p \mathsf{P}_2) \\ f_1(e) & \text{if } e \in E_1 \\ f_2(e) & \text{if } e \in E_2 \end{cases}$$

   It is straightforward to see that f is injective.

2. $\pi(f(e)) = \pi(e)$

   If $e \in E_1$ or $e \in E_2$ we are done, since $\mathsf{P}_1 \sqsubseteq \mathsf{P}'_1$ and $\mathsf{P}_2 \sqsubseteq \mathsf{P}'_2$. If $e = \tau$, then $\pi(f(e)) = \pi(e)$, and we are done.

3. $e \leq e' \Leftrightarrow f(e) \leq' f(e')$

   If $e, e' \in E_1$ or $e, e' \in E_2$ we are done, since $\mathsf{P}_1 \sqsubseteq \mathsf{P}'_1$ and $\mathsf{P}_2 \sqsubseteq \mathsf{P}'_2$. If $e = \tau$ then $f(e)$. Consequently $e \leq e' \Leftrightarrow e \leq' f(e')$, which is trivially satisfied by Definition 5.2.5.

4. $e\#e' \Leftrightarrow f(e)\#'f(e')$

   Since τ is not in conflict with any event, this case is similar to that of Lemma 4.2.16.

5. $\forall x \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2), y \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2) \,.\, f[x] \subseteq y \Rightarrow v(x) \geq v'(y)$

   Let $x \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2)$ and $y \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2)$ such that $f[x] \subseteq y$. We have two cases:

   (a) $x\backslash\tau \in \mathcal{C}(\mathsf{P}_1)$ and $y\backslash\tau \in \mathcal{C}(\mathsf{P}'_1)$
   It follows directly that $v(x) \geq v'(y)$, since $\mathsf{P}_1 \sqsubseteq \mathsf{P}'_1$ and $v(x) = p \cdot v_1(x\backslash\tau) \geq p \cdot v'_1(y\backslash\tau) = v'(y)$.

   (b) $x\backslash\tau \in \mathcal{C}(\mathsf{P}_2)$ and $y\backslash\tau \in \mathcal{C}(\mathsf{P}'_2)$
   It follows directly that $v(x) \geq v'(y)$, since $\mathsf{P}_2 \sqsubseteq \mathsf{P}'_2$ and $v(x) = (1-p) \cdot v_2(x\backslash\tau) \geq (1-p) \cdot v'_2(y\backslash\tau) = v'(y)$.

□

Proof of Lemma 5.2.17

Proof. Let

$$\begin{aligned}
&P = (\mathsf{E}, v)\\
&P\backslash a = (\mathsf{E}', v')
\end{aligned}$$

By Lemma 4.2.20 we know that E' is an event structure. Hence we focus solely on the valuation part.

1. $v'(\emptyset) = 1$

$$v'(\emptyset) = \frac{v_1(\emptyset \cup \{a\})}{v_1(\{a\})} = \frac{v_1(\{a\})}{v_1(\{a\})} = 1$$

2. $d_v^{(n)}[y\,;\, x_1, \ldots, x_n] \geq 0$ for all $n \geq 1$ and $y, x_1, \ldots, x_n \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2)$ with $y \subseteq x_1, \ldots, x_n$
   By [119, Proposition 5] we only need to check the condition for $y \multimap\!\subset x_1, \ldots, x_n$, *i.e.* $y \overset{e_1,\ldots,e_n}{\multimap\!\subset} x_1, \ldots, x_n$.
   Let $I \subseteq \{1, \ldots, n\}$.

$$\begin{aligned}
d_v^{(n)}[y\,;\, x_1, \ldots, x_n] \geq 0 &\Leftrightarrow \sum_I (-1)^{|I|} v\left(y \cup \bigcup_{i \in I} x_i\right) \geq 0 \\
&\Leftrightarrow \sum_I (-1)^{|I|} \frac{v_1\left(\left(y \cup \bigcup_{i \in I} x_i\right) \cup \{a\}\right)}{v_1(\{a\})} \geq 0 \\
&\Leftrightarrow \sum_I (-1)^{|I|} v_1\left(\left(y \cup \bigcup_{i \in I} x_i\right) \cup \{a\}\right) \geq 0 \\
&\Leftrightarrow d_{v_1}^{(n)}[y \cup \{a\}\,;\, x_1 \cup \{a\}, \ldots, x_n \cup \{a\}] \geq 0
\end{aligned}$$

□

Proof of Lemma 5.2.18

Proof. Let

$$\begin{aligned}
&\mathsf{P}_1 = (\mathsf{E}_1,\, v_1) \\
&\mathsf{P}_2 = (\mathsf{E}_2,\, v_2) \\
&\mathsf{P}_1\,;\,\mathsf{P}_2 = (\mathsf{E}_{1;2},\, v_{1;2}) \\
&(\mathsf{P}_1\,;\,\mathsf{P}_2)\backslash l = (\mathsf{E},\, v) \\
&\mathsf{P}_1\backslash l = (\mathsf{E}_1^l,\, v_1^l) \\
&(\mathsf{P}_1\backslash l)\,;\,\mathsf{P}_2 = (\mathsf{E}',\, v') \\
&l \in \mathcal{I}(\mathsf{P}_1\,;\,\mathsf{P}_2)
\end{aligned}$$

Due to Lemma 4.2.22 we only need to show

1. $\forall x \in \mathcal{C}((\mathsf{P}_1\,;\,\mathsf{P}_2)\backslash l), y \in \mathcal{C}((\mathsf{P}_1\backslash l)\,;\,\mathsf{P}_2)\,.\, f[x] \subseteq y \Rightarrow v(x) \geq v'(y)$
   Let $x \in \mathcal{C}((\mathsf{P}_1\,;\,\mathsf{P}_2)\backslash l)$ and $y \in \mathcal{C}((\mathsf{P}_1\backslash l)\,;\,\mathsf{P}_2)$ such that $f[x] \subseteq y$. We have two cases:

   (a) $\{l\} \cup x \in \mathcal{C}(\mathsf{P}_1\,;\,\mathsf{P}_2)$ such that $\{l\} \cup x \in \mathcal{C}(\mathsf{P}_1)$

$$v(x) = \frac{v_{1;2}(\{l\} \cup x)}{v_{1;2}(\{l\})} = \frac{v_1(\{l\} \cup x)}{v_1(\{l\})} = v_1^l(x) = v'(x)$$

   Since $(\mathsf{P}_1\backslash l)\,;\,\mathsf{P}_2$ is a probabilistic event structure, then for $x, y \in \mathcal{C}((\mathsf{P}_1\backslash l)\,;\,\mathsf{P}_2)$ such that $f[x] \subseteq y$, we have $v'(x) \geq v'(y)$, since $d_{v'}^{(1)}[x\,;\,y] \geq 0 \Leftrightarrow v'(x) - v'(y) \geq 0 \Leftrightarrow v'(x) \geq v'(y)$.
   Hence $v(x) \geq v'(y)$.

(b) $\{l\} \cup x \in \mathcal{C}(\mathsf{P}_1 \,;\, \mathsf{P}_2)$ such that $\exists(\{l\} \cup x_1) \in \mathcal{C}_{\max}(\mathsf{P}_1), x_2 \in \mathcal{C}(\mathsf{P}_2)$ where $\{l\} \cup x = (\{l\} \cup x_1) \cup (x_2 \times \{\{l\} \cup x_1\})$

$$v(x) = \frac{v_{1\,;2}(\{l\} \cup x)}{v_{1\,;2}(\{l\})} = \frac{v_1(\{l\} \cup x_1) \cdot v_2(x_2)}{v_1(\{l\})} = v_1^l(x_1) \cdot v_2(x_2) = v'(x)$$

Since $(\mathsf{P}_1 \backslash l) \,;\, \mathsf{P}_2$ is a probabilistic event structure, we obtain $v(x) \geq v'(y)$.

2. $\forall x \in \mathcal{C}((\mathsf{P}_1 \backslash l) \,;\, \mathsf{P}_2), y \in \mathcal{C}((\mathsf{P}_1 \,;\, \mathsf{P}_2) \backslash l) \,.\, f[x] \subseteq y \Rightarrow v'(x) \geq v(y)$

Let $x \in \mathcal{C}((\mathsf{P}_1 \backslash l) \,;\, \mathsf{P}_2)$ and $y \in \mathcal{C}((\mathsf{P}_1 \,;\, \mathsf{P}_2) \backslash l)$ such that $f[x] \subseteq y$. We have two cases:

(a) $x \in \mathcal{C}(\mathsf{P}_1 \backslash l \,;\, \mathsf{P}_2)$ such that $x \in \mathcal{C}(\mathsf{P}_1 \backslash l)$

$$v'(x) = v_1^l(x) = \frac{v_1(\{l\} \cup x)}{v_1(\{l\})} = \frac{v_{1\,;2}(\{l\} \cup x)}{v_{1\,;2}(\{l\})} = v(x)$$

Since $(\mathsf{P}_1 \,;\, \mathsf{P}_2) \backslash l$ is a probabilistic event structure, we obtain $v'(x) \geq v(y)$.

(b) $x \in \mathcal{C}(\mathsf{P}_1 \backslash l \,;\, \mathsf{P}_2)$ such that $\exists x_1 \in \mathcal{C}_{\max}(\mathsf{P}_1 \backslash l), x_2 \in \mathcal{C}(\mathsf{P}_2)$ such that $x = x_1 \cup (x_2 \times \{x_1\})$

$$v'(x) = v_1^l(x_1) \cdot v_2(x_2) = \frac{v_1(\{l\} \cup x_1) \cdot v_2(x_2)}{v_1(\{l\})} = \frac{v_{1\,;2}(\{l\} \cup x)}{v_{1\,;2}(\{l\})} = v(x)$$

Since $(\mathsf{P}_1 \,;\, \mathsf{P}_2) \backslash l$ is a probabilistic event structure, we obtain $v'(x) \geq v(y)$.

□

Proof of Lemma 5.2.19

Proof. Let

$$\begin{aligned}
&\mathsf{P}_1 = (\mathsf{E}_1, v_1)\\
&\mathsf{P}_2 = (\mathsf{E}_2, v_2)\\
&\mathsf{P}_1 \,||\, \mathsf{P}_2 = (\mathsf{E}, v)\\
&(\mathsf{P}_1 \,||\, \mathsf{P}_2) \backslash l = (\mathsf{E}', v')\\
&\mathsf{P}_1 \backslash l = (\mathsf{E}_1^l, v_1^l)\\
&\mathsf{P}_2 \backslash l = (\mathsf{E}_2^l, v_2^l)\\
&(\mathsf{P}_1 \backslash l) \,||\, \mathsf{P}_2 = (\mathsf{E}^l, v^l)\\
&l \in \mathcal{I}(\mathsf{P}_1 \,||\, \mathsf{P}_2)
\end{aligned}$$

Due to Lemma 4.2.22, and similarly to it, we focus when $l \in \mathcal{I}(\mathsf{E}_1)$ and only show

1. $\forall x \in \mathcal{C}((\mathsf{P}_1 \,||\, \mathsf{P}_2) \backslash l), y \in \mathcal{C}((\mathsf{P}_1 \backslash l) \,||\, \mathsf{P}_2) \,.\, f[x] \subseteq y \Rightarrow v'(x) \geq v^l(y)$

$$\begin{aligned}
v'(x) &= \frac{v(\{l\} \cup x)}{v(\{l\})} = \frac{v_1((\{l\} \cup x) \cap E_1) \cdot v_2((\{l\} \cup x) \cap E_2)}{v_1(\{l\})}\\
&= \frac{v_1(\{l\} \cup (x \cap E_1)) \cdot v_2(x \cap E_2)}{v_1(\{l\})} = v_1^l(x \cap E_1) \cdot v_2(x \cap E_2) = v^l(x)
\end{aligned}$$

Since $(\mathsf{P}_1 \backslash l) \,||\, \mathsf{P}_2$ is a probabilistic event structure, we obtain $v'(x) \geq v^l(y)$.

2. $\forall x \in \mathcal{C}((\mathsf{P}_1 \backslash l) \,||\, \mathsf{P}_2), y \in \mathcal{C}((\mathsf{P}_1 \,||\, \mathsf{P}_2) \backslash l) \,.\, f[x] \subseteq y \Rightarrow v^l(x) \geq v'(y)$

$$v^l(x) = v_1^l(x \cap E_1) \cdot v_2(x \cap E_2) = \frac{v_1(\{l\} \cup (x \cap E_1)) \cdot v_2(x \cap E_2)}{v_1(\{l\})}$$
$$= \frac{v_1((\{l\} \cup x) \cap E_1) \cdot v_2((\{l\} \cup x) \cap E_2)}{v_1(\{l\})} = \frac{v(\{l\} \cup x)}{v(\{l\})} = v'(x)$$

Since $(\mathsf{P}_1 \,||\, \mathsf{P}_2) \backslash l$ is a probabilistic event structure, we obtain $v^l(x) \geq v'(y)$.

□

Proof of Lemma 5.2.20

Proof.

- $sk \in \mathcal{I}([\![skip]\!])$

  It follows directly that $v(\{sk\}) = 1$.

- $a \in \mathcal{I}([\![a]\!])$

  It follows directly that $v(\{a\}) = 1$.

- $\tau \in \mathcal{I}([\![C_1 \;+_p\; C_2]\!])$

  It follows directly that $v(\{\tau\}) = 1$.

- $l' \in \mathcal{I}([\![C_1 \,;\, C_2]\!])$

  By Definition 5.2.1 we have $l' \in \mathcal{I}([\![C_1]\!])$. By i.h., $v(\{l'\}) = 1$ and since $l' \in \mathcal{I}([\![C_1 \,;\, C_2]\!])$ we are done.

- $l' \in \mathcal{I}([\![C_1 \,||\, C_2]\!])$

  By Definition 5.2.3 we have $l' \in \mathcal{I}([\![C_1]\!])$ or $l' \in \mathcal{I}([\![C_2]\!])$. By i.h., $v(\{l'\}) = 1$ for both cases. Since $l' \in \mathcal{I}([\![C_1 \,||\, C_2]\!])$ we are done.

□

Proof of Lemma 5.3.3

Proof. Due to Lemma 4.3.4 we only need to check the condition of the valuations. Consider $\mathsf{P}_1 = (\mathsf{E}_1, v_1)$, $\mathsf{P}_2 = (\mathsf{E}_2, v_2)$, and $\mathsf{P}_3 = (\mathsf{E}_3, v_3)$ to be probabilistic event structures.

- Reflexivity: $\mathsf{P}_1 = \mathsf{P}_1$

  We want to show that $\forall x \in \mathcal{C}(\mathsf{P}_1) \,.\, v_1(x) = v_1(x)$. It holds straightforwardly.

- Transitivity: $\mathsf{P}_1 \trianglelefteq \mathsf{P}_2,\ \mathsf{P}_2 \trianglelefteq \mathsf{P}_3 \Rightarrow \mathsf{P}_1 \trianglelefteq \mathsf{P}_3$

  We want to show $\forall x \in \mathcal{C}(\mathsf{P}_1) \,.\, v_1(x) = v_3(x)$. From $\mathsf{P}_1 \trianglelefteq \mathsf{P}_2$, $\forall x \in \mathcal{C}(\mathsf{P}_1) \,.\, v_1(x) = v_2(x)$. From $\mathsf{P}_2 \trianglelefteq \mathsf{P}_3$, $\forall x \in \mathcal{C}(\mathsf{P}_2) \,.\, v_2(x) = v_3(x)$. Hence, $\forall x \in \mathcal{C}(\mathsf{P}_1) \,.\, v_1(x) = v_3(x)$.

- Antisymmetry: $\mathsf{P}_1 \trianglelefteq \mathsf{P}_2,\ \mathsf{P}_2 \trianglelefteq \mathsf{P}_1 \Rightarrow \mathsf{P}_1 = \mathsf{P}_2$

  We want to show $\forall x \in \mathcal{C}(\mathsf{P}_1), \mathcal{C}(\mathsf{P}_2) \,.\, v_1(x) = v_2(x)$. From $\mathsf{P}_1 \trianglelefteq \mathsf{P}_2$, $\forall x \in \mathcal{C}(\mathsf{P}_1) \,.\, v_1(x) = v_2(x)$. From $\mathsf{P}_2 \trianglelefteq \mathsf{P}_1$, $\forall x \in \mathcal{C}(\mathsf{P}_2) \,.\, v_2(x) = v_1(x)$. Hence, $\forall x \in \mathcal{C}(\mathsf{P}_1), \mathcal{C}(\mathsf{P}_2) \,.\, v_1(x) = v_2(x)$.

□

Proof of Lemma 5.3.4

Proof. We first show that $\bot$ is a probabilistic event structure. From Lemma 4.3.5 $\bot = (\emptyset, \emptyset, \emptyset)$ is an event structure. It lacks to see the conditions on the valuations. It follows directly the definition that $v(\emptyset) = 1$. Furthermore the only configuration in $\mathcal{C}(\bot)$ is $\emptyset$. Hence we trivially have that $v(\emptyset) \geq 0$.

To show that $\bot$ is the least element, consider any probabilistic event structure P. We need to show that $\bot \trianglelefteq \mathsf{P}$. Due to Lemma 4.3.5 we focus solely on the valuations. Since the empty configuration is the only one in $\mathcal{C}(\emptyset)$ and since P is a probabilistic event structure it holds that $v_{\bot}(\emptyset) = 1 = v(\emptyset)$. □

Proof of Lemma 5.3.6

Proof. Due to Lemma 4.3.7 we focus only on the valuation part, where we have two conditions to verify:

- $v^{\omega}(\emptyset) = 1$

  From Definition 5.3.5 we know that $\exists n \in \omega \,.\, v^{\omega}(\emptyset) = v_n(\emptyset) = 1$.

- $\forall y, x_1, \ldots, x_m \in \mathcal{C}(\mathsf{P}^{\omega})$ such that $y \subseteq x_1, \ldots, x_m$, $\sum_{I \subseteq \{1,\ldots,m\}} (-1)^{|I|} v^{\omega}(y \cup \cup_{i \in I} x_i) \geq 0$

  Following [119, Propostion 5] we only need to focus on $y \subset\!\!\!\!- x_1, \ldots, x_m$. From Definition 5.3.5 we know it $\exists n \in \omega \,.\, v^{\omega}(y) = v_n(y)$. We then have three cases, depending if the events are in E_n, in E_{n+1}, or in both.

  1. the events are in E_n

     We know that $\sum_{\emptyset \neq I \subseteq \{1,\ldots,m\}} (-1)^{|I+1|} v^{\omega}(\cup_{i \in I} x_i) = \sum_{\emptyset \neq I \subseteq \{1,\ldots,m\}} (-1)^{|I+1|} v_n(\cup_{i \in I} x_i)$ and consequently $v_n(y) - \sum_{\emptyset \neq I \subseteq \{1,\ldots,m\}} (-1)^{|I+1|} v_n(\cup_{i \in I} x_i) \geq 0$, since P_n is a probabilistic event structure

  2. the events are in E_{n+1}

     We know that $v_n(y) = v_{n+1}(y)$ since $\mathsf{P}_n \trianglelefteq \mathsf{P}_{n+1}$. Furthermore

     $$\sum_{\emptyset \neq I \subseteq \{1,\ldots,m\}} (-1)^{|I+1|} v^{\omega}(\cup_{i \in I} x_i) = \sum_{\emptyset \neq I \subseteq \{1,\ldots,m\}} (-1)^{|I+1|} v_{n+1}(\cup_{i \in I} x_i)$$

     and consequently

     $$v_{n+1}(y) - \sum_{\emptyset \neq I \subseteq \{1,\ldots,m\}} (-1)^{|I+1|} v_{n+1}(\cup_{i \in I} x_i) \geq 0$$

     since P_{n+1} is a probabilistic event structure

  3. the events are in both

     Since $\mathsf{P}_n \trianglelefteq \mathsf{P}_{n+1}$ we know that $\forall x \in \mathcal{C}(\mathsf{P}_n) \,.\, v_n(x) = v_{n+1}(x)$, which leads us to the previous case.

□

Proof of Lemma 5.3.7

Proof. Due to Lemma 4.3.8 we focus only on the valuations.

- P^{ω} is an upper bound

  $\forall n \in \omega$ we need to have $\mathsf{P}_n \trianglelefteq \mathsf{P}^{\omega}$. It follows directly from Definition 5.3.2 that $\forall n \in \omega$ we have $\mathsf{P}_n \trianglelefteq \mathsf{P}^{\omega}$ since by Definition 5.3.5, $\forall x \in \mathcal{C}(\mathsf{P}^{\omega})$, $\exists n \in \omega \,.\, v^{\omega}(x) = v_n(x)$.

- P^ω is the least upper bound

  Let $\mathsf{P} = (\mathsf{E}, v)$ be an upper bound of the chain. We need to show that if $\mathsf{P}_n \trianglelefteq \mathsf{P}^\omega$ and $\mathsf{P}_n \trianglelefteq \mathsf{P}$ then $\mathsf{P}^\omega \trianglelefteq \mathsf{P}$. From $\mathsf{P}_n \trianglelefteq \mathsf{P}^\omega$, $\forall x \in \mathcal{C}(\mathsf{P}_n) \,.\, v_n(x) = v^\omega(x)$. From Definition 5.3.5, $\forall x \in \mathcal{C}(\mathsf{P}^\omega),\ \exists n \in \omega \,.\, v^\omega(x) = v_n(x)$. From $\mathsf{P}_n \trianglelefteq \mathsf{P}$, $\forall x \in \mathcal{C}(\mathsf{P}_n) \,.\, v_n(x) = v(x)$. Thus $\forall x \in \mathcal{C}(\mathsf{P}^\omega), \exists n \in \omega \,.\, v^\omega(x) = v_n(x) = v(x)$.

□

Proof of Lemma 5.3.8

Proof. Let

$$\begin{aligned}
\mathsf{P} &= (\mathsf{E}, v) \\
\mathsf{P}_1 &= (\mathsf{E}_1, v_1) \\
\mathsf{P}_2 &= (\mathsf{E}_2, v_2) \\
\mathsf{P} \,;\, \mathsf{P}_1 &= (\mathsf{E}^1, v^1) \\
\mathsf{P} \,;\, \mathsf{P}_2 &= (\mathsf{E}^2, v^2)
\end{aligned}$$

We want to show $\forall x \in \mathcal{C}(\mathsf{P} \,;\, \mathsf{P}_1) \,.\, v^1(x) = v^2(x)$. Due to Lemma 4.3.10 we only focus on the valuations. According to Definition 5.2.1 we have two cases:

1. $x \in \mathcal{C}(\mathsf{P} \,;\, \mathsf{P}_1)$ such that $x \in \mathcal{C}(\mathsf{P})$

   Then we are done because $v^1(x) = v(x) = v^2(x)$.

2. $x \in \mathcal{C}(\mathsf{P} \,;\, \mathsf{P}_1)$ such that $\exists y \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}),\ y' \in \mathcal{C}(\mathsf{P}_1) \,.\, x = y \cup (y' \times \{y\})$

   Then we have $v^1(x) = v(y) \cdot v_1(y')$. Since $\mathsf{P}_1 \trianglelefteq \mathsf{P}_2$, $\forall y' \in \mathcal{C}(\mathsf{P}_1) \,.\, v_1(y') = v_2(y')$. Then $v(y) \cdot v_1(y') = v(y) \cdot v_2(y') = v^2(x)$. Hence $v^1(x) = v^2(x)$.

□

Proof of Lemma 5.3.9

Proof. Let

$$\begin{aligned}
\mathsf{P}_1 &= (\mathsf{E}_1, v_1) \\
\mathsf{P}_2 &= (\mathsf{E}_2, v_2) \\
\mathsf{P}'_1 &= (\mathsf{E}'_1, v'_1) \\
\mathsf{P}'_2 &= (\mathsf{E}'_2, v'_2) \\
\mathsf{P}_1 \,||\, \mathsf{P}_2 &= (\mathsf{E}, v) \\
\mathsf{P}'_1 \,||\, \mathsf{P}'_2 &= (\mathsf{E}', v')
\end{aligned}$$

We want to show $\forall x \in \mathcal{C}(\mathsf{P}_1 \,||\, \mathsf{P}_2) \,.\, v(x) = v'(x)$. Due to Lemma 4.3.11 we only focus on the valuations.

Let $x \in \mathcal{C}(\mathsf{P}_1 \,||\, \mathsf{P}_2) \,.\, v(x) = v_1(x \cap E_1) \cdot v_2(x \cap E_2)$, such that $x_1 = x \cap E_1$ and $x_2 = x \cap E_2$. Since $\mathsf{P}_1 \trianglelefteq \mathsf{P}'_1$ and $\mathsf{P}_2 \trianglelefteq \mathsf{P}'_2$ then $\forall x_1 \in \mathcal{C}(\mathsf{P}_1) \,.\, v_1(x_1) = v'_1(x_1)$ and $\forall x_2 \in \mathcal{C}(\mathsf{P}_2) \,.\, v_2(x_2) = v'_1(x_2)$, respectively. Hence $v(x) = v_1(x_1) \cdot v_2(x_2) = v'_1(x_1) \cdot v'_2(x_2) = v'(x)$. □

Proof of Lemma 5.3.10

Proof. Let

$$\begin{aligned}
\mathsf{P}_1 &= (\mathsf{E}_1, v_1)\\
\mathsf{P}'_1 &= (\mathsf{E}'_1, v'_1)\\
\mathsf{P}_2 &= (\mathsf{E}_2, v_2)\\
\mathsf{P}'_2 &= (\mathsf{E}'_2, v'_2)\\
\mathsf{P}_1 +_p \mathsf{P}_2 &= (\mathsf{E}, v)\\
\mathsf{P}'_1 +_p \mathsf{P}'_2 &= (\mathsf{E}', v')
\end{aligned}$$

The conditions to check are:

1. $E \subseteq E'$
2. $\forall e, e' \,.\, e \leq e' \Leftrightarrow e, e' \in E \wedge e \leq' e'$
3. $\forall e, e' \,.\, e\#e' \Leftrightarrow e, e' \in E \wedge e\#'e'$
4. $\forall x \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2) \,.\, v(x) \geq v'(y)$

The first three conditions follow directly from Definition 5.2.5. Hence we focus on the last one. Let $x \in \mathcal{C}(\mathsf{P}_1 +_p \mathsf{P}_2)$. We have two cases:

1. $x\backslash\tau \in \mathcal{C}(\mathsf{P}_1)$

   It follows directly that $v(x) = v'(x)$, since $\mathsf{P}_1 \trianglelefteq \mathsf{P}'_1$ and $v(x) = p \cdot v_1(x\backslash\tau) = p \cdot v'_1(x\backslash\tau) = v'(x)$.

2. $x\backslash\tau \in \mathcal{C}(\mathsf{P}_2)$

   It follows directly that $v(x) = v'(x)$, since $\mathsf{P}_2 \trianglelefteq \mathsf{P}'_2$ and $v(x) = (1-p) \cdot v_2(x\backslash\tau) = (1-p) \cdot v'_2(y\backslash\tau) = v'(x)$.

□

Proof of Lemma 5.3.11

Proof. By Lemma 5.3.8, the sequential composition is monotone at right. Furthermore, showing that each event of $\mathsf{P}\,;\, \bigsqcup_m \mathsf{P}_m$ is an event of $\bigsqcup_m (\mathsf{P}\,;\, \mathsf{P}_m)$ is already done in Lemma 4.3.14. By Lemma 4.3.13 we are done. □

Proof of Lemma 5.3.12

Proof. By Lemma 5.3.9, the parallel composition is monotone at right. Furthermore, showing that each event of $\bigsqcup_n \mathsf{P}_n \,||\, \bigsqcup_m \mathsf{P}_m$ is an event of $\bigsqcup_{n,m} (\mathsf{P}_n \,||\, \mathsf{P}_m)$ is already done in Lemma 4.3.15. By Lemma 4.3.13 we are done. □

Proof of Lemma 5.3.13

Proof. By Lemma 5.3.10 we know that the probabilistic choice is monotone. It lacks to show that each event of $\bigsqcup_n \mathsf{P}_n +_p \bigsqcup_m \mathsf{P}_m$ is an event of $\bigsqcup_{n,m} (\mathsf{P}_n +_p \mathsf{P}_m)$.

Let $\mathsf{P}_1 \trianglelefteq \cdots \trianglelefteq \mathsf{P}_n \trianglelefteq \ldots$ and $\mathsf{P}'_1 \trianglelefteq \cdots \trianglelefteq \mathsf{P}'_m \trianglelefteq \ldots$ be ω-chains with least upper bound $\bigsqcup_n \mathsf{P}_n$ and $\bigsqcup_m \mathsf{P}_m$, respectively. Let e be an event of $\bigsqcup_n \mathsf{P}_n +_p \bigsqcup_m \mathsf{P}_m$. By Definition 5.2.5 we have three cases:

1. $e = \tau$

   It follows directly from Definition 5.2.5 that τ is an event of $\mathsf{P}_n +_p \mathsf{P}_m$. Consequently it is an event of $\bigsqcup_{n,m}(\mathsf{P}_n +_p \mathsf{P}_m)$.

2. e is an event of $\bigsqcup_n \mathsf{P}_n$

   By Definition 5.3.5, $\exists n \in \omega \,.\, e$ is an event of P_n. By Definition 5.2.5, e is an event of $\mathsf{P}_n +_p \mathsf{P}_m$ and consequently it is an event of $\bigsqcup_{n,m}(\mathsf{P}_n +_p \mathsf{P}_m)$.

3. e is an event of $\bigsqcup_m \mathsf{P}_m$

   Similar to the previous point.

By Lemma 4.3.13 we are done. □

Proof of Lemma 5.3.15

Proof. We only do for the probabilistic choice, since for the remaining cases the prove is the same as in Lemma 4.3.19.

$$
\begin{aligned}
&\Gamma^{C_1 +_p C_2, \gamma}(\bigsqcup_n \mathsf{P}_n) \\
=&\{\text{Definition 5.3.14}\} \\
&[\![C_1 +_p C_2]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{P}_n)} \\
=&\{\text{Definition 5.3.14}\} \\
&[\![C_1]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{P}_n)} +_p [\![C_2]\!]_{\gamma(X \leftarrow \bigsqcup_n \mathsf{P}_n)} \\
=&\{\text{Definition 5.3.14}\} \\
&\bigsqcup_n \Gamma^{C_1,\gamma}(\mathsf{P}_n) +_p \bigsqcup_n \Gamma^{C_2,\gamma}(\mathsf{P}_n) \\
=&\{\text{Lemma 5.3.13}\} \\
&= \bigsqcup_n (\Gamma^{C_1,\gamma}(\mathsf{P}_n) +_p \Gamma^{C_2,\gamma}(\mathsf{P}_n)) \\
=&\{\text{Definition 5.3.14}\} \\
&= \bigsqcup_n ([\![C_1]\!]_{\gamma(X \leftarrow \mathsf{P}_n)} +_p [\![C_2]\!]_{\gamma(X \leftarrow \mathsf{P}_n)}) \\
=&\{\text{Definition 5.3.14}\} \\
&\bigsqcup_n [\![C_1 +_p C_2]\!]_{\gamma(X \leftarrow \mathsf{P}_n)}
\end{aligned}
$$

□

Proof of Lemma 5.3.16

Proof. We only show the probabilistic choice, since the proof for the other cases is in Lemma 4.3.20.

$$
\begin{aligned}
&[\![(C_1 +_p C_2)[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma \\
=&\{\text{Adaptation of Definition 4.3.1}\} \\
&[\![C_1[X \leftarrow [\![\mu X.C]\!]_\gamma] +_p C_2[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma \\
=&\{\text{Definition 5.3.14}\}
\end{aligned}
$$

$$
\begin{aligned}
&[\![C_1]\!][X \leftarrow [\![\mu X.C]\!]_\gamma] \ +_p \ [\![C_2]\!]_\gamma[X \leftarrow [\![\mu X.C]\!]_\gamma] \\
=&\{\text{i.h.}\} \\
&[\![C_1]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \ +_p \ [\![C_2]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \\
=&\{\text{Definition 5.3.14}\} \\
&[\![C_1 \ +_p \ C_2]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}
\end{aligned}
$$

□

Proof of Lemma 5.4.1

Proof. Follows directly from the respective definitions. □

Proof of Lemma 5.4.2

Proof. Let

$$
\begin{aligned}
\mathsf{P} &= (E, \leq, \#, v) \\
\mathsf{P}_i &= (E_i, \leq_i, \#_i, v_i) \\
\mathsf{P}_i \,||\, \mathsf{P} &= (E_i', \leq_i', \#_i', v_i') \\
\sum_i p_i \cdot \mathsf{P}_i &= (E^1, \leq^1, \#^1, v^1) \\
(\sum_i p_i \cdot \mathsf{P}_i) \,||\, \mathsf{P} &= (E^2, \leq^2, \#^2, v^2) \\
\sum_i p_i \cdot (\mathsf{P}_i \,||\, \mathsf{P}) &= (E^3, \leq^3, \#^3, v^3)
\end{aligned}
$$

1. $(\sum_i p_i \cdot \mathsf{P}_i) \,||\, \mathsf{P} \sqsubseteq \sum_i p_i \cdot (\mathsf{P}_i \,||\, \mathsf{P})$
   We know that $E^2 = E^1 \uplus E = (\{\tau\} \uplus \biguplus_i E_i) \uplus E$ and $E^3 = \{\tau\} \uplus \biguplus_i (E_i \uplus E)$. Note that with $\biguplus_i (E_i \uplus E)$, for each i we are making a copy of E. We then denote the events of E from $\biguplus_i (E_i \uplus E)$ as e^i, for each i (it is important to highlight that the plain events of e and e^i are the same) and we write $E^{(i)}$ for the P-copy inside branch i.

   (a) We begin by defining, for a fixed i, $f_i : E^2 \to E^3$ such that

   $$
   f_i(e) = \begin{cases} e & \text{if } e \in \{\tau\} \uplus E_i \subseteq E^1 \\ e^i & \text{if } e \in E \end{cases}
   $$

   Since i is fixed, it is straightforward to see that f_i is injective (f_i is essentially the identity map, where for each i it creates a copy of the events in E).

   (b) $\pi(f_i(e)) = \pi(e)$
   If $e \in \{\tau\} \uplus E_i$ then we are done since $f_i(e) = e$. If $e \in E$, then $\pi(f_i(e)) = \pi(e^i) = \pi(e)$, since e^i is a copy of e for a fixed i.

   (c) $e \leq^2 e' \implies f_i(e) \leq^3 f_i(e')$
   Assume $e \leq^2 e'$. By Definition 5.2.3, $e \leq^1 e'$ or $e \leq e'$. If $e \leq e'$, then by Definition 5.2.3 and for a given i, $f_i(e) \leq_i' f_i(e')$. By Remark 2 $f_i(e) \leq^3 f_i(e')$.
   If $e \leq^1 e'$, then by Remark 2, exists i such that $e \leq_i e'$. By Definition 5.2.3, $e \leq_i' e'$. By Remark 2 $f_i(e) \leq^3 f_i(e')$.

(d) $e\#^2e' \implies f_i(e)\#^3 f_i(e')$
Similar to the previous case.

(e) $\forall x \in \mathcal{C}((\sum_i p_i \cdot \mathsf{P}_i) \,||\, \mathsf{P}), y \in \mathcal{C}(\sum_i p_i \cdot (\mathsf{P}_i \,||\, \mathsf{P}))\,.\, f_i[x] \subseteq y \Rightarrow v^2(x) \geq v^3(y)$
Let $x \in \mathcal{C}((\sum_i p_i \cdot \mathsf{P}_i) \,||\, \mathsf{P})$, $y \in \mathcal{C}(\sum_i p_i \cdot (\mathsf{P}_i \,||\, \mathsf{P}))$, such that $f_i[x] \subseteq y$.
By Definition 5.2.3 and Remark 2 we have:

$$v^2(x) = \begin{cases} p_i v_i((x \cap E^1)\backslash\tau) \cdot v(x \cap E) & \text{if } (x \cap E^1)\backslash\tau \in \mathcal{C}(\mathsf{P}_i) \\ v(x \cap E) & \text{if } (x \cap E^1) = \emptyset \text{ or } (x \cap E^1 = \{\tau\}) \end{cases}$$

By Remark 2 we have:

$$v^3(x) = \begin{cases} p_i v_i'(y\backslash\tau) & \text{if } y\backslash\tau \in \mathcal{C}(\mathsf{P}_i \,||\, \mathsf{P}) \\ 1 & \text{if } (y = \{\tau\}) \text{ or } (y = \emptyset) \end{cases}$$

We have the following cases:

i. $y = \emptyset$
Thus $v^3(y) = 1$. Since $f_i[x] \subseteq y$, then $x = \emptyset$. Thus $v^2(x) = v(\emptyset \cap E) = v(\emptyset) = 1$.

ii. $y = \{\tau\}$
Thus $v^3(y) = 1$. Since $f_i[x] \subseteq y$, then $x = \emptyset$ or $x = \{\tau\}$. If $x = \emptyset$ we are done. If $x = \{\tau\}$, then $v^2(\{\tau\}) = v(\{\tau\}\backslash E)$, which entails $\{\tau\} \cap E^1 = \{\tau\}$. Then $\{\tau\} \cap E = \emptyset$ since $\tau \notin E$. Thus $v(\{\tau\} \cap E) = v(\emptyset) = 1$.

iii. $y\backslash\tau \in \mathcal{C}(\mathsf{P}_i \,||\, \mathsf{P})$
Thus $v^3(y) = p_i \cdot v_i'(y\backslash\tau)$. By Definition 5.2.3, $v_i'(y\backslash\tau) = v_i(y\backslash\tau \cap E_i) \cdot v(y\backslash\tau \cap E^{(i)})$. Since $\tau \notin E_i$ and $\tau \notin E$, then $y\backslash\tau \cap E_i = y \cap E_i$ and $y\backslash\tau \cap E^{(i)} = y \cap E^{(i)}$. Thus $p_i \cdot v_i'(y\backslash\tau) = p_i \cdot v_i(y \cap E_i) \cdot v(y \cap E^{(i)})$.
Since $f_i[x] \subseteq y$, if $x = \emptyset$ or $x = \{\tau\}$ we know that $v^2(x) = 1$. Hence $1 \geq p_i \cdot v_i(y \cap E_i) \cdot v(y \cap E)$, since v_i and v are configuration-valuations.
It lacks to consider the case where $v^2(x) = p_i v_i((x \cap E^1)\backslash\tau) \cdot v(x \cap E)$. To show that $p_i v_i((x \cap E^1)\backslash\tau) \cdot v(x \cap E) \geq p_i \cdot v_i(y \cap E_i) \cdot v(y \cap E^{(i)})$, it suffices to show that $v_i((x \cap E^1)\backslash\tau) \geq v_i(y \cap E_i)$ and $v(x \cap E) \geq v(y \cap E^{(i)})$, since product is monotone in each factor.
We argue first that $v(x \cap E) \geq v(y \cap E^{(i)})$. We know that $f_i[x_i \cap E] = \{f_i(e) \mid e \in x \cap E\} = f_i[x_i] \cap E^{(i)}$. This set is composed of the events of E that belong in x in the i-th copy. Thus, we have $f_i[x_i \cap E] = f_i[x_i] \cap E^{(i)} \subseteq y \cap E^{(i)}$, from $f_i[x_i] \subseteq y$. Since P is a probabilistic event structure, then $v(f_i[x \cap E]) \geq v(y \cap E^{(i)})$. Hence $v(f_i[x \cap E]) = v(x \cap E)$. Thus $v(x \cap E) \geq v(y \cap E^{(i)})$. With similar arguments, we have $v_i((x \cap E^1)\backslash\tau) \geq v_i(y \cap E_i)$. Thus $p_i v_i((x \cap E^1)\backslash\tau) \cdot v(x \cap E) \geq p_i \cdot v_i(y \cap E_i) \cdot v(y \cap E^{(i)})$.

2. $x \in \mathcal{C}_{\mathsf{max}}((\sum_i p_i \cdot \mathsf{P}_i) \,||\, \mathsf{P})$ iff $x \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot (\mathsf{P}_i \,||\, \mathsf{P}))$

For both cases, it is relevant to notice the following: let $\mathsf{P}_1, \mathsf{P}_2$ be two probabilistic event structures such that $x_1 \in \mathcal{C}(\mathsf{P}_1)$ and $x_2 \in \mathcal{C}(\mathsf{P}_2)$. Then $x = x_1 \cup x_2 \in \mathcal{C}(\mathsf{P}_1 \,||\, \mathsf{P}_2)$, since by Definition 5.2.3 there is no conflict between events of P_1 and events of P_2.

$\Leftarrow$ If $x \in \mathcal{C}_{\mathsf{max}}((\sum_i p_i \cdot \mathsf{P}_i) \,||\, \mathsf{P})$ then $x \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot (\mathsf{P}_i \,||\, \mathsf{P}))$
Let $x \in \mathcal{C}_{\mathsf{max}}((\sum_i p_i \cdot \mathsf{P}_i) \,||\, \mathsf{P})$. We can represent x as follows: $x = x \cap (E \uplus E_i) = (x \cap E) \uplus (x \cap E_i)$, for E_i in $\biguplus_i E_i$ and where $x \cap E \in \mathcal{C}_{\mathsf{max}}(\mathsf{P})$ and $x \cap E_i \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot \mathsf{P}_i)$. Hence, it follows directly that $(x \cap E) \uplus (x \cap E_i) = x \cap (E \uplus E_i) = x \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot (\mathsf{P}_i \,||\, \mathsf{P}))$.

$\Rightarrow$ If $x \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot (\mathsf{P}_i \,||\, \mathsf{P}))$ then $x \in \mathcal{C}_{\mathsf{max}}((\sum_i p_i \cdot \mathsf{P}_i) \,||\, \mathsf{P})$
Let $x \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot (\mathsf{P}_i \,||\, \mathsf{P}))$. Hence $\exists x_i \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot \mathsf{P}_i), y \in \mathcal{C}_{\mathsf{max}}(\mathsf{P})$ such that $x = x_i \cup y$. Since $x_i \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot \mathsf{P}_i)$, then $\exists i \,.\, x_i \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}_i)$. We then have $x_i \cup y = x \in \mathcal{C}_{\mathsf{max}}(\mathsf{P}_i \,||\, \mathsf{P})$ and consequently $x \in \mathcal{C}_{\mathsf{max}}((\sum_i p_i \cdot \mathsf{P}_i) \,||\, \mathsf{P})$.

□

Proof of Lemma 5.4.3

Proof. 1. $C \equiv C_1 \,;\, C_2$

We know that $C_1 \,;\, C_2 \rightarrow \sum_i p_i \cdot (\tau, C_i \,;\, C_2)$. By the rules in Figure 43 we have $C_1 \rightarrow \sum_i p_i \cdot (\tau, C_i)$. By i.h. we have $x_1 \in \mathcal{C}_{\mathsf{max}}([\![C_1]\!]_\gamma)$ and $x_1 \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot [\![C_i]\!]_\gamma)$ such that $\exists [\![C_i]\!]_\gamma \,.\, v_1(x_1) = v_i(x_1)$. Let $x_2 \in \mathcal{C}_{\mathsf{max}}([\![C_2]\!]_\gamma)$. By Definition 5.2.1 we have $x_1 \cup (x_2 \times \{x_1\}) \in \mathcal{C}_{\mathsf{max}}([\![C_1 \,;\, C_2]\!]_\gamma)$ and $x_1 \cup (x_2 \times \{x_1\}) \in \mathcal{C}_{\mathsf{max}}((\sum_i p_i \cdot [\![C_i]\!]_\gamma) \,;\, [\![C_2]\!]_\gamma)$, and $v(x) = v_1(x_1) \cdot v_2(x_2) = v_i(x_1) \cdot v_2(x_2)$. By Lemma 5.4.1 we have $x_1 \cup (x_2 \times \{x_1\}) \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot [\![C_i \,;\, C_2]\!]_\gamma)$.

2. $C \equiv C_1 \,||\, C_2$

We know that $C_1 \,||\, C_2 \rightarrow \sum_i p_i \cdot (\tau, C_i \,||\, C_2)$. By the rules in Figure 43 we have $C_1 \rightarrow \sum_i p_i \cdot (\tau, C_i)$. By i.h. we have $x_1 \in \mathcal{C}_{\mathsf{max}}([\![C_1]\!]_\gamma)$ and $x_1 \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot [\![C_i]\!]_\gamma)$ such that $\exists [\![C_i]\!]_\gamma \,.\, v_1(x_1) = v_i(x_1)$. Let $x_2 \in \mathcal{C}_{\mathsf{max}}([\![C_2]\!]_\gamma)$. By Definition 5.2.3 we have $x_1 \cup x_2 \in \mathcal{C}_{\mathsf{max}}([\![C_1 \,||\, C_2]\!]_\gamma)$ and $x_1 \cup x_2 \in \mathcal{C}_{\mathsf{max}}((\sum_i p_i \cdot [\![C_i]\!]_\gamma) \,||\, [\![C_2]\!]_\gamma)$, and $v(x) = v_1(x_1) \cdot v_2(x_2) = v_i(x_1) \cdot v_2(x_2)$. By Lemma 5.4.2 we have $x_1 \cup x_2 \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot [\![C_i \,||\, C_2]\!])_\gamma$.

A similar reasoning is applied when $C_1 \,||\, C_2 \rightarrow \sum_j p_j \cdot (\tau, C_2 \,||\, C_j)$.

□

Proof of Lemma 5.4.4

Proof.

$$\mu X.C \rightarrow \sum_i p_i \cdot (\tau, C_i[X \leftarrow \mu X.C])$$

$\Rightarrow$ {rules in Figure 43}

$$C \rightarrow \sum_i p_i (\tau, C_i)$$

$\Rightarrow$ {i.h.}

$$x \in \mathcal{C}_{\mathsf{max}}([\![C]\!]_\gamma) \text{ and } x \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot [\![C_i]\!]_\gamma) \text{ s.t. } \exists [\![C_i]\!]_\gamma \,.\, v_i(x) = v(x)$$

$\Rightarrow$ {$\gamma = \gamma(X \leftarrow [\![\mu X.C)]\!]_\gamma$}

$$x \in \mathcal{C}_{\mathsf{max}}([\![C]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}) \text{ and } x \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i \cdot [\![C_i]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)})$$

$\Rightarrow$ {Lemma 4.3.21 and Lemma 5.3.16}

$$x \in \mathcal{C}_{\mathsf{max}}(\mu X.C)_\gamma \text{ and } x \in \mathcal{C}_{\mathsf{max}}(\sum_i p_i [\![C_i[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma)$$

□

Proof of Lemma 5.4.5

Proof. Induction over C.

- $C \equiv skip$.

  It follows directly that $skip \twoheadrightarrow 1 \cdot (sk, \checkmark)$

- $C \equiv a$.

  It follows directly that $a \twoheadrightarrow 1 \cdot (a, \checkmark)$

- $C \equiv C_1 +_p C_2$

  By Figure 43, $C_1 +_p C_2 \to p \cdot (\tau, C_1) + (1-p) \cdot (\tau, C_2)$. By i.h., $\exists \sum_n p_n(\omega_n, C_n)$, $\sum_m p_m(\omega_m, C_m)$ s.t. $C_1 \twoheadrightarrow \sum_n p_n(\omega_n, C_n)$ and $C_2 \twoheadrightarrow \sum_m p_m(\omega_m, C_m)$. By Figure 44, $C_1 +_p C_2 \twoheadrightarrow \sum_n p_n(\tau : \omega_n, C_n) + \sum_m p_m(\tau : \omega_m, C_m)$.

- $C \equiv C_1 \,;\, C_2$

  According to Figure 43 we have three cases:

  1. $C_1 \,;\, C_2 \to 1 \cdot (l, C_2)$
     By i.h., $\exists \sum_n p_n(\omega_n, C_n)$ s.t. $C_2 \twoheadrightarrow \sum_n p_n(\omega_n, C_n)$. By Figure 44, $C_1 \,;\, C_2 \twoheadrightarrow \sum_n p_n(l : \omega_n, C_n)$.
  2. $C_1 \,;\, C_2 \to 1 \cdot (l, C_1' \,;\, C_2)$
     By i.h., $\exists \sum_n p_n(\omega_n, C_n)$ s.t. $C_1' \,;\, C_2 \twoheadrightarrow \sum_n p_n(\omega_n, C_n)$. By Figure 44, $C_1 \,;\, C_2 \twoheadrightarrow \sum_n p_n(l : \omega_n, C_n)$.
  3. $C_1 \,;\, C_2 \to \sum_i p_i \cdot (\tau, C_i \,;\, C_2)$
     By i.h., $\forall i, \exists \sum_n p_n(\omega_{in}, C_{in})$ s.t. $C_i \,;\, C_2 \twoheadrightarrow \sum_n p_n(\omega_{in}, C_{in})$. By Figure 44, $C_1 \,;\, C_2 \twoheadrightarrow \sum_i p_i \sum_n p_n(\tau : \omega_{in}, C_{in})$.

- $C \equiv C_1 \,||\, C_2$

  According to Figure 43 we have three cases:

  1. $C_1 \,||\, C_2 \to 1 \cdot (l, C_2)$
     By i.h., $\exists \sum_n p_n(\omega_n, C_n)$ s.t. $C_2 \twoheadrightarrow \sum_n p_n(\omega_n, C_n)$. By Figure 44, $C_1 \,||\, C_2 \twoheadrightarrow \sum_n p_n(l : \omega_n, C_n)$.
  2. $C_1 \,||\, C_2 \to 1 \cdot (l, C_1' \,||\, C_2)$
     By i.h., $\exists \sum_n p_n(\omega_n, C_n)$ s.t. $C_1' \,||\, C_2 \twoheadrightarrow \sum_n p_n(\omega_n, C_n)$. By Figure 44, $C_1 \,||\, C_2 \twoheadrightarrow \sum_n p_n(l : \omega_n, C_n)$.
  3. $C_1 \,||\, C_2 \to \sum_i p_i \cdot (\tau, C_i \,||\, C_2)$
     By i.h., $\forall i, \exists \sum_n p_n(\omega_{in}, C_{in})$ s.t. $C_i \,||\, C_2 \twoheadrightarrow \sum_n p_n(\omega_{in}, C_{in})$. By Figure 44, $C_1 \,||\, C_2 \twoheadrightarrow \sum_i p_i \sum_n p_n(\tau : \omega_{in}, C_{in})$.

- $C \equiv \mu X.D$

  According to Figure 43 we have two cases:

  1. $\mu X.D \to 1 \cdot (l, D'[X \leftarrow \mu X.D])$
     By i.h., $\exists \sum_n p_n(\omega_n, D_n)$ s.t. $D'[X \leftarrow \mu X.D] \twoheadrightarrow \sum_n p_n(\omega_n, D_n)$. By Figure 44, $\mu X.D \twoheadrightarrow \sum_n p_n(l : \omega_n, D_n)$.

2. $\mu X.D \to \sum_i p_i(\tau, D_i[X \leftarrow \mu X.D])$
   By i.h.,$\forall i \; \exists \; \sum_n p_n(\omega_{in}, D_{in})$ s.t. $D_i[X \leftarrow \mu X.D] \twoheadrightarrow \sum_n p_n(\omega_{in}, D_{in})$. By Figure 44, $\mu X.D \twoheadrightarrow \sum_i p_i \sum_n p_n(\tau : \omega_{in}, D_{in})$.

□

Proof of Lemma 5.4.6

Proof. Induction over rules in Figure 43.

- $skip \to 1 \cdot (sk, \checkmark)$
  It follows directly that $[\![\checkmark]\!]_\gamma \equiv [\![skip]\!]_\gamma \backslash sk \equiv \emptyset$.
- $a \to 1 \cdot (a, \checkmark)$
  It follows directly that $[\![\checkmark]\!]_\gamma \equiv [\![a]\!]_\gamma \backslash a \equiv \emptyset$.
- $C_1 \; +_p \; C_2 \to p \cdot (\tau, C_1) + (1-p) \cdot (\tau, C_2)$
  It follows directly that $p \cdot [\![C_1]\!]_\gamma + (1-p) \cdot [\![C_2]\!]_\gamma = [\![C_1 \; +_p \; C_2]\!]_\gamma$.
- $C_1 \,;\, C_2 \to 1 \cdot (l, C_2)$

$$
\begin{array}{l}
\quad C_1 \,;\, C_2 \xrightarrow{l} C_2 \\
\Rightarrow \{\text{Figure 43 entails}\} \\
\quad C_1 \xrightarrow{l} \checkmark \\
\Rightarrow \{\text{i.h.}\} \\
\quad [\![\checkmark]\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l \\
\Rightarrow \{\text{Lemma 5.2.13}\} \\
\quad [\![\checkmark]\!]_\gamma \,;\, [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \backslash l) \,;\, [\![C_2]\!]_\gamma \\
\Rightarrow \{[\![\checkmark]\!]_\gamma \,;\, [\![C_2]\!]_\gamma \equiv [\![C_2]\!]_\gamma, \text{ Lemma 5.2.18}\} \\
\quad [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \,;\, [\![C_2]\!]_\gamma) \backslash l \\
\Rightarrow \{\text{Definition 5.2.8}\} \\
\quad [\![C_2]\!]_\gamma \equiv [\![C_1 \,;\, C_2]\!]_\gamma \backslash l
\end{array}
$$

- $C_1 \,;\, C_2 \to 1 \cdot (l, C_1' \,;\, C_2)$

$$
\begin{array}{l}
\quad C_1 \,;\, C_2 \xrightarrow{l} C_1' \,;\, C_2 \\
\Rightarrow \{\text{Figure 43 entails}\} \\
\quad C_1 \xrightarrow{l} C_1' \\
\Rightarrow \{\text{i.h.}\} \\
\quad [\![C_1']\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l \\
\Rightarrow \{\text{Lemma 5.2.13}\} \\
\quad [\![C_1']\!]_\gamma \,;\, [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \backslash l) \,;\, [\![C_2]\!]_\gamma \\
\Rightarrow \{\text{Lemma 5.2.18}\}
\end{array}
$$

$$
\begin{array}{l}
\quad [\![C_1']\!]_\gamma \,;\, [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \,;\, [\![C_2]\!]_\gamma)\backslash l \\
\Rightarrow \{\text{Definition 5.2.8}\} \\
\quad [\![C_1' \,;\, C_2]\!]_\gamma \equiv [\![C_1 \,;\, C_2]\!]_\gamma \backslash l
\end{array}
$$

- $C_1 \,;\, C_2 \to \sum_i p_i \cdot (\tau, C_i \,;\, C_2)$

$$
\begin{array}{l}
\quad C_1 \,;\, C_2 \to \sum_i p_i \cdot (\tau, C_i \,;\, C_2) \\
\Rightarrow \{\text{Figure 43 entails}\} \\
\quad C_1 \to \sum_i p_i \cdot (\tau, C_i) \\
\Rightarrow \{\text{i.h.}\} \\
\quad [\![C_1]\!]_\gamma \sqsubseteq \sum_i p_i \cdot [\![C_i]\!]_\gamma \\
\Rightarrow \{\text{Lemma 5.2.13}\} \\
\quad [\![C_1 \,;\, C_2]\!]_\gamma \sqsubseteq (\sum_i p_i \cdot [\![C_i]\!]_\gamma) \,;\, [\![C_2]\!]_\gamma \\
\Rightarrow \{\text{Lemma 5.4.1}\} \\
\quad [\![C_1 \,;\, C_2]\!]_\gamma \sqsubseteq \sum_i p_i \cdot [\![C_i \,;\, C_2]\!]_\gamma
\end{array}
$$

- $C_1 \,||\, C_2 \to 1 \cdot (l, C_2)$

$$
\begin{array}{l}
\quad C_1 \,||\, C_2 \xrightarrow{l} C_2 \\
\Rightarrow \{\text{Figure 43 entails}\} \\
\quad C_1 \xrightarrow{l} \checkmark \\
\Rightarrow \{\text{i.h.}\} \\
\quad [\![\checkmark]\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l \\
\Rightarrow \{\text{Lemma 5.2.14}\} \\
\quad [\![\checkmark]\!]_\gamma \,||\, [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \backslash l) \,||\, [\![C_2]\!]_\gamma \\
\Rightarrow \{[\![\checkmark]\!]_\gamma \,||\, [\![C_2]\!]_\gamma \equiv [\![C_2]\!]_\gamma\} \\
\quad [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \backslash l) \,||\, [\![C_2]\!]_\gamma \\
\Rightarrow \{\text{Lemma 5.2.19, Definition 5.2.8}\} \\
\quad [\![C_2]\!]_\gamma \equiv [\![C_1 \,||\, C_2]\!]_\gamma \backslash l
\end{array}
$$

- $C_1 \,||\, C_2 \to 1 \cdot (l, C_1' \,||\, C_2)$

$$
\begin{array}{l}
\quad C_1 \,||\, C_2 \xrightarrow{l} C_1' \,||\, C_2 \\
\Rightarrow \{\text{Figure 43 entails}\} \\
\quad C_1 \xrightarrow{l} C_1' \\
\Rightarrow \{\text{i.h.}\}
\end{array}
$$

$$
\begin{array}{l}
\quad [\![C_1']\!]_\gamma \equiv [\![C_1]\!]_\gamma \backslash l \\
\Rightarrow \{\text{Lemma 5.2.14}\} \\
\quad [\![C_1']\!]_\gamma \,||\, [\![C_2]\!]_\gamma \equiv ([\![C_1]\!]_\gamma \backslash l) \,||\, [\![C_2]\!]_\gamma \\
\Rightarrow \{\text{Lemma 5.2.19, Definition 5.2.8}\} \\
\quad [\![C_1' \,||\, C_2]\!]_\gamma \equiv [\![C_1 \,||\, C_2]\!]_\gamma \backslash l
\end{array}
$$

- $C_1 \,||\, C_2 \to \sum_i p_i \cdot (\tau, C_i \,||\, C_2)$

$$
\begin{array}{l}
\quad C_1 \,||\, C_2 \to \sum_i p_i \cdot (\tau, C_i \,||\, C_2) \\
\Rightarrow \{\text{Figure 43 entails}\} \\
\quad C_1 \to \sum_i p_i \cdot (\tau, C_i) \\
\Rightarrow \{\text{i.h.}\} \\
\quad [\![C_1]\!]_\gamma \sqsubseteq \sum_i p_i \cdot [\![C_i]\!]_\gamma \\
\Rightarrow \{\text{Lemma 5.2.14}\} \\
\quad [\![C_1 \,||\, C_2]\!]_\gamma \sqsubseteq (\sum_i p_i \cdot [\![C_i]\!]_\gamma) \,||\, [\![C_2]\!]_\gamma \\
\Rightarrow \{\text{Lemma 5.4.2, Definition 5.2.8}\} \\
\quad [\![C_1 \,||\, C_2]\!]_\gamma \sqsubseteq \sum_i p_i \cdot [\![C_i \,||\, C_2]\!]_\gamma
\end{array}
$$

- $C_1 \,||\, C_2 \xrightarrow{l} C_1$

$$
\begin{array}{l}
\quad C_1 \,||\, C_2 \xrightarrow{l} C_1 \\
\Rightarrow \{\text{Figure 43 entails}\} \\
\quad C_2 \xrightarrow{l} \checkmark \\
\Rightarrow \{\text{i.h.}\} \\
\quad [\![\checkmark]\!]_\gamma \equiv [\![C_2]\!]_\gamma \backslash l \\
\Rightarrow \{\text{Lemma 5.2.14}\} \\
\quad [\![C_1]\!]_\gamma \,||\, [\![\checkmark]\!]_\gamma \equiv [\![C_1]\!]_\gamma \,||\, ([\![C_2]\!]_\gamma \backslash l) \\
\Rightarrow \{[\![C_1]\!]_\gamma \,||\, [\![\checkmark]\!]_\gamma \equiv [\![C_1]\!]_\gamma\} \\
\quad [\![C_1]\!]_\gamma \equiv [\![C_1]\!]_\gamma \,||\, ([\![C_2]\!]_\gamma \backslash l) \\
\Rightarrow \{\text{Lemma 5.2.19, Definition 5.2.8}\} \\
\quad [\![C_1]\!]_\gamma \equiv [\![C_1 \,||\, C_2]\!]_\gamma \backslash l
\end{array}
$$

- $C_1 \,||\, C_2 \xrightarrow{l} C_1 \,||\, C_2'$

$$
C_1 \,||\, C_2 \xrightarrow{l} C_1 \,||\, C_2'
$$

$$
\begin{array}{l}
\Rightarrow\{\text{Figure 43 entails}\} \\
\quad C_2 \xrightarrow{l} C_2' \\
\Rightarrow\{\text{i.h.}\} \\
\quad [\![C_2']\!]_\gamma \equiv [\![C_2]\!]_\gamma \backslash l \\
\Rightarrow\{\text{Lemma 5.2.14}\} \\
\quad [\![C_1]\!]_\gamma \,||\, [\![C_2']\!]_\gamma \equiv [\![C_1]\!]_\gamma \,||\, ([\![C_2]\!]_\gamma \backslash l) \\
\Rightarrow\{\text{Lemma 5.2.19, Definition 5.2.8}\} \\
\quad [\![C_1 \,||\, C_2']\!]_\gamma \equiv [\![C_1 \,||\, C_2]\!]_\gamma \backslash l
\end{array}
$$

- $C_1 \,||\, C_2 \to \sum_j p_j \cdot (\tau, C_1 \,||\, C_j)$

$$
\begin{array}{l}
\quad C_1 \,||\, C_2 \to \sum_j p_j \cdot (\tau, C_1 \,||\, C_j) \\
\Rightarrow\{\text{Figure 43 entails}\} \\
\quad C_2 \to \sum_j p_j \cdot (\tau, C_j) \\
\Rightarrow\{\text{i.h.}\} \\
\quad [\![C_2]\!]_\gamma \sqsubseteq \sum_j p_j \cdot [\![C_j]\!]_\gamma \\
\Rightarrow\{\text{Lemma 5.2.14}\} \\
\quad [\![C_1 \,||\, C_2]\!]_\gamma \sqsubseteq [\![C_1]\!]_\gamma \,||\, (\sum_j p_j \cdot [\![C_j]\!]_\gamma) \\
\Rightarrow\{\text{Lemma 5.4.2, Definition 5.2.8}\} \\
\quad [\![C_1 \,||\, C_2]\!]_\gamma \sqsubseteq \sum_j p_j \cdot [\![C_1 \,||\, C_j]\!]_\gamma
\end{array}
$$

-

$$
\begin{array}{l}
\quad \mu X.C \to 1 \cdot (l, C'[x \leftarrow \mu X.C]) \\
\Rightarrow\{\text{Figure 43}\} \\
\quad C \to 1 \cdot (l, C') \\
\Rightarrow\{\text{i.h.}\} \\
\quad [\![C]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma \\
\Rightarrow\{\gamma = \gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)\} \\
\quad [\![C]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \backslash l \equiv [\![C']\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \\
\Rightarrow\{\text{Lemma 4.3.21, Lemma 5.3.16}\} \\
\quad [\![\mu X.C]\!]_\gamma \backslash l \equiv [\![C'[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma
\end{array}
$$

-

$$
\mu X.C \to \sum_i p_i \cdot (\tau, C_i[x \leftarrow \mu X.C])
$$

$$
\begin{aligned}
&\Rightarrow\{\text{Figure 43}\}\\
&\quad C \to \sum_i p_i \cdot (\tau, C_i)\\
&\Rightarrow\{\text{i.h.}\}\\
&\quad [\![C]\!]_\gamma \sqsubseteq \sum_i p_i [\![C_i]\!]_\gamma\\
&\Rightarrow\{\gamma = \gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)\}\\
&\quad [\![C]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)} \equiv \sum_I p_i \cdot [\![C_i]\!]_{\gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)}\\
&\Rightarrow\{\text{Lemma 4.3.21, Lemma 5.3.16}\}\\
&\quad [\![\mu X.C]\!]_\gamma \equiv \sum_i p_i [\![C_i[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma
\end{aligned}
$$

□

Proof of Theorem 5.4.7

Proof. Induction over the size of ω_0.

- $|\omega_0| = 1$

  We have that $C \twoheadrightarrow 1 \cdot (l, \checkmark)$. It follows directly that $\{l\} \in \mathcal{C}_{\max}([\![C]\!]_\gamma)$ such that $\emptyset \xrightarrow{l}\subset \{l\}$ and $v(\{l\}) = 1$.

- $|\omega| > 1$

  Let us rewrite $p_0(\omega_0, \checkmark) + \sum_k p_k(\omega_k, C_k)$ as $\sum_{ij} p_{ij}(\omega_{ij}, C_{ij})$

$$
\begin{aligned}
&C \twoheadrightarrow \sum_{ij} p_{ij}(\omega_{ij}, C_{ij})\\
\Rightarrow&\{\text{Figure 44 entails}\}\\
&C \to \sum_i p_i(l', C_i) \qquad \forall i\, C_i \twoheadrightarrow \sum_j p_j(\omega'_{ij}, C_{ij})
\end{aligned}
$$

  We have two cases:

  1. Case $l' \neq \tau$

$$
\begin{aligned}
&C \to 1 \cdot (l', C') \qquad C' \twoheadrightarrow p_0(\omega'_0, \checkmark) + \sum_{j \neq 0} p_j(\omega'_j, C_j)\\
\Rightarrow&\{\text{Lemma 5.4.6, i.h.}\}\\
&[\![C]\!]_\gamma \backslash l' \equiv [\![C']\!]_\gamma \qquad \exists x_0 \in \mathcal{C}_{\max}([\![C']\!]_\gamma) \text{ such that } \emptyset \xrightarrow{\omega'_j}\subset x'_0 \text{ and } p_0 = v'(x'_0)\\
\Rightarrow&\{\text{Definition 5.2.16}\}\\
&\{l'\} \cup x'_0 \in \mathcal{C}_{\max}([\![C]\!]_\gamma) \text{ such that } \emptyset \xrightarrow{l'}\subset \{l'\} \xrightarrow{\omega'_j}\subset \{l'\} \cup x'_0 \text{ and } p_0 = v(\{l'\} \cup x'_0)
\end{aligned}
$$

  2. Case $l' = \tau$

$$
C \to \sum_i p_i(l', C_i) \qquad \exists i\, C_i \twoheadrightarrow p'_0(\omega'_{i0}, \checkmark) + \sum_{j \neq 0} p_j(\omega'_{ij}, C_{ij})
$$

$\Rightarrow$ {Lemma 5.4.6, i.h.}

$$[\![C]\!]_\gamma \sqsubseteq \sum_i p_i [\![C_i]\!]_\gamma$$

$$\exists i, \exists x_{i0} \in \mathcal{C}_{\max}([\![C_i]\!]_\gamma) \text{ such that } \emptyset \xrightarrow{\omega'_{i0}} \subset x_{i0} \text{ and } p'_0 = v_i(x_{i0})$$

Now we have two sub-cases:

(a) Case $C = C_1 +_p C_2$
By Definition 5.2.16, $\exists i, \exists \{\tau\} \cup x_{i0} \in \mathcal{C}_{\max}([\![C]\!]_\gamma)$ such that

$$\emptyset \xrightarrow{\tau} \subset \{\tau\} \xrightarrow{\omega'_{i0}} \subset \{\tau\} \cup x'_{i0},\ v_i(\{\tau\} \cup x'_{i0}) = p_i \cdot p'_0 = p_0$$

(b) Case $C = C_1 \,;\, C_2$ or $C = C_1 \,||\, C_2$
By Remark 2 and Lemma 5.4.3 $\exists i, \exists \{\tau\} \cup x_{i0} \in \mathcal{C}_{\max}([\![C]\!]_\gamma)$ such that

$$\emptyset \xrightarrow{\tau} \subset \{\tau\} \xrightarrow{\omega'_{i0}} \subset \{\tau\} \cup x'_{i0},\ v_i(\{\tau\} \cup x'_{i0}) = p_i \cdot p'_0 = p_0$$

For the recursive command, we only need to add the following sub-case when the size of the word is bigger than one and the transition is made by τ.

- Case $C = \mu X.D$

  By Remark 2 and Lemma 5.4.4

$$\forall i, \exists \{\tau\} \cup x_{i0} \in \mathcal{C}_{\max}([\![\mu X.D]\!]_\gamma)\,.\, \emptyset \xrightarrow{\tau} \subset \{\tau\} \xrightarrow{\omega'_{i0}} \subset \{\tau\} \cup x'_{i0},\ v_i(\{\tau\} \cup x'_{i0}) = p_i \cdot p'_0 = p_0$$

□

Proof of Lemma 5.4.8

Proof.

- $sk \in \mathcal{I}([\![skip]\!]_\gamma)$

  Let $C' = \checkmark$. It follows directly that $skip \to 1 \cdot (sk, \checkmark)$ and that $[\![skip]\!]_\gamma \backslash sk \equiv [\![\checkmark]\!]_\gamma$.

- $a \in \mathcal{I}([\![a]\!]_\gamma)$

  Let $C' = \checkmark$. It follows directly that $a \to 1 \cdot (a, \checkmark)$ and that $[\![a]\!]_\gamma \backslash a \equiv [\![\checkmark]\!]_\gamma$.

- $\tau \in \mathcal{I}([\![C_1 +_p C_2]\!]_\gamma)$

  By Definition 5.2.5 we have that $[\![C_1 +_p C_2]\!]_\gamma = p \cdot [\![C_1]\!]_\gamma + (1-p) \cdot [\![C_2]\!]_\gamma$, hence $\tau \in \mathcal{I}(p \cdot [\![C_1]\!]_\gamma + (1-p) \cdot [\![C_2]\!]_\gamma)$. Let $l \in \mathcal{I}([\![C_1]\!]_\gamma)$ By Definition 5.2.5 and Lemma 5.2.20 we have that $v(\{\tau, l\}) = p \cdot v_1(\{l\}) = p$. It then follows directly that $C_1 +_p C_2 \to p \cdot (\tau, C_1) + (1-p) \cdot (\tau, C_2)$ and $[\![C_1 +_p C_2]\!]_\gamma = p \cdot [\![C_1]\!]_\gamma + (1-p) \cdot [\![C_2]\!]_\gamma$. Similarly we do the same when $l \in \mathcal{I}([\![C_2]\!]_\gamma)$.

- $l' \in \mathcal{I}([\![C_1 \,;\, C_2]\!]_\gamma)$

  We have two cases:

  1. $l' \neq \tau$

     By Definition 5.2.1 we have that $l \in \mathcal{I}([\![C_1]\!]_\gamma)$. By i.h., $\exists C'$ such that $C_1 \to 1 \cdot (l, C')$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma$. We have two cases:

(a) $C' = \checkmark$
We have $C_1 \to 1 \cdot (l, \checkmark)$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![\checkmark]\!]_\gamma$. By the rules in Figure 43, $C_1 \,;\, C_2 \to 1 \cdot (l, C_2)$. By Definition 5.2.1, $([\![C_1]\!]_\gamma \backslash l) \,;\, [\![C_2]\!]_\gamma \equiv [\![\checkmark]\!]_\gamma \,;\, [\![C_2]\!]_\gamma = [\![C_2]\!]_\gamma$.

(b) $C' = C_1'$
We have $C_1 \xrightarrow{l} C_1'$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![C_1']\!]_\gamma$. By the rules in Figure 43, $C_1 \,;\, C_2 \to 1 \cdot (l, C_1' \,;\, C_2)$. By Definition 5.2.1, $([\![C_1]\!]_\gamma \backslash l) \,;\, [\![C_2]\!]_\gamma \equiv [\![C_1']\!]_\gamma \,;\, [\![C_2]\!]_\gamma$. By Definition 5.2.8, $[\![C_1' \,;\, C_2]\!]_\gamma$.

2. $l' = \tau$
We have $\tau \in \mathcal{I}([\![C_1 \,;\, C_2]\!]_\gamma)$, which by Definition 5.2.1 gives us that $\tau \in \mathcal{I}([\![C_1]\!]_\gamma)$. By i.h., $\exists C', C''$ such that $C_1 \to p \cdot (\tau, C') + (1-p) \cdot (\tau, C'')$ with $p = v(\{\tau, e'\})$ and $e' \in \mathcal{I}(C')$, and $[\![C_1]\!]_\gamma \sqsubseteq p \cdot [\![C']\!]_\gamma + (1-p) \cdot [\![C'']\!]_\gamma$. By the rules in Figure 43, we have $C_1 \,;\, C_2 \to p \cdot (\tau, C' \,;\, C_2) + (1-p) \cdot (\tau, C'' \,;\, C_2)$. By Lemma 5.2.13, $[\![C_1]\!]_\gamma \,;\, [\![C_2]\!]_\gamma \sqsubseteq (p \cdot [\![C']\!]_\gamma + (1-p) \cdot [\![C'']\!]_\gamma) \,;\, C_2$. By Lemma 5.4.1 and Definition 5.2.8, $[\![C_1 \,;\, C_2]\!]_\gamma \sqsubseteq p \cdot [\![C' \,;\, C_2]\!]_\gamma + (1-p) \cdot [\![C'' \,;\, C_2]\!]_\gamma$.

- $l \in \mathcal{I}([\![C_1 \,||\, C_2]\!]_\gamma)$
We have two cases:

1. $l' \neq \tau$
By Definition 5.2.3 we have two cases:

(a) $l \in \mathcal{I}([\![C_1]\!]_\gamma)$
By i.h. $\exists C' \,.\, C_1 \to 1 \cdot (l, C')$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma$. By the rules in Figure 43 we have two cases:

i. $C' = \checkmark$
We have $C_1 \to 1 \cdot (l, \checkmark)$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![\checkmark]\!]_\gamma$. By the rules in Figure 19 we have $C_1 \,||\, C_2 \to 1 \cdot (l, C_2)$. By Definition 5.2.3, $([\![C_1]\!]_\gamma \backslash l) \,||\, [\![C_2]\!]_\gamma$. By Lemma 5.2.19 we have $([\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma) \backslash l$. By Definition 5.2.8, $[\![C_1 \,||\, C_2]\!]_\gamma \backslash l$.

ii. $C' = C_1'$
We have $C_1 \to 1 \cdot (l, C_1')$ and $[\![C_1]\!]_\gamma \backslash l \equiv [\![C_1']\!]_\gamma$. By the rules in Figure 43 we have $C_1 \,||\, C_2 \to 1 \cdot (l, C_1' \,||\, C_2)$. By Definition 5.2.3, $([\![C_1]\!]_\gamma \backslash l) \,||\, [\![C_2]\!]_\gamma$. By Lemma 5.2.19 we have $([\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma) \backslash l$. By Definition 5.2.8, $[\![C_1 \,||\, C_2]\!]_\gamma \backslash l$.

(b) $l \in \mathcal{I}([\![C_2]\!]_\gamma)$
By i.h. $\exists C' \,.\, C_2 \to 1 \cdot (l, C')$ and $[\![C_2]\!]_\gamma \backslash l \equiv [\![C']\!]_\gamma$. By the rules in Figure 43 we have two cases:

i. $C' = \checkmark$
We have $C_2 \to 1 \cdot (l, \checkmark)$ and $[\![C_2]\!]_\gamma \backslash l \equiv [\![\checkmark]\!]_\gamma$. By the rules in Figure 43 we have $C_1 \,||\, C_2 \to 1 \cdot (l, C_1)$. By Definition 5.2.3, $[\![C_1]\!]_\gamma \,||\, ([\![C_2]\!]_\gamma \backslash l)$. By Lemma 5.2.19 we have $([\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma) \backslash l$. By Definition 5.2.8, $[\![C_1 \,||\, C_2]\!]_\gamma \backslash l$.

ii. $C' = C_2'$
We have $C_2 \to 1 \cdot (l, C_2')$ and $[\![C_2]\!]_\gamma \backslash l \equiv [\![C_2']\!]_\gamma$. By the rules in Figure 43 we have $C_1 \,||\, C_2 \to 1 \cdot (l, C_1 \,||\, C_2')$. By Definition 5.2.3, $[\![C_1]\!]_\gamma \,||\, ([\![C_2]\!]_\gamma \backslash l)$. By Lemma 5.2.19 we have $([\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma) \backslash l$. By Definition 5.2.8, $[\![C_1 \,||\, C_2]\!] \backslash l$.

2. $l' = \tau$
We have $\tau \in \mathcal{I}([\![C_1 \,||\, C_2]\!]_\gamma)$, which by Definition 5.2.3 entails $\tau \in \mathcal{I}([\![C_1]\!]_\gamma)$ or $\tau \in \mathcal{I}([\![C_2]\!]_\gamma)$. We have two cases:

(a) $\tau \in \mathcal{I}([\![C_1]\!]_\gamma)$
By i.h., $\exists C', C''$ such that $C_1 \to p{\cdot}(\tau, C'){+}(1{-}p){\cdot}(\tau, C'')$ with $p = v(\{\tau, e'\})$ and $e' \in \mathcal{I}([\![C']\!]_\gamma)$, and $[\![C_1]\!]_\gamma \sqsubseteq p \cdot [\![C']\!]_\gamma + (1-p) \cdot [\![C'']\!]_\gamma$. By the rules in Figure 43, we have $C_1 \,||\, C_2 \to p \cdot (\tau, C' \,||\, C_2) + (1-p) \cdot (\tau, C'' \,||\, C_2)$. By Lemma 5.2.14, $[\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma \sqsubseteq (p \cdot [\![C']\!]_\gamma + (1-p) \cdot [\![C'']\!]_\gamma) \,||\, [\![C_2]\!]_\gamma$. By Lemma 5.4.2 and Definition 5.2.8, $[\![C_1 \,||\, C_2]\!]_\gamma \sqsubseteq p \cdot [\![C' \,||\, C_2]\!]_\gamma + (1-p) \cdot [\![C'' \,||\, C_2]\!]_\gamma$.

(b) $\tau \in \mathcal{I}([\![C_2]\!]_\gamma)$
By i.h., $\exists C', C''$ such that $C_2 \to p \cdot (\tau, C') + (1-p) \cdot (\tau, C'')$ with $p = v(\{\tau, e'\})$ and $e' \in \mathcal{I}([\![C']\!]_\gamma)$, and $[\![C_2]\!]_\gamma \sqsubseteq p \cdot [\![C']\!]_\gamma + (1-p) \cdot [\![C'']\!]_\gamma$. By the rules in Figure 43, we have $C_1 \,||\, C_2 \to p \cdot (\tau, C_1 \,||\, C') + (1-p) \cdot (\tau, C_1 \,||\, C'')$. By Lemma 5.2.14, $[\![C_1]\!]_\gamma \,||\, [\![C_2]\!]_\gamma \sqsubseteq C_1 \,||\, (p{\cdot}[\![C']\!]_\gamma{+}(1{-}p){\cdot}[\![C'']\!]_\gamma)$. By Lemma 5.4.2 and Definition 5.2.8, $[\![C_1 \,||\, C_2]\!]_\gamma \sqsubseteq p \cdot [\![C_1 \,||\, C']\!]_\gamma + (1-p) \cdot [\![C_1 \,||\, C'']\!]_\gamma$.

- $l' \neq \tau \in \mathcal{I}([\![\mu X.C]\!]_\gamma)$

  By Definition 5.3.14 and Definition 5.3.5, $l' \in \mathcal{I}([\![C]\!]_\gamma)$. By i.h., $\exists C'$ such that $C \to 1 \cdot (l', C')$ and $[\![C]\!]_\gamma \backslash l' \equiv [\![C']\!]_\gamma$. By Figure 43 and by letting $\gamma = \gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)$ and Lemma 4.3.21 and Lemma 5.3.16, $\mu X.C \to 1 \cdot (l', C'[X \leftarrow \mu X.C])$ and $[\![\mu X.C]\!]_\gamma \backslash l' \equiv [\![C'[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma$.

- $l' = \tau \in \mathcal{I}([\![\mu X.C]\!]_\gamma)$

  By Definition 5.3.14 and Definition 5.3.5, $l' \in \mathcal{I}([\![C]\!]_\gamma)$. By i.h. $\exists C', C''$, $\exists e \in \mathcal{I}([\![C']\!]_\gamma)$ such that $C \to p{\cdot}(\tau, C'){+}(1{-}p){\cdot}(\tau, C'')$ with $p = v(\{\tau, e\})$ and $[\![C]\!]_\gamma \sqsubseteq p{\cdot}[\![C']\!]_\gamma{+}(1{-}p){\cdot}[\![C'']\!]_\gamma$. By Figure 43 and by letting $\gamma = \gamma(X \leftarrow [\![\mu X.C]\!]_\gamma)$ and Lemma 4.3.21 and Lemma 5.3.16, $\mu X.C \to p \cdot (\tau, C'[X \leftarrow \mu X.C]) + (1-p) \cdot (\tau, C''[X \leftarrow \mu X.C])$ and $[\![\mu X.C]\!]_\gamma \sqsubseteq p \cdot [\![C'[X \leftarrow [\![\mu X.C]\!]_\gamma]]\!]_\gamma + (1-p) \cdot [\![C''[X \leftarrow [\![\mu X.C]\!]_\gamma]\!]_\gamma$.

□

Proof of Theorem 5.4.9

Proof. Induction over the size of ω_{x_0}.

- $|\omega_{x_0}| = 1$

  We have $\{l\} \in \mathcal{C}_{\mathsf{max}}([\![C]\!]_\gamma)$. It follows directly that $C \twoheadrightarrow 1 \cdot (l, \checkmark)$ and $v(\{l\}) = 1$.

- $|\omega_{x_0}| > 1$

  We have $x_0 \in \mathcal{C}_{\mathsf{max}}([\![C]\!]_\gamma)$. We know that $\omega_{x_0} = l_0 l_1 \ldots l_n$. Hence $\emptyset \xrightarrow{l_0}\subset \{l_0\} \xrightarrow{\omega_{x'_0}}\subset \{l_0\} \cup x'_0$. We then have $l_0 \in \mathcal{I}([\![C]\!]_\gamma)$. By Lemma 5.4.8 we have two cases:

  1. $l_0 \neq \tau$
     Hence $C \to 1{\cdot}(l_0, C')$ and $[\![C]\!]_\gamma \backslash l_0 \equiv [\![C']\!]_\gamma$. By Definition 5.2.16, $x_0 \backslash l_0 \in \mathcal{C}_{\mathsf{max}}([\![C']\!]_\gamma)$ such that $\emptyset \xrightarrow{\omega_{x'_0}}\subset x_0 \backslash l_0$. By i.h., $C' \twoheadrightarrow v(x_0 \backslash l_0)(\omega_{x'_0}, \checkmark) + \sum_k p_k(\omega_k, C_k)$, for some p_k, ω_k, C_k. By Figure 44, $C \twoheadrightarrow v(x_0 \backslash l_0)(l_0 : \omega_{x'_0}, \checkmark) + \sum_k p_k(l_0 : \omega_k, C_k)$, for some p_k, ω_k, C_k.
  2. $l_0 = \tau$
     Hence $C \to p \cdot (\tau, C') + (1-p) \cdot (\tau, C'')$ and $[\![C]\!]_\gamma \sqsubseteq p[\![C']\!]_\gamma + (1-p)[\![C'']\!]_\gamma$. Now we have two sub-cases:

(a) Case $C = C' +_p C''$
We then have $[\![C]\!]_\gamma = p[\![C']\!]_\gamma + (1-p)[\![C'']\!]_\gamma$, by Definition 5.2.8, and consequently $x_0 \in \mathcal{C}_{\max}(p[\![C']\!]_\gamma + (1-p)[\![C'']\!]_\gamma)$. By Definition 5.2.5, $x_0 \backslash \tau \in \mathcal{C}_{\max}([\![C']\!]_\gamma)$ or $x_0 \backslash \tau \in \mathcal{C}_{\max}([\![C'']\!]_\gamma)$. We only consider the former, since the latter has a similar reasoning. By i.h., $C' \twoheadrightarrow v(x_1)(w_{x_1}, \checkmark) + \sum_n p_n(\omega_n, C_n)$, for some p_n, ω_n, C_n. By Lemma 5.4.5, $C'' \twoheadrightarrow \sum_m p_m(\omega_m, C_m)$. By Figure 44,

$$C \twoheadrightarrow p \cdot \left(v(x_1)(\tau : w_{x_1}, \checkmark) + \sum_n p_n(\tau : \omega_n, C_n) \right) + (1-p) \cdot \sum_m p_m(\tau : \omega_m, C_m)$$

(b) Case $C = C_1 \,;\, C_2$ or $C = C_1 \,||\, C_2$
By Lemma 5.4.3, $x_0 \in \mathcal{C}_{\max}(p[\![C']\!]_\gamma + (1-p)[\![C'']\!]_\gamma)$. By Definition 5.2.5, $x_0 \backslash \tau \in \mathcal{C}_{\max}([\![C']\!]_\gamma)$ or $x_0 \backslash \tau \in \mathcal{C}_{\max}([\![C'']\!]_\gamma)$. We only consider the former, since the latter has a similar reasoning. By i.h., $C' \twoheadrightarrow v(x_1)(w_{x_1}, \checkmark) + \sum_n p_n(\omega_n, C_n)$, for some p_n, ω_n, C_n. By Lemma 5.4.5, $C'' \twoheadrightarrow \sum_m p_m(\omega_m, C_m)$. By Figure 44,

$$C \twoheadrightarrow p \cdot \left(v(x_1)(\tau : w_{x_1}, \checkmark) + \sum_n p_n(\tau : \omega_n, C_n) \right) + (1-p) \cdot \sum_m p_m(\tau : \omega_m, C_m)$$

For the recursive command, we only need to add the following sub-case when the size of the word is bigger than one and the transition is made by τ.

- Case $C = \mu X.D$

  By Lemma 5.4.4, $x_0 \in \mathcal{C}_{\max}(p[\![C'[X \leftarrow \mu X.D]]\!]_\gamma + (1-p)[\![C''[X \leftarrow \mu X.D]]\!]_\gamma)$. By Definition 5.2.5, $x_0 \backslash \tau \in \mathcal{C}_{\max}([\![C'[X \leftarrow \mu X.D]]\!]_\gamma)$ or $x_0 \backslash \tau \in \mathcal{C}_{\max}([\![C''[X \leftarrow \mu X.D]]\!]_\gamma)$. We only consider the former, since the latter has a similar reasoning. By i.h., $C'[X \leftarrow \mu X.D] \twoheadrightarrow v(x_1)(w_{x_1}, \checkmark) + \sum_n p_n(\omega_n, C_n)$, for some p_n, ω_n, C_n. By Lemma 5.4.5, $C''[X \leftarrow \mu X.D] \twoheadrightarrow \sum_m p_m(\omega_m, C_m)$. By Figure 44,

$$C \twoheadrightarrow p \cdot \left(v(x_1)(\tau : w_{x_1}, \checkmark) + \sum_n p_n(\tau : \omega_n, C_n) \right) + (1-p) \cdot \sum_m p_m(\tau : \omega_m, C_m)$$

□

A.3 Proofs of Chapter 6

Proof of Lemma 6.2.2

Proof. Let

$$\begin{aligned}\mathsf{U}_1 &= (E_1, \leq_1, \#_1, Q_1)\\ \mathsf{U}_2 &= (E_2, \leq_2, \#_2, Q_2)\\ \mathsf{U}_1 \,;\, \mathsf{U}_2 &= (E, \leq, \#, Q)\end{aligned}$$

Due to Lemma 4.2.2 we only need show the conditions added in the definition of unitary event structures.

1. $\forall e, e' \in E,\ e \text{ co } e' \Rightarrow [Q(e), Q(e')] = 0$

   Since e co e' only if $e, e' \in E_1$ or $e, e' \in E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{U}_1)$ we are done.

2. $\rightsquigarrow$is transitive

   It follows directly since $\rightsquigarrow$only occurs between events of the same set of events.

3. $\forall e \in E \sum_{e' \in [e]} Q(e')$ is unitary

   We have two cases, since there is no minimal conflict between events in E_1 and $E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{U}_1)$:

   (a) $\forall e \in E_1$

   Since E_1 is a unitary event structure, we are done.

   (b) $\forall e \in E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{U}_1)$

   Since E_2 is a unitary event structure, we are done.

□

Proof of Lemma 6.2.4

Proof. Let

$$\begin{aligned}\mathsf{U}_1 &= (E_1, \leq_1, \#_1, Q_1)\\ \mathsf{U}_2 &= (E_2, \leq_2, \#_2, Q_2)\\ \mathsf{U}_1 \,||\, \mathsf{U}_2 &= (E, \leq, \#, Q)\end{aligned}$$

Due to Lemma 4.2.4 we only need show the conditions added in the definition of unitary event structures.

1. $\forall e, e' \in E,\ e \text{ co } e' \Rightarrow [Q(e), Q(e')] = 0$

   We have two cases:

   (a) $e, e' \in E_1$ or $e, e' \in E_2$

   The condition trivially holds, since U_1 and U_2 are unitary event structures.

   (b) $e \in E_1$ and $e' \in E_2$

   It follows directly from Definition 6.2.3.

2. $\sim\!\!\!\sim$is transitive

   It follows directly since the parallel composition does not create new conflicts and that the conflict relation is inherited from U_1 and U_2 which are unitary event structures.

3. $\forall e \in E, \sum_{e' \in [e]} Q(e')$ is unitary

   Since there is no minimal conflict between events in E_1 and E_2, it follows directly that if $\forall e \in E_1$ or $\forall e \in E_2$ the condition holds since U_1 and U_2 are unitary event structures.

□

Proof of Lemma 6.2.6

Proof. Let

$$\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$$
$$\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$$
$$\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2) = (E, \leq, \#, Q)$$

We need to show that $\leq$ is a partial order and that # is symmetric and irreflexive.

- $\leq$ is a partial order:
  - Reflexivity ($e \leq e$): we have the following cases:
    1. Case $e \in \{\tau_0^n, \tau_1^n\}$. It follows directly from Definition 6.2.5 that either $\tau_0^n \leq \tau_0^n$ or $\tau_1^n \leq \tau_1^n$.
    2. Case $e_1 \leq_1 e_1'$ or $e_2 \leq_2 e_2'$. Since $\leq_1$ and $\leq_2$ are partial orders, we are done.
  - Transitivity ($e \leq e'$ and $e' \leq e''$ then $e \leq e''$): we have the following cases:
    1. Case ($e \leq_1 e'$ and $e' \leq_1 e''$) or ($e \leq_2 e'$ and $e' \leq_2 e''$). Since $\leq_1$ and $\leq_2$ are partial orders, we have $e \leq_1 e''$ or $e \leq_2 e''$. Hence $e \leq e''$.
    2. Case $e = \tau_0^n$ and $e', e'' \in E_1$. We have $\tau_0^n \leq e'$ and $e' \leq e''$. Since $e'' \in E_1$ then $\tau_n^0 \leq e''$.
    3. Case $e = \tau_1^n$ and $e', e'' \in E_2$. We have $\tau_1^n \leq e'$ and $e' \leq e''$. Since $e'' \in E_2$ then $\tau_n^1 \leq e''$.
  - Antisymmetry ($e \leq e'$ and $e' \leq e$ then $e' = e$): we have two cases: ($e \leq_1 e'$ and $e' \leq_1 e$) or ($e \leq_2 e'$ and $e' \leq_2 e$). Since $\leq_1$ and $\leq_2$ are partial orders, we have $e = e'$. Since there are no more cases we are done.

  Hence $\leq$ is a partial order.

- # is symmetric and irreflexive:
  - Symmetric (if $e\#e'$ then $e'\#e$): we have the following cases
    1. Case $e\#_1 e'$ or $e\#_2 e'$. Since $\#_1$ and $\#_2$ are symmetric then we have $e'\#_1 e$ or $e'\#_2 e$. Thus $e'\#e$.
    2. Case ($e = \tau_0^n$ or $e \in E_1$) and ($e' = \tau_1^n$ or $e' \in E_2$). It follows directly from Definition 6.2.5.
    3. Case ($e = \tau_1^n$ or $e \in E_2$) and ($e' = \tau_0^n$ or $e' \in E_1$). It follows directly from Definition 6.2.5.

– Irreflexive ($\neg(e\#e)$): We have either $\neg(e\#_1 e)$ or $\neg(e\#_2 e)$. Since $\#_1$ and $\#_2$ are irreflexive then $\neg(e\#e)$. There are no more cases since either $e \in E_1$ or $e \in E_2$.

Hence # is symmetric and irreflexive.

We need to prove:

1. $\{e' \mid e' \leq e\}$ is finite

   We have four cases:

   (a) $e = \tau_0^n$
   It follows directly that $\{e' \mid e' \leq \tau_0^n\} = \{\tau_0^n\}$ since $\tau_0^n \in \mathcal{I}(\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2))$.

   (b) $e = \tau_1^n$
   It follows directly that $\{e' \mid e' \leq \tau_1^n\} = \{\tau_1^n\}$ since $\tau_1^n \in \mathcal{I}(\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2))$.

   (c) $e \in E_1$
   We have that $\{e' \mid e' \leq e\} = \{\tau_0^n\} \cup \{e' \mid e' \leq_1 e\}$. Since U_1 is a unitary event structure, then we know that $\{e' \mid e' \leq_1 e\}$ is finite. Hence $\{\tau_0^n\} \cup \{e' \mid e' \leq_1 e\}$ is finite.

   (d) $e \in E_2$
   We have that $\{e' \mid e' \leq e\} = \{\tau_1^n\} \cup \{e' \mid e' \leq_2 e\}$. Since U_2 is a unitary event structure, then we know that $\{e' \mid e' \leq_2 e\}$ is finite. Hence $\{\tau_1^n\} \cup \{e' \mid e' \leq_2 e\}$ is finite.

2. $e\#e' \leq e'' \Rightarrow e\#e''$ It follows directly by Definition 6.2.5 that $e\#e''$.

3. e co $e' \Rightarrow [Q(e), Q(e')] = 0$

   The concurrent events are either in U_1 or in U_2, which are unitary event structures, hence the condition trivially holds.

4. $\rightsquigarrow$is transitive

   It follows directly since the conflict relation is inherited from $\mathsf{U}_1, \mathsf{U}_2$, which are unitary event structures, and from the fact that the new events, τ_0^n and τ_1^n, are in minimal conflict between them, *i.e.* $\tau_0^n \rightsquigarrow \tau_1^n$.

5. $\forall e \in E, \sum_{e' \in [e]} Q(e')$ is unitary

   We have two cases (since if $e_1 \in E_1, e_2 \in E_2$ then $\neg(e_1 \rightsquigarrow e_2)$):

   (a) $e = \tau_0^n, e' = \tau_1^n$ or vice-versa
   It follows directly from Definition 6.2.5 that $Q(\tau_0^n) + Q(\tau_1^n) = Id$, which is unitary.

   (b) $\forall e \in E_1$ or $\forall e \in E_2$
   It follows directly from U_1 and U_2 being unitary event structures.

□

Proof of Lemma 6.2.11

Proof. It follows directly from Definition 6.2.3. □

Proof of Lemma 6.2.12

Proof. Let

$$\begin{aligned}
\mathsf{U}_1 &= (E_1, \leq_1, \#_1, Q_1)\\
\mathsf{U}'_1 &= (E'_1, \leq'_1, \#'_1, Q'_1)\\
\mathsf{U}_2 &= (E_2, \leq_2, \#_2, Q_2)\\
\mathsf{U}'_2 &= (E'_2, \leq'_2, \#'_2, Q'_2)\\
\mathsf{U}_1 \,;\, \mathsf{U}_2 &= (E, \leq, \#, Q)\\
\mathsf{U}'_1 \,;\, \mathsf{U}'_2 &= (E', \leq', \#', Q')
\end{aligned}$$

such that $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$ and $\mathsf{U}_2 \sqsubseteq \mathsf{U}'_2$.

Due to Lemma 4.2.14 we only need to show $\forall e \in E \,.\, Q(e) = Q'(f(e))$.

Let $e \in E$. We have two cases:

1. $e \in E_1$

   Since $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$, it follows directly that $Q(e) = Q_1(e) = Q'_1(e) = Q'(e)$.

2. $e = (e_2, x) \in E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{U}_1)$

   By Definition 6.2.1, we know that $Q(e_2, x) = Q_2(e_2)$. Since $\mathsf{U}_2 \sqsubseteq \mathsf{U}'_2$, then $Q_2(e_2) = Q'_2(e_2)$. By Definition 6.2.1, we have that $Q'_2(e_2) = Q'(e_2, m_1(x))$.

□

Proof of Lemma 6.2.13

Proof. Let

$$\begin{aligned}
\mathsf{U}_1 &= (E_1, \leq_1, \#_1, Q_1)\\
\mathsf{U}'_1 &= (E'_1, \leq'_1, \#'_1, Q'_1)\\
\mathsf{U}_2 &= (E_2, \leq_2, \#_2, Q_2)\\
\mathsf{U}'_2 &= (E'_2, \leq'_2, \#'_2, Q'_2)\\
\mathsf{U}_1 \,||\, \mathsf{U}_2 &= (E, \leq, \#, Q)\\
\mathsf{U}'_1 \,||\, \mathsf{U}'_2 &= (E', \leq', \#', Q')
\end{aligned}$$

such that $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$ and $\mathsf{U}_2 \sqsubseteq \mathsf{U}'_2$.

Due to Lemma 4.2.15 we only need to show $\forall e \in E \,.\, Q(e) = Q'(f(e))$.

Let $e \in E$. If $e \in E_1$ then by Definition 6.2.3 we have $Q(e) = Q_1(e)$, which by $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$ gives $Q_1(e) = Q'_1(e)$ that by Definition 6.2.3 gives $Q'_1(e) = Q'(e)$. Similarly when $e \in E_2$. □

Proof of Lemma 6.2.14

Proof. Let

$$\begin{aligned}
\mathsf{U}_1 &= (E_1, \leq_1, \#_1, Q_1)\\
\mathsf{U}'_1 &= (E'_1, \leq'_1, \#'_1, Q'_1)\\
\mathsf{U}_2 &= (E_2, \leq_2, \#_2, Q_2)\\
\mathsf{U}'_2 &= (E'_2, \leq'_2, \#'_2, Q'_2)
\end{aligned}$$

$$\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2) = (E, \leq, \#, Q)$$
$$\mathtt{M}(n, \mathsf{U}'_1, \mathsf{U}'_2) = (E', \leq', \#', Q')$$

such that $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$ and $\mathsf{U}_2 \sqsubseteq \mathsf{U}'_2$.

We have to show that:

1. We start by defining $f : E \to E'$ such that

$$f(e) = \begin{cases} e & \text{if } e \in \mathcal{I}(\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2)) \\ f_1(e) & \text{if } e \in E_1 \\ f_2(e) & \text{if } e \in E_2 \end{cases}$$

   It is straightforward to see that f is injective

2. $\pi(f(e)) = \pi(e)$

   If $e \in E_1$ or $e \in E_2$, we are done since $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$ and $\mathsf{U}_2 \sqsubseteq \mathsf{U}'_2$. If $e \in \mathcal{I}(\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2))$, then $\pi(f(e)) = \pi(e)$, since $f(e) = e$.

3. $e \leq e' \Leftrightarrow f(e) \leq' f(e')$

   If $e \in E_1$ or $e \in E_2$, we are done since $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$ and $\mathsf{U}_2 \sqsubseteq \mathsf{U}'_2$. If $e \in \mathcal{I}(\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2))$, since $f(e) = e$. Consequently $e \leq e' \Leftrightarrow e \leq' f(e')$, which is trivially satisfied by Definition 6.2.5.

4. $e \# e' \Leftrightarrow f(e) \#' f(e')$

   This case is similar to that of Lemma 4.2.16.

5. $\forall e \in E \,.\, Q(e) = Q'(f(e))$

   If $e = \tau_0^n$ or $e = \tau_1^n$ then we are done, since $f(e) = e$ and, consequently, $Q(e) = Q'(e)$. If $e \in E_1$ or $e \in E_2$ we are done since $\mathsf{U}_1 \sqsubseteq \mathsf{U}'_1$ and $\mathsf{U}_2 \sqsubseteq \mathsf{U}'_2$.

The first three conditions follow directly from Definition 6.2.5. For the last condition we argue as follows: □

Proof of Lemma 6.2.16

Proof. Let $\mathsf{U} = (E, \leq, \#, Q)$ and $\mathsf{U} \backslash a = (E', \leq', \#', Q')$.

Due to Lemma 4.2.20 we only need to check the conditions added in the definition of unitary event structures.

1. $\forall e, e' \in E',\ e \text{ co } e' \Rightarrow [Q'(e), Q'(e')] = 0$

   It follows directly from Definition 6.2.15 that $[Q'(e), Q'(e')] = [Q|_{E'}(e), Q|_{E'}(e')] = 0$

2. $\rightsquigarrow$is transitive

   It follows directly since the conflict relation $\#'$ is the restriction of $\#$ to the events of E'.

3. $\forall e \in E, \sum_{e' \in [e]} Q(e')$ is unitary

   It follows directly from Definition 6.2.15 that $\sum_{e' \in [e]} Q'(e') = \sum_{e' \in [e]} Q|_{E'}(e')$ which is unitary.

□

Proof of Lemma 6.2.17

Proof. Let

$$\begin{aligned}
&\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)\\
&\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)\\
&\mathsf{U}_1 \,;\, \mathsf{U}_2 = (E_{1;2}, \leq_{1;2}, \#_{1;2}, Q_{1;2})\\
&(\mathsf{U}_1 \,;\, \mathsf{U}_2)\backslash l = (E, \leq, \#, Q)\\
&\mathsf{U}_1\backslash l = (E_1^l, \leq_1^l, \#_1^l, Q_1^l)\\
&(\mathsf{U}_1\backslash l) \,;\, \mathsf{U}_2 = (E', \leq', \#', Q')\\
&l \in \mathcal{I}(\mathsf{U}_1 \,;\, \mathsf{U}_2)
\end{aligned}$$

Due to Lemma 4.2.22 we focus only on the quantum part.

- $(\mathsf{U}_1 \,;\, \mathsf{U}_2)\backslash l \sqsubseteq (\mathsf{U}_1\backslash l) \,;\, \mathsf{U}_2$

  We need to show $\forall e \in E \,.\, Q(e) = Q'(f(e))$. Let $e \in E$. By Definition 6.2.15, $Q(e) = Q_{1;2}|_E(e)$. By Definition 6.2.1 we have two cases:

  - $e \in E_1$

    By Definition 6.2.1 and since $e \neq l$ and $\neg(e\#l)$, which gives $e \in E_1^l$, then $Q_{1;2}|_E(e) = Q_1|_{E_1^l}(e)$. By Definition 6.2.15, $Q_1|_{E_1^l}(e) = Q_1^l(e)$. By Definition 6.2.1, $Q_1^l(e) = Q'(e)$.
  - $e = (e_2, x) \in E_2 \times \mathcal{C}_{\mathsf{max}}(\mathsf{U}_1)$

    By Definition 6.2.1 and since $l \notin E_2$, $Q_{1;2}|_E(e_2, x) = Q_2(e_2)$. Hence, again by Definition 6.2.1, $Q_2(e_2) = Q'(e_2, m_1(x))$.

- $(\mathsf{U}_1\backslash l) \,;\, \mathsf{U}_2 \sqsubseteq (\mathsf{U}_1 \,;\, \mathsf{U}_2)\backslash l$

  Similar reasoning to the previous bullet.

□

Proof of Lemma 6.2.18

Proof. Let $l \in \mathcal{I}(\mathsf{E}_1)$ and

$$\begin{aligned}
&\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)\\
&\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)\\
&\mathsf{U}_1 \,||\, \mathsf{U}_2 = (E_{1\,||\,2}, \leq_{1\,||\,2}, \#_{1\,||\,2}, Q_{1\,||\,2})\\
&(\mathsf{U}_1 \,||\, \mathsf{U}_2)\backslash l = (E, \leq, \#, Q)\\
&\mathsf{U}_1\backslash l = (E_1^l, \leq_1^l, \#_1^l, Q_1^l)\\
&\mathsf{U}_2\backslash l = (E_2^l, \leq_2^l, \#_2^l, Q_2^l)\\
&(\mathsf{U}_1\backslash l) \,||\, \mathsf{U}_2 = (E', \leq', \#', Q')\\
&l \in \mathcal{I}(\mathsf{U}_1 \,||\, \mathsf{U}_2)
\end{aligned}$$

Due to Lemma 4.2.23 we focus only on the quantum part.

- $(\mathsf{U}_1 \,||\, \mathsf{U}_2)\backslash l \sqsubseteq (\mathsf{U}_1\backslash l) \,||\, \mathsf{U}_2$

  We need to show $\forall e \in E \,.\, Q(e) = Q'(f(e))$. Let $e \in E$. By Definition 6.2.15, $Q(e) = Q_{1\,||\,2}|_E(e)$. By Definition 6.2.3 we have two cases:

1. $e \in E_1$
   Since $l \in \mathcal{I}(\mathsf{U}_1)$ then $e \neq l$ and $\neg(e\#l)$. Thus $e \in E_1^l$. Consequently, $Q_{1\,||\,2}|_E(e) = Q|_{E_1^l}(e)$. By Definition 6.2.15, $Q1|_{E_1^l}(e) = Q_1^l(e)$. By Definition 6.2.3, $Q_1^l(e) = Q'(e)$.
2. $e \in E_2$
   Then $Q_{1\,||\,2}|_E(e) = Q_2(e)$. By Definition 6.2.3, $Q_2(e) = Q'(e)$.

- $(\mathsf{U}_1 \backslash l) \,||\, \mathsf{U}_2 \sqsubseteq (\mathsf{U}_1 \,||\, \mathsf{U}_2) \backslash l$

  We need to show $\forall e \in E' \,.\, Q'(e) = Q(e)$. By Definition 6.2.3 we have two cases:

  1. $e \in E_1^l$
     Then $Q'(e) = Q_1^l(e)$. By Definition 6.2.15, $Q_1^l(e) = Q_1|_{E_1^l}(e)$. By Definition 6.2.3, $Q_1|_{E_1^l}(e) = Q_{1\,||\,2}|_{E_1^l \uplus E_2}$. By Definition 6.2.15, $Q_{1\,||\,2}|_{E_1^l \uplus E_2} = Q(e)$.
  2. $e \in E_2^l$
     Then $Q'(e) = Q_2(e)$. By Definition 6.2.3, $Q_2(e) = Q_{1\,||\,2}(e)$. By Definition 6.2.15, $Q_{1\,||\,2}(e) = Q(e)$.

□

Proof of Lemma 6.2.19

Proof. Let

$$\begin{aligned}
&\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)\\
&\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)\\
&\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2) = (E, \leq, \#, Q)\\
&(\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2))\backslash l = (E', \leq', \#', Q')
\end{aligned}$$

Let us focus on the case where $l = \tau_0^n$, since the reasoning when $l = \tau_1^n$ is equal to the one below.

- $(\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2))\backslash\tau_0^n \sqsubseteq \mathsf{U}_1$

  1. We start by defining $f : E' \to E_1$ as being the identity, which is injective.
  2. $\pi(f(e)) = \pi(e)$
     It follows directly since f is the identity.
  3. $e \leq' e' \Leftrightarrow f(e) \leq_1 f(e')$
     Since f is the identity, $e \leq' e' \Leftrightarrow e \leq_1 e'$. It then follows directly from Definition 6.2.15 and Definition 6.2.5.
  4. $e\#'e' \Leftrightarrow f(e)\#_1 f(e')$
     Since f is the identity, $e\#'e' \Leftrightarrow e\#_1 e'$. It then follows directly from Definition 6.2.15 and Definition 6.2.5.
  5. $\forall e \in E' \,.\, Q'(e) = Q_1(f(e))$
     Since f is the identity, $Q'(e) = Q_1(e)$. Let $e \in E'$. By Definition 6.2.15, $Q'(e) = Q|_{E'}(e)$. Since $\tau_0^n \leq e$ we know that $e \in E_1$, hence by Definition 6.2.5, $Q|_{E'}(e) = Q_1(e)$.

- $\mathsf{U}_1 \sqsubseteq (\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2))\backslash\tau_0^n$

  1. We start by defining $f : E_1 \to E'$ as being the identity, which is injective.

2. $\pi(f(e)) = \pi(e)$
   It follows directly since f is the identity.
3. $e \leq_1 e' \Leftrightarrow f(e) \leq' f(e')$
   Since f is the identity, $e \leq_1 e' \Leftrightarrow e \leq' e'$. It then follows directly from Definition 6.2.15 and Definition 6.2.5.
4. $e\#_1 e' \Leftrightarrow f(e)\#' f(e')$
   Since f is the identity, $e\#_1 e' \Leftrightarrow e\#' e'$. It then follows directly from Definition 6.2.15 and Definition 6.2.5.
5. $\forall e \in E_1 \,.\, Q_1(e) = Q'(f(e))$
   Since f is the identity, $Q_1(e) = Q'(e)$. Let $e \in E_1$. By Definition 6.2.5, $Q(e) = Q_1(e)$. Since $e \neq l$ and $\neg(e\#l)$, then $Q'(e) = Q(e)$. Thus $Q_1(e) = Q'(e)$.

□

Proof of Lemma 6.3.3

Proof. Due to Lemma 4.3.4 we only focus on the condition of the quantum operators. Let $\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$, $\mathsf{U}_2 = (E_2, \leq_2, \#_2, Q_2)$, and $\mathsf{U}_3 = (E_3, \leq_3, \#_3, Q_3)$ be quantum event structures.

- Reflexivity: $\mathsf{U}_1 \trianglelefteq \mathsf{U}_1$
  It follows directly that $\forall e \in E_1 \,.\, Q_1(e) = Q_1(e)$
- Transitivity : $\mathsf{U}_1 \trianglelefteq \mathsf{U}_2$, $\mathsf{U}_2 \trianglelefteq \mathsf{U}_3 \Rightarrow \mathsf{U}_1 \trianglelefteq \mathsf{U}_3$
  From $\mathsf{U}_1 \trianglelefteq \mathsf{U}_2$, $\forall e \in E_1 \,.\, Q_1(e) = Q_2(e)$. From $\mathsf{U}_2 \trianglelefteq \mathsf{U}_3$, $\forall e \in E_2 \,.\, Q_2(e) = Q_3(e)$. Hence $\forall e \in E_1 \,.\, Q_1(e) = Q_2(e) = Q_3(e)$.
- Antisymmetry: $\mathsf{U}_1 \trianglelefteq \mathsf{U}_2$, $\mathsf{U}_2 \trianglelefteq \mathsf{U}_1 \Rightarrow \mathsf{U}_1 = \mathsf{U}_2$
  From Lemma 4.3.4 we know that $E_1 = E_2$. Hence it follows directly that $\mathsf{U}_1 = \mathsf{U}_2$.

□

Proof of Lemma 6.3.4

Proof. We begin by showing that $\bot$ is a unitary event structure. We already know that $(\emptyset, \emptyset, \emptyset)$ is an event structure, hence it lacks to verify the conditions regarding the quantum operator. However, since there are no events, the conditions trivially holds.

To show that $\bot$ is the least element, consider any unitary event structure U. We need to show $\bot \trianglelefteq \mathsf{U}$. Due to Lemma 4.3.5 we only focus on the quantum operator. We need to show that for every event in $\bot$, its mapping through ! and Q is the same. We show it by contradiction. Thus, we need to find an event $e \in \emptyset$ such that its mapping through ! and Q is not the same. However, there are no events in $\bot$. Thus the condition holds. □

Proof of Lemma 6.3.6

Proof. Due to Lemma 4.3.7 we focus on the quantum operator condition.

- $\forall e, e' \in E^\omega \,,\; e \text{ co } e' \Rightarrow [Q^\omega(e), Q^\omega(e')] = 0$
  Let $e, e' \in E^\omega$ such that e co e'. We have two cases:

1. $e, e' \in E_n$

   By Definition 6.3.5 we have $Q^\omega(e) = Q_n(e)$ and $Q^\omega(e') = Q_n(e')$ and since U_n is a unitary event structure we are done.

2. $e \in E_n$ and $e' \in E_m$ such that $\mathsf{U}_n \trianglelefteq \mathsf{U}_m$

   By Definition 6.3.5 we have $Q^\omega(e) = Q_n(e)$ and $Q^\omega(e') = Q_m(e')$. From $\mathsf{U}_n \trianglelefteq \mathsf{U}_m$, we have that $Q_n(e) = Q_m(e)$ and since U_m is a unitary event structure, then we are done.

- $\rightsquigarrow$is transitive

  We want to show that for $e, e', e'' \in E^\omega$ if $e \rightsquigarrow e'$ and $e' \rightsquigarrow e''$ then $e \rightsquigarrow e''$. According to Definition 6.3.5 we have three cases:

  1. $e, e', e'' \in E_n$

     Then we are done, since U_n is a unitary event structure.

  2. $e \in E_n$ and $e', e'' \in E_m$ such that $\mathsf{U}_n \trianglelefteq \mathsf{U}_m$

     From $\mathsf{U}_n \trianglelefteq \mathsf{U}_m$ we know that $E_n \subseteq E_m$, hence $e \in E_m$. Since U_m is a unitary event structure we are done.

  3. $e \in E_n$, $e' \in E_m$, and $e'' \in E_k$ such that $\mathsf{U}_n \trianglelefteq \mathsf{U}_m \trianglelefteq \mathsf{U}_k$

     From $\mathsf{U}_n \trianglelefteq \mathsf{U}_m \trianglelefteq \mathsf{U}_k$ we have that $E_n \subseteq E_m \subseteq E_k$. Hence, $e, e' \in E_k$. Since U_k is a unitary event structure we are done.

- $\forall e \in E^\omega,\ |[e]| > 1 \Rightarrow \sum_{e' \in [e]} Q(e') = U$

  Let $e \in E^\omega \,.\, |[e]| > 1$. By Definition 6.3.5, $\exists n \in \omega$ such that $e \in E_n$ and $Q^\omega(e) = Q_n(e)$. Since U_n is a unitary event structure we have $\sum_{e' \in [e]} Q^\omega(e') = \sum_{e' \in [e]} Q_n(e') = U$.

□

Proof of Lemma 6.3.7

Proof. Due to Lemma 4.3.8 we only need to focus on the quantum condition.

- U^ω is an upper bound

  $\forall n \in \omega$ we need to have $\mathsf{U}_n \trianglelefteq \mathsf{U}^\omega$. We need to check that $\forall e \in E_n \,.\, Q_n(e) = Q^\omega(e)$. It follows directly from Definition 6.3.5 that $\exists n \in \omega \,.\, e \in E_n$ and $Q^\omega(e) = Q_n(e)$.

- U^ω is a least upper bound

  Let $\mathsf{U} = (E, \leq, \#, Q)$ be an upper bound of the chain. We need to show that if $\mathsf{U}_n \trianglelefteq \mathsf{U}^\omega$ and $\mathsf{U}_n \trianglelefteq \mathsf{U}$ then $\mathsf{U}^\omega \trianglelefteq \mathsf{U}$. From $\mathsf{U}_n \trianglelefteq \mathsf{U}^\omega$, $\forall e \in E_n \,.\, Q_n(e) = Q^\omega(e)$. By Definition 6.3.5, $\exists n \in \omega \,.\, e \in E_n$ and $Q^\omega(e) = Q_n(e)$. From $\mathsf{U}_n \trianglelefteq \mathsf{U}$, $\forall e \in E_n \,.\, Q_n(e) = Q(e)$. Thus $\forall e \in \mathsf{U}^\omega,\ \exists n \in \omega \,.\, e \in E_n$ and $Q^\omega(e) = Q_n(e) = Q(e)$.

□

Proof of Lemma 6.3.8

Proof. Let

$$\mathsf{U} = (E, \leq, \#, Q)$$
$$\mathsf{U}_1 = (E_1, \leq_1, \#_1, Q_1)$$

$$\begin{aligned}
\mathsf{U}_2 &= (E_2, \leq_2, \#_2, Q_2) \\
\mathsf{U}\,;\,\mathsf{U}_1 &= (E^1, \leq^1, \#^1, Q^1) \\
\mathsf{U}\,;\,\mathsf{U}_2 &= (E^2, \leq^2, \#^2, Q^2)
\end{aligned}$$

such that $\mathsf{U}_1 \trianglelefteq \mathsf{U}_2$.

We want to show $\forall e \in E^1 \,.\, Q^1(e) = Q^2(e)$. Due to Lemma 4.3.10 we focus solely on the quantum condition. Let $e \in E^1$. By Definition 6.2.1 we have two cases:

1. $e \in E$

   It follows directly that $Q^1(e) = Q(e) = Q^2(e)$.

2. $e = (e_1, x) \in E_1 \times \mathcal{C}_{\mathsf{max}}(\mathsf{U})$

   We have $Q^1(e) = Q_1(e_1)$. From $\mathsf{U}_1 \trianglelefteq \mathsf{U}_2$, $Q_1(e_1) = Q_2(e_1)$. By Definition 6.2.1, $Q_2(e_1) = Q^2(e)$. Thus $Q^1(e) = Q^2(e)$.

□

Proof of Lemma 6.3.9

Proof. Let

$$\begin{aligned}
\mathsf{U}_1 &= (E_1, \leq_1, \#_1, Q_1) \\
\mathsf{U}_2 &= (E_2, \leq_2, \#_2, Q_2) \\
\mathsf{U}'_1 &= (E'_1, \leq'_1, \#'_1, Q'_1) \\
\mathsf{U}_2 &= (E'_2, \leq'_2, \#'_2, Q'_2) \\
\mathsf{U}_1 \,||\, \mathsf{U}_2 &= (E, \leq, \#, Q) \\
\mathsf{U}'_1 \,||\, \mathsf{U}'_2 &= (E', \leq', \#', Q')
\end{aligned}$$

such that $\mathsf{U}_1 \trianglelefteq \mathsf{U}'_1$ and $\mathsf{U}_2 \trianglelefteq \mathsf{U}'_2$.

We want to show that $\forall e \in E \,.\, Q(e) = Q'(e)$. Due to Lemma 4.3.11 we focus solely on the quantum condition. Let $e \in E$. By Definition 6.2.3 we have two cases:

1. $e \in E_1$

   We know that $Q(e) = Q_1(e)$. Since $\mathsf{U}_1 \trianglelefteq \mathsf{U}'_1$, $Q_1(e) = Q'_1(e)$. By Definition 6.2.3, $Q'_1(e) = Q'(e)$. Thus $Q(e) = Q'(e)$.

2. $e \in E_2$

   Similar to the previous.

□

Proof of Lemma 6.3.10

Proof. Let

$$\begin{aligned}
\mathsf{U}_1 &= (E_1, \leq_1, \#_1, Q_1) \\
\mathsf{U}_2 &= (E_2, \leq_2, \#_2, Q_2) \\
\mathsf{U}'_1 &= (E'_1, \leq'_1, \#'_1, Q'_1) \\
\mathsf{U}_2 &= (E'_2, \leq'_2, \#'_2, Q'_2)
\end{aligned}$$

$$\mathtt{M}(n, \mathsf{U}_1, \mathsf{U}_2) = (E, \leq, \#, Q)$$
$$\mathtt{M}(n, \mathsf{U}'_1, \mathsf{U}'_2) = (E', \leq', \#', Q')$$

such that $\mathsf{U}_1 \trianglelefteq \mathsf{U}'_1$ and $\mathsf{U}_2 \trianglelefteq \mathsf{U}'_2$.

The conditions to check are:

1. $E \subseteq E'$
2. $\forall e, e' \,.\, e \leq e' \Leftrightarrow e, e' \in E \wedge e \leq' e'$
3. $\forall e, e' \,.\, e \# e' \Leftrightarrow e, e' \in E \wedge e \#' e'$
4. $\forall e \in E \,.\, Q(e) = Q'(e)$

The first three conditions follow directly from Definition 6.2.5. Hence we focus on the last one. Let $e \in E$. We have the four cases:

1. $e = \tau_0^n$

   By Definition 6.2.5 we are done since, $Q(\tau_0^n) = Q'(\tau_0^n)$.

2. $e = \tau_1^n$

   By Definition 6.2.5 we are done since, $Q(\tau_1^n) = Q'(\tau_1^n)$.

3. $e \in E_1$

   We know that $Q(e) = Q_1(e)$. From $\mathsf{U}_1 \trianglelefteq \mathsf{U}'_1$, $Q_1(e) = Q'_1(e)$. By Definition 6.2.5, $Q'_1(e) = Q'(e)$. Thus $Q(e) = Q'(e)$.

4. $e \in E_2$

   Similar to the previous point.

□

Proof of Lemma 6.3.11

Proof. Similar to Lemma 4.3.14. □

Proof of Lemma 6.3.12

Proof. Similar to Lemma 4.3.15. □

Proof of Lemma 6.3.13

Proof. By Lemma 6.3.10, the measurement is monotone. Then we only need to show that each event of $\mathtt{M}(q, \bigsqcup_n \mathsf{U}_n, \bigsqcup_m \mathsf{U}_m)$ is an event of $\bigsqcup_{n,m}(\mathtt{M}(q, \mathsf{U}_n, \mathsf{U}_m))$. Let e be an event of $\mathtt{M}(q, \bigsqcup_n \mathsf{U}_n, \bigsqcup_m \mathsf{U}_m)$. We have four cases:

1. $e = \tau_0^n$

   It follows directly from Definition 6.2.5.

2. $e = \tau_1^n$

   It follows directly from Definition 6.2.5.

3. e is an event of $\bigsqcup_n \mathsf{U}_n$

   By Definition 6.3.5, $\exists n \in \omega$ such that e is an event of U_n. By Definition 6.2.5, e is an event of $\mathtt{M}(q, \mathsf{U}_n, \mathsf{U}_m)$. By Definition 6.3.5, e is an event of $\bigsqcup_{n,m}(\mathtt{M}(q, \mathsf{U}_n, \mathsf{U}_m))$.

4. e is an event of $\bigsqcup_m \mathsf{U}_m$

   Similar to the previous point.

□

Proof of Lemma 6.4.2

Proof. We show that $\tilde{\leq}$ is a partial order and that $\tilde{\#}$ is symmetric and irreflexive.

- $\tilde{\leq}$ is a partial order:
  - Reflexivity follows directly from Definition 6.4.1.
  - Transitivity we note that the pairs are compose of the same event. Hence transitivity holds.
  - Antisymmetry holds with the same argument as in transitivity.
- $\tilde{\#}$ is symmetric and irreflexive since $\tilde{\#}$ equals # for the events in $\tilde{E}$.

We show $\tilde{\mathsf{U}}_y$ obeys the conditions of a unitary event structure.

- $\{e' \mid e' \tilde{\leq} e\}$ is finite

  Trivially holds because every $e \in \tilde{E}$ is only causally related to itself.

- $e \tilde{\#} e' \tilde{\leq} e'' \Rightarrow e \tilde{\#} e''$

  Trivially holds because every $e \in \tilde{E}$ is only causally related to itself. Hence $e' \tilde{\leq} e''$ is by definition $e' \tilde{\leq} e'$. It then follows directly $e \tilde{\#} e' \tilde{\leq} e' \Rightarrow e \tilde{\#} e'$.

- e co $e' \Rightarrow [\tilde{Q}(e), \tilde{Q}(e')] = 0$

  It follows directly since if e co e' in $\tilde{\mathsf{U}}$, then e co e' in U, in which $[Q(e), Q(e')] = 0$. Hence $[\tilde{Q}(e), \tilde{Q}(e')] = [Q|_{\tilde{E}}(e), Q|_{\tilde{E}}(e')] = 0$

- $\sim\!\!\sim$is transitive

  It follows directly from the fact that $\tilde{\#}$ is inherited from U.

- $\forall e \in \tilde{E}, \sum_{e' \in [e]} Q(e')$ is unitary

  Let $e \in \tilde{E}$. By definition we have $\tilde{Q} = Q|_{\tilde{E}}$. It then follows that $\sum_{e' \in [e]} \tilde{Q}([e]) = \sum_{e' \in [e]} Q|_{\tilde{E}}(e')$ is unitary.

□

Proof of Lemma 6.4.4

Proof. We show that $\hat{\leq}$ is a partial order and that $\hat{\#}$ is symmetric and irreflexive.

- $\hat{\leq}$ is a partial order:
  - Reflexivity follows directly from Definition 6.4.3.
  - Transitivity we note that the pairs are compose of the same event. Hence transitivity holds.
  - Antisymmetry holds with the same argument as in transitivity.
- $\hat{\#}$ is symmetric and irreflexive since $\#$ is the empty relation.

We show $\hat{\mathsf{U}}$ obeys the conditions of a unitary event structure. For convenience we sometimes write U instead of $\sum_{e' \in [e]} Q(e')$.

- $\{[e'] \mid [e'] \hat{\leq} [e]\}$ is finite

  Trivially holds because every $[e] \in \hat{E}$ is only causally related to itself.
- $[e] \hat{\#} [e'] \hat{\leq} [e''] \Rightarrow [e] \hat{\#} [e'']$

  Trivially holds because the conflict relation is empty.
- $[e]$ co $[e'] \Rightarrow [\hat{Q}([e]), \hat{Q}([e'])] = 0$

  We have three cases:

  1. $|[e]| = |[e']| = 1$
     It follows directly that $[\hat{Q}([e]), \hat{Q}([e'])] = [Q(e), Q(e')] = 0$
  2. $|[e]| = 1$ and $|[e']| > 1$
     We have $[\hat{Q}([e]), \hat{Q}([e'])] = [Q(e), U] = 0$, since the event e and all the events in $[e']$ are concurrent.
  3. $|[e]| > 1$ and $|[e']| > 1$
     We have $[\hat{Q}([e]), \hat{Q}([e'])] = [U_1, U_2] = 0$, since all the events in $[e]$ are concurrent with the events in $[e']$.
- $\sim\!\!\sim$is transitive

  It follows directly since $\hat{\#} = \emptyset$.
- $\forall [e] \in \hat{E}, \sum_{[e'] \in [[e]]} \hat{Q}([e'])$ is unitary

  Since $\hat{\#} = \emptyset$ then $\sum_{[e'] \in [[e]]} \hat{Q}([e']) = \hat{Q}([e'])$ since $|[[e]]| = 1$. By Definition 6.4.3 we have two cases:

  1. $|[e']| = 1$
     Then $\hat{Q}([e']) = Q(e')$ which is a unitary.
  2. $|[e]| > 1$
     Then $\hat{Q}([e']) = U$ which is a unitary.

□

Proof of Lemma 6.4.5

Proof. We show that $proj_{\tilde{\mathsf{E}}}$ satisfies the conditions to be a map of event structures.

- $\forall x \in \mathcal{C}(\tilde{\mathsf{E}}) \Rightarrow proj_{\tilde{\mathsf{E}}}(x) \in \mathcal{C}(\hat{\mathsf{E}})$

  It follows straightforwardly since $\hat{\#}$ is empty and $\hat{\leq}$ is the trivial one.

- $\forall (e \neq e') \in x \in \mathcal{C}(\tilde{\mathsf{E}})$, if $proj_{\tilde{\mathsf{E}}}$ is defined in both then $proj_{\tilde{\mathsf{E}}}(e) \neq proj_{\tilde{\mathsf{E}}}(e')$

  Let $(e \neq e') \in x \in \mathcal{C}(\tilde{\mathsf{E}})$. Since $proj_{\tilde{\mathsf{E}}}$ is total, $proj_{\tilde{\mathsf{E}}}$ is defined in both and since $e, e' \in x$ then $\neg(e \tilde{\#} e')$. Hence it follows straightforwardly that $proj_{\tilde{\mathsf{E}}}(e) \neq proj_{\tilde{\mathsf{E}}}(e')$.

□

Proof of Lemma 6.5.1

Proof. Induction over rules in Figure 57.

- $skip \xrightarrow{sk} \checkmark$

  It follows directly that $[\![\checkmark]\!] \equiv [\![skip]\!] \backslash sk \equiv \emptyset$.

- $a \xrightarrow{a} \checkmark$

  It follows directly that $[\![\checkmark]\!] \equiv [\![a]\!] \backslash a \equiv \emptyset$.

- $\mathtt{M}(n, C_1, C_2) \xrightarrow{\tau_0^n} C_1$

  It follows directly since $\mathsf{P}_0^n \,;\, [\![C_1]\!] \sqsubseteq \mathsf{P}_0^n \,;\, [\![C_1]\!] + \mathsf{P}_1^n \,;\, [\![C_2]\!] = [\![\mathtt{M}(n, C_1, C_2)]\!]$.

- $\mathtt{M}(n, C_1, C_2) \xrightarrow{\tau_1^n} C_2$

  It follows directly since $\mathsf{P}_1^n \,;\, [\![C_2]\!] \sqsubseteq \mathsf{P}_0^n \,;\, [\![C_1]\!] + \mathsf{P}_1^n \,;\, [\![C_2]\!] = [\![\mathtt{M}(n, C_1, C_2)]\!]$.

- $C_1 \,;\, C_2 \xrightarrow{l} C_2$

$$
\begin{aligned}
& C_1 \,;\, C_2 \xrightarrow{l} C_2 \\
\Rightarrow & \{\xrightarrow{l} \text{ entails}\} \\
& C_1 \xrightarrow{l} \checkmark \\
\Rightarrow & \{\text{i.h.}\} \\
& [\![\checkmark]\!] \equiv [\![C_1]\!] \backslash l \\
\Rightarrow & \{\text{Lemma 6.2.12}\} \\
& [\![\checkmark]\!] \,;\, [\![C_2]\!] \equiv ([\![C_1]\!] \backslash l) \,;\, [\![C_2]\!] \\
\Rightarrow & \{[\![\checkmark]\!] \,;\, [\![C_2]\!] \equiv [\![C_2]\!], \text{ Lemma 6.2.17}\} \\
& [\![C_2]\!] \equiv ([\![C_1]\!] \,;\, [\![C_2]\!]) \backslash l \\
\Rightarrow & \{\text{Definition 6.2.8}\} \\
& [\![C_2]\!] \equiv [\![C_1 \,;\, C_2]\!] \backslash l
\end{aligned}
$$

- $C_1 \,;\, C_2 \xrightarrow{l} C_1' \,;\, C_2$

$$
\begin{array}{l}
\quad C_1 \,;\, C_2 \xrightarrow{l} C_1' \,;\, C_2 \\
\Rightarrow\{\xrightarrow{l} \text{ entails}\} \\
\quad C_1 \xrightarrow{l} C_1' \\
\Rightarrow\{\text{i.h.}\} \\
\quad [\![C_1']\!] \equiv [\![C_1]\!] \backslash l \\
\Rightarrow\{\text{Lemma 6.2.12}\} \\
\quad [\![C_1']\!] \,;\, [\![C_2]\!] \equiv ([\![C_1]\!] \backslash l) \,;\, [\![C_2]\!] \\
\Rightarrow\{\text{Lemma 6.2.17}\} \\
\quad [\![C_1']\!] \,;\, [\![C_2]\!] \equiv ([\![C_1]\!] \,;\, [\![C_2]\!]) \backslash l \\
\Rightarrow\{\text{Definition 6.2.8}\} \\
\quad [\![C_1' \,;\, C_2]\!] \equiv [\![C_1 \,;\, C_2]\!] \backslash l
\end{array}
$$

- $C_1 \,||\, C_2 \xrightarrow{l} C_2$

$$
\begin{array}{l}
\quad C_1 \,||\, C_2 \xrightarrow{l} C_2 \\
\Rightarrow\{\xrightarrow{l} \text{ entails}\} \\
\quad C_1 \xrightarrow{l} \checkmark \\
\Rightarrow\{\text{i.h.}\} \\
\quad [\![\checkmark]\!] \equiv [\![C_1]\!] \backslash l \\
\Rightarrow\{\text{Lemma 6.2.13}\} \\
\quad [\![\checkmark]\!] \,||\, [\![C_2]\!] \equiv ([\![C_1]\!] \backslash l) \,||\, C_2 \\
\Rightarrow\{[\![\checkmark]\!] \,||\, [\![C_2]\!] \equiv [\![C_2]\!]\} \\
\quad [\![C_2]\!] \equiv ([\![C_1]\!] \backslash l) \,||\, C_2 \\
\Rightarrow\{\text{Lemma 6.2.18, Definition 6.2.8}\} \\
\quad [\![C_2]\!] \equiv [\![C_1 \,||\, C_2]\!] \backslash l
\end{array}
$$

- $C_1 \,||\, C_2 \xrightarrow{l} C_1' \,||\, C_2$

$$
\begin{array}{l}
\quad C_1 \,||\, C_2 \xrightarrow{l} C_1' \,||\, C_2 \\
\Rightarrow\{\xrightarrow{l} \text{ entails}\} \\
\quad C_1 \xrightarrow{l} C_1' \\
\Rightarrow\{\text{i.h.}\} \\
\quad [\![C_1']\!] \equiv [\![C_1]\!] \backslash l \\
\Rightarrow\{\text{Lemma 6.2.13}\} \\
\quad [\![C_1']\!] \,||\, [\![C_2]\!] \equiv ([\![C_1]\!] \backslash l) \,||\, C_2 \\
\Rightarrow\{\text{Lemma 6.2.18, Definition 6.2.8}\}
\end{array}
$$

$$[\![C_1' \,||\, C_2]\!] \equiv [\![C_1 \,||\, C_2]\!] \backslash l$$

- $C_1 \,||\, C_2 \xrightarrow{l} C_1$

$$\begin{aligned}
&C_1 \,||\, C_2 \xrightarrow{l} C_1 \\
\Rightarrow&\{\xrightarrow{l} \text{ entails}\} \\
&C_2 \xrightarrow{l} \checkmark \\
\Rightarrow&\{\text{i.h.}\} \\
&[\![\checkmark]\!] \equiv [\![C_2]\!] \backslash l \\
\Rightarrow&\{\text{Lemma 6.2.13}\} \\
&[\![C_1]\!] \,||\, [\![\checkmark]\!] \equiv [\![C_1]\!] \,||\, ([\![C_2]\!] \backslash l) \\
\Rightarrow&\{[\![C_1]\!] \,||\, [\![\checkmark]\!] \equiv [\![C_1]\!]\} \\
&[\![C_1]\!] \equiv [\![C_1]\!] \,||\, ([\![C_2]\!] \backslash l) \\
\Rightarrow&\{\text{Lemma 6.2.18, Definition 6.2.8}\} \\
&[\![C_1]\!] \equiv [\![C_1 \,||\, C_2]\!] \backslash l
\end{aligned}$$

- $C_1 \,||\, C_2 \xrightarrow{l} C_1 \,||\, C_2'$

$$\begin{aligned}
&C_1 \,||\, C_2 \xrightarrow{l} C_1 \,||\, C_2' \\
\Rightarrow&\{\xrightarrow{l} \text{ entails}\} \\
&C_2 \xrightarrow{l} C_2' \\
\Rightarrow&\{\text{i.h.}\} \\
&[\![C_2']\!] \equiv [\![C_2]\!] \backslash l \\
\Rightarrow&\{\text{Lemma 6.2.13}\} \\
&[\![C_1]\!] \,||\, [\![C_2']\!] \equiv C_1 \,||\, ([\![C_2]\!] \backslash l) \\
\Rightarrow&\{\text{Lemma 6.2.18, Definition 6.2.8}\} \\
&[\![C_1 \,||\, C_2']\!] \equiv [\![C_1 \,||\, C_2]\!] \backslash l
\end{aligned}$$

- $while\ \mathtt{M}(n, C) \xrightarrow{\tau_0^n} \checkmark$

  We know that

$$[\![while\ \mathtt{M}(n, C)]\!] = [\![\mathtt{M}(n, \checkmark, C\,;\ while\ \mathtt{M}(n, C))]\!]$$

  Hence

$$[\![while\ \mathtt{M}(n, C)]\!] \backslash \tau_0^n = [\![\mathtt{M}(n, \checkmark, C\,;\ while\ \mathtt{M}(n, C))]\!] \backslash \tau_0^n$$

  By Lemma 6.2.19, we have $[\![\checkmark]\!]$. Hence $[\![while\ \mathtt{M}(n, C)]\!] \backslash \tau_0^n = [\![\checkmark]\!]$.

- $while\ \mathtt{M}(n, C) \xrightarrow{\tau_1^n} C\,;\ while\ \mathtt{M}(n, C)$

  We know that

$$[\![while\ \mathtt{M}(n, C)]\!] = [\![\mathtt{M}(n, \checkmark, C\,;\ while\ \mathtt{M}(n, C))]\!]$$

Hence

$$[\![while\ \mathtt{M}(n, C)]\!] \backslash \tau_1^n = [\![\mathtt{M}(n, \checkmark, C\ ;\ while\ \mathtt{M}(n, C))]\!] \backslash \tau_1^n$$

By Lemma 6.2.19, we have $[\![C\ ;\ while\ \mathtt{M}(n, C)]\!]$. Hence

$$[\![while\ \mathtt{M}(n, C)]\!] \backslash \tau_0^n = [\![C\ ;\ while\ \mathtt{M}(n, C)]\!]$$

□

Proof of Theorem 6.5.2

Proof.

- $|\omega| = 1$

  It follows directly that $\exists \{l\} \in \mathcal{C}([\![C]\!])\ .\ \emptyset \xrightarrow{l} \subset \{l\}$

- $|\omega| > 1$

$$\begin{array}{l} \quad C \overset{\omega}{\twoheadrightarrow} C' \\ \Rightarrow \{\text{Definition 17}\} \\ \quad C \xrightarrow{l} C'' \qquad C'' \overset{\omega'}{\twoheadrightarrow} C' \\ \Rightarrow \{\text{Lemma 6.5.1, i.h.}\} \\ \quad [\![C'']\!] \equiv [\![C]\!] \backslash l \qquad \exists y \in \mathcal{C}([\![C'']\!])\ .\ \emptyset \xrightarrow{\omega'} \subset y \\ \Rightarrow \{\text{Definition 6.2.15}\} \\ \quad \{l\} \cup y \in \mathcal{C}([\![C]\!])\ .\ \emptyset \xrightarrow{l} \subset \{l\} \xrightarrow{\omega'} \subset \{l\} \cup y = x \end{array}$$

□

Proof of Lemma 6.5.3

Proof. Induction over the interpretation of commands.

- $sk \in \mathcal{I}(skip)$

  Let $C' = \checkmark$. It follows directly that $skip \xrightarrow{sk} \checkmark$ and that $[\![skip]\!] \backslash sk \equiv [\![\checkmark]\!]$.

- $a \in \mathcal{I}(a)$

  Let $C' = \checkmark$. It follows directly that $a \xrightarrow{a} \checkmark$ and that $[\![a]\!] \backslash a \equiv [\![\checkmark]\!]$.

- $l \in \mathcal{I}(\mathtt{M}(n, C_1, C_2))$

  By Definition 6.2.8, $[\![\mathtt{M}(n, C_1, C_2)]\!] = \mathsf{P}_0^n\ ;\ [\![C_1]\!] + \mathsf{P}_1^n\ ;\ [\![C_2]\!]$. We have two cases:

  1. $l' = \tau_0^n$

     By Lemma 6.2.19, $(\mathsf{P}_0^n\ ;\ [\![C_1]\!] + \mathsf{P}_1^n\ ;\ [\![C_2]\!]) \backslash \tau_0^n \equiv [\![C_1]\!]$. Furthermore $\mathtt{M}(n, C_1, C_2) \xrightarrow{\tau_0^n} C_1$.

  2. $l' = \tau_1^n$

     By Lemma 6.2.19, $(\mathsf{P}_0^n\ ;\ [\![C_1]\!] + \mathsf{P}_1^n\ ;\ [\![C_2]\!]) \backslash \tau_1^n \equiv [\![C_2]\!]$. Furthermore $\mathtt{M}(n, C_1, C_2) \xrightarrow{\tau_1^n} C_2$.

- $l' \in \mathcal{I}(C_1 \,;\, C_2)$

  By Definition 6.2.1 we have that $l' \in \mathcal{I}([\![C_1]\!])$. By i.h., $\exists C'$ such that $C_1 \xrightarrow{l} C'$ and $[\![C_1]\!] \backslash l' \equiv [\![C']\!]$. We have two cases:

  1. $C' = \checkmark$

     We have $C_1 \xrightarrow{l'} \checkmark$ and $[\![C_1]\!] \backslash l' \equiv [\![\checkmark]\!]$. By the rules in Figure 57, $C_1 \,;\, C_2 \xrightarrow{l'} C_2$. By Definition 6.2.1, $([\![C_1]\!] \backslash l') \,;\, [\![C_2]\!] \equiv [\![\checkmark]\!] \,;\, [\![C_2]\!] \equiv [\![C_2]\!]$.

  2. $C' = C_1'$

     We have $C_1 \xrightarrow{l'} C_1'$ and $[\![C_1]\!] \backslash l' \equiv [\![C_1']\!]$. By the rules in Figure 57, $C_1 \,;\, C_2 \xrightarrow{l'} C_1' \,;\, C_2$. By Definition 6.2.1, $([\![C_1]\!] \backslash l') \,;\, [\![C_2]\!] \equiv [\![C_1']\!] \,;\, [\![C_2]\!]$. By Definition 6.2.8, $[\![C_1' \,;\, C_2]\!]$.

- $l' \in \mathcal{I}(C_1 \,||\, C_2)$

  By Definition 6.2.3 we have two cases:

  1. $l' \in \mathcal{I}([\![C_1]\!])$

     By i.h. $\exists C' \,.\, C_1 \xrightarrow{l'} C'$ and $[\![C_1]\!] \backslash l' \equiv [\![C']\!]$. By the rules in Figure 57 we have two cases:

     (a) $C' = \checkmark$

     We have $C_1 \xrightarrow{l'} \checkmark$ and $[\![C_1]\!] \backslash l' \equiv [\![\checkmark]\!]$. By the rules in Figure 57 we have $C_1 \,||\, C_2 \xrightarrow{l'} C_2$. By Definition 6.2.3, $([\![C_1]\!] \backslash l') \,||\, [\![C_2]\!]$. By Lemma 6.2.18 we have $([\![C_1]\!] \,||\, [\![C_2]\!]) \backslash l'$. By Definition 6.2.8, $[\![C_1 \,||\, C_2]\!] \backslash l'$.

     (b) $C' = C_1'$

     We have $C_1 \xrightarrow{l'} C_1'$ and $[\![C_1]\!] \backslash l' \equiv [\![C_1']\!]$. By the rules in Figure 57 we have $C_1 \,||\, C_2 \xrightarrow{l'} C_1' \,||\, C_2$. By Definition 6.2.3, $([\![C_1]\!] \backslash l') \,||\, [\![C_2]\!]$. By Lemma 6.2.18 we have $([\![C_1]\!] \,||\, [\![C_2]\!]) \backslash l'$. By Definition 6.2.8, $[\![C_1 \,||\, C_2]\!] \backslash l'$.

  2. $l' \in \mathcal{I}([\![C_2]\!])$

     By i.h. $\exists C' \,.\, C_2 \xrightarrow{l} C'$ and $[\![C_2]\!] \backslash l' = [\![C']\!]$. By the rules in Figure 57 we have two cases:

     (a) $C' = \checkmark$

     We have $C_2 \xrightarrow{l'} \checkmark$ and $[\![C_2]\!] \backslash l' \equiv [\![\checkmark]\!]$. By the rules in Figure 57 we have $C_1 \,||\, C_2 \xrightarrow{l'} C_1$. By Definition 6.2.3, $[\![C_1]\!] \,||\, ([\![C_2]\!] \backslash l')$. By Lemma 6.2.18 we have $([\![C_1]\!] \,||\, [\![C_2]\!]) \backslash l'$. By Definition 6.2.8, $[\![C_1 \,||\, C_2]\!] \backslash l'$.

     (b) $C' = C_2'$

     We have $C_2 \xrightarrow{l'} C_2'$ and $[\![C_2]\!] \backslash l' \equiv [\![C_2']\!]$. By the rules in Figure 57 we have $C_1 \,||\, C_2 \xrightarrow{l'} C_1 \,||\, C_2'$. By Definition 6.2.3, $[\![C_1]\!] \,||\, ([\![C_2]\!] \backslash l')$. By Lemma 6.2.18 we have $([\![C_1]\!] \,||\, [\![C_2]\!]) \backslash l'$. By Definition 6.2.8, $[\![C_1 \,||\, C_2]\!] \backslash l'$.

- $l \in \mathcal{I}([\![while\ \mathtt{M}(n, C)]\!])$

  Since $[\![while\ \mathtt{M}(n, C)]\!] = [\![\mathtt{M}(n, \checkmark, C \,;\, while\ \mathtt{M}(n, C))]\!]$, we have that $l = \tau_0^n$ or $l = \tau_1^n$. We have two cases:

  1. $l = \tau_0^n$

     Let $C' = \checkmark$. We know that

$$[\![while\ \mathtt{M}(n, C)]\!] = [\![\mathtt{M}(n, \checkmark, C \,;\, while\ \mathtt{M}(n, C))]\!]$$

Hence

$$[\![\mathtt{M}(n, \checkmark, C\,;\, while\ \mathtt{M}(n, C))]\!] \backslash \tau_0^n$$

By Lemma 6.2.19, we have $[\![\checkmark]\!]$. Thus it follows directly that $while\ \mathtt{M}(n, C) \xrightarrow{\tau_0^n} \checkmark$.

2. $l = \tau_1^n$
   Let $C' = C\,;\, while\ \mathtt{M}(n, C)$. We know that

$$[\![while\ \mathtt{M}(n, C)]\!] = [\![\mathtt{M}(n, \checkmark, C\,;\, while\ \mathtt{M}(n, C))]\!]$$

Hence

$$[\![\mathtt{M}(n, \checkmark, C\,;\, while\ \mathtt{M}(n, C))]\!] \backslash \tau_1^n$$

By Lemma 6.2.19, we have $[\![\checkmark]\!]$. Thus it follows directly that

$$while\ \mathtt{M}(n, C) \xrightarrow{\tau_1^n} C\,;\, while\ \mathtt{M}(n, C)$$

□

Proof of Theorem 6.5.4

Proof. Induction over the length of ω.

- $|\omega| = 1$

  We have $\{l'\} \in \mathcal{C}(C)$ such that $\emptyset \xrightarrow{l'}\!\subset \{l'\}$. Furthermore $l' \in \mathcal{I}([\![C]\!])$. By Lemma 6.5.3, $C \xrightarrow{l'} C'$ and $[\![C']\!] \equiv [\![C]\!] \backslash l'$. By the rules in Figure 17, $C \xrightarrow{l'}\!\!\!\!\rightarrow C'$.

- $|\omega| > 1$

  We have $x \in \mathcal{C}([\![C]\!])$ such that $\emptyset \xrightarrow{\omega}\!\subset x$. Since $\omega = l_0 l_1 \ldots l_n$, then $\emptyset \xrightarrow{l_0}\!\subset \{l_0\} \xrightarrow{\omega'}\!\subset x$. Hence $l_0 \in \mathcal{I}([\![C]\!])$. By Lemma 6.5.3, $C \xrightarrow{l_0} C'$ and $[\![C']\!] \equiv [\![C]\!] \backslash l_0$. By Definition 6.2.15, $\exists y \in \mathcal{C}([\![C']\!])$ such that $\emptyset \xrightarrow{\omega'}\!\subset y$. By i.h. $\exists C''$ such that $C' \xrightarrow{\omega'}\!\!\!\!\rightarrow C''$. By the rules in Figure 17, $C \xrightarrow{\omega}\!\!\!\!\rightarrow C''$, where $\omega = l_0 : \omega'$.

□

Chapter B
Parser details

B.1 Implementation of the parser

We now describe the parser, implemented using Parsec [66], a monadic combinator parser for Haskell. Parsec enables the construction of complex parsers by combining small, composable parsing functions in a modular way.

As previously mentioned, the parser is organized into three modules: *ParserAEBE*, *ParserCom*, and *ParserFile*. We start by describing the first of these, *ParserAEBE*, which is responsible for parsing integer and Boolean expressions, as well as some other minor parsers.

Parsing integer expressions

Integer expressions are represented in Haskell by the data type shown in Figure 105.

```
data E = Num Integer
       | Id String
       | PlusE E E
       | MinusE E E
       | MultE E E
       | DivE E E
       deriving (Show, Eq)
```

Figure 105: Definition of data type E

The constructor `Num Integer` represents an integer, while `Id String` represents a classical variable. The remaining constructors encode the standard arithmetic operations: `PlusE`, `MinusE`, `MultE`, and `DivE` correspond to addition, subtraction, multiplication, and division, respectively.

The brain behind the parsing of integer expressions is `parseESelect`, defined as follows:

```
parseESelect :: GenParser Char st E
parseESelect = try(parseMultDiv)
               <|> try(parsePlusMinus)
               <|> try(parseNum)
               <|> try(parseId)
```

```
  <|> parsePar
```

Intuitively, this parser sequentially applies each parser that composes it, choosing the one that matches the input. For instance, if the input is the string `"1+2"`, `parseESelect` picks up `parsePlusMinus` to parse the string. This behavior is possible due to the combination of two parsers: `<|>` and `try`. Recall that the former represents a choice (if one fails, the next is attempted) and the latter ensures that when a parser fails, the input is not consumed.

Let us leave the parser for multiplication and division, and addition and subtraction to last.

Parsing integer numbers

The parser of integer numbers is given as follows:

```
parseNum :: GenParser Char st E
parseNum = do
  minus <- option "" (string "-")
  x <- many1 digit
  return (Num (read (minus ++ x) :: Integer))
```

Intuitively, this parser allows us to parse integer numbers that are either positive or negative. This capability comes from using `minus <- option "" (string "-")`, where we store in `minus` either an empty string or the minus symbol. Then, we capture one or more numbers in `x` through `x <- many1 digit`. Finally, `minus` and `x` are concatenated and converted into an integer value with `Num (read (minus ++ x) :: Integer)`. For example, the parsing of -1 returns `Right (Num (-1))`, and the parsing of 1 returns solely `Right (Num 1)`.

Parsing classical variables

Next, we have the parsing of classical variables. For the name of classical variables, we consider the following restrictions:

- They may contain letters, digits, and underscores.
- The first character must be either a letter or an underscore (it cannot be a digit).
- They cannot consist solely of underscores.

Furthermore, there are some names that cannot be given to classical variables, since they represent keywords, *i.e.* words that have a defined meaning in the language. Those words are as follows: skip, if, else, then, while, do, true, false, and await. The definition of the parser of classical variables is as follows:

```
parseId :: GenParser Char st E
parseId = do
    notFollowedBy (reservedWord)
    i <- (try(startUnderscore) <|> startLetter)
    return (Id i)
```

where `notFollowedBy (reservedWord)` guarantees that reserved words are not used as names of classical variables, since if the parser `reservedWord` is able to parse successfully, then `notFollowedBy (reservedWord)` fails. The parser then proceeds to parse an identifier that either starts with an underscore (`startUnderscore`) or with a letter (`startLetter`), storing the resulting string in `i` and returning `Id i`.

Parsing parenthesized expressions

Before going to more complex parsers, we define the parser for expressions enclosed in parentheses:

```
parsePar :: GenParser Char st E
parsePar = do
  char '('
  spaces
  x <- parseESelect
  spaces
  char ')'
  return x
```

This parser first matches the opening parenthesis `'('`. If successful, it skips any whitespace, parses an integer expression using `parseESelect`, and stores the result in `x`. It then skips further whitespace and matches the closing parenthesis `')'`. If all steps succeed, `parsePar` returns the parsed integer expression `x`.

Parsing multiplication/division expressions

At last, it lacks an explanation of the parsers `parseMultDiv` and `parsePlusMinus`. Let us begin with the former. The parser `parseMultDiv` comes accompanied with an auxiliary parser, `termMultDiv`, which acts similarly to `parseESelect`. In other words, `termMultDiv` is the "brain" and the "worker" is `parseMultDiv`. The definition of `termMultDiv` is as follows:

```
termMultDiv :: GenParser Char st E
termMultDiv =  try(parsePlusMinus)
               <|> try(parseNum)
               <|> try(parseId)
               <|> parsePar
```

As it is possible to see, its definition resembles the definition of `parseESelect`, only lacking the parser `parseMultDiv`. The absence of `parseMultDiv` is on purpose, since its presence would result in an infinite loop.

The definition of `parseMultDiv` is as follows:

```
parseMultDiv :: GenParser Char st E
parseMultDiv = do
  x <- termMultDiv
  spaces
  op <- try(char '*') <|> (char '/')
  spaces
  y <- termMultDiv
  case op of
    '*' -> loop (MultE x y)
    '/' -> loop (DivE x y)
    where
      f x = do
        spaces
        op <- try(char '*') <|> (char '/')
        spaces
        y <- termMultDiv
        spaces
        case op of
          '*' -> loop (MultE x y)
          '/' -> loop (DivE x y)
      loop t = try(f t) <|> return t
```

This parser behaves as follows: it starts by trying to parse a term from `termMultDiv` and stores the result in the variable `x` if the attempt is successful. Later on, ignoring white spaces (spaces, tabs, and newlines), it tries to parse either the multiplication symbol `'*'` or the division symbol `'/'`, storing the result in `op` if successful. Once again, after ignoring white spaces, it attempts to parse another expression from `termMultDiv`. Similarly to `x`, if the parser succeeds, it stores the result in `y`. Now, the parser proceeds with `loop (MultE x y)` if `op` is a multiplication, or with `loop (DivE x y)` if `op` is a division. If there is nothing more to parse, then the loop returns `MultE x y` or `DivE x y`. Otherwise, it will execute `f (MultE x y)` or `f (DivE x y)`, *i.e.* it will loop. The execution of `f` is similar to the initial part of `parseMultDiv`, which we just described here. The only difference is that in `f` we do not need to parse anything to a variable `x`, since the argument given to `f` is the variable `x`. Hence, we need to parse `op` and `y`, *i.e.* which operation are we performing, either multiplication or division, and the next expression given by `termMultDiv`, before repeating the loop call. For instance, the expressions `"1*2*3"`, `"1/(2*3)"`, and `"(2*4)/(1/2)"` are respectively parsed to:

```
Right (MultE (MultE (Num 1) (Num 2)) (Num 3))
```

```
Right (DivE (Num 1) (MultE (Num 2) (Num 3)))
Right (DivE (MultE (Num 2) (Num 4)) (DivE (Num 1) (Num 2)))
```

Parsing addition/subtraction expressions

At last, we describe the parser `parsePlusMinus`, whose rationale is identical to that of `parseMultDiv`. In fact, one can obtain `parsePlusMinus` from `parseMultDiv` simply by replacing the multiplication and division symbols with the addition and subtraction symbols, and by replacing `termMultDiv` with `termPlusMinus`. The auxiliary parser `termPlusMinus`, analogous to `termMultDiv`, is defined as follows:

```
termPlusMinus :: GenParser Char st E
termPlusMinus = try(parseNum)
                <|> try(parseId)
                <|> parsePar
```

The absence of `parseMultDiv` in `termPlusMinus` highlights an important point. For instance, if we want to parse the string `"2*2+3"` as "the sum of 2*2 with 3", we need to put `2*2` between parentheses, *i.e.* `"(2*2)+3"`. By putting these two strings, `"2*2+3"` and `"(2*2)+3"`, in our parser we respectively obtain the following results:

```
Right (MultE (Num 2) (PlusE (Num 2) (Num 3)))
Right (PlusE (MultE (Num 2) (Num 2)) (Num 3))
```

Here we have that `"2*2+3"` is parsed as `"2*(2+3)"`, which gives priority to addition instead of multiplication, while `"(2*2)+3"` is parsed as an addition whose first operand is a multiplication. The choice to omit `parseMultDiv` in `termPlusMinus` obliges us to use parentheses when having multiplication and division operations around addition and subtraction. We suspect that, by including `parseMultDiv` in `termPlusMinus` and excluding `parsePlusMinus` in `termMultDiv` we would have the desired behavior.

The rationale behind `parseMultDiv` and `parsePlusMinus` shall be adopted and adapted when parsing Boolean expressions and commands. To illustrate this general parsing pattern, Figure 106 provides a schematic view of how such parsers operate.

The diagram is read as follows: we input a string, which the parser attempts to parse. If it succeeds, it means that it parsed the left operand of an expression, which is stored in `x`. We then attempt to parse the operation symbol, which, if successful, is stored in `op`. Next, we try to parse the right operand, which, if successful, is stored in `y`. At this point, if the input string has finished, then we return `op x y`. Otherwise, we continue parsing with `loop(op x y)`.

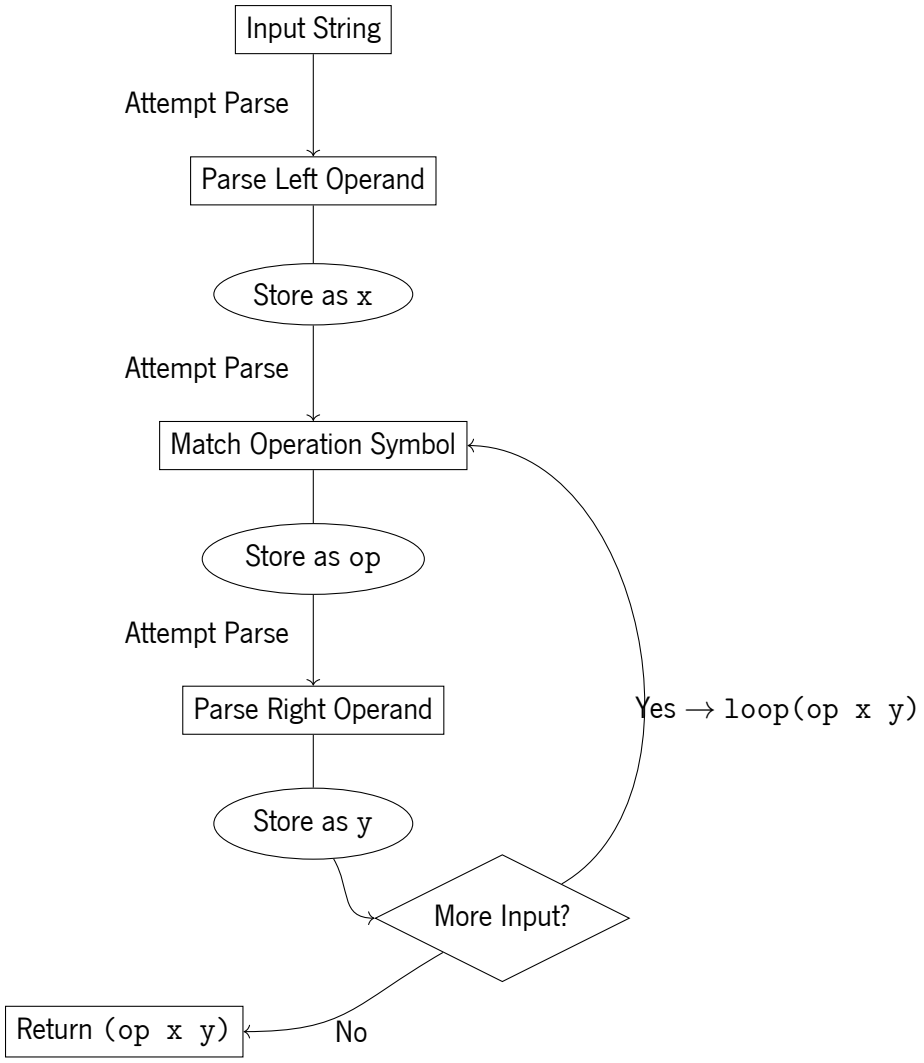


Figure 106: Generic parser of multiplication, division, addition, and subtraction expressions

In order to test if an integer expression is well written, we implemented a test parser called `testParseE`. For instance, the result of testing the string `"1+2*3/(2+1)"` is as follows:

```
ghci> testParseE "1+2*3/(2+1)"
Right (DivE (MultE (PlusE (Num 1) (Num 2)) (Num 3))
            (PlusE (Num 2) (Num 1)))
```

i.e. `"1+2*3/(2+1)"` is parsed as `"((1+2)*3)/(2+1)"`.

Parsing Boolean expressions

We now turn our attention to Boolean expressions. In Haskell, they are encoded by the following data type.

where `BTrue` and `BFalse` denote the true and false values, respectively, `Not B` represents the negation of a Boolean expression, `And B B` is the conjunction of Boolean expressions, and `OrB B B` is the disjunction of Boolean expressions. Furthermore, `Equ E E`, `Leq E E`, `Geq E E`, `Less E E`, and `Gre E E`

```
data B = BTrue
       | BFalse
       | Not B
       | And B B
       | OrB B B
       | Equ E E
       | Leq E E
       | Geq E E
       | Less E E
       | Gre E E
       deriving (Show, Eq)
```

Figure 107: Definition of data type B

represent the equality, less-than-or-equal, greater-than-or-equal, less-than, and greater-than relations over integer expressions, respectively.

The parser of Boolean expressions is organized similarly to the parser of integer expressions. Specifically, there exists a "brain" that selects the parser that best fits the input. Such a parser is called `parseBSelect` and is defined as follows:

```
parseBSelect :: GenParser Char st B
parseBSelect = try(parseAndOr)
               <|> try(parseNot)
               <|> try(parseEq)
               <|> try(parseLeq)
               <|> try(parseGeq)
               <|> try(parseLess)
               <|> try(parseGre)
               <|> try(parseTrue)
               <|> try(parseFalse)
               <|> parseParB
```

The *modus operandi* of `parseBSelect` mirrors that of `parseESelect`, hence we skip its explanation. We note, however, that Boolean expressions enclosed in parentheses are parsed by `parseParB` rather than `parsePar`, since the latter produces results of type `E` while the former produces results of type `B`. This distinction prevents type mismatches in the parser.

Parsing and/or expressions

Different from what was done in the parsers for integer expressions, here we explain the parsers of Boolean expressions by the order in which they appear in `parseBSelect`. We start with `parseAndOr`. The rationale behind this parser is the same as that behind `parseMultDiv` and `parsePlusMinus`. Consequently, it comes accompanied with a "brain", called `termAndOr`, which is defined as follows:

```
termAndOr :: GenParser Char st B
termAndOr =  try(parseNot)
           <|> try(parseEq)
           <|> try(parseLeq)
           <|> try(parseGeq)
           <|> try(parseLess)
           <|> try(parseGre)
           <|> try(parseTrue)
           <|> try(parseFalse)
           <|> parseParB
```

Inside conjunctive or disjunctive Boolean expressions, `termAndOr` selects the appropriate parser for the input, behaving analogously to `termMultDiv` and `termPlusMinus`.

The definition of `parseAndOr` is given below:

```
parseAndOr :: GenParser Char st B
parseAndOr = do
  x <- termAndOr
  spaces
  op <- try(char '&') <|> (char '|')
  spaces
  y <- termAndOr
  case op of
    '&' -> loop (And x y)
    '|' -> loop (OrB x y)
    where
      f x = do
        spaces
        op <- try(char '&') <|> (char '|')
        spaces
        y <- termAndOr
        spaces
        case op of
          '&' -> loop (And x y)
          '|' -> loop (OrB x y)
      loop t = try(f t) <|> return t
```

As it is possible to see, the definition of `parseAndOr` resembles the definition of `parseMultDiv` and `parsePlusMinus`. Hence we skip their explanation. We only note that conjunctions are represented by the symbol `'&'`, while disjunctions are represented by `'|'`.

Parsing negation of Boolean expressions

Next, we focus on the parsing of Boolean expressions that are negated. For this purpose, two parsers are defined: `bNot` and `parseNot`. The former is simply an alias for `termAndOr`, introduced for readability when defining `parseNot`:

```
parseNot :: GenParser Char st B
parseNot = do
  char '-'
  x <- bNot
  return (Not x)
```

The negation operator is represented by the symbol `'-'`, which coincides with subtraction in arithmetic expressions. However, since they occur in distinct contexts, no ambiguity arises. This can be seen with the following example, where we negate the result of the expression $(1 - 2) == 1$:

```
ghci> testParseB "-((1-2) == 1)"
Right (Not (Equ (MinusE (Num 1) (Num 2)) (Num 1)))
```

Parsing relations between integer expressions

We now focus on `parseEq`, which parses strings representing equalities between integer expressions (e.g., 1==1):

```
parseEq :: GenParser Char st B
parseEq = do
  x <- parseESelect
  spacesOnly
  string "=="
  spacesOnly
  y <- parseESelect
  return (Equ x y)
```

In sum, this parser attempts to parse an integer expression and, if successful, stores the result in `x`. Then, by ignoring solely spaces, the parser tries to parse the operator, *i.e.* ==. If it succeeds, it continues by ignoring solely spaces, until it attempts to parse the other integer expression. Again, if it successfully parses it, the result is stored in `y`. After this, it returns the parsed expression `Equ x y`.

The parsers `parseLeq`, `parseGeq`, `parseLess`, and `parseGre` behave in the same way as `parseEq`, hence we skip their explanation. However, it is essential to illustrate the symbols used in these parsers. For `parseLeq`, `parseGeq`, `parseLess`, and `parseGre`, the symbols used are, respectively, <=, >=, <, and >.

Parsing true/false Boolean values

Now, we present the parsers for the Boolean values “true” and “false”. The parsers are quite simple, since they just catch the string `"true"` or `"false"` and return the respective data type in Haskell. More concretely:

```
parseTrue :: GenParser Char st B
parseTrue = do
  string "true"
  return BTrue

parseFalse :: GenParser Char st B
parseFalse = do
  string "false"
  return BFalse
```

Parsing parenthesized Boolean expressions

At last, we have `parseParB`, which behaves similarly to `parsePar`. The difference between both is that the former parses Boolean expressions based on `parseBSelect` and the latter parses integer expressions based on `parseESelect`.

Similarly to what has done with the parsing of integer expressions, here we also defined a function, `testParseB`, to verify if a Boolean expression is well written or not. For instance, the result of parsing the string `"1 <= 4 & false | true"` is as follows:

```
ghci> testParseB "1 <= 4 & false | true"
Right (OrB (And (Leq (Num 1) (Num 4)) BFalse) BTrue)
```

The (controlled) shift gates introduced in Section 9.1.1 receive a phase as an argument. This argument is also parsed in the module *ParserAEBE*, using code very similar to that of integer expressions. Hence, we skip their explanation. Furthermore, the *ParserAEBE* module defines several auxiliary parsers for handling whitespace. Since these are either self-explanatory or explained when used, we also omit their details. The reader may consult the code of these parsers in the git repository [37].

Parsing commands

We now turn our attention to the module *ParserCom*. As suggested by the name, this module contains the parsers responsible for interpreting commands. To provide a brief overview of this module, the rationale behind it follows the same principles as the parsers for integer and Boolean expressions. In particular, the parsers for sequential, non-deterministic, concurrent, and probabilistic compositions follow a similar ideology to the one depicted in Figure 106. Having said this, let us show the data type that encodes commands.

where `Skip` denotes $skip$, `Asg String E` represents the deterministic assignment of integer expressions, `Reset QVar` denotes the reset of a quantum variable, `U G QVarList` represents the application of a unitary gate to a list of quantum variables, `Meas (String, QVar)` denotes the measurement of

```
data C = Skip
       | Asg String E
       | Reset QVar
       | U G QVarList
       | Meas (String, QVar)
       | Seq C C
       | Or C C
       | Paral C C
       | P Rational C C
       | IfC B C C
       | Whl B C
       | Await B C
       | Atom C
       deriving (Show, Eq)
```

Figure 108: Definition of data type C

a quantum variable and the storage of its result in a classical variable (here represented by the type `String`). `Seq C C`, `Or C C`, `Paral C C`, `P Rational C C` denote, respectively, the sequential, non-deterministic, concurrent, and probabilistic composition of commands. `IfC B C C` represents a usual if-then-else command, `Whl B C` denotes the usual while loop, `Await B C` represents the await command, and `Atom C` denotes the atom constructor.

The parser `parseCSelect` behaves similarly to `parseESelect` and `parseBSelect`. Its definition is as follows:

```
parseCSelect :: GenParser Char st C
parseCSelect = try(parseParal)
               <|> try(parseSeq)
               <|> try(parseOr)
               <|> try(parseProb)
               <|> try(parseSkip)
               <|> try(parseGate)
               <|> try(parseAsg)
               <|> try(parseMeas)
               <|> try(parseReset)
               <|> try(parseIf)
               <|> try(parseWhile)
               <|> try(parseAwait)
               <|> try(parseAtom)
               <|> parseParen
```

Parsing sequential/probabilistic/non-deterministic/parallel commands

The parsers `parseParal`, `parseSeq`, `parseOr`, and `parseProb`, follow a similar recursive structure as that shown in Figure 106. The main difference is that each operator is parsed separately, rather

than jointly (as is the case in, for example, `parseAndOr`). Consequently, there is no need to store the operator symbol in a variable such as `op` to handle multiple cases. Furthermore, each of these parsers is accompanied by an auxiliary parser - `comParal`, `comSeq`, `comOr`, and `comProb` - whose behavior is similar to that of `termAndOr`. To illustrate the difference, we show `parseParal` and omit the remaining parsers (`parseSeq`, `parseOr`, and `parseProb`) due to their similarity.

```
parseParal :: GenParser Char st C
parseParal = do
  x <- comParal
  spaces
  string "||"
  spaces
  y <- comParal
  loop (Paral x y)
    where
      f x = do
        spaces
        string "||"
        spaces
        y <- comParal
        loop (Paral x y)
      loop t = try(f t) <|> return t
```

The order in which the parsers appear in `parseCSelect` reflects their precedence when parentheses are not present. Parallel composition has the lowest precedence, followed by sequential composition, non-deterministic choice, and finally probabilistic choice. As in the case of integer expressions, this precedence is enforced by omitting lower-precedence constructs from the auxiliary parsers `com_`. Finally, we present the symbols that represent each operator. For parallel composition, we use the symbol `||`. For the sequential composition, we use the symbol `;`. For the non-deterministic choice, we use the word `or`. At last, for the probabilistic choice we use the symbol `(+)(p)`, where `p` is a rational number, parsed by `parseProbRat`.

Parsing $skip$ command

Let us now focus on the remaining parsers that are in `parseCSelect`. Starting with `parseSkip`. It simply parses the string `"skip"` and returns the respective Haskell data type `Skip`.

Parsing unitary gates

Next on line is the parser of unitary gates, `parseGate`, defined as follows:

```
parseGate :: GenParser Char st C
parseGate = try(pGate1Q)
```

```
        <|> try(pGate1QArg)
        <|> try(pGate2Q)
        <|> try(pGate2QArg)
        <|> pGate3Q
```

This parser has in its composition the following parsers: `pGate1Q`, which parses 1-qubit unitary gates; `pGate1QArg`, which parses 1-qubit unitary gates together with an argument for the phase; `pGate2Q`, which parses 2-qubit unitary gates; `pGate2QArg`, which parses 2-qubit unitary gates together with an argument for the phase; and `pGate3Q`, which parses 3-qubit unitary gates. Let us show the definition of `pGate1Q`.

```
pGate1Q :: GenParser Char st C
pGate1Q = do
    g <- gate1Q
    spacesOnly
    char '('
    spacesOnly
    q <- parseQVar
    qs <- (try(parseQVars) <|> return [])
    spacesOnly
    char ')'
    return (U g (q:qs))
```

This parser resorts to `gate1Q`, `parseQVar`, and `parseQVars` to correctly parse a 1-qubit unitary gate. The first parser recognizes one-qubit gate names (e.g., `"H"` for the Hadamard gate). The second parses the first quantum variable to which the gate applies, and behaves analogously to `parseId` in the *ParserAEBE* module. The third parses any additional qubits on which the gate acts. Note that, if the gate acts on a single qubit, then `qs` is the empty list. For example, the string `"H(q1,q2)"` is parsed as `Right (U H ["q1","q2"])`, and the string `"H(q1)"` as `Right (U H ["q1"])`.

Similarly to the parser `pGate1Q`, we have the parser `pGate1QArg`. What distinguishes these parsers is that the latter also needs to parse an argument, corresponding to the input phase of the shift gates. Recall that this phase is parsed in the module *ParserAEBE*. The code of `pGate1QArg`, as well as the code of `pGate2Q`, `pGate2QArg`, and `pGate3Q` can be consulted in [37].

Parsing assignments

We now focus our attention in the parser of assignments. Its definition is as follows:

```
parseAsg :: GenParser Char st C
parseAsg = do
    i <- parseIdeStr
    spacesOnly
```

```
    string ":="
    spacesOnly
    e <- parseESelect
    return (Asg i e)
```

This parser first parses the variable name using `parseIdeStr`, which behaves similarly to `parseId` in the module *ParserAEBE*. If it succeeds, it ignores only spaces (note that no tabs nor new lines are allowed) to parse the assignment symbol `:=`. By once again ignoring only spaces, it then parses an integer expression, resorting to `parseESelect`, with the result stored in `e`. The result is returned as `Asg i e`.

Parsing measurements and resets

Next in queue is the parser for measurements, whose definition is as follows:

```
parseMeas :: GenParser Char st C
parseMeas = do
    string "Meas"
    spacesOnly
    char '('
    spacesOnly
    c <- parseIdeStr
    spacesOnly
    char ','
    spacesOnly
    q <- parseQVar
    spacesOnly
    char ')'
    return (Meas (c,q))
```

In sum, this parser tries to parse the string `"Meas"`. Later on, it tries to parse the character `'('`, followed by the parsing of the classical variable, which is stored in `c`. Then, it parses the character `','`, followed by the quantum variable, which is stored in `q`. At last, it parses the char `')'` and returns `Meas (c,q)`. Note that, in between these parsers, spaces are ignored. In other words, between each element composing `Meas (c,q)` there is zero or more spaces.

We now proceed to the reset command, which is very similar to the measurement command. The parser `parseReset` differs from `parseMeas` by not including the classical variable, as shown next:

```
parseReset :: GenParser Char st C
parseReset = do
  string "Reset"
  spacesOnly
  char '('
  spacesOnly
```

```
  q <- parseQVar
  spacesOnly
  char ')'
  return (Reset q)
```

Parsing if-then-else commands

The parser for if-then-else statements, `parseIf`, is defined as follows:

```
parseIf :: GenParser Char st C
parseIf = do
    string "if"
    spacesOnly
    b <- parseBSelect
    spaces
    string "then"
    spaces
    char '{'
    spaces
    c1 <- parseCSelect
    spaces
    char '}'
    spaces
    string "else"
    spaces
    char '{'
    spaces
    c2 <- parseCSelect
    spaces
    char '}'
    return (IfC b c1 c2)
```

After parsing the string `"if"` and by ignoring only spaces, the parser proceeds to collect the Boolean expression resorting to `parseBSelect` and storing the result in `b`. From this moment, all the remaining elements are separated by white spaces, *i.e.* spaces, tabs, and newlines. The parser then collects the commands corresponding to the `"then"` and `"else"` branches, both enclosed in braces, *i.e.* `{` and `}`. The resulting command is returned as `IfC b c1 c2`.

Parsing while/await commands

The parser for while loops, `parseWhile`, is as follows:

```
parseWhile :: GenParser Char st C
parseWhile = do
  string "while"
  spacesOnly
  b <- parseBSelect
```

```
  spacesOnly
  string "do"
  spaces
  char '{'
  spaces
  c <- parseCSelect
  spaces
  char '}'
  return (Whl b c)
```

This parser recognizes constructs of the form `while b do { c }`, where `b` is a Boolean condition and `c` a command. The strings `"while"` and `"do"` and the Boolean condition `b` are separated only by spaces. After parsing `"do"`, the remaining elements are all separated through white spaces.

The parser for the $await$ command follows the same structure, differing only in that it begins by parsing the keyword `"await"`.

Parsing atomic regions

The parser for atomic regions is:

```
parseAtom :: GenParser Char st C
parseAtom = do
  string "Atom"
  spacesOnly
  char '('
  spaces
  c <- parseCSelect
  spaces
  char ')'
  return (Atom c)
```

The parser `parseAtom` parses commands of the form `Atom(c)`, where the string `"Atom"` and the opening parenthesis are separated only by spaces. The remaining parts are all separated by white spaces.

Parsing parenthesized commands

At last, we have the parser of commands between parentheses. However, its definition is similar to the previous case where parentheses were involved. Thus, we skip its explanation.

Similarly to what was previously done, we have the function `parseInputC` which allows us to verify if a string can be parsed into a Haskell data type command. For instance, the string `"H(q1); Meas(x1,q1)"` is parsed into the following Haskell data type command:

```
ghci> parseInputC "H(q1); Meas(x1,q1)"
Right (Seq (U H ["q1"]) (Meas ("x1","q1")))
```

With this, we finish the module *ParserCom*. We are ready to proceed to the last module regarding parsers, *ParserFile*, which handles input files containing one or more programs to be executed.

Parsing input file

Let us recall the specific format we use in the input file for each program.

```
---ProgramName---
hist: (nsamples, variables_to_plot)
k: nsteps_big_step
<
  command,
  classical_state,
  linking_function,
  quantum_state
>
---ProgramName---
```

In *ParserFile*, the parser responsible for parsing each program is the `splitRun` parser. Its definition is as follows:

```
splitRun :: GenParser Char st [((String, (Int, [[String]]), Int),
                                (C,SC,L,SQ))]
splitRun = do
  string "---"
  name <- many1 alphaNum
  string "---"
  char '\n'
  string "hist:"
  spacesOnly
  (rep, llvar) <- parseHist
  char '\n'
  string "k:"
  spacesOnly
  k <- many1 digit
  char '\n'
  (c,sc,l,sq) <- pConf
  separateOrJoined
  string "---"
  many1 alphaNum
  string "---"
  separateOrJoined
  listRun <- option [] splitRun
  return (((name, (rep, llvar), read k :: Int),(c,sc,l,sq)):listRun)
```

It begins by parsing the initial identifier, which consists of the string `"---"`, followed by `name`, and ending again with `"---"`. Note that `name` can only contain alphanumeric characters. After this, a newline

character must follow. If a space is placed before the newline, such as in `"---ProgramName--- \n"`, an error will occur. Next, information about the histogram is parsed. For that, the parser matches the string `"hist:"` and, ignoring spaces, extracts the histogram information using `parseHist`. The result is stored in the pair `(rep, llvar)`. Similarly to the initial identifier, this section is followed by a newline character. Then, the parser processes the information related to the big-step semantics. It matches the string `"k:"`, ignores spaces, and parses an integer, stored in `k`. This is followed by a newline character (`\n`). The next step parses the configuration of the program using the parser `pConf`, whose result is stored in `(c, sc, l, sq)`. After that, `separateOrJoined` ignores zero or more spaces and newline characters before parsing the ending identifier. Since the input file may contain more than one program, programs are separated by zero or more spaces and/or newline characters. Using the `option` parser, we either return an empty list (if no additional programs are found) or recursively parse the remaining programs. The result of this recursive step is stored in `listRun`, and finally we return: `((name, (rep, llvar), read k :: Int),(c,sc,l,sq)) : listRun`.

Parsing histogram information

Let us now delve into `parseHist`, whose definition is as follows:

```
parseHist :: GenParser Char st (Int, [[String]])
parseHist = do
  char '('
  rep <- many1 digit
  string ", "
  lvar <- parseListListVarHist
  char ')'
  return (read rep :: Int, lvar)
```

This parser begins by matching the opening parenthesis `'('`, immediately followed by one or more digits, which are stored in `rep` (representing the number of samples for the histogram). Then, it parses the string `", "`, followed by a list of variable lists using `parseListListVarHist`, whose result is stored in `lvar`. Note that the separator between `rep` and `lvar` must be exactly `", "` (a comma followed by a space). For instance, an input such as `"(100,__[x])"` (where underscores mark extra spaces) would produce an error.

As stated in Section 10.1, histograms can be built for individual or collective variables. For instance, if we want to build a histogram for two variables together, then the `hist` parameter can be `(10, [x,y])`. On the other hand, if we want to build a histogram for each variable, then the `hist` parameter can be`(10, [[x],[y]])`. Observe the difference in how variables are represented. The type `hist` is `(Int, [[String]])`, but we can also write `(10, [x,y])`, because `parseListListVarHist`

recognizes that only a single list is provided and automatically encloses it in an outer list. The definition of `parseListListVarHist` is as follows:

```
parseListListVarHist :: GenParser Char st [[String]]
parseListListVarHist = try (between (char '[') (char ']')
                           (sepBy parseListVarHist (char ',')))
                     <|> fmap (:[]) parseListVarHist
```

This parser first attempts to parse elements between `'['` and `']'`, separated by commas, using `parseListVarHist`. If that fails, it falls back to parsing a single list via `parseListVarHist` and wraps the result inside another list using `fmap (:[])`. The definition of `parseListVarHist` is as follows:

```
parseListVarHist :: GenParser Char st [String]
parseListVarHist = between (char '[') (char ']')
                           (sepBy (many1 alphaNum) (char ','))
```

Intuitively, this parser extracts alphanumeric variable names located between square brackets, separated by commas. For example, `[x1,x2]` is parsed as `["x1","x2"]`.

Parsing configurations

Now it is time to focus on `pConf`, where we tried to be as close to hand-writing as possible. Its definition is as follows:

```
pConf :: GenParser Char st (C,SC,L,SQ)
pConf = do
  char '<'
  spacesAndEnters
  com <- parseCSelect
  char ','
  spacesAndEnters
  sc <- pClassicStates
  (l,sq) <- option ([],fromLists [[]]) (do
                                           char ','
                                           spacesAndEnters
                                           pLSQ
                                       )
  spacesAndEnters
  char '>'
  return (com,sc,l,sq)
```

Configurations are delimited by the characters `'<'` and `'>'`. Each component inside is separated by zero or more spaces and/or newline characters. The command, classical state, linking function, and quantum state are separated by commas. The command is parsed using `parseCSelect` and

stored in com. The classical state is parsed by pClassicStates and stored in sc. The parser option allows the linking function and quantum state to be omitted or not. If they are present, they are parsed via pLSQ, and the result is stored in (l, sq). For instance, "<skip, [(x,0)]>" and "<skip, [(x,0)], [(q1,1)], (1.0)|0><0|>" are valid configurations.

Parsing classical states

The parser of classical states, pClassicStates, is defined as follows:

```
pClassicStates :: GenParser Char st SC
pClassicStates = do
  cs <- option [] (between (char '[') (char ']') (sepBy state separator))
  return cs
  where state = do
          char '('
          x <- parseIdeStr
          char ','
          v <- parseNum
          char ')'
          return (x, rmvNat v)
        separator = char ',' <* spaces
```

This parser uses the option, between, and sepBy combinators from *parsec*. A classical state is written between square brackets, with its elements separated by commas and optional whitespace. Each state is enclosed in parentheses '(' and ')', where x is the classical variable name (parsed with parseIdeStr) and v is its integer value (parsed with parseNum). The only separator between x and v is a comma. For example, (x,1) is valid, while (x, 1) would produce an error. The helper function rmvNat simply removes the Num constructor from v.

Parsing linking function and quantum states

We now turn to pLSQ, whose definition is as follows:

```
pLSQ :: GenParser Char st (L,SQ)
pLSQ = do
  l <- parserL
  char ','
  spacesAndEnters
  sq <- pOpDen
  return (l,sq)
```

This parser first parses the linking function via parserL, storing the result in l. Then, after parsing a comma and ignoring any spaces or newlines, it parses the density operator through pOpDen and stores it in sq. Finally, it returns the pair (l, sq).

The parser for linking functions is very similar to `pClassicalStates`, so we omit its details here. Furthermore, we remind the reader that the code can be consulted in [37].

At last, we have the parsing of quantum states. Quantum states can be written in two forms: the first form isolates the probability from the state, while the second form accompanies the state with its respective probability. For example, `(0.5)(|0><0| + |1><1|)` and `(0.5)|0><0| + (0.5)|1><1|` are valid quantum states according to the first and second forms, respectively. The parsers for each way quantum states can be written follow a similar rationale to the parallel composition. Thus, we omit their code, which can be consulted in [37].

Chapter C
Proofs of Part II

C.1 Proofs of Chapter 10

Proof of Proposition 10.2.2

We start by showing that `fmap id = id`.

Proof. `fmap id (ProbT mdist)`
= {Definition 10.2.1}

```
 ProbT $ do
     dist <- mdist
     return [(id x, p) | (x, p) <- dist]
```

=

```
 ProbT $ do
     dist <- mdist
     return [(x, p) | (x, p) <- dist]
```

=

```
 ProbT $ do
     dist <- mdist
     return dist
```

= {Desugar}

```
 ProbT(mdist >>= \dist -> return dist)
```

=

```
 ProbT(mdist >>= return)
```

= {Right identity `m`}

```
 ProbT mdist
```

=

```
 id(ProbT mdist)
```

□

We now show that `fmap (g . h) = fmap g . fmap h`. We begin by expanding the LHS.

Proof. `fmap (g . h) (ProbT mdist)`
={Definition 10.2.1}

```
ProbT $ do
   dist <- mdist
   return [((g . h) x, p) | (x, p) <- dist]
```

=

```
ProbT $ do
   dist <- mdist
   return [(g (h x), p) | (x, p) <- dist]
```

We now expand the RHS.

```
(fmap g . fmap h) (ProbT mdist)
```

=

```
fmap g (fmap h (ProbT mdist))
```

={Definition 10.2.1}

```
fmap g (ProbT $ do
            dist <- mdist
            return [(h x, p) | (x, p) <- dist])
```

={Definition 10.2.1}

```
ProbT $ do
   dist' <- do
               dist <- mdist
               return [(h x, p) | (x, p) <- dist]
   return [(g y, q) | (y, q) <- dist']
```

={do sugar}

```
ProbT $ do
   dist <- mdist
   dist' <- return [(h x, p) | (x, p) <- dist]
   return [(g y, q) | (y, q) <- dist']
```

=

```
ProbT $ do
   dist <- mdist
   return [(g y, q) | (y, q) <- [(h x, p) | (x, p) <- dist]]
```

=

```
ProbT $ do
   dist <- mdist
   return [(g (h x), p) | (x, p) <- dist]
```

=

```
fmap  (g . h ) (ProbT mdist)
```

□

Hence `ProbT` is part of the Functor instance.

Proof of Proposition 10.2.4

We start by showing `pure id <*> x = x`

Proof. `pure id <*> (ProbT mdist)`
={Definition 10.2.3}

```
(ProbT $ return [(id, 1)]) <*> (ProbT mdist)
```

={Definition 10.2.3}

```
ProbT $ do
   distf <- return [(id, 1)]
   dist <- mdist
   return [(f x, pf * px) | (f, pf) <- distf, (x, px) <- dist]
```

={`distf = [(id,1)]`}

```
ProbT $ do
   dist <- mdist
   return [(x, px) | (x, px) <- dist]
```

=

```
ProbT $ do
   dist <- mdist
   return dist
```

={Desugar, Right identity `m`}

```
ProbT mdist
```

□

We now show that `pure f <*> pure x = pure (f x)`.

Proof. `pure f <*> pure x`
={Definition 10.2.3}

```
ProbT $ return [(f, 1)] <*> ProbT (return [(x, 1)])
```

={Definition 10.2.3, `distf=[(f, 1)]`, `dist=[(x,1)]`}

```
ProbT $ do
   distf <- return [(f, 1)]
   dist <- return [(x, 1)]
   return [(f x, 1)]
```

=

```
ProbT $ do
   return [(f x, 1)]
```

={Desugar, Right identity `m`}

```
ProbT $ return [(f x, 1)]
```

={Definition 10.2.3}

```
  pure (f x)
```

□

We now show `x <*> pure y = pure ($ y) <*> x`.

Proof. We begin by expanding the LHS.

```
 x <*> pure y
```

={let `x = ProbT mdistf`, Definition 10.2.3}

```
 (ProbT mdistf) <*> (ProbT $ return [(y,1)])
```

={Definition 10.2.3}

```
 ProbT $ do
    distf <- mdistf
    dist <- return [(y, 1)]
    return [(f x, pf * px) | (f, pf) <- distf, (x, px) <- dist]
```

={`dist = [(y, 1)]`}

```
 ProbT $ do
    distf <- mdistf
    return [(f y, pf) | (f, pf) <- distf]
```

Now we expand the RHS.

```
 pure ($ y) <*> x
```

={let `x = ProbT mdistf`, Definition 10.2.3}

```
 (ProbT $ return [($ y, 1)]) <*> (ProbT mdistf)
```

={Definition 10.2.3}

```
 ProbT $ do
    distf <- return [($ y, 1)]
    dist <- mdistf
    return [(x f, px * pf) | (x, px) <- distf, (f, pf) <- dist]
```

={`distf = [($ y, 1)]`}

```
 ProbT $ do
    dist <- mdistf
    return [(($ y) f, pf) | (f, pf) <- dist]
```

=

```
 ProbT $ do
    dist <- mdistf
    return [(f y, pf) | (f, pf) <- dist]
```

Since the LHS and the RHS are equal, what we want to show holds. □

We now show `x <*> (y <*> z) = pure (.) <*> x <*> y <*> z`

Proof. We start by expanding the LHS.

```
 x <*> (y <*> z)
```

={Let `x = ProbT mdistfx`, `y = ProbT mdistfy`, `z = ProbT mdistz`}

```
 (ProbT mdistfx) <*> ((ProbT mdistfy) <*> (ProbT mdistz))
```

={Definition 10.2.3}

```
 (ProbT mdistfx) <*> (ProbT $ do
                         distfy <- mdistfy
                         distz <- mdistz
                         return [(fy z, pfy * pz) | (fy, pfy) <- distfy,
                                                    (z, pz) <- distz])
```

={Definition 10.2.3}

```
 ProbtT $ do
    distfx <- mdistfx
    dist <- do
       distfy <- mdistfy
       distz <- mdistz
       return [(fy z, pfy * pz) | (fy, pfy) <- distfy, (z, pz) <- distz]
    return [(fx v, pfx * pv) | (fx, pfx) <- distfx, (v, pv) <- dist]
```

={do sugar}

```
 ProbtT $ do
    distfx <- mdistfx
    distfy <- mdistfy
    distz <- mdistz
    dist <- return [(fy z, pfy * pz) | (fy, pfy) <- distfy,
                                       (z, pz) <- distz]
    return [(fx v, pfx * pv) | (fx, pfx) <- distfx, (v, pv) <- dist]
```

={`dist = [(fy z, pfy * pz) | (fy, pfy) <- distfy, (z, pz) <- distz]`}

```
 ProbtT $ do
    distfx <- mdistfx
    distfy <- mdistfy
    distz <- mdistz
    return [((fx . fy) z, pfx * pfy * pz) | (fx, pfx) <- distfx,
                                            (fy z, pfy * pz) <- dist]
```

We now expand the RHS.

```
 pure (.) <*> x <*> y <*> z
```

={Let `x = ProbT mdistfx`, `y = ProbT mdistfy`, `z = ProbT mdistz`, Definition 10.2.3}

```
 ((ProbT $ return [((.),1)]) <*> (ProbT mdistfx))
                             <*> (ProbT mdistfy)
                             <*> (ProbT mdistz)
```

={Definition 10.2.3}

```
 (ProbT $ do
     distf <- return [((.)),1]
     distfx <- mdistfx
     return [(f fx, pf * pfx) | (f, pf) <- distf, (fx, pfx) <- distfx]
 ) <*> (ProbT mdistfy) <*> (ProbT mdistz)
```

={distf = [((.)),1]}

```
 ((ProbT $ do
     distfx <- mdistfx
     return [((.) fx, pfx) | (fx, pfx) <- distfx]) <*> (ProbT mdistfy)
 ) <*> (ProbT mdistz)
```

={Definition 10.2.3}

```
 ( ProbT $ do
      distf' <- do
          distfx <- mdistfx
          return [((.) fx, pfx) | (fx, pfx) <- distfx])
      distfy <- mdistfy
      return [(f fy, pf * pfy) | (f, pf) <- distf', (fy, pfy) <- distfy]
 ) <*> (ProbT $ mdistz)
```

={do sugar}

```
 ( ProbT $ do
      distfx <- mdistfx
      distf' <- return [((.) fx, pfx) | (fx, pfx) <- distfx]
      distfy <- mdistfy
      return [(f fy, pf * pfy) | (f, pf) <- distf', (fy, pfy) <- distfy]
 ) <*> (ProbT $ mdistz)
```

={[((.) fx, pfx) | (fx, pfx) <- distfx]}

```
 ( ProbT $ do
      distfx <- mdistfx
      distfy <- mdistfy
      return [(((.) fx) fy, pfx * pfy) | ((.) fx, pfx) <- distf',
                                         (fy, pfy) <- distfy]
 ) <*> (ProbT $ mdistz)
```

=

```
 ( ProbT $ do
      distfx <- mdistfx
      distfy <- mdistfy
      return [(fx . fy, pfx * pfy) | ((.) fx, pfx) <- distf',
                                     (fy, pfy) <- distfy]
 ) <*> (ProbT $ mdistz)
```

={Definition 10.2.3}

```
 ProbT $ do
    distf'' <- do
       distfx <- mdistfx
```

```
      distfy <- mdistfy
      return [(fx . fy, pfx * pfy) | ((.) fx, pfx) <- distf',
                                     (fy, pfy) <- distfy]
   distz <- mdistz
   return [(f z, pf * pz) | (f, pf) <- distf'',
                            (z, pz) <- distz]
```

={do sugar}

```
ProbT $ do
   distfx <- mdistfx
   distfy <- mdistfy
   distf'' <- return [(fx . fy, pfx * pfy) | ((.) fx, pfx) <- distf',
                                             (fy, pfy) <- distfy]
   distz <- mdistz
   return [(f z, pf * pz) | (f, pf) <- distf'', (z, pz) <- distz]
```

={distf'' = [(fx . fy, pfx * pfy) | ((.) fx, pfx) <- distf',
                                    (fy, pfy) <- distfy]}

```
ProbT $ do
   distfx <- mdistfx
   distfy <- mdistfy
   distz <- mdistz
   return [((fx . fy) z, pfx*pfy*pz) | (fx . fy, pfx*pfy)<-distf'',
                                       (z, pz) <- distz]
```

Since LHS and RHS are the same, we are done. □

At last, we show that `pure f <*> x = fmap f x`

Proof. We start with the LHS.

```
pure f <*> x
```

={Definition 10.2.3, let `x = ProbT mdist`}

```
(ProbT $ return [(f, 1)]) <*> (ProbT mdist)
```

={Definition 10.2.3}

```
ProbT $ do
   distf <- return [(f, 1)]
   dist <- mdist
   return [(g y, pg * p) | (g, pg) <- distf, (y, p) <- dist]
```

={`distf = [(f,1)]`}

```
ProbT $ do
   dist <- mdist
   return [(f y, p) | (y, p) <- dist]
```

We now go to the RHS.

```
fmap f x
```

={Let `x = ProbT mdist`}

```
fmap f (ProbT mdist)
```

={Definition 10.2.1}

```
ProbT $ do
   dist <- mdist
   return [(f y, p) | (y, p) <- dist]
```

Since the LHS and RHS are equal, we are done. □

Thus, `ProbT` belongs to the Applicative instance.

Proof of Proposition 10.2.6

Proof. We start by showing that `return x >>= f = f x`. This law is also known as "Left identity".

```
return  x >>= f
```

={Definition 10.2.5}

```
(ProbT $ return [(x, 1)]) >>= f
```

={Definition 10.2.5}

```
ProbT $ do
   dist <- return [(x, 1)]
   appf <- joinProb f dist
   return (concat appf)
```

={`dist = [(x, 1)]`}

```
ProbT $ do
   appf <- joinProb f [(x,1)]
   return (concat appf)
```

={Reducing `joinProb f [(x,1)]`}

```
ProbT $ do
   appf <- do
       dist' <- runProbT $ f x
       return [dist']
   return (concat appf)
```

={do sugar}

```
ProbT $ do
   dist' <- runProbT $ f x
   appf <- return [dist']
   return (concat appf)
```

={`appf = [dist']`, apply `concat`}

```
ProbT $ do
   dist' <- runProbT $ f x
   return dist'
```

={Desugar do notation}

```
ProbT $ (runProbT $ f x >>= \dist' -> return dist')
```

=

```
 ProbT (runProbT $ f x >>= return)
```

={Right identity on `m`}

```
 ProbT (runProbT $ f x)
```

=

```
 f x
```

□

The step "Reducing `joinProb f [(x,1)]`" is as follows:

```
 joinProb f [(x,1)]
```

={Definition 10.2.5}

```
 forM [(x,1)] $ \(x', px') -> do
      let ProbT mdist = f x'
      dist' <- mdist
      return [(y, py * px) | (y, py) <- dist']
```

={unroll `forM`}

```
   do
 let ProbT mdist = f x
 dist' <- mdist
 return [(y, py) | (y, py) <- dist']
```

={`mdist = runProbT $ ProbT mdist`}

```
   do
 let ProbT mdist = f x
 dist' <- runProbT $ ProbT mdist
 return [(y, py) | (y, py) <- dist']
```

=

```
   do
 dist' <- runProbT $ f x
 return [dist']
```

We now proceed to `m >>= return = m`, which is also known as "Right identity".

Proof. `m >>= return`
={Let `m = ProbT mdist`}

```
 (ProbT mdist) >>= return
```

={Definition 10.2.5}

```
 ProbT $ do
    dist <- mdist
    appf <- joinProb return dist
    return (concat appf)
```

={Reducing `joinProb return dist` }

```
 ProbT $ do
   dist <- mdist
   appf <- return [(x, px) | (x, px) <- dist]
   return (concat appf)
```

={`appf = [[(x, px)] | (x, px) <- dist]`, apply `concat`}

```
 ProbT $ do
    dist <- mdist
    return dist
```

={Desugar do}

```
 ProbT (mdist >>= \dist -> return dist)
```

=

```
 ProbT (mdist >>= return)
```

={`(mdist >>= return)` = `mdist`}

```
 ProbT mdist
```

=

```
 m
```

□

The step "Reducing `joinProb return dist` " is as follows:

```
 joinProb return dist
```

={Definition 10.2.5}

```
 forM dist $ \(x, px) -> do
    let ProbT mdist = return x
    dist' <- mdist
    return [(y, py * px) | (y, py) <- dist']
```

={`return x = ProbT $ return [(x,1)]`}

```
 forM dist $ \(x, px) -> do
    let ProbT mdist = ProbT $ return [(x,1)]
    dist' <- mdist
    return [(y, py * px) | (y, py) <- dist']
```

={`mdist = return[(x,1)]`, `dist' = [(x,1)]`}

```
forM dist $ \(x, px) -> do
   return [(x, px)]
```

={unroll `forM`}

```
 return [[(x, px)] | (x, px) <- dist]
```

At last, we check `(m >>= f) >>= g = (m >>= (\x -> fx >>= g))`. We start by the LHS.

Proof. `(m >>= f) >>= g`
={Let `m = ProbT mdist`}

```
(ProbT mdist >>= f) >>= g
```

={Definition 10.2.5}

```
(ProbT $ do
    dist1 <- mdist
    appf <- joinProb f dist1
    return (concat appf)) >>= g
```

={Definition 10.2.5}

```
ProbT $ do
   dist2 <- do
       dist1 <- mdist
       appf <- joinProb f dist1
       return (concat appf)
   appg <- joinProb g dist2
   return (concat appg)
```

={do sugar}

```
ProbT $ do
   dist1 <- mdist
   appf <- joinProb f dist1
   dist2 <- return (concat appf)
   appg <- joinProb g dist2
   return (concat appg)
```

={Definition 10.2.5}

```
ProbT $ do
   dist1 <- mdist
   appf <- forM dist1 $ \(x, px) -> do
        let ProbT mdist' = f x
        dist' = mdist'
        return [(y, py * px) | (y, py) <- dist']
   dist2 <- return (concat appf)
   appg <- forM dist2 $ \(y', py') -> do
        let ProbT mdist'' = g y'
        dist'' <- mdist''
        return [(z, pz * py') | (z, pz) <- dist'']
   return (concat appg)
```

□

We now note the following:

- for each `(x, px)` in `dist1`, `f` is applies to `x`, which later originates `dist'`, whose elements are of the form `(y, py)`
- we then multiply the probabilities of `dist'` with `px`; hence elements of `dist'` are now of the form `(y, py * px)`
- for each `(y, py * px)` in `dist'`, we apply `g` to `y`, which later originates `dist''`, whose elements are of the form `(z, pz)`
- we then multiply the probabilities of `dist''` with `py * px`; hence the elements of `dist''` are now of the form `(z, pz * py * px)`

Now we expand the RHS.

Proof.
```
m >>= (\x -> f x >>= g)
```
={Let `m = ProbT mdist`}
```
(ProbT mdist) >>= (\x -> f x >>= g)
```
={Definition 10.2.5}
```
ProbT $ do
   dist1 <- mdist
   appf <- joinProb (\x -> f x >>= g) dist1
   return (concat appf)
```
={Reducing `joinProb (\x -> f x >>= g) dist1`, with `x'` $\rightarrow$ `x`}
```
ProbT $ do
   dist1 <- mdist
   appf <- forM dist1 $ \(x, px) -> do
        dist' <- runProbT $ f x
        appg <- joinProb g dist'
        return [(y, px * py) | (y, py) <- concat appg]
   return (concat appf)
```
={Definition 10.2.5}
```
ProbT $ do
   dist1 <- mdist
   appf <- forM dist1 $ \(x, px) -> do
        dist' <- runProbT $ f x
        appg <- forM dist' $ \(y', py') -> do
             let ProbT mdist'' = g y'
             mdist'' <- mdist''
             return [(z, pz * py') | (z, pz) <- dist'']
```

```
        return [(y, px * py) | (y, py) <- concat appg]
    return (concat appf)
```

□

We now note the following:

- for each `(x, px)` in `dist1`, `f` is applied to `x` which later originates `dist'`, whose elements are of the form `(y, py)`
- for each `(y, py)` in `dist'`, we apply `g` to `y`, which later originates `dist''`, whose elements are of the form `(z, pz)`
- we then multiply the probabilisties of `dist''` with `py`; the resultant elements are of the form `(z, pz * py)`, which are elements of `concat appg`
- lastly, we multiply the probabilisties of `concat appg` with `px`; this results in elements whose form is `(z, pz * py * px)`

In the RHS, the step "Reducing `joinProb (\x -> f x >>= g) dist1`" is obtained as follows:

```
joinProb (\x -> f x >>= g) dist1
```

={Definition 10.2.5}

```
forM dist1 $ \(x', px') -> do
   let ProbT mdist = (\x -> f x >>= g) x'
   dist2 <- mdist
   return [(y, px' * py) | (y, py) <- dist2]
```

=

```
forM dist1 $ \(x', px') -> do
   let ProbT mdist = f x' >>= g
   dist2 <- mdist
   return [(y, px' * py) | (y, py) <- dist2]
```

Since `runProbT (f x' >>= g)` equals

```
runProbT (ProbT $ do
   dist' <- runProbT $ f x'
   appg <- joinProb g dist'
   return (concat appg))
```

=

```
    do
  dist' <- runProbT $ f x'
  appg <- joinProb g dist'
  return (concat appg)
```

Then `mdist` equals

```
    do
  dist' <- runProbT $ f x'
  appg <- joinProb g dist'
  return (concat appg)
```

Thus

```
 forM dist1 $ \(x', px') -> do
  dist2 <- do
    dist' <- runProbT $ f x'
    appg <- joinProb g dist'
    return (concat appg)
  return [(y, px' * py) | (y, py) <- dist2]
```

={do sugar}

```
 forM dist1 $ \(x', px') -> do
  dist' <- runProbT $ f x'
  appg <- joinProb g dist'
  dist2 <- return (concat appg)
  return [(y, px' * py) | (y, py) <- dist2]
```

={dist2 = concat appg}

```
 forM dist1 $ \(x', px') -> do
  dist' <- runProbT $ f x'
  appg <- joinProb g dist'
  return [(y, px' * py) | (y, py) <- concat appg]
```

Since the LHS and RHS give the same elements, then we can conclude that they are equal.

Thus `ProbT` belongs to the Monad instance.

Proof of Proposition 10.2.8

Proof. We start by showing that `lift(return a) = return a`.

```
 lift(return a)
```

={Definition 10.2.7}

```
 ProbT $ liftM (\x -> [(x,1)]) (return a)
```

={Definition 10.2.7}

```
ProbT $ do
   a <- return a
   return [(a,1)]
```

={Desugar}

```
ProbT $ ((return a) >>= \a' -> return [(a',1)])
```

={Left identity m}

```
ProbT $ ((\a' -> return [(a',1)]) a)
```

=

```
ProbT $ return [(a,1)]
```

={Definition 10.2.5}

```
return a
```

□

We now show that `lift (m >>= f)` equals `lift m >>= (lift . f)`. We begin by unrolling the LHS.

```
lift (ma >>= f)
```

={Definition 10.2.7}

```
ProbT $ liftM (\x -> [(x,1)]) (ma >>= f)
```

={Definition 10.2.7}

```
ProbT $ do
   y <- (ma >>= f)
   return [(y,1)]
```

={do notation}

```
ProbT $ do
   y <- do
     a <- ma
     f a
   return [(y,1)]
```

={do sugar}

```
ProbT $ do
   a <- ma
   y <- f a
   return [(y,1)]
```

We now unroll the RHS.

```
 lift ma >>= (lift . f)
```

={Definition 10.2.7}

```
 (ProbT $ liftM (\x -> [(x,1)]) ma) >>= (lift . f)
```

={Definition 10.2.5}

```
 ProbT $ do
    dist <- liftM (\x -> [(x,1)]) ma
    appf <- joinProb (lift . f) dist
    return (concat appf)
```

={Definition 10.2.7}

```
 ProbT $ do
    dist <- do
       a <- ma
       return [(a,1)]
    appf <- joinProb (lift . f) dist
    return (concat appf)
```

={Desugar}

```
 ProbT $ do
    a <- ma
    dist <- return [(a,1)]
    appf <- joinProb (lift . f) dist
    return (concat appf)
```

={dist = [(a,1)]}

```
 ProbT $ do
    a <- ma
    appf <- joinProb (lift . f) [(a,1)]
    return (concat appf)
```

={Reducing `joinProb (lift . f) [(a,1)]`}

```
 ProbT $ do
    a <- ma
    appf <- do
       y <- f a
       return [[(y,1)]]
   return (concat appf)
```

={do sugar}

```
 ProbT $ do
    a <- ma
    y <- f a
    appf <- return [[(y,1)]]
    return (concat appf)
```

={appf = [[(y,1)]], applying concat}

```
ProbT $ do
   a <- ma
   y <- f a
   return [(y,1)]
```

Since the LHS and the RHS are the same, we can conclude that `ProbT` belongs to the instance `MonadTrans`.

The step "Reducing `joinProb (lift . f) [(a,1)]`" is as follows:

```
joinProb (lift . f) [(a,1)]
```

={Definition 10.2.5}

```
forM [(a,1)] $ \(x, px) -> do
     let ProbT mdist = (lift . f) x
     dist' <- mdist
     return [(y', px * py') | (y', py') <- dist']
```

=(lift . f) x = lift(f x)

```
forM [(a,1)] $ \(x, px) -> do
     let ProbT mdist = lift(f x)
     dist' <- mdist
     return [(y', px * py') | (y', py') <- dist']
```

={unroll forM}

```
    do
  let ProbT mdist = lift(f a)
  dist' <- mdist
  return [[(y', py') | (y', py') <- dist']]
```

={dist' = [(y', py') | (y', py') <- dist']}

```
    do
  let ProbT mdist = lift(f a)
  dist' <- mdist
  return [dist']
```

={Definition 10.2.7}

```
    do
  let ProbT mdist = ProbT $ liftM (\x -> [(x,1)]) (f a)
  dist' <- mdist
  return [dist']
```

={Definition 10.2.7}

```
    do
  let ProbT mdist = ProbT $ do
                      y <- f a
                      return [(y,1)]
  dist' <- mdist
  return [dist']

=

    do
  dist' <- do
        y <- f a
        return [(y,1)]
  return [dist']
```

={do sugar}

```
    do
  y <- f a
  dist' <- return [(y,1)]
  return [dist']
```

={dist' = [(y,1)]}

```
    do
  y <- f a
  return [[(y,1)]]
```

Chapter D
Data collected for testing the tool

The data collected for the branch explosion case is shown in Table 6.

it	BigStep (s)	Histogram (s)
1	0.005731	0.01788
2	0.005803	0.01747
3	0.005868	0.01809
4	0.005776	0.01794
5	0.006041	0.01793
6	0.005920	0.01724
7	0.006444	0.01809
8	0.006814	0.01823
9	0.009250	0.01783
10	0.01449	0.01757
11	0.02492	0.01712
12	0.04861	0.01754
13	0.11800	0.01742
14	0.22970	0.01713
15	0.48840	0.01767
16	1.10300	0.01737
17	2.34800	0.01700
18	4.96400	0.01720
19	10.1500	0.01744
20	21.9500	0.01749
21	89.5000	0.01720
22	623.800	0.01697
23	3086.00	0.01693

Table 6: Execution times of BigStep and Histogram semantics.

The data collected for the qubit scaling case is shown in Table 7.

qbits	Tool (s)	QIO (s)
1	0.000216	0.000001539
2	0.000501	0.000005392
3	0.001800	0.000018510
4	0.012800	0.000065640
5	0.121000	0.000238700
6	1.500000	0.000928900
7	18.30000	0.003640000
8	260.0000	0.014900000
9	4530.000	0.060350000
10	113000.0	0.254600000

Table 7: Execution times of the tool and QIO for increasing numbers of qubits.

Chapter E
Calculations for the Quantum Dining Philosophers

To investigate why [5, Theorem 3] does not hold, we followed the calculations in [86] for the case with three philosophers. In [86, Table 3] it is showed that the states arising from $|111111\rangle$ should cancel those arising from $|000000\rangle$. However, for the relevant states, the amplitude coming from $|000000\rangle$ is $\frac{1}{2\sqrt{2}}$, whereas the corresponding amplitudes are either $-\frac{1}{2\sqrt{2}}$, $\frac{(e^{i\frac{\pi}{3}})^2e^{-i\frac{\pi}{3}}}{2\sqrt{2}}$, or $\frac{(e^{-i\frac{\pi}{3}})^2e^{i\frac{\pi}{3}}}{2\sqrt{2}}$. The amplitudes of the last two terms are not equal to $-\frac{1}{2\sqrt{2}}$, since $(e^{i\frac{\pi}{3}})^2e^{-i\frac{\pi}{3}} \neq -1$ and $(e^{-i\frac{\pi}{3}})^2e^{i\frac{\pi}{3}} \neq 1$. Hence, the states that should cancel do not, and consequently, the symmetry is not broken.

To see this more clearly, let us focus only on the symmetry-breaking protocol – specifically, considering the case where the protocol advances to the third phase, as this is where the magic unitary gate V_n is applied. At the end of this phase, we expect that the states of qubits q1, q2, and q3 should not all be equal to $|0\rangle$ or all equal to $|1\rangle$, since these configurations would mean that all philosophers attempt to pick either their left or right chopstick, respectively, which could lead to a deadlock.

At the beginning of the third phase, the quantum state of the system is as follows (assuming that the qubits are organized as $|\mathtt{q1q12qa1}\rangle \otimes |\mathtt{q2q23qa2}\rangle \otimes |\mathtt{q3q31qa3}\rangle$):

$$\frac{1}{\sqrt{2}}\left(|000000000\rangle + |110110110\rangle\right)$$

After applying all the required CNOT gates, and before executing any V_n gate, the quantum state becomes:

$$\frac{1}{\sqrt{2}}(|000000000\rangle + |101101101\rangle)$$

Summing up, the pairs of qubits q12 and qa1, q23 and qa2, and q31 and qa3 have exchanged their values.

The next step is to apply the V_n gates to qubits q1, q2, q3, qa1, qa2, and qa3. Note that q12, q23, and q31 are not affected by V_n, so we exclude them from the calculation. This reduces the state to $\frac{1}{\sqrt{2}}(|000000\rangle + |111111\rangle)$, with the qubits ordered as $|\mathtt{q1qa1}\rangle \otimes |\mathtt{q2qa2}\rangle \otimes |\mathtt{q3qa3}\rangle$. We then compute the effect of V_n on $|000000\rangle$ and $|111111\rangle$ separately, and sum the results.

Let us start the calculations for $|000000\rangle$. We begin by applying V_n to $|00\rangle$ (ignoring the normalization

factor of V_n, hence the usage of $\propto$ instead of $=$):

$$V_n|00\rangle \propto \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \sqrt{R_n} & \frac{e^{i\frac{\pi}{n}}}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\sqrt{R_n}e^{-i\frac{\pi}{n}} & \frac{e^{-i\frac{\pi}{n}}}{\sqrt{2}} \\ \sqrt{R_n} & 0 & \frac{e^{-i\frac{\pi}{2n}}I_n}{i\sqrt{2}R_{2n}} & -\sqrt{R_n} \\ 0 & \sqrt{1+R_n} & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \propto \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle + \sqrt{R_n}|10\rangle$$

Now let us start by applying V_n to q1, q2, q3, qa1, qa2, and qa3. For this, it is useful to note that R_n, with $n = 3$, is equal to $\frac{1}{2}$.

$$\begin{aligned}
&V_n|00\rangle \otimes V_n|00\rangle \otimes V_n|00\rangle \\
&\propto \left(\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle + \sqrt{R_n}|10\rangle\right) \otimes \left(\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle + \sqrt{R_n}|10\rangle\right) \\
&\qquad \otimes \left(\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle + \sqrt{R_n}|10\rangle\right) \\
&= \left(\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|10\rangle\right) \otimes \left(\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|10\rangle\right) \\
&\qquad \otimes \left(\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|10\rangle\right) \\
&= \frac{1}{2\sqrt{2}}(|00\rangle + |01\rangle + |10\rangle) \otimes (|00\rangle + |01\rangle + |10\rangle) \otimes (|00\rangle + |01\rangle + |10\rangle) \\
&= \frac{1}{2\sqrt{2}}(|0000\rangle + |0001\rangle + |0010\rangle + |0100\rangle + |0101\rangle + |0110\rangle + |1000\rangle + |1001\rangle + |1010\rangle) \\
&\qquad \otimes (|00\rangle + |01\rangle + |10\rangle) \\
&= \frac{1}{2\sqrt{2}}(|000000\rangle + |000001\rangle + |000010\rangle + |000100\rangle + |000101\rangle + |000110\rangle \\
&\qquad + |001000\rangle + |001001\rangle + |001010\rangle + |010000\rangle + |010001\rangle + |010010\rangle \\
&\qquad + |010100\rangle + |010101\rangle + |010110\rangle + |011000\rangle + |011001\rangle + |011010\rangle \\
&\qquad + |100000\rangle + |100001\rangle + |100010\rangle + |100100\rangle + |100101\rangle + |100110\rangle \\
&\qquad + |101000\rangle + |101001\rangle + |101010\rangle)
\end{aligned}$$

Let us proceed with the calculations for $|111111\rangle$. We start by applying V_n to $|11\rangle$ (again ignoring

the normalization factor):

$$V_n|11\rangle \propto \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \sqrt{R_n} & \frac{e^{i\frac{\pi}{n}}}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\sqrt{R_n}e^{-i\frac{\pi}{n}} & \frac{e^{-i\frac{\pi}{n}}}{\sqrt{2}} \\ \sqrt{R_n} & 0 & \frac{e^{-i\frac{\pi}{2n}}I_n}{i\sqrt{2}R_{2n}} & -\sqrt{R_n} \\ 0 & \sqrt{1+R_n} & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

$$\propto \frac{1}{\sqrt{2}}e^{i\frac{\pi}{n}}|00\rangle + \frac{1}{\sqrt{2}}e^{-i\frac{\pi}{n}}|01\rangle - \sqrt{R_n}|10\rangle$$

When $n = 3$, the value of R_n is $\frac{1}{2}$. By substituting R_n by $\frac{1}{2}$, and n by 3 the previous state becomes

$$\frac{1}{\sqrt{2}}(e^{i\frac{\pi}{3}}|00\rangle + e^{-i\frac{\pi}{3}}|01\rangle - |10\rangle)$$

Applying V_n to q1, q2, q3, qa1, qa2, and qa3 yields

$$\begin{aligned}
&V_n|11\rangle \otimes V_n|11\rangle \otimes V_n|11\rangle \\
\propto&\frac{1}{\sqrt{2}}(e^{i\frac{\pi}{3}}|00\rangle + e^{-i\frac{\pi}{3}}|01\rangle - |10\rangle) \otimes \frac{1}{\sqrt{2}}(e^{i\frac{\pi}{3}}|00\rangle + e^{-i\frac{\pi}{3}}|01\rangle - |10\rangle) \\
&\otimes \frac{1}{\sqrt{2}}(e^{i\frac{\pi}{3}}|00\rangle + e^{-i\frac{\pi}{3}}|01\rangle - |10\rangle) \\
=&\frac{1}{2\sqrt{2}}(e^{i\frac{\pi}{3}}|00\rangle + e^{-i\frac{\pi}{3}}|01\rangle - |10\rangle) \otimes (e^{i\frac{\pi}{3}}|00\rangle + e^{-i\frac{\pi}{3}}|01\rangle - |10\rangle) \\
&\otimes (e^{i\frac{\pi}{3}}|00\rangle + e^{-i\frac{\pi}{3}}|01\rangle - |10\rangle) \\
=&\frac{1}{2\sqrt{2}}(e^{i\frac{\pi}{3}}e^{i\frac{\pi}{3}}|0000\rangle + e^{i\frac{\pi}{3}}e^{-i\frac{\pi}{3}}|0001\rangle - e^{i\frac{\pi}{3}}|0010\rangle \\
&+ e^{-i\frac{\pi}{3}}e^{i\frac{\pi}{3}}|0100\rangle + e^{-i\frac{\pi}{3}}e^{-i\frac{\pi}{3}}|0101\rangle - e^{-i\frac{\pi}{3}}|0110\rangle \\
&- e^{i\frac{\pi}{3}}|1000\rangle - e^{-i\frac{\pi}{3}}|1001\rangle + |1010\rangle) \otimes (e^{i\frac{\pi}{3}}|00\rangle + e^{-i\frac{\pi}{3}}|01\rangle - |10\rangle) \\
=&\frac{1}{2\sqrt{2}}(e^{i\frac{2\pi}{3}}|0000\rangle + |0001\rangle - e^{i\frac{\pi}{3}}|0010\rangle \\
&+ |0100\rangle + e^{-i\frac{2\pi}{3}}|0101\rangle - e^{-i\frac{\pi}{3}}|0110\rangle \\
&- e^{i\frac{\pi}{3}}|1000\rangle - e^{-i\frac{\pi}{3}}|1001\rangle + |1010\rangle) \otimes (e^{i\frac{\pi}{3}}|00\rangle + e^{-i\frac{\pi}{3}}|01\rangle - |10\rangle) \\
=&\frac{1}{2\sqrt{2}}(e^{i\frac{2\pi}{3}}e^{i\frac{\pi}{3}}|000000\rangle + e^{i\frac{2\pi}{3}}e^{-i\frac{\pi}{3}}|000001\rangle - e^{i\frac{2\pi}{3}}|000010\rangle \\
&+ e^{i\frac{\pi}{3}}|000100\rangle + e^{-i\frac{\pi}{3}}|000101\rangle - |000110\rangle \\
&- e^{i\frac{\pi}{3}}e^{i\frac{\pi}{3}}|001000\rangle - e^{i\frac{\pi}{3}}e^{-i\frac{\pi}{3}}|001001\rangle + e^{i\frac{\pi}{3}}|001010\rangle
\end{aligned}$$

$$
\begin{aligned}
&+ e^{i\frac{\pi}{3}}|010000\rangle + e^{-i\frac{\pi}{3}}|010001\rangle - |010010\rangle \\
&+ e^{-i\frac{2\pi}{3}}e^{i\frac{\pi}{3}}|010100\rangle + e^{-i\frac{2\pi}{3}}e^{-i\frac{\pi}{3}}|010101\rangle - e^{-i\frac{2\pi}{3}}|010110\rangle \\
&- e^{-i\frac{\pi}{3}}e^{i\frac{\pi}{3}}|011000\rangle - e^{-i\frac{\pi}{3}}e^{-i\frac{\pi}{3}}|011001\rangle + e^{-i\frac{\pi}{3}}|011010\rangle \\
&- e^{i\frac{\pi}{3}}e^{i\frac{\pi}{3}}|100000\rangle - e^{i\frac{\pi}{3}}e^{-i\frac{\pi}{3}}|100001\rangle + e^{i\frac{\pi}{3}}|100010\rangle \\
&- e^{-i\frac{\pi}{3}}e^{i\frac{\pi}{3}}|100100\rangle - e^{-i\frac{\pi}{3}}e^{-i\frac{\pi}{3}}|100101\rangle + e^{-i\frac{\pi}{3}}|100110\rangle \\
&+ e^{i\frac{\pi}{3}}|101000\rangle + e^{-i\frac{\pi}{3}}|101001\rangle - |101010\rangle) \\
=&\frac{1}{2\sqrt{2}}(-|000000\rangle + e^{i\frac{\pi}{3}}|000001\rangle - e^{i\frac{2\pi}{3}}|000010\rangle \\
&+ e^{i\frac{\pi}{3}}|000100\rangle + e^{-i\frac{\pi}{3}}|000101\rangle - |000110\rangle \\
&- e^{i\frac{2\pi}{3}}|001000\rangle - |001001\rangle + e^{i\frac{\pi}{3}}|001010\rangle \\
&+ e^{i\frac{\pi}{3}}|010000\rangle + e^{-i\frac{\pi}{3}}|010001\rangle - |010010\rangle \\
&+ e^{-i\frac{\pi}{3}}|010100\rangle - |010101\rangle - e^{-i\frac{2\pi}{3}}|010110\rangle \\
&- |011000\rangle - e^{-i\frac{2\pi}{3}}|011001\rangle + e^{-i\frac{\pi}{3}}|011010\rangle \\
&- e^{i\frac{2\pi}{3}}|100000\rangle - |100001\rangle + e^{i\frac{\pi}{3}}|100010\rangle \\
&- |100100\rangle - e^{-i\frac{2\pi}{3}}|100101\rangle + e^{-i\frac{\pi}{3}}|100110\rangle \\
&+ e^{i\frac{\pi}{3}}|101000\rangle + e^{-i\frac{\pi}{3}}|101001\rangle - |101010\rangle)
\end{aligned}
$$

At this point, the calculations for $|000000\rangle$ and $|111111\rangle$ are done. We then examine only the states where q1, q2, and q3 are all $|0\rangle$ or all $|1\rangle$, as those correspond to cases where symmetry should break. To do so, we construct Table 8, following the format of [86, Table 3], to track the amplitudes and verify where the expected cancellation fails.

How should Table 8 be read? From left to right, we start with the state $|000000\rangle$. Its amplitude in the calculations from $|000000\rangle$ is $\frac{1}{2\sqrt{2}}$, while in those from $|111111\rangle$ it is $-\frac{1}{2\sqrt{2}}$. Their sum is therefore 0. Proceeding similarly for the remaining entries, we find that 6 out of 9 terms that should cancel do not. Hence, there is a non-zero probability that all philosophers belong to the 0-group. These calculations are consistent with the behavior observed in practice.

	$\|000000\rangle$	$\|111111\rangle$	$+$
$\|000000\rangle$	$\frac{1}{2\sqrt{2}}$	$-\frac{1}{2\sqrt{2}}$	0
$\|000001\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{i\frac{\pi}{3}})$
$\|000100\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{i\frac{\pi}{3}})$
$\|000101\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{-i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{-i\frac{\pi}{3}})$
$\|010000\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{i\frac{\pi}{3}})$
$\|010001\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{-i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{-i\frac{\pi}{3}})$
$\|010100\rangle$	$\frac{1}{2\sqrt{2}}$	$\frac{1}{2\sqrt{2}}e^{-i\frac{\pi}{3}}$	$\frac{1}{2\sqrt{2}}(1+e^{-i\frac{\pi}{3}})$
$\|010101\rangle$	$\frac{1}{2\sqrt{2}}$	$-\frac{1}{2\sqrt{2}}$	0
$\|101010\rangle$	$\frac{1}{2\sqrt{2}}$	$-\frac{1}{2\sqrt{2}}$	0

Table 8: Amplitudes of the states that should cancel out