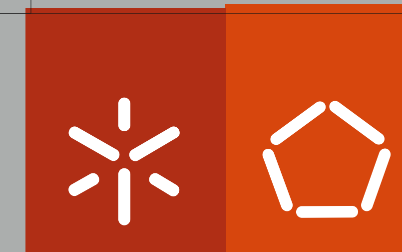
Cláudio Filipe Belo da Silva Lourenço **Single-assignment Program Verification**

UMinho | 2018



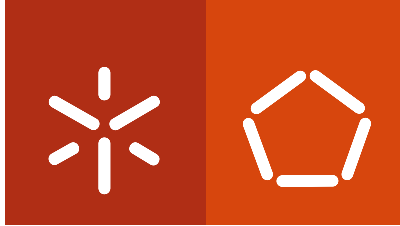
Universidade do Minho
Escola de Engenharia

Cláudio Filipe Belo da Silva Lourenço

Single-assignment Program Verification

Programa de Doutoramento em Informática (MAP-i)
das Universidades do Minho, de Aveiro e do Porto

julho de 2018



Universidade do Minho
Escola de Engenharia

Cláudio Filipe Belo da Silva Lourenço

Single-assignment Program Verification

Programa de Doutoramento em Informática (MAP-i)
das Universidades do Minho, de Aveiro e do Porto

Trabalho realizado sob a orientação do
Professor Doutor Jorge Sousa Pinto

julho de 2018

DECLARAÇÃO

Nome: Cláudio Filipe Belo da Silva Lourenço

Endereço electrónico: belolourenco@gmail.com

Número do Bilhete de Identidade: 13352548

Título da tese: Single-assignment Program Verification

Orientador: Professor Doutor Jorge Sousa Pinto

Ano de conclusão: 2018

Designação do Doutoramento: Programa Doutoral em Informática MAP-i

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE

Universidade do Minho, 13/07/2018

Assinatura: ____

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

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

University of Minho, 13/07/2018

Full name: Cláudio Filipe Belo da Silva Lourenço

Signature: Cláudio Belo Lourenço

Acknowledgments

Several people made this journey possible. Foremost I would like to thank my supervisor Jorge for his guidance, generosity, and friendship throughout these years. His enthusiasm, wise ideas and recommendations were greatly helpful. Jorge has not only guided me through my PhD studies, but has also showed me how to balance work with personal life and gave me greatest tips about music, movies, and TV series. I am also very grateful to Maria João. Her real, true and generous co-supervision was a supplement of energy across this challenging process. Especially I want to thank her for her tremendous commitment to providing insights on the proof of both SA translations – her meticulous methods and detailed reasoning were crucial for a successful outcome.

I thank Shin Nakajima for having accepted me as an intern researcher at National Institute of Informatics, and for having shared with me his vast knowledge about formal methods. To all members of the Why3 team my greatest gratitude for having me and allowing me to be part of their team during the seven months I spent with them. In particular I would like to mention Andrei, who spent several hours guiding me through the Why3 internals.

To my colleagues and friends of *laboratório(s)*, I am grateful for their friendship, for many funny moments, and for the chill out time eating *francesinha*, drinking beer, and playing board games. Many thanks to the senior members of *departamento de informática* for having helped me during this journey. A special thanks to José Bernardo for having guided me in my teaching duties.

To all my friends who accompanied me since the beginning of this journey and to those I met on the way, my deepest thanks. They helped me to relax from the hard work, and were there when I needed them – in all the good times and in the less good ones.

Above all, a huge thanks to my family for having been, evermore, my *porto seguro*. Especially to my mother and my father – to whom I owe my life and education, and all important things in life that I cannot put into words. To my sister, for all the healthy fights. To all my cousins for still being in touch after all the years bullying them. To my godfather for teaching me how to fight hard for things I want in life. To my uncles and aunts for their care and love. To my paternal grandmother for her love through silence. To my maternal grandparents and paternal grandfather who bid farewell to life, for the good memories of all love and inspiration. To all of you: a deep, long, high and wide thank you.

FCT Fundação para a Ciência e a Tecnologia

This work was funded by FCT – Foundation for Science and Technology, the Portuguese Ministry of Science, Technology and Higher Education, through the Operacional Programme for Human Capital (POCH). Grant reference: PD/BD/52236/2013.

Abstract

Many *program verification* tools rely on the translation of code annotated with properties into an intermediate *single-assignment* form (in a more or less explicit way), and then on an algorithm that generates *verification conditions* from it. In this thesis, we revisit two major methods that are widely used to produce verification conditions for single-assignment programs: *predicate transformers* (used mostly by *deductive verification* tools) and the *conditional normal form transformation* (used in *bounded model checking of software*). We identify different aspects in which the methods differ and show that they can be combined to produce new hybrid verification condition generators; together with the initial algorithms they form what we call the *VCGen cube*, which we propose as a framework for synthesizing and comparing verification condition generators. Optimizations implemented by verification tools are then integrated into the cube.

At the theoretical level we propose two fully proved verification frameworks based on the translation into single-assignment and subsequent generation of verification conditions. On one hand we formalize program verification based on the translation of *While* programs annotated with *loop invariants* into an *iterating single-assignment* language with a dedicated iterating construct. *Soundness* and *completeness* proofs are given for the entire workflow, including the translation of annotated programs into iterating single-assignment form. The formalization is based on a *program logic* that we show to be *adaptation-complete*.

On the other hand we formally define an iteration-free single-assignment language with *assume*, *assert*, and *exceptions*, and introduce a program logic for this language which allows us to prove the soundness and completeness of the VCGen cube. A verification framework based on the translation of programs into (iteration-free) single-assignment form is then proposed, and the entire workflow is proved to be sound and complete. We also suggest a concrete single-assignment translation that transforms annotated loops into assumes and asserts to check that the annotated invariants are valid and preserved during the iterations.

Finally, we compare the verification condition generators empirically, both for programs of the LLVM intermediate representation (a concrete popular intermediate language that is based on SA form), and in the context of the Why3 deductive verification tool. Although the results do not indicate absolute superiority of any given method, they do allow us to identify interesting trends.

Resumo

No contexto da verificação de programas, são várias as ferramentas que assentam na tradução de programas numa forma *single-assignment* (umas implicitamente e outras explicitamente), e depois num algoritmo que gera condições de verificação. Nesta tese, revisitamos dois grandes métodos tipicamente utilizados para a geração de condições de verificação a partir de programas *single-assignment*: *predicate transformers* (utilizados maioritariamente por ferramentas dedutivas) e a transformação baseada em *conditional normal form* (utilizados em *bounded model checking* de software). Com isto, identificamos vários aspetos nos quais estes métodos diferem e mostramos como estes podem ser combinados para produzir novos geradores de condições de verificação. Os métodos resultantes, juntamente com os iniciais, formam aquilo que designamos por *VCGen cube* e que propomos como uma *framework* para sintetizar e comparar a geração de condições de verificação.

A nível teórico, propomos duas abordagens para a verificação baseada na tradução de programas em *single-assignment* e subsequente geração de condições de verificação. Por um lado, formalizamos uma técnica assente na tradução de programas *While*, anotados com invariantes de ciclo, em programas *single-assignment* contendo um comando especial para captar os respetivos ciclos e invariantes. Provas sobre a *soundness* e *completeness* do fluxo completo da técnica são apresentadas. A formalização da técnica é baseada numa lógica de programas que é *adaptation-complete*.

Numa segunda abordagem, definimos formalmente a noção de programa *single-assignment* (sem ciclos) contendo os comandos *assume* e *assert*, e também exceções, e introduzimos uma lógica para estes programas que nos permite mostrar que o *VCGen cube* é *sound* e *complete*. Uma técnica de verificação que assenta na tradução de programas em *single-assignment* é então proposta e mais uma vez, o fluxo completo da *framework* é provado como sendo *sound* e *complete*. Em particular, a tradução concreta (esta também provada correta) traduz programas contendo ciclos em programas sem ciclos.

Para a avaliação empírica dos geradores de condições de verificação, recorremos a duas abordagens, uma no contexto da linguagem intermédia *LLVM*, e outra no contexto da ferramenta de verificação dedutiva *Why3*. Embora os resultados não indiquem superioridade absoluta de nenhum método, estes revelam tendências interessantes.

Contents

1 Introduction **1**

2 Background **7**

2.1 While Language . . . 8
2.2 Hoare Logic . . . 14
2.3 An Inference System for Annotated Programs . . . 20
2.4 Predicate Transformers . . . 23
2.5 Verification Condition Generator . . . 31
2.6 Bounded Model Checking of Software . . . 33
2.7 More Preliminary Notation and Results . . . 39

3 Iterating SA Programs as a Basis for Program Verification **43**

3.1 Iterating Single-assignment Language . . . 44
3.2 Hoare Logic and Verification Conditions for Iterating SA Programs . . . 47
3.3 Program Verification Using Iterating SA Form . . . 51
3.4 Adaptation Completeness of Iterating SA Programs . . . 52
3.5 Iterating SA Translation . . . 58
3.6 Related Work . . . 70

4 Verification Conditions for SA Programs with Assume, Assert, and Exceptions **73**

4.1 Verification Conditions for SA Programs with Asserts . . . 74
4.2 A Cube of VCGens . . . 79
4.3 VCGens Optimizations . . . 88
4.4 Generic VCGens for Programs with Exceptions . . . 94
4.5 Unifying the Generation of Verification Conditions . . . 104
4.6 Related Work . . . 105

5 A Verification Workflow Based on SA Programs **107**

5.1 Semantics for Programs with Assume, Assert, and Exceptions . . . 108
5.2 SA Program with Assume, Assert, and Exceptions . . . 114
5.3 Correctness of the Cube of VCGens . . . 120

5.4 A Proved Verification Framework 130
5.5 Related Work 132

6 A Translation of Iterating Programs into SA Form 133
6.1 A Small-step Semantics 134
6.2 SA Translation 139
6.3 Soundness of SA Translation 147
6.4 Completeness of SA Translation 155
6.5 Related Work 156

7 Evaluation and Experiments 159
7.1 Experimental Evaluation with SNIPER-VCGen 160
7.2 Experiments with Why3 165
7.3 Related Work 174

8 Conclusion 177

A Derivations 181

B Why3 Experimental Programs 185

Chapter 1

Introduction

The need for methods to reason about algorithms was identified in the late forties [57, 101], but it was during the late sixties that Floyd and Hoare proposed a systematic approach for reasoning about programs [53, 64]. The initial idea of Floyd was to reason about flowcharts, while Hoare proposed to reason about text representing programming constructs. Since then, program verification has gained an important role in the programming languages community, and several progresses have been made. Tools can nowadays be effectively used in daily tasks when producing and reasoning about programs.

In the last years *deductive program verification* has reached a stage of a certain maturity, to the point that a number of tools are now available allowing users to prove properties of programs written in real-world languages like C, C#, Java, or SPARK [74, 12, 49, 89]. Deductive techniques attempt to establish the correctness of a program implementation with respect to a specification, usually given as a set of *contracts* (e.g. pre and postcondition) expressed in first-order logic. The success of the verification procedure depends on information provided by the user in the form of *annotations*, in particular *loop invariants*.

Bounded model checking of software is an alternative approach to program verification that has been in use for more than a decade now, and solutions have been proposed for multiple languages [31, 91, 86, 93]. Properties are normally inserted in the code through *assume* and *assert* clauses to be checked. The goal of bounded model checkers is to be as automatic as possible, and for this reason the precision of the verification technique may be compromised. For this reason they are sometimes described as *bug finders* rather than program verification tools, but in our view the two families of tools, bounded model checking and deductive verification tools, share many principles and complement each other.

To start with, at the language level, the programs to be verified typically contain pre and postconditions, as well as a generic mechanism that allows for properties to be annotated at any point in the program. *Assume* and *assert* statements are part of the core specification language of most bounded model checkers of software, and more generally of *software model checkers* [71]. The assert command is also included in the behavior specification languages employed by deductive verification platforms such as Why3 [50], Dafny [83], OpenJML [35], SPARK [89], or Frama-C [74]. Even though such tools do not usually allow for the use of

the assume command at the programmer's level, they often resort to it at some point in the intermediate form.

A second shared principle is that modern tools employ internally a *verification conditions generator* (VCGen), a component that takes as input a program together with a specification, and outputs a number of proof obligations known as *verification conditions* (VCs), that are sent to a backend proof tool for validity checking.

Finally, tools based on the two approaches resort, in a more or less explicit way, to an intermediate form in which programs do not contain iterating constructs (loops) or subprogram calls, and are in what is known as *single-assignment* (SA) form [40, 102], which means that variables may not be assigned more than once. In BoogiePL [10], a language tailored for deductive verification, assume and assert statements are used to encode loops annotated with invariants as non-iterating commands, and the resulting program is transformed into SA. In bounded model checking of software, on the other hand, loops are expanded a fixed number of times (property violations may thus be found only up to a bound) and the resulting program is then translated into SA form. Procedure and function calls are similarly eliminated, either by inserting assume and assert statements encoding their contracts, or by inlining their code.

Typically the intermediate SA form will still contain some constructs allowing for rich control-flow, such as a *goto* command or a *try-catch* exception mechanism, since this is usually allowed at source level [52, 11, 50]. There are multiple reasons for the use of a single-assignment intermediate form: (i) programs are easy to encode in this form, since no variable substitutions have to be applied in order to capture assignments to variables; (ii) it allows for the generation of compact VCs in a sense that was first identified by Flanagan and Saxe [52]; (iii) and it is easy to capture aspects of specification languages such as the ability to refer to the value of a variable at a given program point.

This thesis is born from the observation that the major approaches to the generation of verification conditions based on the translation of programs into SA have not yet been sufficiently investigated, nor compared, either theoretically or practically. In particular, as far as we know, no SA translation has ever been shown to be sound or complete with respect to the programming languages semantics, or to preserve the validity of the loop invariants annotated in the code. At the VCGen level, there is a clear theoretical gap between the VCGens implemented by these tools and the correctness of the verification method as a whole. A key point is that the VCGens used by different tools may differ substantially in the format of the VCs they generate and optimizations they employ. The VCGen and the optimization algorithms are usually coupled tightly, and both are hidden in the internals of tools. Identifying the baseline features that influence efficacy and efficiency of the verification techniques is far from trivial, and a thorough study of these aspects is thus desirable to understand the intricacies of each technique, contributing towards improving current verification tools, and allowing for new techniques to be investigated systematically.

Contributions. The contributions of this thesis can be divided in two parts. In the first part, we formalize a verification technique for *While* programs annotated with invariants, based on their conversion to an intermediate *iterating single-assignment* (ISA) form. The technique relies on (i) a novel notion of single-assignment program that supports loops annotated with invariants; (ii) a notion of translation of *While* programs annotated with loop invariants (resp. Hoare triples containing such programs) into the ISA programs (resp. Hoare triples containing ISA programs); (iii) a Hoare-style logic for these programs; and (iv) a VCGen generating compact verification conditions for Hoare triples containing ISA programs. The entire workflow is proved to be sound and complete – in particular, we show how invariants annotated in the initial *While* program are translated into the intermediate ISA form in a way that guarantees the completeness of the approach. This means that if the invariants annotated in the original program are appropriate for showing its correctness, then the verification of the translated ISA program will be successful.

Adaptation completeness is an important property that establishes that a triple can be derived from another whenever the validity of the first implies the validity of the second. An adaptation-complete variant of the logic for ISA is also proposed, by adding to the inference system a dedicated consequence rule with a simple side condition, that provides the highest degree of adaptation, without the need to check any additional complicated conditions or rules, as used to be the case in existing adaptation-complete presentations of Hoare logic [5, 4, 75].

The second part aims to formally capture the typical workflow implemented by state of the art verification tools that rely on the translation of programs into SA. We start by presenting a systematic study and categorization of the VCGens commonly used in deductive verification and bounded model checking of software, and compare the verification conditions they generate. By taking as point of departure the logical encoding of SA programs containing assumes and asserts, (i) we identify three dimensions in which they differ, and by combining these dimensions (ii) we introduce 6 new hybrid VCGens and (iii) organize them graphically in what we call the *VCGen cube*; (iv) we propose a set of *optimized* hybrid VCGens, by considering optimizations implemented by two flagship tools; (v) we extend the previous VCGens to handle programs with exceptions; and (vi) aggregate all the VCGens into a single unifying definition.

Aiming at providing a common proved verification framework, based on the translation of programs annotated with loop invariants into SA and the subsequent generation of verification conditions, we propose a set of theoretical tools. More specifically, we (i) study operational and axiomatic semantics of a language with assume, assert, and exceptions; (ii) introduce a sound and complete logic for (non-iterating) SA programs with assume, assert, and exceptions; (iii) prove the soundness and completeness of the VCGens in the cube; (iv) introduce an appropriate notion of SA translation; and (v) prove that the resulting verification framework is sound and complete.

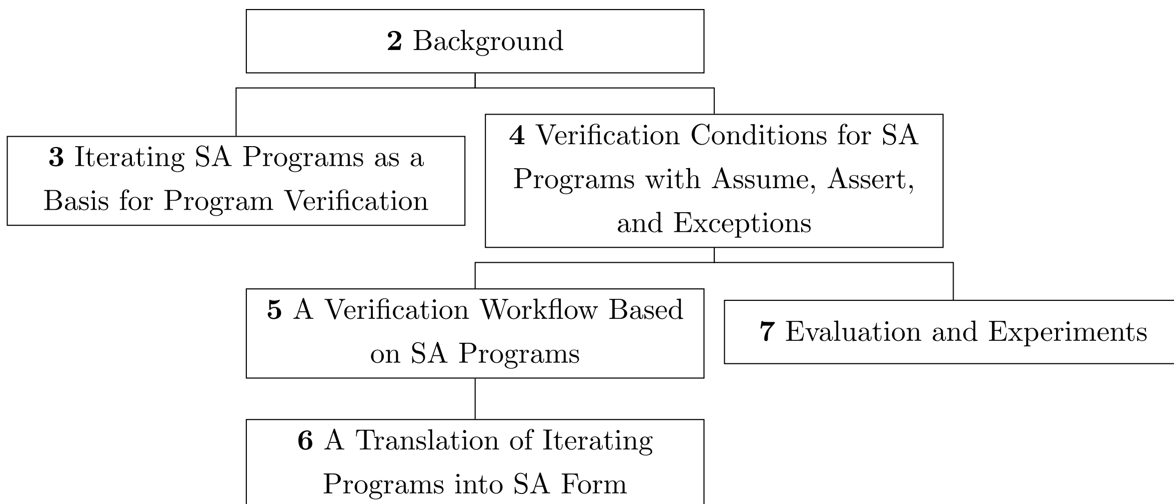
A concrete SA translation is also provided, together with the proof that it complies with the notion of SA translation. Note that this is not immediate since the translation must convert programs with loops into iteration-free programs. Even though this is a translation that is applied by several verification tools, as far as we know, it is the first time that such a translation

is proved to be sound and complete.

Besides comparing the VCGens asymptotically, and demonstrating that all the VCGens generate VCs that are equivalent in terms of validity, we also present some empirical results. In a first approach we show the result of comparing them empirically with a large set of benchmark programs in the context of LLVM intermediate representation [80], and in a second approach we show the result of different experiments using the Why3 deductive verification tool.

The work regarding the verification of programs based on the translation into iterating single-assignment form was published in the Proceedings of the *25th European Symposium on Programming* [87]. An initial approach at comparing the generation of verification conditions was published in the Proceedings of the *15th International Workshop on Automated Verification of Critical Systems* [88], and the VCGen cube together with the empirical evaluation in the context of LLVM intermediate representation is to appear in the *Proceedings of the 37th Annual IEEE Computer Software and Applications Conference* [85].

Organization of the Document. The dependencies between the chapters are depicted in the diagram below:



Chapter 2 provides an overview over the area of program verification and presents a set of frameworks to reason about programs. The chapter may also be seen as an introduction to program verification. In particular it presents a commonly used While language together with the operational and axiomatic semantics, and Dijkstra's guarded commands together with their predicate transformers. The notion of verification condition generation and the basis of bounded model checking of software are also presented. The chapter ends by introducing some notation that will be used along the thesis.

Chapter 3 proposes a first approach to formalize program verification based on the use of a single-assignment intermediate language. We propose a workflow for program verification based on a novel iterating SA programming language and a program logic for these programs.

Chapter 4 is somewhat less formal, in the sense that its contents are justified by resorting to what tools based on the use of single-assignment form typically do, instead of starting from

the theoretical foundations of program verification. We propose the VCGen cube as a device to organize and unify the presentation of the different methods used in the generation of VCs for SA programs containing assume, assert, and try-catch exceptions.

Chapter 5 aims at bridging the gap between the previous chapter, that discusses different methods for generating VCs, and the theoretical foundations of program verification, in particular Hoare logic and program semantics. It formalizes the notion of single-assignment program, provides an inference system to reason about these programs, and shows that the VCGens of Chapter 4 are all equivalent, sound, and complete with respect to the inference system. Contrarily to the approach from Chapter 3, the single-assignment language contains assume and assert commands but does not contain any iterating construct, which implies that iteration has to be captured resorting to the former constructs.

Chapter 6 proposes a single-assignment translation that is shown to be indeed an SA translation. More specifically, the translation is shown to be sound (if a translated triple is valid then so is the original) and complete (if the original triple is valid, then so is the translated one). This allows us to demonstrate that the workflow based on the translation of programs into single-assignment is not vacuous, and there is at least one translation that preserves the restrictions from the previous chapter.

Chapter 7 presents some experiments. The first part of the chapter provides a comparative evaluation of the considered VCGens in a tool that was developed on top of the SNIPER fault localization tool [79], and the second part carries out a comparative analysis on some experiments that were made with the Why3 deductive verification tool [50].

Chapter 8 discusses the main conclusions of our work and presents suggestions for future studies.

Chapter 2

Background

This chapter introduces the necessary background for this thesis and also some preliminary notation and results. The first part of the chapter is written in the form of a tutorial and can also be seen as an introduction to program verification, both from the *deductive* and from the *bounded model checking* point of view.

The typical *While language* that will serve as a basis for the whole thesis is presented and immediately used to explore deductive verification of programs. We study the notion of *Hoare triple* and a logic formalism to reason about those triples known as *Hoare Logic*. Since the traditional Hoare logic requires user intervention, the concept of annotated program is introduced and a new inference logic system to reason about annotated programs is presented.

With the intention of abstracting from all the intricacies of the theoretical foundations of programming languages, Dijkstra proposed *guarded commands* and a method known as *predicate transformers* to reason about them. These concepts have had a great impact on program verification, in particular on the topic of *(compact) verification conditions* generation, which we review here.

A light-weight introduction to *bounded model checking of software* is then presented. The main concepts of the technique are exposed and explored, and a concrete example is given of how to transform an iterating-program into a logical formula.

This chapter is organized in the following way. The next section introduces the basic principles about the semantics of programming languages: it introduces the notion of program expression, a concrete syntax for a While language, and also a small and big-step semantics. Section 2.2 shows the basic principles of program verification, in particular the notion of Hoare triple and Hoare logic. The previous principles are then extended to annotated programs with invariants in Section 2.3. We then focus on the generation of verification conditions, by first presenting some work in the context of Dijkstra's guarded commands (Section 2.4) and then in the context of our While language (Section 2.5). Section 2.6 is dedicated to bounded model checking of software and Section 2.7 presents some additional preliminary notation and results.

2.1 While Language

This section presents the While programming language that will be used as a basis for the work presented in this thesis. Even though the language will be extended during the thesis to incorporate other constructs, here we present it in its simplest form as normally found in the literature [103, 96, 95]. There are two aspects that are particularly relevant when formally defining a programming language: the *programming language syntax* and the *programming language semantics*. While the former is concerned with the grammar structure of the language, the latter is concerned with the meaning of the constructs in that structure.

Let us start by the syntax. For our purposes only the *abstract syntax* tree of the language is relevant: the *concrete syntax*, which captures how the structure is represented and handled, can be left undefined.

The language for program expressions **Exp** and for Boolean expressions $\mathbf{Exp^{bool}}$, both constructed over variables from a set **Var**, will not be fixed. This will make our results more general in the sense that they are independent from the expression language. A standard instantiation is for **Exp** to be a language of integer expressions and $\mathbf{Exp^{bool}}$ constructed from comparison operators over **Exp** and Boolean operators: such a language is defined in Example 2.1. We will use the following meta-variables: e, e_1, e_2 to range over **Exp**, b, b_1, b_2 to range over $\mathbf{Exp^{bool}}$, and x to range over **Var**.

Example 2.1. *Let* $e \in \mathbf{Exp}$, $b \in \mathbf{Exp^{bool}}$, $x \in \mathbf{Var}$ *and* $n \in \mathbb{Z}$. *An instantiation for* **Exp** *and* $\mathbf{Exp^{bool}}$ *is given by the abstract syntax tree below:*

$$\begin{array}{rcl} e & ::= & n \mid x \mid -e \mid e + e \mid e - e \mid e * e \mid e/e \\ b & ::= & \top \mid \bot \mid e = e \mid e < e \mid e \leq e \mid b \wedge b \mid b \vee b \mid \neg b \end{array}$$

The syntax notation used in the previous example, as well as in the forthcoming syntactic notions are based on BNF (Backus-Naur form) [76]. The syntax of the While language, $C \in$ **Comm** is given by the following *abstract syntax*:

$$C \ ::= \ \mathbf{skip} \ \mid \ x := e \ \mid \ C\,;C \mid \ \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C\ \mathbf{fi} \ \mid \ \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}$$

Along the thesis we will use the word *statement* and *command* indiscriminately to refer to the constructs of the respective programming language. The following is taken from Dijkstra [44]:

> As everybody does, I shall use the term 'statement' because it has found its firm place in jargon; when people suggested that 'command' was perhaps a more appropriate term, it was already too late!

The informal meaning of each command C is the expected: **skip** does not produces any effect on the state when it is executed; $x := e$ assigns to x the value obtained by evaluating e (we assume that the evaluation of e has no side-effects); the sequence command, $C_1\,; C_2$, executes

C_1 and then C_2; the conditional command **if** b **then** C_1 **else** C_2 **fi** executes C_1 if the result of evaluating b is $\top$ or C_2 otherwise; finally **while** b **do** C **od** executes C while the condition b evaluates to $\top$. When adequate, we will write **if** b **then** C **fi** instead of **if** b **then** C **else skip fi** for simplification purposes.

Example 2.2. *The sequence* $x := y\,;\, y := z\,;\, z := x$ *is an example of a command in* **Comm**.

Example 2.3. *Assuming the instantiation of* **Exp** *and* $\mathbf{Exp^{bool}}$ *as in Example 2.1, the following is an example of a program from* **Comm**.

$$\begin{array}{l} f := 1\,;\, i := 1\,; \\ \textbf{while}\ i \leq n\ \textbf{do} \\ \quad f := f * i\,; \\ \quad i := i + 1\ \textbf{od} \end{array}$$

Intuitively one can observe that the program in Example 2.2 swaps the value of the variables y and z and the program in Example 2.3 is an iterative implementation of a factorial program: the counter i ranges from 1 to n and the accumulator f contains at each step the factorial of $i - 1$. Nonetheless, in a formal setting, intuition is not enough to describe precisely what constructs over an abstract syntax tree are intended to mean. Some formal framework is required for this and as such, we will use *programming language semantics* [62, 103, 95] to describe the meaning of the programs and the expressions in them. Three different styles of programming semantics are normally considered when reasoning about programming languages: *operational semantics*, *denotational semantics*, and *axiomatic semantics*. These styles should not be seem as being competing between each other, instead they simply target different purposes: operational semantics should be used to reason about the execution of a program: basically, given a command and the state from where the command is to be executed, it returns the resulting state; denotational semantics describes the results of the computation as mathematical objects, and therefore, the focus is on the result that is obtained and not on how it is obtained; finally, axiomatic semantics is intended to reason about the properties that can be observed when executing program statements. In this thesis we will be using operational semantics to give meaning to programs and expressions, and then we will reason about them using axiomatic semantics.

Since we are not fixing the language of expressions, we will consider an undefined *interpretation structure* $\mathcal{M} = (D, I)$ for their evaluation. Such a structure provides an interpretation domain D as well as a concrete interpretation of constants and operators, given by I. We want expressions to contain variables, therefore, their interpretation depends on a *state*, which is a total function that maps each variable into its value in domain D (the function must be total in the sense that it must be defined for every variable). We will write $\Sigma = \mathbf{Var} \to D$ for the

set of states. Note that if variables of different types are to be considered, this approach can be extended to a multi-sorted setting by letting Σ become a *generic function space* [54]. For $s \in \Sigma$, $s[x \mapsto a]$ will denote the state that maps x to a and every other variable y to $s(y)$. The interpretation of $e \in \mathbf{Exp}$ in $\mathcal{M}$ will be given by a function $[\![e]\!]_{\mathcal{M}} : \Sigma \to D$, and the interpretation of $b \in \mathbf{Exp^{bool}}$ will be given by $[\![b]\!]_{\mathcal{M}} : \Sigma \to \{\bot, \top\}$. This reflects our assumption that an expression has a value at every state (evaluation always terminates without error) and that expression evaluation never changes the state (the language is free of *side effects*). We will often omit the $\mathcal{M}$ subscripts for the sake of readability and leave the interpretation structure implicit.

Example 2.4. *Consider the expression language of Example 2.1. We define $[\![e]\!]_{\mathcal{M}}$ and $[\![b]\!]_{\mathcal{M}}$ for $\mathcal{M} = (D, I)$, some $s \in \Sigma$, $\odot \in \{+, -, *, /, =, <, \leq, >, \geq\}$, and $\oslash \in \{\wedge, \vee\}$ as follows:*

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
[\![n]\!](s) & = & I(n) & [\![\top]\!](s) & = & \top \\
[\![x]\!](s) & = & s(x) & [\![\bot]\!](s) & = & \bot \\
[\![-e]\!](s) & = & I(-)([\![e]\!](s)) & [\![\neg b]\!](s) & = & I(\neg)([\![b]\!](s)) \\
[\![e_1 \odot e_2]\!](s) & = & I(\odot)([\![e_1]\!](s), [\![e_2]\!](s)) & [\![b_1 \oslash b_2]\!](s) & = & I(\oslash)([\![b_1]\!](s), [\![b_2]\!](s))
\end{array}
$$

An obvious instantiation for $\mathcal{M}$ is for D to be $\mathbb{Z}$ and I to be the function that transforms each operator from our language into the mathematical operator represented by the same symbol. For instance $[\![e_1 + e_2]\!](s) = [\![e_1]\!](s) + [\![e_2]\!](s)$, and the $+$ symbol on the right hand side is the mathematical plus operation. Note however that, the division symbol cannot be interpreted directly as the mathematical division, since the latter is not defined for the case in which the denominator is zero. A solution is for it to return some default value when the denominator is zero, or simply omit the division from the language.

When it comes to the operational semantics of commands, two different approaches come into hand. *Structural operational semantics*, also known as *small-step semantics* describes how each individual step of the evaluation evolves. On the other hand, *natural operational semantics*, also known as *big-step semantics* or *evaluation semantics*, describes the overall result of evaluating a command. Once again, these two styles of operational semantics should not be seen as competing with each other, but rather as different mechanisms to address different problems. Even though the while language used in this section can be adequately described using either of the two styles, we will present below both small-step and big-step semantics. In fact, it is normally easier and more natural to reason with a big-step semantics, but for certain languages and intermediate properties about the execution of programs from such a language, it becomes crucial to use a small-step semantics.

The big-step semantics is given by a deterministic *evaluation relation* $\leadsto \subseteq \mathbf{Comm} \times \Sigma \times \Sigma$ that depends on an implicit interpretation structure $\mathcal{M}$ for program expressions. A natural semantics describes the final program states that may result from running a program in a given initial state. We write $\langle C, s \rangle \leadsto s'$ to denote the fact that if C is executed in the initial state s, then its execution terminates, and the final state is s'.

Definition 2.1 (Natural semantics)**.** *The evaluation relation for* **Comm** *is defined as the smallest relation* $\rightsquigarrow\ \subseteq\ \mathbf{Comm} \times \Sigma \times \Sigma$ *satisfying the following set of rules:*

1. $\langle \mathbf{skip}, s\rangle \rightsquigarrow s$.
2. $\langle x := e, s\rangle \rightsquigarrow s[x \mapsto [\![e]\!](s)]$.
3. *if* $\langle C_1, s\rangle \rightsquigarrow s'$ *and* $\langle C_2, s'\rangle \rightsquigarrow s''$, *then* $\langle C_1 \,;\, C_2, s\rangle \rightsquigarrow s''$.
4. *if* $[\![b]\!](s) = \top$ *and* $\langle C_1, s\rangle \rightsquigarrow s'$, *then* $\langle \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s\rangle \rightsquigarrow s'$.
5. *if* $[\![b]\!](s) = \bot$ *and* $\langle C_2, s\rangle \rightsquigarrow s'$, *then* $\langle \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s\rangle \rightsquigarrow s'$.
6. *if* $[\![b]\!](s) = \top$, $\langle C, s\rangle \rightsquigarrow s'$ *and* $\langle \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, s'\rangle \rightsquigarrow s''$, *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, s\rangle \rightsquigarrow s''$.
7. *if* $[\![b]\!](s) = \bot$, *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, s\rangle \rightsquigarrow s$.

Example 2.5. *Consider the program from Example 2.2 and let* $s \in \Sigma$ *be some arbitrary state such that* $s(y) = 3$ *and* $s(z) = 5$. *Then, according to Definition 2.1 we have* $\langle x := y, s\rangle \rightsquigarrow s[x \mapsto 3]$ *and* $\langle y := z, s[x \mapsto 3]\rangle \rightsquigarrow s[x \mapsto 3, y \mapsto 5]$. *Therefore* $\langle x := y \,;\, y := z, s\rangle \rightsquigarrow s[x \mapsto 3, y \mapsto 5]$. *Finally,* $\langle z := x, s[x \mapsto 3, y \mapsto 5]\rangle \rightsquigarrow s[x \mapsto 3, y \mapsto 5, z \mapsto 3]$, *and thus* $\langle x := y \,;\, y := z \,;\, x := 3, s\rangle \rightsquigarrow s[x \mapsto 3, y \mapsto 5, z \mapsto 3]$.

In the previous example we considered an arbitrary s because states must be defined for the complete set of variables. Therefore it would not be enough to depart from a state described by the set of pairs $\{(y, 3), (z, 5)\}$.

The small-step semantics is given by a deterministic transition relation $\Rightarrow\subseteq \mathbf{Comm} \times \Sigma \times (\Sigma + \mathbf{Comm} \times \Sigma)$ that depends on an implicit interpretation for program expressions. The emphasis is now on each individual step of the computation. A configuration can evolve into a final state, progress into an intermediate configuration leaving part of the program to be evaluated, or simply get stuck. For a program $C \in \mathbf{Comm}$ and a state $s \in \Sigma$, the transition relation can be represented by one of the following:

- $\langle C, s\rangle \Rightarrow s_f$, to denotes that the configuration $\langle C, s\rangle$ evolves into a final state $s_f \in \Sigma$ in one step.
- $\langle C, s\rangle \Rightarrow \langle C', s'\rangle$, to denote that the execution of C from s has not yet terminated, and it remains to compute $C' \in \mathbf{Comm}$ from the state $s' \in \Sigma$.
- $\langle C, s\rangle \not\Rightarrow$, to denote that the program C cannot evolve from state s.

We define below a small-step semantics for programs defined over **Comm**. We note however, that this definition does not allow for configurations to be stuck because it is defined for every command possible combination of **Comm** and Σ.

Definition 2.2 (Structural operational semantics)**.** *The evaluation relation for* **Comm** *is defined as the smallest relation* $\Rightarrow\subseteq \mathbf{Comm} \times \Sigma \times (\Sigma + \mathbf{Comm} \times \Sigma)$ *satisfying the following set of rules:*

1. $\langle \mathbf{skip}, s\rangle \Rightarrow s$.

2. $\langle x := e, s\rangle \Rightarrow s[x \mapsto [\![e]\!](s)]$.

3. *if* $\langle C_1, s\rangle \Rightarrow s'$ *then* $\langle C_1\,;\, C_2, s\rangle \Rightarrow \langle C_2, s'\rangle$.

4. *if* $\langle C_1, s\rangle \Rightarrow \langle C_1', s'\rangle$, *then* $\langle C_1\,;\, C_2, s\rangle \Rightarrow \langle C_1'\,;\, C_2, s'\rangle$.

5. *if* $[\![b]\!] = \top$, *then* $\langle \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s\rangle \Rightarrow \langle C_1, s\rangle$.

6. *if* $[\![b]\!] = \bot$, *then* $\langle \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s\rangle \Rightarrow \langle C_2, s\rangle$.

7. $\langle \mathbf{while}\ b\ \mathbf{do}\ C_1\ \mathbf{od}, s\rangle \Rightarrow \langle \mathbf{if}\ b\ \mathbf{then}\ \{C_1\,;\, \mathbf{while}\ b\ \mathbf{do}\ C_1\ \mathbf{od}\}\ \mathbf{else\ skip\ fi}, s\rangle$.

A *derivation sequence* describes how the execution of a program evolves using a small-step semantics. It can be *finite*, if the configuration evolves into a final state or into a stuck configuration, or *infinite* if it diverges. For a finite derivation we write $\delta_0 \Rightarrow \ldots \Rightarrow \delta_n$ to express that the configuration δ_0 evolves into δ_n in n steps, and δ_n is a final state, or a stuck configuration (again, in our current setting only the former can happen). The notation $\delta_0 \Rightarrow^n \delta_n$ expresses that δ_n is reached after n steps when departing from δ_0. If there exists a finite number of steps n such that $\delta_0 \Rightarrow^n \delta_n$, we can also write $\delta_0 \Rightarrow^* \delta_n$ (or $\delta_0 \Rightarrow^+ \delta_n$ when $n > 0$). Also, if $\delta_0 \Rightarrow^n \langle C', s'\rangle$ and $\langle C', s'\rangle \not\Rightarrow$ then we write $\delta_0 \not\Rightarrow^n$ (in this case we can also write $\delta_0 \not\Rightarrow^*$, or $\delta_0 \not\Rightarrow^+$ if $n > 0$).

If the derivation sequence is infinite (e.g. in the presence of an infinite loop), we write $\delta_0 \Rightarrow \ldots$ to express the fact that from the configuration δ_0 it is not possible to reach a final state, neither a stuck configuration.

Example 2.6. *Consider again the program from Example 2.2 and let $s \in \Sigma$ be some arbitrary state such that $s(y) = 3$ and $s(z) = 5$. Then, using Definition 2.2 it is possible to obtain the following derivation sequence $\langle x := y\,;\, y := z\,;\, z := x, s\rangle \Rightarrow \langle y := z\,;\, z := x, s[x \mapsto 3]\rangle \Rightarrow \langle z := x, s[x \mapsto 3, y \mapsto 5]\rangle \Rightarrow s[x \mapsto 3, y \mapsto 5, z \mapsto 3]$.*

Example 2.5 and Example 2.6 highlight the difference between both semantics approaches. In the first example the emphasis, while using a big-step semantics, is on the final result; in the second example the emphasis, while using a small-step semantics, is on the evolution of the computation. It is also clear that the same final state is reached: $s[y \mapsto 5, z \mapsto 3, x \mapsto 3]$. This is actually a general result stated in the next proposition: either a final state is not reached in neither of the semantics or if it is in one, then so it is in the other and both styles will agree on the same final state.

Proposition 2.1. *Let $C \in$ **Comm** and $s, s' \in \Sigma$. Then:*

1. *If $\langle C, s\rangle \rightsquigarrow s'$, then $\langle C, s\rangle \Rightarrow^* s'$.*

2. *If $\langle C, s\rangle \Rightarrow^k s'$, for some natural k, then $\langle C, s\rangle \rightsquigarrow s'$.*

Proof. 1. follows by induction on the structure of $\langle C, s\rangle \rightsquigarrow s'$; 2. follows by induction on the length of the derivation for $\langle C, s\rangle \Rightarrow^k s'$. See Nielson et al. [95] for a complete proof. □

Let us now introduce some auxiliary functions that will be useful all along this thesis: we will be referring to the variables occurring and assigned in a program through the respective functions Vars and Asgn defined below. The sets $\mathsf{Vars}(e) \subseteq \mathbf{Var}$ and $\mathsf{Vars}(b) \subseteq \mathbf{Var}$ denote respectively the free variables of $e \in \mathbf{Exp}$ and $b \in \mathbf{Exp^{bool}}$.

Definition 2.3. *Let $C \in \mathbf{Comm}$. The sets $\mathsf{Vars}(C)$ and $\mathsf{Asgn}(C)$ of variables occurring and assigned in C are defined below.*

$$\begin{aligned}
\mathsf{Vars}(\mathbf{skip}) &= \emptyset \\
\mathsf{Vars}(x := e) &= \{x\} \cup \mathsf{Vars}(e) \\
\mathsf{Vars}(C_1 \,;\, C_2) &= \mathsf{Vars}(C_1) \cup \mathsf{Vars}(C_2) \\
\mathsf{Vars}(\mathbf{if}\, b\, \mathbf{then}\, C_1\, \mathbf{else}\, C_2\, \mathbf{fi}) &= \mathsf{Vars}(b) \cup \mathsf{Vars}(C_1) \cup \mathsf{Vars}(C_2) \\
\mathsf{Vars}(\mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}) &= \mathsf{Vars}(b) \cup \mathsf{Vars}(C)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Asgn}(\mathbf{skip}) &= \emptyset \\
\mathsf{Asgn}(x := e) &= \{x\} \\
\mathsf{Asgn}(C_1 \,;\, C_2) &= \mathsf{Asgn}(C_1) \cup \mathsf{Asgn}(C_2) \\
\mathsf{Asgn}(\mathbf{if}\, b\, \mathbf{then}\, C_1\, \mathbf{else}\, C_2\, \mathbf{fi}) &= \mathsf{Asgn}(C_1) \cup \mathsf{Asgn}(C_2) \\
\mathsf{Asgn}(\mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}) &= \mathsf{Asgn}(C)
\end{aligned}$$

Example 2.7. *Recall the Example 2.1. For this language of integer and boolean expressions the function Vars is defined as follows:*

$$\begin{aligned}
\mathsf{Vars}(n) &= \emptyset \\
\mathsf{Vars}(x) &= \{x\} \\
\mathsf{Vars}(-e) &= \mathsf{Vars}(e) \\
\mathsf{Vars}(e_1 \odot e_2) &= \mathsf{Vars}(e_1) \cup \mathsf{Vars}(e_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Vars}(b) &= \emptyset \\
\mathsf{Vars}(\neg b) &= \mathsf{Vars}(b) \\
\mathsf{Vars}(b_1 \oslash b_2) &= \mathsf{Vars}(b_1) \cup \mathsf{Vars}(b_2)
\end{aligned}$$

With the presented set of tools it is already possible to write programs and to observe their execution. It is even possible to reason about executions as in the following example.

Example 2.8. *Consider the program from Example 2.2 and let $s \in \Sigma$, such that $s(y) = Y$ and $s(z) = Z$, for some constants $Y, Z \in \mathbb{Z}$. It can be proved that if $\langle x := y \,;\, y := z \,;\, z := x, s\rangle \rightsquigarrow s'$ then $s'(y) = Z$ and $s'(z) = Y$.*

Proof. *Let $s \in \Sigma$ such that $s(y) = Y$ and $s(z) = Z$. Since we have $\langle x := y \,;\, y := z \,;\, z := x, s\rangle \rightsquigarrow s'$, by the definition of $\rightsquigarrow$ there must exist some s'', such that $\langle x := y \,;\, y := z, s\rangle \rightsquigarrow s''$ and $\langle z := x, s''\rangle \rightsquigarrow s'$, and thus also, $\langle x := y, s\rangle \rightsquigarrow s'''$ and $\langle y := z, s'''\rangle \rightsquigarrow s''$. Again from the definition of $\rightsquigarrow$, we have $s''' = s[x \mapsto s(y)]$, $s'' = s'''[y \mapsto z]$, and $s' = s''[z \mapsto x]$. Thus $s'' = s[x \mapsto s(y), y \mapsto s(z)]$, and $s' = s[x \mapsto s(y), y \mapsto s(z), z \mapsto s(y)]$, which allows us to conclude that $s'(y) = s(z)$ and $s'(z) = s(y)$.*

The same proof can be done using the transition relation $\Rightarrow$ from the small-step semantics.

Even though the property that we wanted to prove in Example 2.8 was fairly simple, its proof got complicated due to details that are actually irrelevant for that property. The problem is that even though the semantics we used is very convenient for reasoning about concrete executions and to observe how the state evolves, it is not so practical for relating pre and post states. The following section introduces a new style of semantics, called *axiomatic semantics* which is intended for reasoning about properties to be established before and after the execution of a program.

2.2 Hoare Logic

Hoare logic [64] deals with the notion of correctness of a program w.r.t. a *specification*. A *Hoare triple*, written as $\{\phi\}\, C\, \{\psi\}$, expresses the fact that the program C conforms to the specification (ϕ, ψ). The intuitive meaning is that if the program C is executed in an initial state in which the precondition ϕ is true, then either execution of C does not terminate or if it does, the postcondition ψ will be true in the final state. Because termination is not guaranteed, this is called a *partial correctness* specification, which contrasts with the notion of *total correctness* that forces termination. Total correctness is normally seen as an extra layer on top of partial correctness [5], that is:

$$\text{total correctness} = \text{partial correctness} + \text{termination}$$

In practical terms if one wants to ensure total correctness of a program that has already been shown to be partially correct w.r.t. a specification, it is enough to annotate every loop construct with a variant and prove respectively that each variant decreases in every iteration. In this thesis, unless stated otherwise, we will be working with the notion of partial correctness.

In addition to commands and program expressions, we need formulas that express properties of particular states of the program, in particular for the pre and postconditions found in Hoare triples. This language will be called *specification* or *assertion language*, and **Assert** will denote the set of all formulas over it. Formulas from **Assert** should be much richer than $\mathbf{Exp}^{\mathbf{bool}}$ because they need to capture the behavior of programs. In this thesis we will assume that **Assert** is an expansion of $\mathbf{Exp}^{\mathbf{bool}}$, because it is also useful to refer to program expressions from the specification (a setting in which $\mathbf{Exp}^{\mathbf{bool}}$ and **Assert** are distinct would also be possible, see for instance [2]). Lower case Greek letters will normally be used to represent formulas from **Assert**. Having this in mind, program assertions $\phi, \theta, \psi \in \mathbf{Assert}$ will be formulas of a first-order language obtained as an expansion of $b \in \mathbf{Exp}^{\mathbf{bool}}$ with at least the universal and existential quantifier.

Example 2.9. *Let $b \in \mathbf{Exp}^{\mathbf{bool}}$. The language of assertions can be defined by the following syntax:*

$$\mathbf{Assert} \ni \phi \quad ::= \quad b \mid \phi \to \phi \mid \phi \leftrightarrow \phi \mid \forall\, x.\ \phi \mid \exists\, x.\ \phi$$

For the interpretation of assertions we take the usual interpretation of first-order formulas, noting two facts: since assertions build on the language of program expressions, their interpretation also depends on an interpretation structure $\mathcal{M}$ (possibly extended to account for user-defined predicates and functions), and states from Σ can be used as *variable assignments* in the interpretation of assertions. The interpretation of assertions $\phi \in \mathbf{Assert}$ is then given by an interpretation function $[\![\phi]\!]_{\mathcal{M}} : \Sigma \to \{\bot, \top\}$. We will write $s \models_{\mathcal{M}} \phi$ as a shorthand for $[\![\phi]\!]_{\mathcal{M}}(s) = \top$, and $s \models_{\mathcal{M}} \Gamma$ when $s \models_{\mathcal{M}} \phi$ for all $\phi \in \Gamma$. Similarly to the evaluation of program expressions we will omit the $\mathcal{M}$ subscript in the rest of the thesis. Moreover, the notation $[\![.]\!]$ for interpretation functions is overloaded, and applies to $e \in \mathbf{Exp}$, $b \in \mathbf{Exp^{bool}}$, and $\phi \in \mathbf{Assert}$. The concrete function to apply should be clear from the syntax of the expression.

Example 2.10. *The interpretation function $[\![.]\!]$ for assertions when* **Assert** *is defined as in Example 2.9 is given as follows:*

$$\begin{array}{rcl} [\![\phi_1 \to \phi_2]\!](s) & = & \top \quad \mathsf{iff}\ [\![\phi_1]\!](s) = \bot \vee [\![\phi_2]\!](s) = \top \\ [\![\phi_1 \leftrightarrow \phi_2]\!](s) & = & \top \quad \mathsf{iff}\ [\![\phi_1 \to \phi_2]\!](s) = \top \wedge [\![\phi_2 \to \phi_1]\!](s) = \top \\ [\![\forall\, x.\ \phi]\!](s) & = & \top \quad \mathsf{iff}\ [\![\phi]\!](s[x \mapsto a]) = \top,\ \textit{for all } a \in D \\ [\![\exists\, x.\ \phi]\!](s) & = & \top \quad \mathsf{iff}\ [\![\phi]\!](s[x \mapsto a]) = \top,\ \textit{for some } a \in D \end{array}$$

The set $\mathsf{FV}(\phi) \subseteq \mathbf{Var}$ denoting the free variables in $\phi \in \mathbf{Assert}$ extends the notion of $\mathsf{Vars}(.)$ to the language of assertions taking into account the bounded variables, as those that are bounded to quantifiers.

Example 2.11. *Let $\odot \in \{\to, \leftrightarrow\}$ and $\mathbb{Q} \in \{\forall, \exists\}$. The set of* free variables *of $\phi \in$* **Assert** *is given by $\mathsf{FV}(\phi)$ where FV is defined as follows:*

$$\begin{array}{rcl} \mathsf{FV}(b) & = & \mathsf{Vars}(b),\ \textit{for } b \in \mathbf{Exp^{bool}} \\ \mathsf{FV}(\phi \odot \psi) & = & \mathsf{FV}(\phi) \cup \mathsf{FV}(\psi) \\ \mathsf{FV}(\mathbb{Q}\, x.\ \phi) & = & \mathsf{FV}(\phi) \setminus \{x\} \end{array}$$

The section started by introducing the informal meaning of a Hoare triple $\{\phi\}\, C\, \{\psi\}$. We now switch to a more formal setting and define formally the validity of a Hoare triple in terms of operational semantics of programs.

Definition 2.4. *The Hoare triple $\{\phi\}\, C\, \{\psi\}$ is said to be* valid*, denoted $\models \{\phi\}\, C\, \{\psi\}$, whenever for all $s, s' \in \Sigma$, if $s \models \phi$ and $\langle C, s\rangle \rightsquigarrow s'$, then $s' \models \psi$.*

In fact, the validity of a triple also depends on an interpretation structure $\mathcal{M}$, and thus it should actually be written $\models_{\mathcal{M}} \{\phi\}\, C\, \{\psi\}$. Nonetheless, as we did with the evaluation of expressions and assertions, we opt for omitting the subscript. If $\models \{\phi\}\, C\, \{\psi\}$ holds, we say that C is (partially) *correct* w.r.t. the specification pair (ϕ, ψ) or that C conforms with the specification pair (ϕ, ψ).

$$\text{(skip)} \quad \frac{}{\{\phi\}\,\textbf{skip}\,\{\phi\}} \qquad \text{(assign)} \quad \frac{}{\{\psi[e/x]\}\,x := e\,\{\psi\}}$$

$$\text{(seq)} \quad \frac{\{\phi\}\,C_1\,\{\theta\} \qquad \{\theta\}\,C_2\,\{\psi\}}{\{\phi\}\,C_1\,;\,C_2\,\{\psi\}} \qquad \text{(if)} \quad \frac{\{\phi \wedge b\}\,C_1\,\{\psi\} \qquad \{\phi \wedge \neg b\}\,C_2\,\{\psi\}}{\{\phi\}\,\textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}\,\{\psi\}}$$

$$\text{(while)} \quad \frac{\{\theta \wedge b\}\,C\,\{\theta\}}{\{\theta\}\,\textbf{while}\ b\ \textbf{do}\ C\ \textbf{od}\,\{\theta \wedge \neg b\}} \qquad \text{(conseq)} \quad \frac{\{\phi\}\,C\,\{\psi\}}{\{\phi'\}\,C\,\{\psi'\}}\ \text{if}\ \begin{array}{l}\phi' \to \phi\ \text{ and}\\ \psi \to \psi'\end{array}$$

Figure 2.1: System H

Hoare introduced an inference system to reason about triples. Such a system can be seen as a framework to check the validity of triples that hides all the intricacies of operational semantics. The system is shown in Figure 2.1 and we will call it system H. Each rule consists in a set (possibly empty) of assumptions and a conclusion. The (skip) and (assign) rules are called *axioms* because they have an empty set of assumptions. On the other hand, in order to infer the validity of, for instance, $\{\phi\}\,C_1\,;\,C_2\,\{\psi\}$ using the (seq) rule, one must first derive the triples $\{\phi\}\,C_1\,\{\theta\}$ and $\{\theta\}\,C_2\,\{\psi\}$, for some $\theta \in \textbf{Assert}$. The same applies to the (if) and (while) rules. The (conseq) rule requires more than that. If the (conseq) rule is to be applied, in order to derive $\{\phi'\}\,C\,\{\psi'\}$, one must derive $\{\phi\}\,C\,\{\psi\}$, for some $\phi, \psi \in \textbf{Assert}$ and then, since the rule is guarded by first-order conditions, these conditions must be shown to be valid. We will write $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi\}$ to denote the fact that the triple is derivable in system H and that all side conditions from the derivation hold.

System H admits multiple derivations for the same Hoare triple, and does not impose any particular strategy for constructing them. Note for instance how the (seq) rule allows assertions to be propagated either forward or backwards: one can calculate a postcondition of C_1 w.r.t. ϕ and then propagate it to be the precondition of C_2 or the other way around, calculate the precondition of C_2 w.r.t. ψ and then propagate it to the postcondition of C_1. Since, the (assign) rule is based on a weakest precondition calculation, derivations based on backward propagation are in a sense more natural in this system. Note also that the (conseq) rule can be applied at any point in the derivation.

A derivation using system H is shown in the following example. Instead of using a tree structure for the proof (which would not fit in the page), we write it vertically using numbers and indentation to justify each step. The triple to derive is on the top, and each direct nested triple is the proof for the application of the indicated rule in blue.

Example 2.12. *Consider the factorial program shown in Example 2.3. Obviously, it only makes sense to calculate the factorial value of positive numbers, therefore the precondition must include that* $n \geq 0$. *For the postcondition it is fundamental to express that the variable* f *contains*

effectively the factorial of n*. To express this we consider that the language of assertions contains the postfix operator* $!$*, specified through the axioms* $0! = 1$ *and* $n! = n * (n-1)!$.

In the postcondition, we still need to refer to the initial value of n*. For that we will use an* auxiliary variable *that will only be used in the specification, to freeze the value of* n *in the precondition: note that* n *is updated during the execution of the program and therefore it cannot be used in the postcondition. Hence, the precondition of our program can be for instance* $n \geq 0 \land n_{aux} = n$*, and the postcondition* $f = n_{aux}!$*. A derivation of the triple* $\{n \geq 0 \land n_{aux} = n\}\, \mathsf{Fact}\, \{f = n_{aux}!\}$ *in system* H *is shown below.*

$\{n \geq 0 \land n_{aux} = n\}\, \mathsf{Fact}\, \{f = n_{aux}!\}$
(seq)

1. $\{n \geq 0 \land n_{aux} = n\}\, f := 1; i := 1\, \{f = 1 \land i = 1 \land n \geq 0 \land n_{aux} = n\}$
   (conseq)
   1. $\{1 = 1 \land 1 = 1 \land n \geq 0 \land n_{aux} = n\}\, f := 1\,;\, i := 1\, \{f = 1 \land i = 1 \land n \geq 0 \land n_{aux} = n\}$
      (seq)
      1. *(assign)* $\{1 = 1 \land 1 = 1 \land n \geq 0 \land n_{aux} = n\}\, f := 1\, \{f = 1 \land 1 = 1 \land n \geq 0 \land n_{aux} = n\}$
      2. *(assign)* $\{f = 1 \land 1 = 1 \land n \geq 0 \land n_{aux} = n\}\, i := 1\, \{f = 1 \land i = 1 \land n \geq 0 \land n_{aux} = n\}$
2. $\{f = 1 \land i = 1 \land n \geq 0 \land n_{aux} = n\}\, \textbf{while}\ i \leq n\ \textbf{do}\ f := f * i; i := i + 1\ \textbf{od}\, \{f = n_{aux}!\}$
   (conseq)
   1. $\{f = (i-1)! \land i \leq n + 1 \land n_{aux} = n\}\, \textbf{while}\ i \leq n\ \textbf{do}\ f := f * i; i := i + 1\ \textbf{od}\, \{f = (i-1)! \land i \leq n + 1 \land n_{aux} = n \land \neg i \leq n\}$
      (while)
      1. $\{f = (i-1)! \land i \leq n + 1 \land n_{aux} = n \land i \leq n\}\, \textbf{while}\ i \leq n\ \textbf{do}\ f := f * i; i := i + 1\ \textbf{od}\, \{f = (i-1)! \land i \leq n + 1 \land n_{aux} = n\}$
         (conseq)
         1. $\{f * i = (i + 1 - 1)! \land i + 1 \leq n + 1 \land n_{aux} = n\}\, f := f * i; i := i + 1\, \{f = (i-1)! \land i \leq n + 1 \land n_{aux} = n\}$
            (seq)
            1. *(assign)* $\{f * i = (i+1-1)! \land i+1 \leq n+1 \land n_{aux} = n\}\, f := f * i\, \{f = (i+1-1)! \land i+1 \leq n + 1 \land n_{aux} = n\}$
            2. *(assign)* $\{f = (i + 1 - 1)! \land i + 1 \leq n + 1 \land n_{aux} = n\}\, i := i + 1\, \{f = (i-1)! \land i \leq n + 1 \land n_{aux} = n\}$

Side conditions for application of the (conseq) rule:

- $n \geq 0 \land n_{aux} = n \implies 1 = 1 \land 1 = 1 \land n \geq 0 \land n_{aux} = n$.
- $f = 1 \land i = 1 \land n \geq 0 \land n_{aux} = n \implies f = (i-1)! \land i \leq n + 1 \land n_{aux} = n$ *and* $f = (i-1)! \land i \leq n + 1 \land n_{aux} = n \land \neg i \leq n \implies f = n_{aux}!$.
- $f = (i-1)! \land i \leq n + 1 \land n_{aux} = n \land i \leq n \implies f * i = (i + 1 - 1)! \land i + 1 \leq n + 1 \land n_{aux} = n$.

The fact that a triple is derivable in an arbitrary system does not necessarily imply that the triple is valid. This implication only holds if the inference system is shown to be sound w.r.t.

the operational semantics of the language. System H is in fact sound, and thus, it does not derive triples that are not valid.

Proposition 2.2 (Soundness of system H)**.** *Let $C \in$ **Comm** and $\phi, \psi \in$ **Assert**. If $\vdash_{\mathsf{H}}$ $\{\phi\}\, C\, \{\psi\}$, then $\models \{\phi\}\, C\, \{\psi\}$.*

Proof. By induction on the derivation of $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$. For the while case, induction on the definition of the evaluation relation is also required. See for instance Nielson et al. [95] for a complete proof. □

On the other hand, the notion of a system being sound does not make an inference system useful, in the sense that an inference system that does not derive any triple is sound. The notion of *completeness* of the inference system must also be analysed. This however deserves a deeper discussion than the soundness result. The account that follows is based on [5]. First of all, it is clear that only a *relative* notion of completeness can be achieved, for the simple reason that the application of the consequence rule is guarded by assertions. System H does not contain any rules for deriving these assertions, and moreover it would likely not be possible to extend the system with an adequate axiomatization, since interesting assertion languages may themselves be incomplete (it is difficult to imagine reasoning about programs without requiring arithmetics, for instance). Cook [36] notes that restricting the specification language (i.e. the language of assertions occurring in preconditions and postconditions) does not solve the problem, as shown by triples of the form $\{\top\}\, C\, \{\bot\}$, expressing that no execution of C halts. If the programming language is such that the halting problem is undecidable, the validity of such a triple does not imply that it can be derived in system H, which again confirms the incompleteness of the system.

The solution to this first problem is easy. Recall that the interpretation of assertions and Hoare triples is implicitly indexed by a model $\mathcal{M}$. We will equally assume that derivation of Hoare triples in system H is indexed by the *complete theory* of this structure (the set of all valid assertions under $\mathcal{M}$), so $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$ becomes shorthand for $\mathrm{Th}(\mathcal{M}) \vdash_{\mathsf{H}} \{\phi\}\, \mathrm{C}\, \{\psi\}$, which means that when constructing derivations in system H one simply checks, when applying the guarded (conseq) rule, whether the side conditions are elements of $\mathrm{Th}(\mathcal{M})$.[1]

As to the second problem, rather than explaining it now we will formulate the completeness result in a restricted way that already solves it. The problem will manifest itself as we write the proof of completeness; we will comment on it then. Let $C \in$ **Comm** and $\phi \in$ **Assert**, we denote by $\mathrm{post}(\phi, \mathrm{C})$ the set of states $\{s' \in \Sigma \mid \langle C, s\rangle \rightsquigarrow s' \text{ for some } s \in \Sigma \text{ such that } s \models \phi\}$.

Definition 2.5 (Expressiveness)**.** *The assertion language* **Assert** *is said to be* expressive *with respect to the command language* **Comm** *and interpretation structure $\mathcal{M}$, if for every $\phi \in$* **Assert** *and $C \in$* **Comm** *there exists $\psi \in$* **Assert** *such that $s \models \psi$ iff $s \in \mathrm{post}(\phi, \mathrm{C})$ for any $s \in \Sigma$.*

[1] In fact this should not be very surprising with our presentation of the (conseq) rule: the fact that the first-order guards appear as side conditions indicates that they are not supposed to be derived within the inference system. It is not unusual to see the guards presented as premisses of the rule, but strictly speaking the resulting systems are incomplete.

In other words, expressiveness guarantees the existence of the *postcondition* ψ of C with respect to ϕ [5].

Proposition 2.3 (Completeness of system H in the sense of Cook)**.** *Let* $C \in$ **Comm***,* $\phi, \psi \in$ **Assert***, and* $\mathcal{M}$ *be an interpretation structure such that* **Assert** *is expressive with respect to* **Comm** *and* $\mathcal{M}$*. If* $\models \{\phi\}\, C\, \{\psi\}$ *then* $\mathrm{Th}(\mathcal{M}) \vdash_{\mathsf{H}} \{\phi\}\, \mathrm{C}\, \{\psi\}$*.*

Proof. By induction on the structure of the program C. We show here the sequence case that illustrates the need for the expressiveness assumption. In what follows we write $[\![\cdot]\!]$ for $[\![\cdot]\!]_{\mathcal{M}}$, $\models$ for $\models_{\mathcal{M}}$, and $\vdash_{\mathsf{H}}$ for $\mathrm{Th}(\mathcal{M}) \vdash_{\mathsf{H}}$.

Assume C is $C_1\, ;\, C_2$. We assume that for all states s and s'', if $[\![\phi]\!](s) = \top$ and $\langle C_1\, ;\, C_2, s\rangle \rightsquigarrow s''$ then $[\![\psi]\!](s'') = \top$. We have the following induction hypotheses, where we fix the formulas ϕ and ψ :

IH1: For every formula θ, if $\models \{\phi\}\, C_1\, \{\theta\}$ then $\vdash_{\mathsf{H}} \{\phi\}\, C_1\, \{\theta\}$.

IH2: For every formula θ, if $\models \{\theta\}\, C_2\, \{\psi\}$ then $\vdash_{\mathsf{H}} \{\theta\}\, C_2\, \{\psi\}$.

Now it is apparently straightforward to conclude this proof case by applying both induction hypotheses. But in fact we must be careful: from the semantics we know that there must exist some s' such that $\langle C_1, s\rangle \rightsquigarrow s'$ and $\langle C_2, s'\rangle \rightsquigarrow s''$, but the assumptions do not immediately indicate a concrete intermediate assertion θ. It is here that Cook's expressiveness requirement becomes useful: we know there exists $\theta \in$ **Assert** such that $s' \models \theta$ iff $s' \in \mathrm{post}(\phi, \mathrm{C}_1)$ for any $s' \in \Sigma$. So we immediately have $\models \{\phi\}\, C_1\, \{\theta\}$, and since $\models \{\phi\}\, C_1\, ;\, C_2\, \{\psi\}$ we also necessarily have that $\models \{\theta\}\, C_2\, \{\psi\}$. Both induction hypotheses then apply, which allows for the proof case to be concluded using the (seq) rule of system H. □

As a final remark, we note that completeness means in particular that (under the expressiveness assumption) it is always possible to write appropriate invariants for a program to be deductively shown correct with respect to a specification, if indeed it is correct with respect to it.

The following definition establishes notation for identifying programs that do not assign free variables of an assertion. In particular this notation is used in the next lemma to express the fact that a precondition is preserved after the execution of a program, if the program does not assign free variables of it.

Definition 2.6. *If* $\mathsf{FV}(\phi) \cap \mathsf{Asgn}(C) = \emptyset$ *we say that* C *does not assign free variables of* ϕ*, and write this as* $\phi\#C$*.*

Lemma 2.4. *Let* $\phi, \psi \in$ **Assert** *and* $C \in$ **Comm***, such that* $\phi\#C$*. If* $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$*, then* $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\phi \wedge \psi\}$*.*

Proof. Consider the *conjunction of assertions* rule below that is admissible in system H (see for instance [96]).

$$\frac{\vdash_{\mathsf{H}} \{\phi_1\}\, C\, \{\psi_1\} \qquad \vdash_{\mathsf{H}} \{\phi_2\}\, C\, \{\psi_2\}}{\vdash_{\mathsf{H}} \{\phi_1 \wedge \phi_2\}\, C\, \{\psi_1 \wedge \psi_2\}}$$

By induction on the structure of C one proves that $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\phi\}$. Then it follows from $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$, by the conjunction of assertions rule shown above that $\vdash_{\mathsf{H}} \{\phi \wedge \phi\}\, C\, \{\phi \wedge \psi\}$. The triple $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\phi \wedge \psi\}$ is obtained by applying (conseq). □

2.3 An Inference System for Annotated Programs

In the previous section a systematic approach for verifying whether a Hoare triple is valid was presented: if one is able to construct a derivation using system H and if all side conditions hold, then the triple is valid. Constructing derivations manually is tiresome and thus a mechanized process is desired. Nonetheless, system H is not suitable for this for two reasons that we will now discuss.

The consequence rule of Hoare logic provides the ‘glue’ that allows derivations to be plugged together, when the formulas do not match because they are either too strong or too weak. This also allows for proofs to be *reused*. For instance if the code contains some form of sub-routine, then instead of producing one different proof for each call of such a routine, one may construct a single one with respect to a sufficiently rich specification (usually called a *contract*). This same proof may then be plugged into different places in the overall derivation, using the consequence rule to adapt it to the local requirements of each call. The problem with this rule is that it is a source of nondeterminacy – it is the only rule in system H whose application is not directed by the structure of the program, and may thus occur at any point.

Another relevant aspect to be taken into account in system H is that loop invariants must be invented during the construction of the derivation tree. Even though research exists on the generation of loop invariants, user intervention is often required. To avoid the need for inventing loop invariants during the derivation we consider a syntactic class of *annotated programs*, which differs from **Comm** only in the case of while commands [54]. The syntax of a command $C \in \mathbf{AComm}$, with annotated invariants $\theta \in \mathbf{Assert}$, is defined as:

$$C \;::=\; \mathbf{skip} \;\mid\; x := e \;\mid\; C\,;C \mid\; \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C\ \mathbf{fi} \;\mid\; \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \mathbf{od}$$

Example 2.13. *The program below is an example of an annotated program in* **AComm**.

$$\begin{array}{l} f := 1\,;\, i := 1\,; \\ \mathbf{while}\ i \leq n\ \mathbf{do}\ \{f = (i-1)! \wedge i \leq n+1 \wedge n_{aux} = n\} \\ \quad f := f * i\,; \\ \quad i := i + 1 \\ \ \mathbf{od} \end{array}$$

$$(\text{skip}) \quad \frac{}{\{\phi\}\,\textbf{skip}\,\{\psi\}} \text{ if } \phi \to \psi \qquad (\text{assign}) \quad \frac{}{\{\phi\}\, x := e\,\{\psi\}} \text{ if } \phi \to \psi[e/x]$$

$$(\text{seq}) \quad \frac{\{\phi\}\, C_1\,\{\theta\} \qquad \{\theta\}\, C_2\,\{\psi\}}{\{\phi\}\, C_1\,;\, C_2\,\{\psi\}} \qquad (\text{if}) \quad \frac{\{\phi \wedge b\}\, C_1\,\{\psi\} \qquad \{\phi \wedge \neg b\}\, C_2\,\{\psi\}}{\{\phi\}\,\textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}\,\{\psi\}}$$

$$(\text{while}) \quad \frac{\{\theta \wedge b\}\, C\,\{\theta\}}{\{\phi\}\,\textbf{while}\ b\ \textbf{do}\ \{\theta\}\ C\ \textbf{od}\,\{\psi\}} \text{ if } \begin{array}{l} \phi \to \theta \text{ and} \\ \theta \wedge \neg b \to \psi \end{array}$$

Figure 2.2: System Hg

Annotations do not affect the operational semantics of programs, therefore we will use the function $\lfloor\cdot\rfloor : \textbf{AComm} \to \textbf{Comm}$ below to erase all the annotations from a program.

Definition 2.7. *The function* $\lfloor\cdot\rfloor : \textbf{AComm} \to \textbf{Comm}$ *is defined as follows:*

$$\begin{array}{rcl} \lfloor\textbf{skip}\rfloor & = & \textbf{skip} \\ \lfloor x := e\rfloor & = & x := e \\ \lfloor C_1\,;\, C_2\rfloor & = & \lfloor C_1\rfloor\,;\, \lfloor C_2\rfloor \\ \lfloor\textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}\rfloor & = & \textbf{if}\ b\ \textbf{then}\ \lfloor C_1\rfloor\ \textbf{else}\ \lfloor C_2\rfloor\ \textbf{fi} \\ \lfloor\textbf{while}\ b\ \textbf{do}\ \{\theta\}\ C\ \textbf{od}\rfloor & = & \textbf{while}\ b\ \textbf{do}\ \lfloor C\rfloor\ \textbf{od} \end{array}$$

The following definition expands the notion of variables of a program and variables assigned in a program for annotated programs.

Definition 2.8. *The definition of* Vars *and* Asgn *is extended in the obvious way:* $\mathsf{Vars}(\textbf{while}\ b\ \textbf{do}\ \{\theta\}\ C\ \textbf{od}) = \mathsf{Vars}(b) \cup \mathsf{FV}(\theta) \cup \mathsf{Vars}(C)$ *and* $\mathsf{Asgn}(\textbf{while}\ b\ \textbf{do}\ \{\theta\}\ C\ \textbf{od}) = \mathsf{Asgn}(C)$*. All other cases are as in Definition 2.3 and Example 2.11 respectively.*

Determinacy is important for the purpose of verifying programs mechanically, so Figure 2.2 presents system Hg [54], a *goal-directed* version of Hoare logic for triples containing annotated programs. This system is intended for mechanical construction of derivations: loop invariants are not invented at this point but taken from the annotations, and there is no ambiguity in the choice of rule to apply, since the consequence rule is absent. The 'glue' is provided instead by modifying the remaining rules of the system.

Proposition 2.5 (Soundness of Hg)**.** *Let* $C \in \textbf{AComm}$ *and* $\phi, \psi \in \textbf{Assert}$*. If* $\vdash_{\mathsf{Hg}} \{\phi\}\, C\,\{\psi\}$ *then* $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor C\rfloor\,\{\psi\}$.

Proof. By induction on the derivation of $\vdash_{\mathsf{Hg}} \{\phi\}\, C\,\{\psi\}$. All cases are straightforward. $\square$

The converse implication does not hold, since the annotated invariants may be inadequate for deriving the triple. Instead we need the following definition:

Definition 2.9. *Let $C \in$ **AComm** and $\phi, \psi \in$ **Assert**. We say that C is* correctly-annotated *w.r.t. (ϕ, ψ) if $\vdash_{\mathsf{H}} \{\phi\} \lfloor C \rfloor \{\psi\}$ implies $\vdash_{\mathsf{Hg}} \{\phi\} C \{\psi\}$.*

Note that the loop invariants annotated in a program may well depend on the specification of the program, since their purpose is to allow for the program to be proved correct with respect to it. Finally, the following lemma states the admissibility of the consequence rule in Hg.

Lemma 2.6. *Let $C \in$ **AComm** and $\phi, \psi, \phi', \psi' \in$ **Assert** such that $\vdash_{\mathsf{Hg}} \{\phi\} C \{\psi\}$, $\models \phi' \to \phi$, and $\models \psi \to \psi'$. Then $\vdash_{\mathsf{Hg}} \{\phi'\} C \{\psi'\}$.*

Proof. By induction on the derivation of $\vdash_{\mathsf{Hg}} \{\phi\} C \{\psi\}$. □

Derivations in system Hg have a fixed shape – the structure of the programs determines uniquely the rules to be applied. Different derivations can still be constructed for the same Hoare triple, differing only in the intermediate formulas used when applying the (seq) or (try) rules. System Hg is agnostic with respect to a strategy for choosing these assertions; two commonly used strategies are forward propagation (based on *strongest postcondition* computations), and backward propagation (based on *weakest precondition*). We will postpone this discussion to the following section.

Example 2.14. *Consider the factorial program shown in Example 2.13. It is easy to show that it is correct with respect to the specification $(n \geq 0 \wedge n_{aux} = n, f = n_{aux}!)$. We show below a derivation of this triple in system* Hg*. Again, the axioms $0! = 1$ and $n! = n * (n-1)!$ are required to prove the side conditions.*

$\{n \geq 0 \wedge n_{aux} = n\}$ Fact $\{f = n_{aux}!\}$
(seq)

1. $\{n \geq 0 \wedge n_{aux} = n\}\ f := 1; i := 1\ \{n \geq 0 \wedge n_{aux} = n \wedge f = 1 \wedge i = 1\}$
   (seq)
   1. *(assign)* $\{n \geq 0 \wedge n_{aux} = n\}\ f := 1\ \{n \geq 0 \wedge n_{aux} = n \wedge f = 1\}$
   2. *(assign)* $\{n \geq 0 \wedge n_{aux} = n \wedge f = 1\}\ i := 1\ \{n \geq 0 \wedge n_{aux} = n \wedge f = 1 \wedge i = 1\}$
2. $\{n \geq 0 \wedge n_{aux} = n \wedge f = 1 \wedge i = 1\}$ **while** $i \leq n$ **do** $\{f = (i-1)! \wedge i \leq n+1 \wedge n_{aux} = n\}\ f := f * i; i := i+1$ **od** $\{f = n_{aux}!\}$
   (while)
   1. $\{f = (i-1)! \wedge i \leq n+1 \wedge n_{aux} = n \wedge i \leq n\}\ f := f*i; i := i+1\ \{f = (i-1)! \wedge i \leq n+1 \wedge n_{aux} = n\}$
      (seq)
      1. *(assign)* $\{f = (i-1)! \wedge i \leq n+1 \wedge n_{aux} = n \wedge i \leq n\}\ f := f*i\ \{f = (i-1)!*i \wedge i \leq n+1 \wedge n_{aux} = n \wedge i \leq n\}$
      2. *(assign)* $\{f = (i-1)!*i \wedge i \leq n+1 \wedge n_{aux} = n \wedge i \leq n\}\ i := i+1\ \{f = (i-1)! \wedge i \leq n+1 \wedge n_{aux} = n\}$

Side conditions for application of the (assign) rules:

- $n \geq 0 \land n_{aux} = n \implies (n \geq 0 \land n_{aux} = n \land f = 1)[1/f]$.
- $n \geq 0 \land n_{aux} = n \land f = 1 \implies (n \geq 0 \land n_{aux} = n \land f = 1 \land i = 1)[1/i]$.
- $f = (i-1)! \land i \leq n+1 \land n_{aux} = n \land i \leq n \implies (f = (i-1)! * i \land i \leq n+1 \land n_{aux} = n \land i \leq n)[f * i/f]$.
- $f = (i-1)! * i \land i \leq n+1 \land n_{aux} = n \land i \leq n \implies (f = (i-1)! \land i \leq n+1 \land n_{aux} = n)[i+1/i]$.

Side conditions for application of the (while) rule:

- $n \geq 0 \land n_{aux} = n \land f = 1 \land i = 1 \implies f = (i-1)! \land i \leq n+1 \land n_{aux} = n$.
- $f = (i-1)! \land i \leq n+1 \land n_{aux} = n \land \neg(i \leq n) \implies f = n_{aux}!$.

Before moving forward, note that there exists a line of research on *loop invariant generation*. The idea is that loop invariants are automatically (or semi-automatically, since some tools allow for user guidance) invented by the verification tool, or else, they are invented by some external tool and then annotated in the code. Since this falls outside the context of this thesis we do not extend this discussion, but it should be mentioned that it has been a hot topic basically since verification of programs started to be explored [73, 38, 92, 61, 99, 3].

2.4 Predicate Transformers

Predicate transformers offer an alternative view of the semantics of programs, which are interpreted as transforming logical formulas characterizing states. For instance the *strongest postcondition* (SP) interpretation of a program maps a formula ϕ (a *precondition*) into another formula ψ that characterizes final states of the program starting from initial states satisfying ϕ. Our interest in the use of predicate transformers is that computing them allows for the generation of a set of formulas that can ensure the correctness of Hoare triples [84, 52, 51] (the topic of verification conditions will be addressed in the next section).

Aiming to focus on the core functionality of algorithms and their beauty for human readers, Dijkstra proposed a language baptized as *guarded commands* and a calculus to reason about programs in such a language [44, 45, 46]. The calculus is used as a mechanism that abstracts away all the intricacies of operational semantics and allows one to reason about program properties without the trouble of building derivations, as done in axiomatic semantics. In some way, it resembles denotational semantics, in the sense that it operates on mathematical objects. Instead of considering the native guarded commands proposed by Dijkstra [45], we consider instead a version of this language that has been used along the years in the context of program verification.

A Kind of Guarded Commands. Even though the language we will introduce in this section differs from the initial guarded commands of Dijkstra, and it actually does not contain the typical guards that give the name to the language (they are replaced by assume commands, as will be seen below), in the rest of this thesis we will refer to it as the *guarded command* language. Moreover, in what follows we will omit iteration commands, because for verification

purposes they can be captured by the rest of the commands (we will see more on this in the last part of this section). The following abstract syntax tree defines the guarded command language that we will use in the rest of this section over a set of variables $x \in \mathbf{GVar}$.

$$\mathbf{GComm} \ni C \quad ::= \quad x := e \mid C \,;\, C \mid C \,\Box\, C \mid \mathbf{assert}\, e \mid \mathbf{assume}\, e$$

For the language of expressions $e \in \mathbf{GExp}$, it is assumed that it is pure (expressions do not contain side-effects) and it includes at least the boolean constants *true* and *false*. It is also assumed that $\Box$ has higher precedence than ; .

It should be mentioned that a guarded command can terminate normally, not terminate at all, or it can go wrong. Let us start by analyzing each command separately. The idea of the two first constructs is exactly the same as in **Comm**. In particular $x := e \in \mathbf{GComm}$ always terminates normally, and $C_1; C_2 \in \mathbf{GComm}$ can go wrong if either C_1 goes wrong or if C_1 terminates normally and C_2 goes wrong. The third command, known as the *choice command*, executes one and only one of the commands, non-deterministically. A command $C_1 \,\Box\, C_2 \in \mathbf{GComm}$ can terminate normally or go wrong depending on whether C_1 or C_2 is executed and whether it terminates normally goes wrong. The command $\mathbf{assert}\, e$ is the only atomic command that can go wrong. More precisely, $\mathbf{assert}\, e$ goes wrong if e is not satisfied in the current state, or it terminates normally without any effect otherwise. On the other hand an $\mathbf{assume}\, e$ statement cannot be executed in a state where the condition e is false (when this happens we say that it blocks). Otherwise it does not produce any effect and terminates normally. Even though the language does not contain an explicit deterministic **if** statement, this can be encoded using the choice command together with the assume and sequence commands: a statement with the form $\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}$ can be written as $(\mathbf{assume}\, b;\, C_1) \,\Box\, (\mathbf{assume}\, \neg b;\, C_2)$.

We should mention here that even though the pair of commands assume/assert are normally used in context of deductive verification, their use goes well beyond it. For instance as we will see in Section 2.6, they are both used in *bounded model checking of software*, and in *software model checking* in general.

Predicate Transformers. The question that we should ask now is how can one reason about guarded programs. Traditionally there is a clear distinction between the operational semantics, which interprets programs as mechanisms that update the current state, and the axiomatic semantics, used for reasoning about programs. Dijkstra proposed to reason about commands using a form of program calculus known as *predicate transformer calculus*. The main idea is to look at guarded commands as functions from predicates to predicates: instead of reasoning with concrete states and observing how they evolve, Dijkstra proposes to reason with predicates and observe how commands transform those predicates. As a first example, the weakest precondition predicate transformer is given by the following definition.

Definition 2.10. *Let $C \in \mathbf{GComm}$ and $\psi \in \mathbf{GAssert}$. The weakest precondition of C with respect to the postcondition ψ is given by $\mathsf{wp}(C, \psi)$ where $\mathsf{wp} : \mathbf{GComm} \times \mathbf{GAssert} \to \mathbf{GAssert}$*

is defined as follows:

$$\begin{aligned}
\mathsf{wp}(\textbf{assert}\, e, \psi) &= e \wedge \psi \\
\mathsf{wp}(\textbf{assume}\, e, \psi) &= e \rightarrow \psi \\
\mathsf{wp}(x := e, \psi) &= \psi[e/x] \\
\mathsf{wp}(C_1 \,;\, C_2, \psi) &= \mathsf{wp}(C_1, \mathsf{wp}(C_2, \psi)) \\
\mathsf{wp}(C_1 \,\square\, C_2, \psi) &= \mathsf{wp}(C_1, \psi) \wedge \mathsf{wp}(C_2, \psi)
\end{aligned}$$

The set of logical formulas **GAssert** extends **GExp** with at least the boolean operators $\{\wedge, \vee, \rightarrow, =\}$ and quantifiers. The predicate $\mathsf{wp}(C, \psi)$, with $C \in \textbf{GComm}$ and $\psi \in \textbf{GAssert}$, captures all the initial states from which every non-blocking execution of C terminates in a state where ψ holds. The required precondition for **assert** e to terminate normally in a state where ψ holds, is given by $e \wedge \psi$. For the command **assume** e one must have as precondition $e \rightarrow \psi$. Note that ψ is only required to be true in states in which the command **assume** e can be executed, i.e. whenever e holds. The assignment command $x := e$ requires the condition $\psi[e/x]$ to be met for it to terminate normally in a state where the condition ψ holds. For a sequence of commands, one must first calculate the weakest precondition of the second command and use it when calculating the weakest precondition of the first command. In the choice command the weakest precondition must satisfy both $\mathsf{wp}(C_1, \psi)$ and $\mathsf{wp}(C_2, \psi)$, because independently of the command that is executed, in the end, the postcondition ψ must be met.

The function wp presented in Definition 2.10 is just a predicate transformer among others. As stated before this function intends to capture the weakest precondition for a command to be executed and to *terminate* in a state where *a postcondition is met*: it is directly linked with the notion of *total correctness*. An alternative predicate transformer known as *weakest liberal precondition* [46] is closely linked with *partial correctness*: the obtained predicate does not impose that the program has to terminate. In our setting since we do not have loops, this distinction does not make much sense. As an alternative, in the presence of the assert command, it is customary to define the weakest liberal precondition as being the predicate that captures all the states from which the execution either goes wrong (some assert fails during the execution) or terminates in a state satisfying ψ. In other words, termination is interpreted as 'not going wrong'. The definition of weakest liberal precondition is then given by the following definition.

Definition 2.11. *Let* $C \in \textbf{GComm}$ *and* $\psi \in \textbf{GAssert}$*. The weakest liberal precondition of* C *with respect to the postcondition* ψ *is given by* $\mathsf{wlp}(C, \psi)$ *where* $\mathsf{wlp} : \textbf{GComm} \times \textbf{GAssert} \rightarrow \textbf{GAssert}$ *is defined as follows:*

$$\begin{aligned}
\mathsf{wlp}(\textbf{assert}\, e, \psi) &= e \rightarrow \psi \\
\mathsf{wlp}(\textbf{assume}\, e, \psi) &= e \rightarrow \psi \\
\mathsf{wlp}(x := e, \psi) &= \psi[e/x] \\
\mathsf{wlp}(C_1 \,;\, C_2, \psi) &= \mathsf{wlp}(C_1, \mathsf{wlp}(C_2, \psi)) \\
\mathsf{wlp}(C_1 \,\square\, C_2, \psi) &= \mathsf{wlp}(C_1, \psi) \wedge \mathsf{wlp}(C_2, \psi)
\end{aligned}$$

Note that the only difference is in the interpretation of the assert command: the condition ψ has to be met only if e holds. The inverse of weakest liberal precondition is the *strongest postcondition* predicate transformer [46] that captures the final state of computations that start in a state where a certain condition holds. This predicate transformer is given through the following definition.

Definition 2.12. *Let* $C \in \mathbf{GComm}$ *and* $\phi \in \mathbf{GAssert}$*. The strongest postcondition of* C *with respect to the precondition* ϕ *is given by* $\mathsf{sp}(\phi, C)$ *where* $\mathsf{sp} : \mathbf{GAssert} \times \mathbf{GComm} \to \mathbf{GAssert}$ *is defined as follows:*

$$
\begin{array}{rcl}
\mathsf{sp}(\phi, \mathbf{assert}\, e) & = & \phi \wedge e \\
\mathsf{sp}(\phi, \mathbf{assume}\, e) & = & \phi \wedge e \\
\mathsf{sp}(\phi, x := e) & = & \exists\, x'.\ x = e[x'/x] \wedge \phi[x'/x] \\
\mathsf{sp}(\phi, C_1 \,;\, C_2) & = & \mathsf{sp}(\mathsf{sp}(\phi, C_1), C_2) \\
\mathsf{sp}(\phi, C_1 \,\square\, C_2) & = & \mathsf{sp}(\phi, C_1) \vee \mathsf{sp}(\phi, C_2)
\end{array}
$$

As shown in the definition above, the strongest postcondition of an assignment introduces existential quantifiers. This is the main reason why the weakest precondition predicate transformers is normally preferred to strongest postcondition, in both the theoretical context, and verification tools. For a detailed monograph about important properties of the predicate transformers, and their different forms and relations see [46].

Predicate transformers have had a great impact on the verification tools we use nowadays. Since they are based on a calculation mechanism, they naturally provide a better foundation for automatic verification tools than the traditional Hoare logic [43], more specifically concerning the generation of verification conditions. In particular, the verification of a Hoare triple $\{\phi\}\, C\, \{\psi\}$ is reduced to the verification of one of the formulas $\phi \to \mathsf{wlp}(C, \psi)$ or $\mathsf{sp}(\phi, C) \to \psi$ [58] (more on this in the next section).

A clear consequence of using predicate transformers as a replacement for the operational and axiomatic semantics is that the formal meaning of the language statements becomes less clear, either operationally or axiomatically. For instance, assume and assert statements were initially introduced in the context of guarded commands, nonetheless, it is not specified operationally what they mean. Operationally both of them can be seen as the skip command, but, the assert command can also be seen as a command that checks in runtime whether the given property holds in the current state, as in ANSI C. If an operational semantics is provided for the language, either small-step or big-step, these aspects become clear.

Avoiding Exponential Explosion. The size of the formulas generated by the predicate transformers above can grow exponentially in the worse case w.r.t. the size of the program. Let us focus on Definition 2.10. There are two factors contributing towards this explosion. The first is the definition of wp in the case of the choice command: the received postcondition ψ appears twice in the result. Consider a command consisting in a sequence of n choice commands, and a postcondition ψ. When applying wp the condition ψ is propagated through both branches

of the last choice command, and produces two assertions ψ_1, ψ_2. These will be combined in a predicate $\psi_1 \wedge \psi_2$, where ψ may occur twice. The (seq) rule will then use $\psi_1 \wedge \psi_2$ as postcondition for the prefix of the program, repeating the process and generating the exponential pattern.

Example 2.15. *Take the program* ChoiceEg *as being* $(\mathbf{assume}\,\theta_1 \,\Box\, \mathbf{assume}\,\theta_2); (\mathbf{assume}\,\theta_3 \,\Box\, \mathbf{assume}\,\theta_4); (\mathbf{assume}\,\theta_5 \,\Box\, \mathbf{assume}\,\theta_6)$. *The function* wp *will generate a formula where the desired postcondition will occur eight times. If we add one more choice command to the sequence then the number of occurrences of the postcondition will be sixteen in the end:*

$$\begin{aligned}\mathsf{wp}(\mathsf{ChoiceEg}, \psi) = (\theta_1 \to (\theta_3 \to (\theta_5 \wedge \psi) \wedge (\theta_6 \to \psi)) \wedge (\theta_4 \to (\theta_5 \wedge \psi) \wedge (\theta_6 \to \psi))) \wedge \\ (\theta_2 \to (\theta_3 \to (\theta_5 \wedge \psi) \wedge (\theta_6 \to \psi)) \wedge (\theta_4 \to (\theta_5 \wedge \psi) \wedge (\theta_6 \to \psi)))\end{aligned}$$

A similar exponential pattern may be generated by duplicating variables rather than assertions in a sequence of assignment statements whose right-hand sides contain multiple occurrences of the same variable.

Example 2.16. *Consider a postcondition* ψ *containing a single occurrence of* z. *The* wp *of the sequence of assignments* $y := x + x;\ z := y + y$ *produces a formula containing at least four occurrences of* x, *since* $\mathsf{wp}(y := x + x;\ z := y + y, \psi) = \psi[y + y/z][x + x/y]$. *Note that after* z *is substituted by* $y + y$, *each* y *will be substituted by* $x + x$.

It was observed by Flanagan and Saxe that assignments are the main cause of explosion and the problem can be solved by converting programs into *passive form*, a form in which assignments are removed [52]. Leino [82] later clarified that such an approach works because passive commands are exactly the class of programs that do not change the state of the program during execution (programs in this class are said to enjoy from the *dream property*).

To be able to remove assignment commands while keeping the same program behavior, one must make use of the other commands. The idea when converting a program into passive form consists roughly in replacing assignments by assumes. However, it is obvious that for an assignment of the form $x := e$ one cannot simply write $\mathbf{assume}\, x = e$, because the condition $x = e$ does not make sense if x occurs in e, in which case it will probably be a contradiction (consider for instance $x = x + 1$). To circumvent this limitation, a new fresh variable, which we will call a *variant* of x, can be introduced each time x is assigned. This new variant of the variable has to be used in all the subsequent reads of x (at least until x is assigned again). This way, the assignment $x := e$ can be replaced by the command $\mathbf{assume}\, x' = e$ and if x is read afterwards, then it must be substituted by x'.

The function that transforms a program into passive form presented in [52] is recursive and carries a substitution map from variables to their variant. When the current command is an assignment the received substitution map is used to rename its right hand side and the introduced fresh variable is used to update the map for the subsequent commands. For the

assume and assert commands the function basically substitutes the variables according to the received substitution map.

The choice command introduces a new challenge: after the command the substitution map coming from each branch must be merged, and every variable must be synchronized with the variant resulting from each branch. Consider for instance the example $x := e \,\Box\, \mathbf{assume}\,\theta$. The variable x is only assigned by the left hand side of the command. One can fall into the temptation of translating this into $x' := e' \,\Box\, \mathbf{assume}\,\theta'$, where e' and θ' are the result of the replacing the variables with their respective current variant, and x' is a fresh variable. The problem here is that in the translated program, x' is not assigned in the right hand side of the choice command, and thus its value is undefined, as opposed to the original program where the value of x was preserved. The solution is to introduce a new assignment on the right hand side for x' to keep the previous variant of x. The result would then be $x' := e' \,\Box\, (\mathbf{assume}\,\theta';\, x' := x'')$, assuming that x'' is the previous variant of x.

Let $\mathsf{passify}(\sigma, C)$ denote the function that translates guarded commands into a passive form. The following result taken from [52] states that the translation preserves the wp semantics.

Proposition 2.7. *Let $C \in$ **GComm**, σ be a substitution map,* $\mathsf{passify}$ *the translation function, $(\sigma', C') = \mathsf{passify}(\sigma, C)$, and $x_1, \ldots, x_n$ the additional variables introduced by the translation. Then for every $\psi \in$ **GExp** containing no free occurrences of $x_1, \ldots, x_n$, $\sigma(\mathsf{wp}(C, \psi)) \equiv \forall\, x_1, \ldots, x_n.\ \mathsf{wp}(C', \sigma'(\psi))$, where $\sigma(_)$ substitutes the variables according to the substitution map σ.*

Once a program is converted into passive form, its execution will not change the program state, because there are no assignments. Thus, the initial state is kept throughout the execution and the only thing that can vary is the outcome of the execution: it can terminate normally or it can go wrong. In order to generate weakest preconditions for these programs, it suffices to capture when they terminate normally and when they go wrong. The functions N and W in the definition below do precisely this.

Definition 2.13. *Let $C \in$ **GComm** and consider the functions N : **GComm** $\to$ **GAssert** and W : **GComm** $\to$ **GAssert** defined below. The predicate $\mathsf{N}(C)$ characterizes the states from which the program terminates normally; the predicate $\mathsf{W}(C)$ characterizes the states from which the program goes wrong.*

$$
\begin{array}{rclrcl}
\mathsf{N}(\mathbf{assert}\,e) & = & e & \mathsf{W}(\mathbf{assert}\,e) & = & \neg e \\
\mathsf{N}(\mathbf{assume}\,e) & = & e & \mathsf{W}(\mathbf{assume}\,e) & = & \bot \\
\mathsf{N}(C_1;\, C_2) & = & \mathsf{N}(C_1) \wedge \mathsf{N}(C_2) & \mathsf{W}(C_1;\, C_2) & = & \mathsf{W}(C_1) \vee (\,\mathsf{N}(C_1) \wedge \mathsf{W}(C_2)\,) \\
\mathsf{N}(C_1 \,\Box\, C_2) & = & \mathsf{N}(C_1) \vee \mathsf{N}(C_2) & \mathsf{W}(C_1 \,\Box\, C_2) & = & \mathsf{W}(C_1) \vee \mathsf{W}(C_2)
\end{array}
$$

Both the assert and assume commands terminate normally if the condition is true, but only the assert command can go wrong in the case where the condition is false – the assume command can never go wrong. A sequence terminates normally if both parts terminate normally and it goes wrong if either the first component goes wrong or it terminates normally and the second

goes wrong. The choice command terminates normally (resp. goes wrong) if at least one of the components terminate normally (goes wrong). The relation between wp and the newly defined predicates is as follows:

Proposition 2.8. *Let $C \in$ **GComm** and $\psi \in$ **GAssert** such that C is in passive form. Then* $\mathsf{wp}(C, \psi) \equiv \neg\mathsf{W}(C) \wedge (\,\mathsf{N}(C) \rightarrow \psi\,)$.

Let us go back to Examples 2.15 and 2.16 that caused an exponential explosion when fed to wp. Following the preceding method both approaches generate a formula of linear-size.

Example 2.17. *Consider the programs from Examples 2.15 and 2.16. Assume that* ChoiceEg *is already in passive form, and let* AsgnEg *be the passive program* **assume** $y_1 = x_0 + x_0$; **assume** $z_1 = y_1 + y_1$. *Let also $\psi \in$ **GExp**. Then:*

- $\neg\mathsf{W}(\mathsf{ChoiceEg}) \wedge (\,\mathsf{N}(\mathsf{ChoiceEg}) \rightarrow \psi\,) = \neg\bot \wedge ((\theta_1 \vee \theta_2) \wedge (\theta_3 \vee \theta_4) \wedge (\theta_5 \vee \theta_6) \rightarrow \psi)$.
- $\neg\mathsf{W}(\mathsf{AsgnEg}) \wedge (\,\mathsf{N}(\mathsf{AsgnEg}) \rightarrow \psi\,) = \neg\bot \wedge (y_1 = x_0 + x_0 \wedge z_1 = y_1 + y_1 \rightarrow \psi)$.

It can be proved by structural induction on $C \in$ **GComm** that $\neg\mathsf{W}(C)$ is of quadratic size and $\mathsf{N}(C)$ of linear size with respect to the size of the program C.

Dealing with Loops. At this point we should recall that our interest in predicate transfomers is motivated by the generation of verification conditions for iterating programs. There are different ways of transforming an iterating program into a loop-free program such as those constructed over **GComm**. Let us first consider an approach that does not require loop invariants. Such an approach was named as *even weaker precondition* (ewp) in Detlefs et al. [43] and was initially used in ESC [43] and ESC Java [84]. The basic idea is to consider only the first iterations (zero, one or more) of the loop. Obviously this makes the technique unsound, but Detlefs et al. [43] state the following:

> This apparently crude technique is remarkably effective in practice. Of course, it is not sound. In fact, it is a good example of the wonderful liberation we get by dropping the shackles of soundness.

Assume that we have a source programming language with a loop constructor (for instance, **GComm** extended with the while constructor) of the form **while** b **do** C **od**. We can capture all the executions that iterate at most once with the guarded command **assume** $\neg b$ $\square$ (**assume** b; C; **assume** $\neg b$). The left-hand side of the first choice command captures executions that do not iterate at all. The right-hand side captures executions that iterate exactly once. For excluding executions that iterate more than once the **assume** $\neg b$ is used. Note that if the internal program C does contain inner loops the same transformation should be applied recursively. Naturally, this approach can be extended to take into account executions with

more iterations repeating the following pattern the desired number of times: $\textbf{assume}\, \neg b \;\square\; (\textbf{assume}\, b;\; C;\; (\textbf{assume}\, \neg b \;\square\; (\textbf{assume}\, b;\; C;\; (\textbf{assume}\, \neg\theta \;\square\; \ldots))))$.

If invariants are annotated in the program the same approach can be used to give some evidence of their validity, i.e. to show that the invariant holds in the first few iterations. Consider the annotated loop **while** b **do** $\{\theta\}$ C **od**. Then the following program can be obtained: $\textbf{assert}\, \theta;\; (\textbf{assume}\, \neg b \;\square\; (\textbf{assume}\, b;\; C;\; \textbf{assert}\, \theta;\; \textbf{assume}\, \neg b))$ that will take into account executions that iterate zero or one time. As before the same approach can be extended for considering more iterations.

If invariants are annotated in the program, one will in principle prefer a sound mechanism to remove loops. Remember that an invariant must be true at the beginning of the loop (invariant initialisation) and when the loop body is executed in a state where the invariant is true then if it terminates, it should be in a state where the invariant is kept valid. Consider that we have a loop **while** b **do** $\{\theta\}$ C **od** and, for simplification purposes, that C does not contain nested loops. Then we can capture its behavior as follows:

$$
\begin{array}{l}
\textbf{assert}\, \theta; \\
x_1 := y_1;\; \ldots;\; x_n := y_n; \\
\textbf{assume}\, \theta; \\
(\textbf{assume}\, b;\; C;\; \textbf{assert}\, \theta;\; \textbf{assume}\, \bot) \\
\quad \square\; \textbf{assume}\, \neg b
\end{array}
$$

The first assert is used to ensure the loop invariant initialisation. After that every variable that is assigned in the loop body is assigned with a non-deterministic value (here we assume that variables $x_1, \ldots, x_n$ are assigned in the loop body and that $y_1, \ldots, y_n$ have some non deterministic value). Then, the possible executions in the transformed program are restricted to those that satisfy the loop invariant (this is done with the $\textbf{assume}\, \theta$). W.r.t. the choice command, the left hand side corresponds to a loop iteration and thus it is assumed that the loop condition holds and an assert statement is inserted at the end to ensure the invariant preservation; an $\textbf{assume}\, \bot$ is inserted after the assert because the execution of the program cannot continue from a random iteration (possibly not final). The other side of the choice command corresponds to the loop termination, and therefore the negation of the loop condition is placed in an assume statement: in the rest of the program it is possible to rely on the conditions θ and $\neg b$. This is the approach followed by most tools based on deductive verification using verification condition generation [43, 84, 83, 50].

Program Logics and the Dijkstra's Predicate Transformers. The foundations of deductive program verification have traditionally lied in two different frameworks: Dijkstra's predicate transformers and program logics. Guarded commands have been used as an intermediate language in tools like ESC/Java [84] and more recently the Boogie generic verifier. Many pragmatic aspects of program verification have been addressed and described in this setting, in particular the generation of efficiently provable verification conditions and the treatment of

unstructured programs [11]. The program logic tradition on the other hand is based on separate operational and axiomatic semantics of programming languages, which has allowed for the study of properties like soundness and (relative) completeness of Hoare logic with respect to the standard semantics of a While language [6], an approach that has been extended with the treatment of pointers and aliasing in separation logic [97].

2.5 Verification Condition Generator

Let us now go back to System Hg from Section 2.3. It is possible to write an algorithm, known as a *verification conditions generator*, that simply collects the side conditions of a derivation without actually constructing it. Even though the system Hg is agnostic with respect to a strategy for propagating assertions, the VCGen will necessarily impose one such strategy. What we mean here is that in the (seq) rule of system Hg nothing is said about the assertion θ. That assertion can be calculated as a postcondition of C_1 w.r.t. the precondition ϕ and then propagated forward to the precondition of C_2, or the other way around: as a precondition of C_2 w.r.t. the postcondition ψ and then propagated back to the postcondition of C_1. This is where the predicate transformers introduced by Dijkstra and described in the previous section come in handy. Let us see how the weakest precondition predicate transformer can be written for **AComm**.

Definition 2.14. *The weakest precondition approximation function* $\mathsf{wp} : \mathbf{AComm} \times \mathbf{Assert} \to \mathbf{Assert}$ *is defined as follows:*

$$\begin{array}{rcl}
\mathsf{wp}(\mathbf{skip}, \psi) & = & \psi \\
\mathsf{wp}(x := e, \psi) & = & \psi[e/x] \\
\mathsf{wp}(C_1; C_2, \psi) & = & \mathsf{wp}(C_1, \mathsf{wp}(C_2, \psi)) \\
\mathsf{wp}(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, \psi) & = & (b \to \mathsf{wp}(C_1, \psi)) \wedge (\neg b \to \mathsf{wp}(C_2, \psi)) \\
\mathsf{wp}(\mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \mathbf{od}, \psi) & = & \theta
\end{array}$$

The function wp calculates an approximation of the weakest precondition required for a command to satisfy ψ if it terminates. It is now possible to use wp to generate verification conditions in a mechanized way, following the structure of system Hg.

$$\begin{array}{rcl}
\mathsf{VC}_{\mathsf{wp}}(\phi, \mathbf{skip}, \psi) & = & \{\phi \to \psi\} \\
\mathsf{VC}_{\mathsf{wp}}(\phi, x := e, \psi) & = & \{\phi \to \psi[e/x]\} \\
\mathsf{VC}_{\mathsf{wp}}(\phi, C_1\, ;\, C_2, \psi) & = & \mathsf{VC}_{\mathsf{wp}}(\phi, C_1, \mathsf{wp}(C_2, \psi)) \cup \mathsf{VC}_{\mathsf{wp}}(\mathsf{wp}(C_2, \psi), C_2, \psi) \\
\mathsf{VC}_{\mathsf{wp}}(\phi, \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, \psi) & = & \mathsf{VC}_{\mathsf{wp}}(\phi \wedge b, C_1, \psi) \cup \mathsf{VC}_{\mathsf{wp}}(\phi \wedge \neg b, C_2, \psi) \\
\mathsf{VC}_{\mathsf{wp}}(\phi, \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \mathbf{od}, \psi) & = & \{\phi \to \theta, \theta \wedge \neg b \to \psi\} \cup\ \mathsf{VC}_{\mathsf{wp}}(\theta \wedge b, C, \theta)
\end{array}$$

The suggested VCGen creates many tautologies of the form $\phi \to \phi$. What is more, the precondition of the Hoare triple that we want to validate is only used once. For instance, for

the triple $\{\phi\}\, C\, \{\psi\}$, the condition ϕ will be used only once. As a matter of fact, if we consider the additional verification condition $\phi \to \mathsf{wp}(C, \psi)$ it is possible to create a VCGen based on a function that only takes into account the program and the postcondition, and will avoid the tautologies described above.

Definition 2.15 (Backward-propagation VCGen)**.** *The verification conditions generator function* $\mathsf{VC_{wp}} : \textbf{Assert} \times \textbf{AComm} \times \textbf{Assert} \to \mathcal{P}(\textbf{Assert})$ *and* $\mathsf{VC^{aux}_{wp}} : \textbf{AComm} \times \textbf{Assert} \to \mathcal{P}(\textbf{Assert})$ *are defined as follows:*

$$\begin{array}{rcl}
\mathsf{VC^{aux}_{wp}}(\textbf{skip}, \psi) & = & \emptyset \\
\mathsf{VC^{aux}_{wp}}(x := e, \psi) & = & \emptyset \\
\mathsf{VC^{aux}_{wp}}(C_1\,;\, C_2, \psi) & = & \mathsf{VC^{aux}_{wp}}(C_1, \mathsf{wp}(C_2, \psi)) \cup \mathsf{VC^{aux}_{wp}}(C_2, \psi) \\
\mathsf{VC^{aux}_{wp}}(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}, \psi) & = & \mathsf{VC^{aux}_{wp}}(C_1, \psi) \cup \mathsf{VC^{aux}_{wp}}(C_2, \psi) \\
\mathsf{VC^{aux}_{wp}}(\textbf{while } b \textbf{ do } \{\theta\}\ C \textbf{ od}, \psi) & = & \{\theta \wedge \neg b \to \psi, \theta \wedge b \to \mathsf{wp}(C, \theta)\} \cup\ \mathsf{VC^{aux}_{wp}}(C, \theta)
\end{array}$$

$$\mathsf{VC_{wp}}(\phi, C, \psi) = \{\phi \to \mathsf{wp}(C, \psi)\} \cup \mathsf{VC^{aux}_{wp}}(C, \psi)$$

The VCGen described in Definition 2.15 is sound, and for correctly-annotated programs it is complete. From now on, when referring to the functions $\mathsf{VC_{wp}}$, we intend to refer to the function from the previous definition, and not the one initially introduced.

Proposition 2.9 (Soundness and completeness of $\mathsf{VC_{wp}}$)**.** *Let* $C \in \textbf{AComm}$ *and* $\phi, \psi \in \textbf{Assert}$*. Then:*

1. $\models \mathsf{VC_{wp}}(\phi, C, \psi)$ *iff* $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$.
2. *If* $\models \mathsf{VC_{wp}}(\phi, C, \psi)$ *then* $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$.
3. *If* C *is correctly-annotated with respect to* (ϕ, ψ) *and* $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$ *then* $\models \mathsf{VC_{wp}}(\phi, C, \psi)$.

Proof. 1. By induction on the structure of C. 2. Follows directly from 1., and Proposition 2.5. 3. Follows directly from 1., Definition 2.9, and Proposition 2.3. □

If instead of using a backward propagation strategy, one wants to use a forward propagation strategy, then the strongest postcondition predicate transformer should be used.

Definition 2.16. *The strongest postcondition approximation function* $\mathsf{sp} : \textbf{Assert} \times \textbf{AComm} \to \textbf{Assert}$ *is defined as follows:*

$$\begin{array}{rcl}
\mathsf{sp}(\phi, \textbf{skip}) & = & \phi \\
\mathsf{sp}(\phi, x := e) & = & \exists\, x'.\ x = e[x'/x] \wedge \phi[x'/x] \\
\mathsf{sp}(\phi, C_1; C_2) & = & \mathsf{sp}(\mathsf{sp}(\phi, C_1), C_2) \\
\mathsf{sp}(\phi, \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}) & = & \mathsf{sp}(\phi \wedge b, C_1) \vee \mathsf{sp}(\phi \wedge \neg b, C_2) \\
\mathsf{sp}(\phi, \textbf{while } b \textbf{ do } \{\theta\}\ C \textbf{ od}) & = & \theta \wedge \neg b
\end{array}$$

Similarly to $\mathsf{VC_{wp}}$ it is possible to create a VCGen based on forward propagation.

Definition 2.17 (Forward-propagation VCGen)**.** *The verification conditions generator function* $\mathsf{VC_{sp}} : \textbf{Assert} \times \textbf{AComm} \times \textbf{Assert} \rightarrow \mathcal{P}(\textbf{Assert})$ *and the auxiliary function* $\mathsf{VC^{aux}_{sp}} : \textbf{Assert} \times \textbf{AComm} \rightarrow \mathcal{P}(\textbf{Assert})$ *are defined as follows:*

$$\begin{array}{rcl}
\mathsf{VC^{aux}_{sp}}(\phi, \textbf{skip}) & = & \emptyset \\
\mathsf{VC^{aux}_{sp}}(\phi, x := e) & = & \emptyset \\
\mathsf{VC^{aux}_{sp}}(\phi, C_1 \,;\, C_2) & = & \mathsf{VC^{aux}_{sp}}(\phi, C_1) \cup \mathsf{VC^{aux}_{sp}}(\mathsf{sp}(\phi, C_1), C_2) \\
\mathsf{VC^{aux}_{sp}}(\phi, \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}) & = & \mathsf{VC^{aux}_{sp}}(\phi \wedge b, C_1) \cup \mathsf{VC^{aux}_{sp}}(\phi \wedge \neg b, C_2) \\
\mathsf{VC^{aux}_{sp}}(\phi, \textbf{while } b \textbf{ do } \{\theta\}\ C \textbf{ od}) & = & \{\phi \rightarrow \theta, \mathsf{sp}(\theta \wedge b, C) \rightarrow \theta\} \cup \ \mathsf{VC^{aux}_{sp}}(\theta \wedge b, C)
\end{array}$$

$$\mathsf{VC_{sp}}(\phi, C, \psi) \ = \ \mathsf{VC^{aux}_{sp}}(\phi, C) \cup \{\mathsf{sp}(\phi, C) \rightarrow \psi\}$$

The following proposition establishes the equivalence of the VCGens $\mathsf{VC_{wp}}$ and $\mathsf{VC_{sp}}$. This means that either one of these VCGens can be used to generate VCs and that $\mathsf{VC_{sp}}$ is sound and complete.

Proposition 2.10. *Let* $C \in \textbf{AComm}$ *and* $\phi, \psi \in \textbf{Assert}$*. Then* $\models \mathsf{VC_{wp}}(\phi, C, \psi)$ *iff* $\models \mathsf{VC_{sp}}(\phi, C, \psi)$*.*

Proof. By structural induction on the program C. □

The observations made about the predicate size in the previous section can be expanded to the VCGens from this section. In particular, $\mathsf{VC_{wp}}$ and $\mathsf{VC_{sp}}$ can generate VCs of exponential size, and in order to circumvent such growth, programs can be transformed into passive form. We postpone further discussions about this topic to Chapter 4.

2.6 Bounded Model Checking of Software

Naturally this section could start by presenting the well established theoretical foundations of *model checking* [30, 9]. Nonetheless, instead of doing that, we will sate the basic principles of model checking and depart from there to the notion of *bounded model checking of software*: to understand the typical approach on what is called bounded model checking of software, one does not necessarily need to know in great depth all the details and intricacies of the model checking technique and its optimizations.

Model checking consists basically in checking automatically whether or not a given model satisfies a certain specification (basically a set of properties about the given model). The model is normally created using some adequate mathematical structures and the specification is given through a specific logic. Given the mathematical structure and a logic sentence (the specification), a model checking algorithm checks if that model satisfies the given property.

Initially a model would be described through a transition graph, with a number of concrete states and another number of concrete transitions between those states, and the specification would be described through temporal logic. An algorithm would then check whether or not the specification is valid in the model, and if not a counter-example would be returned. Such an approach allows for simple systems behaviors to be captured, but it is easy to imagine that as the complexity of the system evolves so does the number of states, which leads to an explosion of states for more complicated system behaviors. Approaches focusing on encoding states symbolically instead of encoding them concretely [23, 90], and also on using Binary Decision diagrams (BDDs) were proposed [22], nonetheless they only alleviate the problem and do not solve it.

Currently the main goal in model checking (and this also applies to model checking of software) is to control the state explosion. Two techniques are normally used for this [71]: one is based on *abstraction techniques* [29], and the other is based on *bounded model checking* [19]. In model checking using abstraction techniques, the model only captures an abstract description of the real system design. Therefore, these techniques are sound, but not complete: if the model is shown to be correct w.r.t. a specification, then the respective system is correct w.r.t. that specification. Nonetheless, if the specification is violated in the model, then it is not granted that the specification is actually violated in the system. So, when a counter-example is reported by the model checking algorithm, that counter-example must be confronted with the concrete system to check whether or not it is a real counter-example. If it is a false counter-example then the model can in principle be refined.

On the other hand, bounded model checking only captures bounded behaviors of the system. The idea is to look for counter-examples in paths of at most length k. If a counter-example is not found, then k can be incremented until a counter-example is found or the problem becomes intractable. The bounded model checking technique is complete but not sound, in the sense that if the system is correct w.r.t. a specification then the technique will return no counter-example, but if the technique reports that the model is correct it does not mean that the concrete system respects the specification for paths bigger than k. Nonetheless, since some systems have naturally bounded behaviors, it is possible in those cases to conclude that the system is correct w.r.t. a certain specification. A great advantage of bounded model checking is that the problem can be reduced to a satisfiability problem [28, 20], and therefore fed to a SAT solver for validity checking.

When it comes to bounded model checking of software the idea is fairly simple. The bounded encoding of the system comes from unwinding loops and recursive functions calls a certain number of times, let us say k. The specification is normally annotated in the code through *assume* and *assert* commands similar (but not exactly with the same meaning) to those adopted in the context of predicate transformers. This approach allows one to conclude that executions not requiring more than k iterations (resp. recursive calls) have indeed the expected behavior, in the sense that they do not violate any assertion in the code. As expected, for executions requiring more than k iterations or recursive calls nothing is said. The main distinction between

bounded model checking of software and the approach described in Section 2.4 for removing loops (based on the use of assumes, asserts, and non-deterministic values) refers to how the program is transformed before logical formulas are generated. Below we present the typical workflow followed by bounded model checking tools such as CBMC [32, 31] and ESBMC [37] and SMT-CBMC [8].

Since the theoretical foundations of bounded model checking of software come from the model checking area, and this falls outside the context of this thesis, in the rest of this section we will use a less formal approach to describe programming languages and the transformations applied to programs. We will extend the language introduced in Section 2.1 with a **nondet** expression (an expression that returns a non deterministic value), and support for arrays: the expression $a[i]$ is used to get the element in index i from array a; the assignment instruction $a[i] := x$ assigns x to the index i in array a.

Example 2.18. *Consider the following program and assume that the variable l contains the length of the array a. The program calculates the index of an occurrence of the maximum element in the array, and saves it into the variable max.*

$$
\begin{array}{l}
\textbf{assume}\, l > 0\,; \\
max := 0\,; \\
i := 1\,; \\
\textbf{while}\; i < l\; \textbf{do} \\
\quad \textbf{if}\; a[i] > a[max]\; \textbf{then}\; max := i\; \textbf{else skip fi}\,; \\
\quad i := i + 1 \\
\;\textbf{od}\,; \\
ii := \textbf{nondet}\,; \\
\textbf{assume}\, ii >= 0 \wedge ii < l\,; \\
\textbf{assert}\, a[max] \geq a[ii]
\end{array}
$$

An assume is used in the beginning to restrict only executions in which the length of the array is greater than zero, which can be seen as a precondition. After the loop a new variable ii is assigned with a non deterministic value, and the assume that follows is intended to filter executions in which ii is a value within the bounds of the array a. The subsequent assert is intended to check that whatever the index ii is, the value $a[ii]$ is always small or equal to the value $a[max]$. This can be seen as the postcondition of the program, and intends to capture what would be a quantification over the indexes of the array.

The bounded model checking method follows normally a number of stages that start with the input program and terminate with a set of logical formulas to be fed to a logic solver. In what follows we will describe each one of these stages and show how the program from Example 2.18 is transformed into a set of logical formulas.

Bounding Programs. Naturally these tools start by simplifying the target program into the simplest form possible. Examples of this might include the removal of side effects in program expressions and the elimination of syntactic sugar commands. This process can be summarized by translating a high level programming language into an intermediate form that is suitable for analysis: it is easy to imagine that the fewer commands the language has, the fewer cases have to be dealt with. The next step consists in inserting safety properties through asserts, such as, for instance, properties to check for overflow and/or array out of bounds errors.

The workflow continues by unwinding loops k times and inlining functions calls (if a function is recursive, a bound is also applied during the inline process). For clarity purposes we assume here that every loop and recursive function call is unwound the same number of times. Nonetheless, tools have mechanisms for allowing different loops and recursive calls to be unwound a different number of times. The idea in loop unwinding is that the transformation

$$\textbf{while}\ b\ \textbf{do}\ C\ \textbf{od} \longrightarrow C\,;\ \textbf{if}\ b\ \textbf{then while}\ b\ \textbf{do}\ C\ \textbf{od else skip fi}$$

is applied k times and then *one* of the following transformations is applied:

1. $\textbf{while}\ b\ \textbf{do}\ C\ \textbf{od} \longrightarrow \textbf{assume}\,\neg b$.

2. $\textbf{while}\ b\ \textbf{do}\ C\ \textbf{od} \longrightarrow \textbf{assert}\,\neg b$.

The transformation 1. is used to exclude executions requiring more than k iterations from being taken into account by the bounded model. If the bounded model checking algorithm reports that the model is correct w.r.t. the specification then one must bear in mind that nothing is said about executions requiring more than k iterations, therefore, the concrete program may still contain erroneous executions. This is the reason why bounded model checkers of software are sometimes referred as a bug finders. On the other hand, the idea behind transformation 2. is to check if longer executions exist. If this is actually the case, the bounded model checker will report a counter-example, otherwise, when no counter-example exists the technique becomes sound, because it is guaranteed that loops will not iterate more than the number they were unwound.

Example 2.19. *Recall the program from Example 2.18. The result of unwinding its loop twice is shown below. In this case we introduced an* **assume** $\bot$*, so executions requiring more than two iterations are not taken into account. In practice this means that we are just taking into account executions where the length of the array is at most two. If one wants to check if executions requiring more than two iterations exists, the* **assume** $\bot$ *could be replaced by* **assert** $\bot$*. Note that the use of an* **assume** $\bot$ *(resp.* **assert** $\bot$*) at that point of the program is equivalent to use an* **assume** $\neg(i < l)$ *(resp.* **assert** $\neg(i < l)$*), because it appears inside a branching structure with the condition* $i < l$*.*

$\textbf{assume}\ l > 0\,;$

$max := 0\,;$

$i := 1$;
if $i < l$ **then**
  if $a[i] > a[max]$ **then** $max := i$ **else skip fi** ;
  $i := i + 1$;
  if $i < l$ **then**
    if $a[i] > a[max]$ **then** $max := i$ **else skip fi** ;
    $i := i + 1$;
    if $i < l$ **then assume** $\perp$ **else skip fi**
  else skip fi
else skip fi ;
$ii :=$ **nondet** ;
assume $ii >= 0 \wedge ii < l$;
assert $a[max] \geq a[ii]$

Towards a Satisfiability Problem. The first step towards obtaining a logical formula containing the model of the program is to convert the program into static single-assignment (SSA) form [40]. This form limits the syntactic occurrence of each variable as L-value of a single assignment instruction. A construct called 'Φ-function' is used to synchronize versions of the same variable used in different paths. For instance the fragment **if** $x > 0$ **then** $x := x + 10$ **else** $x := x + 20$ **fi** could be translated as **if** $x_0 > 0$ **then** $x_1 := x_0 + 10$ **else** $x_2 := x_0 + 20$ **fi** ; $x_3 := \Phi(x_1, x_2)$. This means that the value assigned to x_3 depends on whether execution has reached this point through the first or the second branch of the conditional. In bounded model checking of software, this is typically done using *conditional expressions* as found in the C programming language [77]. For instance the program above can be translated to the following SSA form: **if** $x_0 > 0$ **then** $x_1 := x_0 + 10$ **else** $x_2 := x_0 + 20$ **fi** ; $x_3 := x_0 > 0?x_1 : x_2$.

Example 2.20. *The program from Example 2.19 can be easily transformed into SSA form as below. Note that after each if command we introduce an assignments with a conditional expression on the right hand side to synchronize variables that are assigned inside the if body. Note also that the assignment with the* **nondet** *expression was also removed. The reason for this is that we assume that the 'version' zero of each variable contains some non deterministic value.*

assume $l_1 > 0$;
$max_1 := 0$;
$i_1 := 1$;
if $i_1 < l_1$ **then**
  if $a_1[i_1] > a_1[max_1]$ **then** $max_2 := i_1$ **else skip fi** ;

$max_3 := a_1[i_1] > a[max_1]\ ?\, max_2\ :\ max_1\,;$
$i_2 := i_1 + 1\,;$
if $i_2 < l_1$ **then**
  if $a_1[i_2] > a_1[max_3]$ **then** $max_4 := i_2$ **else skip fi** ;
  $max_5 := a_1[i_2] > a_1[max_3]\ ?\, max_4\ :\ max_3\,;$
  $i_3 := i_2 + 1\,;$
  if $i_3 < l_1$ **then assume** $\perp$ **else skip fi**
 else skip fi ;
$i_4 := i_2 < l_1\ ?\, i_3\ :\ i_2\,;$
$max_6 := i_2 < l_1\ ?\, max_5\ :\ max_3$
else skip fi ;
$i_5 := i_1 < l_1\ ?\, i_4\ :\ i_1\,;$
$max_7 := i_1 < l_1\ ?\, max_6\ :\ max_1\,;$
assume $ii_0 >= 0 \wedge ii_0 < l_1\,;$
assert $a_1[max_7] \geq a_1[ii_0]$

The resulting SSA program is then transformed into *Conditional Normal Form* (CNF)[2]. In this form programs are sequences of single-branch conditionals, each containing an atomic command. Conversion of programs to CNF involves a number of transformation steps that are sound for SSA programs. The resulting form consists in a sequence of commands of the form **if** b **then** C **else skip fi**, where C is an atomic command and b is the necessary condition for the command C to be executed. The following function implements precisely such a transformation:

$$
\begin{array}{rcl}
\mathsf{toCNF}(\pi, \mathbf{skip}) & = & \mathbf{if}\ \pi\ \mathbf{then\ skip\ fi} \\
\mathsf{toCNF}(\pi, x := e) & = & \mathbf{if}\ \pi\ \mathbf{then}\ x := e\ \mathbf{fi} \\
\mathsf{toCNF}(\pi, C_1\,;\, C_2) & = & \mathsf{toCNF}(\pi, C_1)\,;\,\mathsf{toCNF}(\pi, C_2) \\
\mathsf{toCNF}(\pi, \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \mathbf{fi}) & = & \mathsf{toCNF}(\pi \wedge b, C_t)\,;\,\mathsf{toCNF}(\pi \wedge \neg b, C_f) \\
\mathsf{toCNF}(\pi, \mathbf{assume}\,\theta) & = & \mathbf{if}\ \pi\ \mathbf{then\ assume}\,\theta\ \mathbf{fi} \\
\mathsf{toCNF}(\pi, \mathbf{assert}\,\theta) & = & \mathbf{if}\ \pi\ \mathbf{then\ assert}\,\theta\ \mathbf{fi}
\end{array}
$$

Example 2.21. *The CNF of the SSA program from Example 2.20 is as follows:*

if $\top$ **then assume** $l_1 > 0$ **fi** ;
if $\top$ **then** $max_1 := 0$ **fi** ;
if $\top$ **then** $i_1 := 1$ **fi** ;
if $i_1 < l_1 \wedge a_1[i_1] > a_1[max_1]$ **then** $max_2 := i_1$ **fi** ;
if $i_1 < l_1$ **then** $max_3 := a_1[i_1] > a[max_1]\ ?\, max_2\ :\ max_1$ **fi** ;

[2] In the rest of this thesis we will use the CNF acronym to refer to the Conditional Normal Form and never to the most common Conjunctive Normal Form.

$$
\begin{array}{l}
\textbf{if } i_1 < l_1 \textbf{ then } i_2 := i_1 + 1 \textbf{ fi}\,; \\
\textbf{if } i_1 < l_1 \wedge i_2 < l_1 \wedge a_1[i_2] > a_1[max_3] \textbf{ then } max_4 := i_2 \textbf{ fi}\,; \\
\textbf{if } i_1 < l_1 \wedge i_2 < l_1 \textbf{ then } max_5 := a_1[i_2] > a_1[max_3]\ ?\, max_4\ :\ max_3 \textbf{ fi}\,; \\
\textbf{if } i_1 < l_1 \wedge i_2 < l_1 \textbf{ then } i_3 := i_2 + 1 \textbf{ fi}\,; \\
\textbf{if } i_1 < l_1 \wedge i_2 < l_1 \wedge i_3 < l_1 \textbf{ then assume} \perp \textbf{fi} \\
\textbf{if } i_1 < l_1 \textbf{ then } i_4 := i_2 < l_1\ ?\, i_3\ :\ i_2 \textbf{ fi}\,; \\
\textbf{if } i_1 < l_1 \textbf{ then } max_6 := i_2 < l_1\ ?\, max_5\ :\ max_3 \textbf{ fi} \\
\textbf{if } \top \textbf{ then } i_5 := i_1 < l_1\ ?\, i_4\ :\ i_1 \textbf{ fi}\,; \\
\textbf{if } \top \textbf{ then } max_7 := i_1 < l_1\ ?\, max_6\ :\ max_1 \textbf{ fi}\,; \\
\textbf{if } \top \textbf{ then assume } ii_0 >= 0 \wedge ii_0 < l_1 \textbf{ fi}\,; \\
\textbf{if } \top \textbf{ then assert } a_1[max_7] \geq a_1[ii_0] \textbf{ fi}
\end{array}
$$

For a single-assignment program C, the encoding $\mathcal{C}$ of the operational behavior, and $\mathcal{P}$ of the properties to be verified can now be extracted using $(\mathcal{C}, \mathcal{P}) = \mathsf{cp}(\mathsf{toCNF}(\top, C))$, with cp the function defined below. We omit for now the definition for the assume statement and postpone it to Chapter 4.

$$
\begin{array}{rcl}
\mathsf{cp}(\textbf{if } b \textbf{ then skip fi}) & = & (\emptyset, \emptyset) \\
\mathsf{cp}(\textbf{if } b \textbf{ then } x := e \textbf{ fi}) & = & (\{b \to x = e\}, \emptyset) \\
\mathsf{cp}(C_1\,;\,C_2) & = & (\mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{P}_1 \cup \mathcal{P}_2), \\
 & & \text{where } (\mathcal{C}_1, \mathcal{P}_1) = \mathsf{cp}(C_1) \\
 & & \text{and } \quad (\mathcal{C}_2, \mathcal{P}_2) = \mathsf{cp}(C_2) \\
\mathsf{cp}(\textbf{if } b \textbf{ then assert } \theta \textbf{ fi}) & = & (\emptyset, \{b \to \theta\})
\end{array}
$$

With the two obtained sets of formulas, the satisfiability problem is now reduced to $\bigwedge \mathcal{C} \wedge \neg(\bigwedge \mathcal{P})$. If such a formula is satisfiable then any model that satisfies it will be a counter-example corresponding to the violation of a property. On the other hand, if the formula is UNSAT then the program is correct in the following sense: if unwinding assumes were used, the program is correct w.r.t. the annotated specification for executions not requiring more than k iterations; if unwinding assertions were used, the program is correct w.r.t. the specification, because there are no executions requiring more than k iterations.

2.7 More Preliminary Notation and Results

This section provides some additional notation, definitions and properties that will be used in different parts of this thesis.

First of all let us introduce the notation we will use for functions. A function f from A to B will be denoted by $f : A \to B$ when it is total (it is defined for all $x \in A$), and by $f : A \hookrightarrow B$ if it is partial. Given a function f, $\mathsf{dom}(f)$ will denote the domain of f, and $\mathsf{rng}(f)$ the codomain

of f. As usual, $f[x \mapsto a]$ is the function that maps x to a and any other value y to $f(y)$.

For the particular case of partial functions we will use the following notation. The empty function, that is, the function whose domain is the empty set will be represented by $[]$. The function whose domain is $\{x_1, \ldots, x_n\}$ and maps each x_i to a_i will be represented by $[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$. Let A be a set. Then $[x \mapsto g(x) \mid x \in A]$ will represent the function with domain A generated by g (known as the restriction of a function). In this case, g must be defined for all $x \in A$.

Finally with respect to functions, we introduce an operator to compose functions.

Definition 2.18. *Given the partial functions $f, g : A \hookrightarrow B$, the function represented as $f \oplus g : A \hookrightarrow B$ is defined as:*

$$(f \oplus g)(x) = \begin{cases} g(x) & \text{if } x \in \mathsf{dom}(g) \\ f(x) & \text{if } x \notin \mathsf{dom}(g) \text{ and } x \in \mathsf{dom}(f) \end{cases}$$

Note in particular that, if $f : A \to B$ then $f \oplus g : A \to B$.

It is possible to compose multiple functions and the following result establishes that in such a case, the operator is associative.

Lemma 2.11. *Let $f, g, h : A \hookrightarrow B$. Then $f \oplus (g \oplus h) = (f \oplus g) \oplus h$.*

Proof. Straightforward by expanding $(f \oplus (g \oplus h))(x)$ and $((f \oplus g) \oplus h)(x)$ using the previous definition, for some $x \in A$. □

The operator is not commutative for arbitrary functions, although, for some particular cases the operands can be swapped as indicated by the following lemma. Note in particular that for the first function to be total, the second has to be the empty partial function.

Lemma 2.12. *Let $f, g : A \hookrightarrow B$ such that $\mathsf{dom}(f) \cap \mathsf{dom}(g) = \emptyset$. Then $f \oplus g = g \oplus f$.*

Proof. Straightforward by expanding $(f \oplus g)(x)$ and $(g \oplus f)(x)$ using Definition 2.18, for some $x \in A$. □

Changing topics, let us now introduce a new class of programs whose commands are just sequences of assignments from variables to variables.

Definition 2.19 (Renaming)**.** *The set* **Rnm** $\subseteq$ **Comm** *of* renamings *consists of all programs of the form $\{x_1 := y_1 \,;\, \ldots \,;\, x_n := y_n\}$ such that all x_i and y_i are distinct. The empty renaming will be written as* **skip**.

A renaming $\mathcal{R} = \{x_1 := y_1 \,;\, \ldots \,;\, x_n := y_n\}$ represents also a finite bijection $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$, which we will also denote by $\mathcal{R}$. We will write $\mathsf{dom}(\mathcal{R})$ and $\mathsf{rng}(\mathcal{R})$ to denote the domain and range of $\mathcal{R}$, respectively. Furthermore, $\mathcal{R}(\phi)$ will denote the assertion that results

from applying the substitution $[y_1/x_1, \ldots, y_n/x_n]$ to ϕ. Also, for $s \in \Sigma$ we define the state $\mathcal{R}(s)$ as follows: $\mathcal{R}(s)(x) = s(\mathcal{R}(x))$ if $x \in \mathsf{dom}(\mathcal{R})$, and $\mathcal{R}(s)(x) = s(x)$ otherwise.

This class of programs has some interesting properties that can be described by the lemma below, which will be particularly useful when transforming programs into single-assignment form.

Lemma 2.13. *Let* $\mathcal{R} \in \mathbf{Rnm}$, $\phi, \psi \in \mathbf{Assert}$ *and* $s \in \Sigma$.

1. $\langle \mathcal{R}, s \rangle \leadsto \mathcal{R}(s)$.
2. $[\![\mathcal{R}(\phi)]\!](s) = [\![\phi]\!](\mathcal{R}(s))$.
3. $\models \{\phi\}\,\mathcal{R}\,\{\psi\}$ *iff* $\models \phi \to \mathcal{R}(\psi)$.

Proof. 1. By inspection on the evaluation relation. 2. By induction on the interpretation assertions. 3. Follows from 1 and 2.

□

The previous sections cover the background that is shared by forthcoming chapters of this thesis. Additional background will be provided as needed.

Chapter 3

Iterating SA Programs as a Basis for Program Verification

Translating programs into single-assignment (SA) form has been part of the standard compilation pipeline for decades now [40]. As we saw in Section 2.4, passive form (a form very similar to SA) has been used in the context of program verification to avoid exponential explosion when generating verification conditions. Verification tools typically opt for removing loops a priori which implies that the obtained program cannot be used for compilation purposes, and thus the program being verified and the program being compiled diverge.

In this chapter we formalize a verification technique for *While* programs annotated with invariants, based on their translation into an intermediate iterating single-assignment (ISA) form. Loops in ISA are annotated with invariants, have single-assignment bodies, and a renaming (see Section 2.7) allows for the values of the initial variables to be updated between iterations. An inference system that admits only derivations guided by the annotated loop invariants following a forward-propagation strategy is also provided, which serves as a basis for an algorithm that generates compact verification conditions (in the sense of Flanagan and Saxe [52]) for a given Hoare triple.

Instead of reasoning with a concrete translation, we identify the semantic requirements that are expected from such a translation and validate the workflow of the framework by showing that the generation of VCs from the ISA form is sound and complete for the verification of the initial program. The framework is completed by providing a concrete ISA translation and proving that it complies with the given semantic restrictions.

An important issue is that of *modular verification* and proof reuse. Ideally, one produces a separate proof of correctness for each occurrence (or call) of a subprogram C inside a program P, and then *adapts* the proved specification of C to different 'local' specifications. A formalism that always allows for this to be done is said to be *adaptation-complete* [75]; in its original formulation Hoare logic is not adaptation-complete. This is a problem in the presence of recursive procedures, since it leads to incompleteness of the program logic itself, but it is also a problem for the implementation of tools where the correctness of each procedure is proved once

and for all with respect to a *contract* that must be adapted to the local context of each call to it. We will show that adaptation-completeness is a natural property of reasoning in the single-assignment setting. An adaptation-complete variant of the logic will be proposed by adding to the inference system a dedicated consequence rule with a simple side condition. This new consequence rule is restricted to reasoning about triples in which the program does not assign any variable occurring free in the precondition; since the Hoare logic for ISA programs propagates preconditions forward in a way that preserves this property, the rule can be applied at any point in a derivation. It provides the highest degree of adaptation, without the need to check any additional complicated conditions or rules, as used to be the case in adaptation-complete presentations of Hoare logic [6, 4, 75].

The chapter is organized as follows: the next section introduces a language of iterating single-assignment programs, Section 3.2 proposes a Hoare logic and a VCGen for this language, and Section 3.3 considers the verification workflow based on the translation of annotated While programs into ISA form. In Section 3.4 we show how the program logic can be extended with a special consequence rule that makes it *adaptation-complete*, and in Section 3.4 we provide a concrete ISA translation together with the proof that it complies with the semantic restrictions. Finally Section 3.6 discusses related work.

3.1 Iterating Single-assignment Language

In what follows we introduce a language based on dynamic single-assignment (DSA) form that contains a loop construct. Such an approach allows us to identify properties about programs constructed over the proposed language, and to study from a formal perspective the use of SA in program verification.

It should be noted that in a strict sense it is not possible to write iterating programs in DSA form because the body of the loop is executed multiple times. So what we propose here is a syntactically controlled violation of the single-assignment constraints that allows for structured reasoning. Loop bodies are still SA blocks, but two renamings, responsible for propagating the values inside, outside and between iterations, are free of single-assignment restrictions. This will allow us to keep loop constructs in the language, while taking the benefits of single-assignment form.

Definition 3.1. *Let* $\mathbf{AComm}^{\text{ISA}}$ *be the class of* annotated single-assignment programs*. Its abstract syntax is defined by*

$$C \ ::= \ \mathbf{skip} \ | \ x := e \ | \ C\,;C \ | \ \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C\ \mathbf{fi} \ | \ \mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\ \{\theta\}\ C\ \mathbf{od}$$

where:

- $\mathbf{skip} \in \mathbf{AComm}^{\text{ISA}}$.
- $x := e \in \mathbf{AComm}^{\text{ISA}}$ *if* $x \notin \mathsf{Vars}(e)$.

- $C_1\,;\,C_2 \in \mathbf{AComm}^{\text{ISA}}$ *if* $C_1, C_2 \in \mathbf{AComm}^{\text{ISA}}$ *and* $\mathsf{Vars}(C_1) \cap \mathsf{Asgn}(C_2) = \emptyset$.
- $\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \mathbf{fi} \in \mathbf{AComm}^{\text{ISA}}$ *if* $C_t, C_f \in \mathbf{AComm}^{\text{ISA}}$ *and* $\mathsf{Vars}(b) \cap (\mathsf{Asgn}(C_t) \cup \mathsf{Asgn}(C_f)) = \emptyset$.
- $\mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\,\{\theta\}\, C\ \mathbf{od} \in \mathbf{AComm}^{\text{ISA}}$ *if* $C \in \mathbf{AComm}^{\text{ISA}}$, $\mathcal{I}, \mathcal{U} \in \mathbf{Rnm}$, $\mathsf{Asgn}(\mathcal{I}) = \mathsf{Asgn}(\mathcal{U})$, $\mathsf{rng}(\mathcal{U}) \subseteq \mathsf{Asgn}(C)$, *and* $(\mathsf{Vars}(\mathcal{I}) \cup \mathsf{Vars}(b) \cup \mathsf{FV}(\theta)) \cap \mathsf{Asgn}(C) = \emptyset$.

and Vars *and* Asgn *are extended to cope with the* **for** *command as follows:*

- $\mathsf{Vars}(\mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\,\{\theta\}\, C\ \mathbf{od}) = \mathsf{Vars}(\mathcal{I}) \cup \mathsf{Vars}(b) \cup \mathsf{FV}(\theta) \cup \mathsf{Vars}(C)$.
- $\mathsf{Asgn}(\mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\,\{\theta\}\, C\ \mathbf{od}) = \mathsf{Asgn}(\mathcal{I}) \cup \mathsf{Asgn}(C)$.

The previous definition imposes the typical single-assignment constraints, in particular that a variable is only assigned once, and after it has been used, it cannot be assigned. The commands of the language do not bring any additional novelty except for the case of loops. A **skip**, command is always an ISA command. An assignment command of the form $x := e$ is only an ISA command if the variable x does not occur in expression e. This restriction is required to ensure that a variable is not assigned after being used. A similar restriction is applied to the sequence command of the form $C_1\,;\,C_2$ imposing that the latter command of the sequence C_2 cannot assign variables that occur in the former command C_1. Also in the branching command of the form $\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}$ the commands C_1 and C_2 cannot assign variables that occur in b. Loops have the form of $\mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\,\{\theta\}\, C\ \mathbf{od}$, where $\mathcal{I}$ and $\mathcal{U}$ are renamings (see Section 2.7), b is a boolean condition, θ is the loop invariant and C is the loop body. The initialization code $\mathcal{I}$ contains a renaming that runs exactly once, even if no iterations take place. On the other hand the code in $\mathcal{U}$ is executed after every iteration. This ensures that the variables in $\mathsf{dom}(\mathcal{U})$ (which are the same as $\mathsf{dom}(\mathcal{I})$) always contain the appropriate output values at the beginning of each iteration and when the loop terminates. Note that the definition of $\phi\#C$, initially introduced in Definition 2.6 and stating that C does not assign free variables of ϕ, extends to annotated programs as expected.

To understand the use of the for construct in our ISA language, let us first write a program in a form where blocks of code consisting in a sequence of assignments are converted to SA form. The following example shows a program where the assignments before and inside the loop were converted to SA form (the variables occurring in the loop are signaled with an 'a' subscript for clarity, but any other fresh variables would do).

Example 3.1. *The program below represents the program from Example 2.13 with the blocks converted to SA form.*

$f_1 := 1\,;$
$i_1 := 1\,;$
$\mathcal{I}$
$\mathbf{while}\ (i_{a0} \leq n)\ \mathbf{do}\ \{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1\}$

$f_{a1} := f_{a0} * i_{a0}\,;$
$i_{a1} := i_{a0} + 1\,;$
$\mathcal{U}$
od

The initial version variables of the loop body f_{a0} and i_{a0} are the ones used in the Boolean expression, which is evaluated at the beginning of each iteration. They are also used in the invariant annotation. We have placed in the code the required renamings $\mathcal{I}$ and $\mathcal{U}$, and it is straightforward to instantiate them. The renaming $\mathcal{I}$ should be $\{i_{a0} := i_1\,;\, f_{a0} := f_1\}$, and $\mathcal{U}$ should be $\{i_{a0} := i_{a1}\,;\, f_{a0} := f_{a1}\}$. Note that without $\mathcal{U}$ the new values of the counter and of the accumulator would not be transported to the next iteration. The initial version variables can be used after the loop to access the value of the counter and accumulator. A specification for this program can be as follows: $(n \geq 0 \wedge n_{aux} = n, f_{a0} = n_{aux}!)$.

It is now immediate to write the program with a *for* command encapsulating the structure of the loop, in accordance with Definition 3.1. This is shown in the next example. Incidentally, note that the invariant does not contain the 'continuous' part $n_{aux} = n$ of the initial code, since it becomes unnecessary in the ISA version.

Example 3.2. *The program initially presented in Example 2.13, and then converted into a pseudo SA in Example 3.1, can now be written using only commands from* **AComm**$^{\text{ISA}}$*. In the rest of this chapter, we will refer to this annotated ISA program as* $\mathsf{Fact}^{\mathsf{isa}}$*.*

$f_1 := 1\,;$
$i_1 := 1\,;$
for $(\{i_{a0} := i_1\,;\, f_{a0} := f_1\}, i_{a0} \leq n, \{i_{a0} := i_{a1}\,;\, f_{a0} := f_{a1}\})$ **do**
$\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1\}$
$f_{a1} := f_{a0} * i_{a0}\,;$
$i_{a1} := i_{a0} + 1$
od

To be able to use the operational and axiomatic semantics introduced in the previous chapter to reason about **AComm**$^{\text{ISA}}$ programs without modifications, we will consider a function that transforms **AComm**$^{\text{ISA}}$ into **AComm**, by simply inlining in the assignments I and U in the appropriate places.

Definition 3.2. *The function* $\mathcal{W}$: **AComm**$^{\text{ISA}}$ $\rightarrow$ **AComm** *translates ISA programs to (an-*

(skip) $$\overline{\{\phi\}\,\mathbf{skip}\,\{\phi \wedge \top\}}$$

(assign) $$\overline{\{\phi\}\,x := e\,\{\phi \wedge x = e\}}$$

(seq) $$\frac{\{\phi\}\,C_1\,\{\phi \wedge \psi_1\} \qquad \{\phi \wedge \psi_1\}\,C_2\,\{\phi \wedge \psi_1 \wedge \psi_2\}}{\{\phi\}\,C_1\,;\,C_2\,\{\phi \wedge \psi_1 \wedge \psi_2\}}$$

(if) $$\frac{\{\phi \wedge b\}\,C_t\,\{\phi \wedge b \wedge \psi_t\} \qquad \{\phi \wedge \neg b\}\,C_f\,\{\phi \wedge \neg b \wedge \psi_f\}}{\{\phi\}\,\mathbf{if}\;b\;\mathbf{then}\;C_t\;\mathbf{else}\;C_f\;\mathbf{fi}\,\{\phi \wedge ((b \wedge \psi_t) \vee (\neg b \wedge \psi_f))\}}$$

(for) $$\frac{\{\theta \wedge b\}\,C\,\{\theta \wedge b \wedge \psi\}}{\{\phi\}\,\mathbf{for}\;(\mathcal{I}, b, \mathcal{U})\;\mathbf{do}\,\{\theta\}\,C\;\mathbf{od}\,\{\phi \wedge \theta \wedge \neg b\}} \quad \text{if} \quad \begin{array}{l} \phi \rightarrow \mathcal{I}(\theta) \quad \text{and} \\ \theta \wedge b \wedge \psi \rightarrow \mathcal{U}(\theta) \end{array}$$

Figure 3.1: Inference system for annotated ISA triples – System Hisa

notated) While programs as follows:

$$\begin{array}{rcl}
\mathcal{W}(\mathbf{skip}) & = & \mathbf{skip} \\
\mathcal{W}(x := e) & = & x := e \\
\mathcal{W}(C_1\,;\,C_2) & = & \mathcal{W}(C_1)\,;\,\mathcal{W}(C_2) \\
\mathcal{W}(\mathbf{if}\;b\;\mathbf{then}\;C_t\;\mathbf{else}\;C_f\;\mathbf{fi}) & = & \mathbf{if}\;b\;\mathbf{then}\;\mathcal{W}(C_t)\;\mathbf{else}\;\mathcal{W}(C_f)\;\mathbf{fi} \\
\mathcal{W}(\mathbf{for}\;(\mathcal{I}, b, \mathcal{U})\;\mathbf{do}\,\{\theta\}\,C\;\mathbf{od}) & = & \mathcal{I}\,;\,\mathbf{while}\;b\;\mathbf{do}\;\{\theta\}\;\mathcal{W}(C)\,;\,\mathcal{U}\;\mathbf{od}
\end{array}$$

Whenever we need to observe the execution of $C \in \mathbf{AComm}^{\mathrm{ISA}}$ using the small or big-step semantic introduced in the previous chapter, we use $\mathcal{W}(C)$ to convert C into **AComm** and then $\lfloor \mathcal{W}(C) \rfloor$ to convert it into **Comm**. The same method will be used for checking the derivability of triples in system H. For system Hg it is enough to apply $\mathcal{W}$. The next section introduces a new inference system and VCGen intended for $\mathbf{AComm}^{\mathrm{ISA}}$.

3.2 Hoare Logic and Verification Conditions for Iterating SA Programs

Regardless of the strategy used (forward or backward propagation), derivations in system Hg (and similarly in H) may produce formulas (and thus side conditions) of exponential size in the length n of the program, similarly to the VCGens described in Section 2.4. It suffices to consider

a program containing 2^n execution paths, for instance a sequence of n conditional statements. If a forward (resp. backward) propagation strategy is used to construct a derivation for this program, each instance of the (if) rule will have as conclusion a Hoare triple whose postcondition (resp. precondition) contains two occurrences of the precondition (resp. postcondition). Repeating this process throughout the sequence will generate the exponential pattern.

We propose in Figure 3.1 an inference system for Hoare triples containing annotated ISA programs that avoids such an explosion. The system Hisa is a goal-directed like system Hg but it incorporates a strategy, based on forward propagation (reminiscent of strongest postcondition computations). Note that Hisa derives triples of the form $\{\phi\}\, C\, \{\phi \wedge \psi\}$, where the program does not interfere with the truth of the precondition. For this reason we restrict our results to triples satisfying the $\phi\#C$ condition (ISA translations should generate triples that satisfy this restriction).

Definition 3.3 (ISA triple)**.** *Let $C \in \mathbf{AComm}^{\text{ISA}}$ and $\phi, \psi \in \mathbf{Assert}$. A Hoare triple $\{\phi\}\, C\, \{\psi\}$ is said to be an ISA triple if $\phi\#C$.*

Some properties that will be useful in the rest of this chapter can be extracted from triples that are derived in system Hisa. In particular the free variables from the postcondition come either from the precondition or are variables occurring in the program. Moreover, every triple occurring in the derivation is an ISA triple.

Lemma 3.1. *Let $C \in \mathbf{AComm}^{\text{ISA}}$ and $\phi, \psi \in \mathbf{Assert}$ such that $\phi\#C$, $\vdash_{\mathsf{Hisa}} \{\phi\}\, C\, \{\psi\}$. Then:*

1. *$\mathsf{FV}(\psi) \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$.*
2. *all triples $\{\alpha\}\, C'\, \{\beta\}$ occurring in this derivation satisfy $\alpha\#C'$.*

Proof. Both are proved by induction on the derivation of $\vdash_{\mathsf{Hisa}} \{\phi\}\, C\, \{\psi\}$. □

The system can now be proved to be sound and complete. We show that it is sound w.r.t. system H from Section 2.2, and complete w.r.t. Hg from Section 2.3.

Proposition 3.2 (Soundness of system Hisa)**.** *Let $C \in \mathbf{AComm}^{\text{ISA}}$ and ϕ, $\psi' \in \mathbf{Assert}$ such that $\phi\#C$. If $\vdash_{\mathsf{Hisa}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$, then $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\phi \wedge \psi'\}$.*

Proof. By induction on the derivation of $\vdash_{\mathsf{Hisa}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$, using Lemma 3.1. We show the interesting case, where the last rule applied is (for) – all the other are straightforward. Assume the last step is

$$\frac{\{\theta \wedge b\}\, C\, \{\theta \wedge b \wedge \psi\}}{\{\phi\}\, \mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\, \{\theta\}\, C\ \mathbf{od}\, \{\phi \wedge \theta \wedge \neg b\}} \quad \text{with} \quad \begin{array}{l} \phi \rightarrow \mathcal{I}(\theta) \ \text{ and} \\ \theta \wedge b \wedge \psi \rightarrow \mathcal{U}(\theta) \end{array}$$

By Lemma 3.1, we have that $(\theta \wedge b)\#C$. So, by IH, we have $\vdash_{\mathsf{H}} \{\theta \wedge b\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\theta \wedge b \wedge \psi\}$. From the validity of the side conditions, by Lemma 2.13 and completeness of H, we have $\vdash_{\mathsf{H}} \{\theta \wedge b \wedge \psi\} \mathcal{U}\, \{\theta\}$ and $\vdash_{\mathsf{H}} \{\phi\}\, \mathcal{I}\, \{\theta\}$. Now applying sequentially the rules (seq), (while) and again (seq), we get $\vdash_{\mathsf{H}} \{\phi\}\, \mathcal{I}\, ;\, \mathbf{while}\ b\ \mathbf{do}\ \lfloor \mathcal{W}(C) \rfloor\, ;\, \mathcal{U}\ \mathbf{od}\, \{\theta \wedge \neg b\}$. Hence, by definition of $\mathcal{W}$ and Lemma 2.4, we have $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor \mathcal{W}(\mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\, \{\theta\}\, C\ \mathbf{od}) \rfloor\, \{\phi \wedge \theta \wedge \neg b\}$. □

Proposition 3.3 (Completeness of system Hisa). *Let $C \in \mathbf{AComm}^{\text{ISA}}$ and ϕ, $\psi \in \mathbf{Assert}$ such that $\phi \# C$ and $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathcal{W}(C)\, \{\psi\}$. Then $\vdash_{\mathsf{Hisa}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$ for some $\psi' \in \mathbf{Assert}$ such that $\models \phi \wedge \psi' \to \psi$.*

Proof. By induction on the structure of C. Assume $\phi \# C$ and $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathcal{W}(C)\, \{\psi\}$.

- Case $C \equiv x := e$, we must have $\models \phi \to \psi[e/x]$. Since $x \notin (\mathsf{FV}(e) \cup \mathsf{FV}(\phi))$, it follows that $\models \phi \wedge x = e \to \psi$. As $\vdash_{\mathsf{Hisa}} \{\phi\}\, x := e\, \{\phi \wedge x = e\}$ we are done.

- Case $C \equiv C_1; C_2$, we must have for some $\gamma \in \mathbf{Assert}$ $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathcal{W}(C_1)\, \{\gamma\}$ and $\vdash_{\mathsf{Hg}} \{\gamma\}\, \mathcal{W}(C_2)\, \{\psi\}$. Since $\phi \# C_1; C_2$ we have $\phi \# C_1$. Hence by IH we have $\vdash_{\mathsf{Hisa}} \{\phi\}\, C_1\, \{\phi \wedge \gamma'\}$ for some $\gamma' \in \mathbf{Assert}$ such that $\models \phi \wedge \gamma' \to \gamma$. Therefore, by Lemma 2.6, $\vdash_{\mathsf{Hg}} \{\phi \wedge \gamma'\}\, \mathcal{W}(C_2)\, \{\psi\}$. From Lemma 3.1 we have that $\mathsf{FV}(\phi \wedge \gamma') \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C_1)$, and thus $(\phi \wedge \gamma') \# C_2$. Hence by IH $\vdash_{\mathsf{Hisa}} \{\phi \wedge \gamma'\}\, C_2\, \{\phi \wedge \gamma' \wedge \psi'\}$ for some $\psi' \in \mathbf{Assert}$ such that $\models \phi \wedge \gamma' \wedge \psi' \to \psi$. Applying rule (seq) we then get $\vdash_{\mathsf{Hisa}} \{\phi\}\, C_1\, ;\, C_2\, \{\phi \wedge \gamma' \wedge \psi'\}$.

- Case $C \equiv \mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\, \{\theta\}\, C_t\ \mathbf{od}$, we must have, for some $\gamma \in \mathbf{Assert}$, that $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathcal{I}\, \{\theta\}$, $\vdash_{\mathsf{Hg}} \{\theta \wedge b\}\, \mathcal{W}(C_t)\, \{\gamma\}$, $\vdash_{\mathsf{Hg}} \{\gamma\}\, \mathcal{U}\, \{\theta\}$, and $\models \theta \wedge \neg b \to \psi$. We have that $(\theta \wedge b) \# C_t$, so it follows by IH that $\vdash_{\mathsf{Hisa}} \{\theta \wedge b\}\, C_t\, \{\theta \wedge b \wedge \gamma'\}$, for some $\gamma' \in \mathbf{Assert}$ and $\models \theta \wedge b \wedge \gamma' \to \gamma$. Therefore, by Lemma 2.6, $\vdash_{\mathsf{Hg}} \{\theta \wedge b \wedge \gamma'\}\, \mathcal{U}\, \{\theta\}$. Since Hg is sound, by Lemma 2.13, it follows that $\models \phi \to \mathcal{I}(\theta)$ and $\models \theta \wedge b \wedge \gamma' \to \mathcal{U}(\theta)$, which allow us to apply rule (for) and get the conclusion $\vdash_{\mathsf{Hisa}} \{\phi\}\, \mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\, \{\theta\}\, C\ \mathbf{od}\, \{\phi \wedge \theta \wedge \neg b\}$.

The remaining cases are routine. □

All the rules of system Hisa propagate the precondition ϕ forward. Note that in the (for) rule this happens even though ϕ is not implied by the annotated loop invariant. Observe also how in this same rule we reason structurally about the body of the loop (an ISA piece of code), with the renamings applied to the invariant in the side conditions.

Example 3.3. *Let $\mathsf{Fact}^{\mathsf{isa}}$ be the factorial single-assignment program from Example 3.2. The derivation of the triple $\{n \geq 0 \wedge n_{aux} = n\}\, \mathsf{Fact}^{\mathsf{isa}}\, \{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1 \wedge f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge \neg(i_{a0} \leq n)\}$ in system Hisa is shown below. In particular, it is possible to conclude that the triple $\{n \geq 0 \wedge n_{aux} = n\}\, \mathsf{Fact}^{\mathsf{isa}}\, \{\mathsf{f}_{\mathsf{a0}} = \mathsf{n}_{\mathsf{aux}}!\}$ is valid.*

$\{n \geq 0 \wedge n_{aux} = n\}$
$\mathsf{Fact}^{\mathsf{isa}}$
$\{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1 \wedge f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge \neg(i_{a0} \leq n)\}$
(seq)

1. $\{n \geq 0 \wedge n_{aux} = n\}\, f_1 := 1; i_1 := 1\, \{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1\}$
   (seq)

   1. *(assign)* $\{n \geq 0 \wedge n_{aux} = n\}\, f_1 := 1\, \{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1\}$
   2. *(assign)* $\{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1\}\, i_1 := 1\, \{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1\}$

2. $\{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1\}$
for $(i_{a0} := i_1; f_{a0} := f_1, i_{a0} \leq n, i_{a0} := i_{a1}; f_{a0} := f_{a1})$ **do** $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1\}\, f_{a1} := f_{a0} * i_{a0}; i_{a1} := i_{a0} + 1$ **od**
$\{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1 \wedge f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge \neg(i_{a0} \leq n)\}$
(for)

1. $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n\}$
$f_{a1} := f_{a0} * i_{a0}; i_{a1} := i_{a0} + 1$
$\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0} \wedge i_{a1} = i_{a0} + 1\}$
(seq)

1. *(assign)* $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n\}\, f_{a1} := f_{a0} * i_{a0}\, \{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0}\}$

2. *(assign)* $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0}\}\, i_{a1} := i_{a0} + 1\, \{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0} \wedge i_{a1} = i_{a0} + 1\}$

Side conditions for application of the (for) rule:

- $n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1 \rightarrow f_1 = (i_1 - 1)! \wedge i_1 \leq n + 1$.
- $f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0} \wedge i_{a1} = i_{a0} + 1 \rightarrow f_{a1} = (i_{a1} - 1)! \wedge i_{a1} \leq n + 1$.

Note that in the previous example, the application of the (for) rule introduces two side conditions, which are both valid. The derivation generates a unique postcondition for the program, with the given precondition. Other valid triples with the same precondition may be obtained by weakening this postcondition, following Proposition 3.3.

A set of verification conditions for a triple $\{\phi\}\, C\, \{\psi\}$ can be obtained from a candidate derivation of a triple of the form $\{\phi\}\, C\, \{\phi \wedge \psi'\}$ in system Hisa. The VCs are the side conditions introduced by the (for) rule, together with $\phi \wedge \psi' \rightarrow \psi$: the triple is valid if and only if all these VCs are valid. It is possible to calculate the VCs and the formula ψ' without explicitly constructing the derivation. The following function does precisely this.

Definition 3.4 (Verification conditions generator)**.** *The VCGen function* $\mathsf{VC}_{\text{ISA}} : \textbf{Assert} \times \textbf{AComm}^{\text{ISA}} \rightarrow \textbf{Assert} \times \mathcal{P}(\textbf{Assert})$ *is defined as follows:*

$$
\begin{aligned}
\mathsf{VC}_{\text{ISA}}(\phi, \textbf{skip}) &= (\top, \emptyset) \\
\mathsf{VC}_{\text{ISA}}(\phi, x := e) &= (x = e, \emptyset) \\
\mathsf{VC}_{\text{ISA}}(\phi, C_1 \,;\, C_2) &= (\psi_1 \wedge \psi_2, \Gamma_1 \cup \Gamma_2), \textit{ where} \\
& \quad (\psi_1, \Gamma_1) = \mathsf{VC}_{\text{ISA}}(\phi, C_1), \textit{ and} \\
& \quad (\psi_2, \Gamma_2) = \mathsf{VC}_{\text{ISA}}(\phi \wedge \psi_1, C_2) \\
\mathsf{VC}_{\text{ISA}}(\phi, \textbf{if } b \textbf{ then } C_t \textbf{ else } C_f \textbf{ fi}) &= ((b \wedge \psi_t) \vee (\neg b \wedge \psi_f), \Gamma_t \cup \Gamma_f), \textit{ where} \\
& \quad (\psi_t, \Gamma_t) = \mathsf{VC}_{\text{ISA}}(\phi \wedge b, C_t) \textit{ and} \\
& \quad (\psi_f, \Gamma_f) = \mathsf{VC}_{\text{ISA}}(\phi \wedge \neg b, C_f) \\
\mathsf{VC}_{\text{ISA}}(\phi, \textbf{for } (\mathcal{I}, b, \mathcal{U}) \textbf{ do } \{\theta\}\, C \textbf{ od}) &= (\theta \wedge \neg b, \Gamma \cup \{\phi \rightarrow \mathcal{I}(\theta), \theta \wedge b \wedge \psi \rightarrow \mathcal{U}(\theta)\}), \textit{ where} \\
& \quad (\psi, \Gamma) = \mathsf{VC}_{\text{ISA}}(\theta \wedge b, C)
\end{aligned}
$$

Let $(\psi', \Gamma) = \mathsf{VC}_{\text{ISA}}(\phi, C)$. The verification conditions of C with the precondition ϕ are given by the set Γ, and the formula ψ' approximates (since it relies on loop invariants) a logical encoding of the program; it is clear from the definition that ψ' does not depend on the formula ϕ. The VCs of a Hoare triple $\{\phi\}\, C\, \{\psi\}$ are then given by $\Gamma \cup \{\phi \wedge \psi' \to \psi\}$. The VCGen is sound and complete w.r.t. system Hisa.

Proposition 3.4. *Let* $C \in \mathbf{AComm}^{\text{ISA}}$, $\phi, \psi', \psi'' \in \mathbf{Assert}$ *and* $\Gamma \subseteq \mathbf{Assert}$, *such that* $(\psi', \Gamma) = \mathsf{VC}_{\text{ISA}}(\phi, C)$. *Then:*

1. *If* $\models \Gamma$, *then* $\vdash_{\mathsf{Hisa}} \{\phi\}\, C\, \{\phi \wedge \psi'\}$.
2. *If* $\vdash_{\mathsf{Hisa}} \{\phi\}\, C\, \{\phi \wedge \psi''\}$ *for some assertion* ψ'', *then* $\models \Gamma$ *and* $\psi'' \equiv \psi'$.

Proof. 1. By induction on the structure of C. 2. By induction on the derivation of $\vdash_{\mathsf{Hisa}} \{\phi\}\, C\, \{\phi \wedge \psi''\}$. □

Example 3.4. *For our factorial example we have that* $\mathsf{VC}_{\text{ISA}}(n \geq 0 \wedge n_{aux} = n, \mathsf{Fact}^{\mathsf{isa}}) = (f_1 = 1 \wedge i_1 = 1 \wedge f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge \neg(i_{a0} \leq n), \{n \geq 0 \wedge n_{aux} = n \wedge f_1 = 1 \wedge i_1 = 1 \to f_1 = (i_1 - 1)! \wedge i_1 \leq n + 1, f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1 \wedge i_{a0} \leq n \wedge f_{a1} = f_{a0} * i_{a0} \wedge i_{a1} = i_{a0} + 1 \to f_{a1} = (i_{a1} - 1)! \wedge i_{a1} \leq n + 1\})$, *in accordance with the derivation in Example 3.3.*

Consider the calculation of $\mathsf{VC}_{\text{ISA}}(\phi, \{\mathbf{if}\ b\ \mathbf{then}\, C_t\, \mathbf{else}\, C_f\ \mathbf{fi}\}; C_2)$. The recursive call on C_2 will be $\mathsf{VC}_{\text{ISA}}(\phi \wedge ((b \wedge \psi_t) \vee (\neg b \wedge \psi_f)), C_2)$, where ψ_t, ψ_f do not depend on ϕ. The resulting VCs avoid the exponential pattern described at the beginning of Section 3.2, since a single copy of the precondition ϕ is propagated to C_2. In fact the size of the VCs is *quadratic* in the size of the program. It is clear from the $\mathsf{VC}_{\text{ISA}}(\phi, C_1\, ;\, C_2)$ clause of the definition that the propagated precondition ϕ is duplicated, with one copy used to generate VCs for C_1, and another propagated to C_2 together with the encoding of C_1. Now observe that each loop in the program generates two VCs, one corresponding to the initialization of the invariant $(\phi \to \mathcal{I}(\theta))$, and another to its preservation. The size of loop preservation VCs depends only on the size of the loop's body, but initialization conditions contain an encoding of the prefix of the program leading to the loop (propagated in the ϕ parameter), so they have size linear in the size of that prefix. The worst case occurs for a program consisting in a sequence of n loops: the ith loop will generate an initialization VC of size $\mathcal{O}(i)$, so the total size of the VCs is $\mathcal{O}(n^2)$.

3.3 Program Verification Using Iterating SA Form

We will now put up a framework for the verification of annotated While programs, based on their translation to single-assignment form and the subsequent generation of compact verification conditions from this intermediate code.

The translation into ISA form will operate at the level of Hoare triples, rather than of isolated annotated programs. Such a translation must of course abide by the syntactic restrictions of $\mathbf{AComm}^{\text{ISA}}$ (as ilustrated by the factorial example), with additional requirements of a semantic

nature. In particular, the translation will annotate the ISA program with loop invariants (produced from those contained in the original program), and Hg-derivability guided by these annotations must be preserved. On the other hand, the translation must be sound: it will not translate invalid triples into valid ones. These requirements are expressed by translating back to While programs.

Definition 3.5 (ISA translation)**.** *A function* $\mathcal{T} : \mathbf{Assert} \times \mathbf{AComm} \times \mathbf{Assert} \to \mathbf{Assert} \times \mathbf{AComm}^{\text{ISA}} \times \mathbf{Assert}$ *is said to be a* single-assignment translation *if when* $\mathcal{T}(\phi, C, \psi) = (\phi', C', \psi')$ *we have* $\phi' \# C'$*, and both the following hold:*

1. *If* $\models \{\phi'\} \lfloor \mathcal{W}(C') \rfloor \{\psi'\}$*, then* $\models \{\phi\} \lfloor C \rfloor \{\psi\}$*.*
2. *If* $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$*, then* $\vdash_{\mathsf{Hg}} \{\phi'\}\, \mathcal{W}(C')\, \{\psi'\}$*.*

The following propositions establish that translating annotated programs to ISA form before generating VCs results in a sound and complete technique for deductive verification.

Proposition 3.5 (Soundness of verification technique)**.** *Let* $C \in \mathbf{AComm}$*,* $C' \in \mathbf{AComm}^{\text{ISA}}$*,* $\phi, \phi', \psi, \psi', \gamma \in \mathbf{Assert}$ *and* $\Gamma \subseteq \mathbf{Assert}$*, such that* $(\phi', C', \psi') = \mathcal{T}(\phi, C, \psi)$ *for some ISA translation* $\mathcal{T}$*, and* $(\gamma, \Gamma) = \mathsf{VC}_{\text{ISA}}(\phi', C')$*. If* $\models \Gamma,\ \phi' \wedge \gamma \to \psi'$ *then* $\models \{\phi\} \lfloor C \rfloor \{\psi\}$*.*

Proof. From Proposition 3.4(1) we have $\vdash_{\mathsf{Hisa}} \{\phi'\}\, C'\, \{\phi' \wedge \gamma\}$ and from Definition 3.5 we have $\phi' \# C'$. Thus Proposition 3.2 applies yielding $\vdash_{\mathsf{H}} \{\phi'\} \lfloor \mathcal{W}(C') \rfloor \{\phi' \wedge \gamma\}$. From soundness of H, and because $\models \phi' \wedge \gamma \to \psi'$, it follows that $\models \{\phi'\} \lfloor \mathcal{W}(C') \rfloor \{\psi'\}$. Finally, by Definition 3.5, we have $\models \{\phi\} \lfloor C \rfloor \{\psi\}$. □

Proposition 3.6 (Completeness of verification technique)**.** *Let* $C \in \mathbf{AComm}$*,* $C' \in \mathbf{AComm}^{\text{ISA}}$*,* $\phi, \phi', \psi, \psi', \gamma \in \mathbf{Assert}$ *and* $\Gamma \subseteq \mathbf{Assert}$ *such that* $(\phi', C', \psi') = \mathcal{T}(\phi, C, \psi)$ *for some ISA translation* $\mathcal{T}$*, and* $(\gamma, \Gamma) = \mathsf{VC}_{\text{ISA}}(\phi', C')$*. If* $\models \{\phi\} \lfloor C \rfloor \{\psi\}$ *and* C *is correctly-annotated w.r.t.* (ϕ, ψ)*, then* $\models \Gamma,\ \phi' \wedge \gamma \to \psi'$*.*

Proof. First note that by completeness of system H we have $\vdash_{\mathsf{H}} \{\phi\} \lfloor C \rfloor \{\psi\}$. By Definitions 2.9 and 3.5 it follows that $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$ and $\vdash_{\mathsf{Hg}} \{\phi'\}\, \mathcal{W}(C')\, \{\psi'\}$. The latter definition implies that $\phi' \# C'$, and by Proposition 3.3 $\vdash_{\mathsf{Hisa}} \{\phi'\}\, C'\, \{\phi' \wedge \psi''\}$ for some $\psi'' \in \mathbf{Assert}$ such that $\models \phi' \wedge \psi'' \to \psi'$. Proposition 3.4(2) then gives us $\models \Gamma$ and $\psi'' \equiv \gamma$, which concludes the proof. □

Even though the rest of the results are formulated w.r.t. Definition 3.5, and we do not impose any concrete translation, Section 3.5 presents an example of such a concrete translation, together with the proof that it complies with Definition 3.5.

3.4 Adaptation Completeness of Iterating SA Programs

Ideally one would like to create a derivation for a triple and then to be able to use it to derive all other triples whose validity is semantically implied by the validity of the first. For a more

formal explanation, let (ϕ, ψ) and (ϕ', ψ') be specifications, and assume that (ϕ, ψ) is satisfiable (there exists some program that is correct w.r.t. it). Suppose now that C is a program such that if the Hoare triple $\{\phi\}\, C\, \{\psi\}$ is valid then so is $\{\phi'\}\, C\, \{\psi'\}$. An inference system for Hoare logic is said to be *adaptation-complete* if whenever this happens, then $\{\phi'\}\, C\, \{\psi'\}$ is derivable in that system from the triple $\{\phi\}\, C\, \{\psi\}$.

Adaptation is closely linked to the existence of a *consequence rule* that dictates when a triple is derivable in one step from another triple containing the same program. Therefore, it is by design entirely absent from goal-directed systems like Hg or Hisa, which have no consequence rule. System H is capable of adaptation, but not in a complete way. For a simple example of how adaptation fails in system H, consider for instance the triple $\{n \geq 0 \wedge n_{aux} = n\}\, \mathsf{Fact}\, \{f = n_{aux}!\}$. The specification makes use of an *auxiliary variable* n_{aux}. These variables do not have a special status; they are simply not used as program variables, and can be safely employed for writing specifications relating the pre-state and post-state. According to the above, the program Fact computes the factorial of the *initial* value of n. Now suppose Fact is part of a bigger program, and one would like to establish the validity of the triple $\{n = K\}\, \mathsf{Fact}\, \{f = K!\}$, with K a positive constant. Adaptation-completeness would mean that one would be able to derive this from the specification of Fact without constructing a dedicated proof – indeed, it should not even be necessary to know the implementation of Fact, since it has already been proved correct. The (conseq) rule of Hoare logic is meant precisely for this, but it cannot be applied here, since both side conditions are clearly *not valid.*

$$\frac{\{n \geq 0 \wedge n_{aux} = n\}\, \mathsf{Fact}\, \{f = n_{aux}!\}}{\{n = K\}\, \mathsf{Fact}\, \{f = K!\}} \quad \text{if} \quad \begin{array}{l} n = K \to n \geq 0 \wedge n_{aux} = n \ \text{ and} \\ f = n_{aux}! \to f = K! \end{array}$$

For an even simpler example consider the triple $\{x > 0\}\, P\, \{y = x\}$ where x is now a program variable, used outside P, but *not assigned in* P. Again let K be some positive constant. Clearly if the triple is valid then so is $\{x = K\}\, P\, \{y = K\}$, since the value of x is preserved. However, attempting to apply the consequence rule would yield the following, where the first side condition is valid, but the second is invalid

$$\frac{\{x > 0\}\, P\, \{y = x\}}{\{x = K\}\, P\, \{y = K\}} \quad \text{if} \quad x = K \to x > 0 \ \text{ and } \ y = x \to y = K$$

The problem of adaptation was raised by the study of complete extensions of Hoare logic for reasoning about recursive procedures. Assuming that an identity axiom is present, and system H rules are lifted to work with Gentzen-style sequents [55], the initial proposal by Hoare [65] was to derive a triple concerning a procedure call by assuming that same triple as a hypothesis when reasoning about the procedure's body:

$$\frac{\{\phi\}\, \mathbf{call\ p}\, \{\psi\} \vdash \{\phi\}\, \mathbf{body\ p}\, \{\psi\}}{\vdash \{\phi\}\, \mathbf{call\ p}\, \{\psi\}}$$

Example 3.5. *As an example of a recursive procedure consider the following program taken from [5]. Assume that the procedure* $\mathsf{FACT_R}$ *is defined as:*

$$\mathbf{if}\ x = 0\ \mathbf{then}\ y := 1\ \mathbf{else}\ x := x - 1\,;\ \mathbf{call}\ \mathsf{FACT_R}\,;\ x := x + 1\,;\ y := y * x\ \mathbf{fi}$$

The triple $\{x \geq 0\}\,\mathbf{call}\ \mathsf{FACT_R}\,\{y = x!\}$ *is derivable in system* H *with the addition of the recursion rule shown above. According to the recursion rule, we must prove that* $\{x \geq 0\}\,\mathbf{call}\ \mathsf{FACT_R}\,\{y = x!\} \vdash \{x \geq 0\}\,\mathbf{body}\ \mathsf{FACT_R}\,\{y = x!\}$*. For simplification purposes, we leave implicit the antecedent of the sequents which is always* $\{x \geq 0\}\,\mathbf{call}\ \mathsf{FACT_R}\,\{y = x!\}$*:*

$\{x \geq 0\}\,\mathbf{body}\ \mathsf{FACT_R}\,\{y = x!\}$
(if)

1. $\{x \geq 0 \wedge x = 0\}\, y := 1\,\{y = x!\}$
   (conseq)
   1. $\{1 = x!\}\, y := 1\,\{y = x!\}$ *(assign)*
2. $\{x \geq 0 \wedge \neg x = 0\}\, x := x - 1\,;\ \mathbf{call}\ \mathsf{FACT_R}\,;\ x := x + 1\,;\ y := y * x\,\{y = x!\}$
   (seq)
   1. $\{x \geq 0 \wedge \neg x = 0\}\, x := x - 1\,\{x \geq 0\}$
      (conseq)
      1. $\{x - 1 \geq 0\}\, x := x - 1\,\{x \geq 0\}$ *(assign)*
   2. $\{x \geq 0\}\,\mathbf{call}\ \mathsf{FACT_R}\,;\ x := x + 1\,;\ y := y * x\,\{y = x!\}$
      (seq)
      1. $\{x \geq 0\}\,\mathbf{call}\ \mathsf{FACT_R}\,\{y * (x + 1) = (x + 1)!\}$ *(conseq)*
         1. $\{x \geq 0\}\,\mathbf{call}\ \mathsf{FACT_R}\,\{y = x!\}$ *[cancellation using the antecedent* $\{x \geq 0\}\,\mathbf{call}\ \mathsf{FACT_R}\,\{y = x!\}$*]*
      2. $\{y * (x + 1) = (x + 1)!\}\, x := x + 1\,;\ y := y * x\,\{y = x!\}$
         (conseq)
         1. $\{y * (x + 1) = (x + 1)!\}\, x := x + 1\,\{y * x = x!\}$ *(assign)*
         2. $\{y * x = x!\}\, y := y * x\,\{y = x!\}$ *(assign)*

Side conditions for application of the (conseq) rules:

- $x \geq 0 \wedge x = 0 \implies 1 = x!$.
- $x \geq 0 \wedge \neg x = 0 \implies x - 1 \geq 0$.
- $y = x! \implies y * (x + 1) = (x + 1)!$.

As stated by Hoare [65] in his seminal work, the traditional Hoare inference system with the addition of this rule is not complete:

> It has been shown that it is possible by axiomatic methods to define an important programming language feature in such a way as to facilitate the demonstration of the correctness of programs and at the same time to permit flexibility and high efficiency of implementation. The combination of these two advantages can be achieved **only**

> **if the programmer is willing to observe certain disciplines in his use of the feature**, namely that all actual parameters which may be changed by a procedure must be distinct from each other, and must not be contained in any of the other parameters.

Even though the arguments given by Hoare for the incompleteness of the inference system are based on parameter passing, Apt [5] shows that this incompleteness is visible even in parameterless procedures, if one wants to reason with auxiliary variables (it was later shown that those are orthogonal issues [75]). For instance if we consider the program $\mathsf{FACT_R}$ from Example 3.5, it is not possible to show that the program never changes the value of x, that is, it is not possible to derive the triple $\{x = z\}\,\mathsf{FACT_R}\,\{x = z\}$. The proof of such impossibility follows by contradiction and is shown in [5].

Along the years several attempts were made to create a sound and complete inference system capable of dealing with procedure calls and/or auxiliary variables. Among those works are the work of Gorelick [59] and Sokolwski [100] which were used as a basis by Apt [5] to introduce a new set of rules to create a system that is shown to be complete. Nonetheless, America and de Boer [4] showed that such a system becomes unsound if total correctness is to be considered and propose a set of additional structural rules [4] to fix it.

Years later the topic was addressed again by Kleymann [75], who showed that the adaptation problem is orthogonal to the handling of recursive procedures: if the base system is made adaptation-complete, then Hoare's rule for recursive procedure calls is sufficient to achieve a Cook-complete system, with no need for further structural rules. Kleymann obtains an adaptation-complete inference system for Hoare logic by proposing the following consequence rule, whose side condition is a meta-level formula with quantification over states/variable assignments:

$$(\text{conseq}_K) \qquad \frac{\{\phi\}\,C\,\{\psi\}}{\{\phi'\}\,C\,\{\psi'\}} \quad \begin{array}{l}\text{if } \forall Z.\forall\sigma.[\![\phi']\!](Z,\sigma) \rightarrow \\ \quad \forall\tau.(\forall Z_1.[\![\phi]\!](Z_1,\sigma) \rightarrow [\![\psi]\!](Z_1,\tau)) \\ \quad \rightarrow [\![\psi']\!](Z,\tau)\end{array}$$

where $[\![\phi']\!](Z,\sigma)$ denotes the truth value of ϕ' in the state (Z,σ), partitioned between auxiliary (Z) and program (σ) variables. With the addition of this rule, for reasoning about partial correctness in the presence of recursive functions calls one would use the initial Hoare rule for recursive calls (presented above). Note however that, the (conseq_K) rule as it is presented above, is not a first-order formula due to the quantification over states, and therefore it cannot be handled directly by an SMT solver.

In what follows, we will show that reasoning with single-assignment programs is advantageous from the point of view of adaptation: our Hisa system will be made adaptation-complete by adding a rule with a simple syntactic side condition.

Let us start with a result showing that the side condition of a consequence rule that always leads to adaptation-completeness, in general terms, turns out to be the result of stripping away

the states and quantifiers in the side condition of (conseq_K) above.

Lemma 3.7. *Let* $\phi, \phi', \psi, \psi' \in \textbf{Assert}$*. If there exists at least one program* $C_0 \in \textbf{AComm}$ *such that* $\models \{\phi\}\, C_0\, \{\psi\}$*, and for arbitrary* C *one has that* $\models \{\phi\}\, C\, \{\psi\}$ *implies* $\models \{\phi'\}\, C\, \{\psi'\}$*, then it must be the case that* $\models \phi' \to (\phi \to \psi) \to \psi'$*.*

Proof. We assume $\not\models \phi' \to (\phi \to \psi) \to \psi'$, i.e. there exists a state s_0 such that $s_0 \models \phi'$, $s_0 \models \phi \to \psi$, and $s_0 \not\models \psi'$. To show that in this context $\models \{\phi'\}\, C\, \{\psi'\}$ does not follow from $\models \{\phi\}\, C\, \{\psi\}$ for arbitrary C, we construct a particular program C_1 with the following behavior: $\langle C_1, s_0\rangle \rightsquigarrow s_0$ and for $s \neq s_0$,$\langle C_1, s\rangle \rightsquigarrow s'$ whenever $\langle C_0, s\rangle \rightsquigarrow s'$. To see that $\{\phi\}\, C_1\, \{\psi\}$ is a valid triple, observe that if $s_0 \models \phi$ and C_1 is executed in state s_0 we will have $s_0 \models \psi$ since $s_0 \models \phi \to \psi$, and for other executions we note that $\models \{\phi\}\, C_0\, \{\psi\}$. The triple $\{\phi'\}\, C_1\, \{\psi'\}$ is however not valid, since $s_0 \models \phi'$, but $\langle C_1, s_0\rangle \rightsquigarrow s_0$ and $s_0 \not\models \psi'$. □

The problem is that a consequence rule with side condition $\phi' \to (\phi \to \psi) \to \psi'$ would not be sound. But it is sound for triples satisfying the simple syntactic restriction that free variables of the precondition are not assigned in the program.

Lemma 3.8. *Let* $C \in \textbf{Comm}$ *and* $\phi \in \textbf{Assert}$*. If* $\phi \# C$ *and* $\langle C, s\rangle \rightsquigarrow s'$*, then* $[\![\phi]\!](s) = [\![\phi]\!](s')$*.*

Proof. Since $\phi \# C$, $s(x) = s'(x)$ for every $x \in \mathsf{FV}(\phi)$. Hence, $[\![\phi]\!](s) = [\![\phi]\!](s')$. □

Lemma 3.9. *Let* $C \in \textbf{Comm}$ *and* $\phi, \phi', \psi, \psi' \in \textbf{Assert}$*, such that* $\phi \# C$ *and* $\phi' \# C$*. If* $\models \{\phi\}\, C\, \{\psi\}$ *and* $\models \phi' \to (\phi \to \psi) \to \psi'$*, then* $\models \{\phi'\}\, C\, \{\psi'\}$*.*

Proof. Assume $s \models \phi'$ and $\langle C, s\rangle \rightsquigarrow s'$. Since $\phi' \# C$, by Lemma 3.8, we get $s' \models \phi'$. We also have $s' \models \phi \to \psi$ because, if $s' \models \phi$, then $s \models \phi$ (by Lemma 3.8, since $\phi \# C$) so, as $\models \{\phi\}\, C\, \{\psi\}$, we get $s' \models \psi$. Now, $s' \models \psi'$ follows directly from $\models \phi' \to (\phi \to \psi) \to \psi'$, $s' \models \phi'$ and $s' \models \phi \to \psi$. □

Recall from Lemma 3.1 that Hisa derivations consist entirely of triples $\{\phi\}\, C\, \{\psi\}$ satisfying the $\phi \# C$ condition, which means that an adaptation rule with the side condition given above can be naturally incorporated in the system. We must however be careful to ensure that the new rule *preserves* Lemma 3.1; in particular, the postcondition ψ' should not contain free occurrences of variables not occurring either in the program or free in the precondition ϕ'. The following result will allow us to eliminate these free occurrences.

Lemma 3.10. *Let* $C \in \textbf{Comm}$*,* $\phi, \psi \in \textbf{Assert}$ *and* $x \in \textbf{Var}$*, such that* $x \notin \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$*. If* $\models \{\phi\}\, C\, \{\psi\}$ *then* $\models \{\phi\}\, C\, \{\forall\, x.\ \psi\}$*.*

Proof. Assume $s \models \phi$ and $\langle C, s\rangle \rightsquigarrow s'$. As $x \notin \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$, for every $a \in D$, $s[x \mapsto a] \models \phi$ and $\langle C, s[x \mapsto a]\rangle \rightsquigarrow s'[x \mapsto a]$. Since $\models \{\phi\}\, C\, \{\psi\}$, it follows that $s'[x \mapsto a] \models \psi$. Hence, $s' \models \forall\, x.\ \psi$. □

Let Hisa^+ be the inference system consisting of all the rules of system Hisa together with the following rule:

$$(\text{conseq}_a) \qquad \frac{\{\phi\}\, C\, \{\phi \wedge \psi\}}{\{\phi'\}\, C\, \{\phi' \wedge (\forall\, \vec{x}.\ \phi \to \psi)\}} \quad \text{if } \phi \# C \text{ and } \vec{x} = \mathsf{FV}(\phi) \backslash (\mathsf{FV}(\phi') \cup \mathsf{Vars}(C))$$

Recall that Hisa is a forward propagation system, so the rule will be applied when we reach C with the propagated precondition ϕ' (in which case Lemma 3.1 ensures that $\phi' \# C$ holds). The rule will produce a postcondition not directly by propagating ϕ' through the structure of C, but instead by adapting the triple $\{\phi\}\, C\, \{\phi \wedge \psi\}$. The conditions ϕ and ψ may well contain occurrences of variables not occurring either in C or free in ϕ', but the quantification ensures that Lemma 3.1 remains valid in system Hisa^+. Note that the lemma guarantees that $\mathsf{FV}(\psi) \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$, so $\mathsf{FV}(\psi)$ does not need to be included in $\vec{x}$.

System Hisa^+ is not goal-directed, since there will always be a choice to apply the rule specific to the command at hand or the (conseq_K) rule, but it is still a forward-propagation system (the postcondition is the strongest allowed by Lemma 3.9).

Proposition 3.11 (Soundness of Hisa^+)**.** *Let $C \in \mathbf{AComm}^{\text{ISA}}$ and ϕ, $\psi' \in \mathbf{Assert}$ such that $\phi \# C$ and $\vdash_{\mathsf{Hisa}^+} \{\phi\}\, C\, \{\phi \wedge \psi'\}$. Then $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\phi \wedge \psi'\}$.*

Proof. The proof, by induction on the derivation of $\vdash_{\mathsf{Hisa}^+} \{\phi\}\, C\, \{\phi \wedge \psi'\}$, extends the proof of Proposition 3.2 with the (conseq_a) rule case. Assume the last step is

$$\frac{\{\phi_1\}\, C\, \{\phi_1 \wedge \psi_1\}}{\{\phi\}\, C\, \{\phi \wedge (\forall\, \vec{x}.\ \phi_1 \to \psi_1)\}} \quad \text{with } \ \phi_1 \# C \text{ and } \vec{x} = \mathsf{FV}(\phi_1) \backslash (\mathsf{FV}(\phi) \cup \mathsf{Vars}(C))$$

By IH we have $\vdash_{\mathsf{H}} \{\phi_1\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\phi_1 \wedge \psi_1\}$ and since H is sound it follows that $\models \{\phi_1\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\phi_1 \wedge \psi_1\}$. As $\models \phi \to (\phi_1 \to \phi_1 \wedge \psi_1) \to \phi \wedge (\phi_1 \to \psi_1)$, we get $\models \{\phi\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\phi \wedge (\phi_1 \to \psi_1)\}$ by Lemma 3.9. Now note that $\vec{x} \cap (\mathsf{FV}(\phi) \cup \mathsf{Vars}(C)) = \emptyset$, thus Lemma 3.10 can be applied, and it follows that $\models \{\phi\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\phi \wedge (\forall\, \vec{x}.\ \phi_1 \to \psi_1)\}$. Finally, by completeness of H we obtain $\vdash_{\mathsf{H}} \{\phi\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\phi \wedge (\forall\, \vec{x}.\ \phi_1 \to \psi_1)\}$. □

The system is obviously complete in the same sense as Hisa, since it extends it. But unlike Hisa it is also adaptation-complete.

Proposition 3.12 (Adaptation completeness of Hisa^+)**.**
Assume that (ϕ, ψ) is satisfiable and that for all $C \in \mathbf{AComm}^{\text{ISA}}$, such that $\phi \# C$, it holds that $\models \{\phi\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\psi\}$ implies $\models \{\phi'\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\psi'\}$.

Let $C' \in \mathbf{AComm}^{\text{ISA}}$ such that $\phi \# C'$, and $\vdash_{\mathsf{Hisa}^+} \{\phi\}\, C'\, \{\phi \wedge \gamma\}$ for some $\gamma \in \mathbf{Assert}$ such that $\models \phi \wedge \gamma \to \psi$. Then $\{\phi'\}\, C'\, \{\phi' \wedge (\forall\, \vec{x}.\ \phi \to \gamma)\}$ with $\vec{x} = \mathsf{FV}(\phi) \backslash (\mathsf{FV}(\phi') \cup \mathsf{Vars}(C'))$ can be derived from that triple in system Hisa^+, and $\models \phi' \wedge (\forall\, \vec{x}.\ \phi \to \gamma) \to \psi'$.

Proof. From $\vdash_{\mathsf{Hisa}^+} \{\phi\}\, C'\, \{\phi \wedge \gamma\}$ we can apply the (conseq_a) rule to produce $\vdash_{\mathsf{Hisa}^+} \{\phi'\}\, C'\, \{\phi' \wedge (\forall\, \vec{x}.\ \phi \to \gamma)\}$ with $\vec{x} = \mathsf{FV}(\phi) \backslash (\mathsf{FV}(\phi') \cup \mathsf{Vars}(C'))$, since $\phi \# C'$. So it just remains to prove the validity of the formula $\phi' \wedge (\forall\, \vec{x}.\ \phi \to \gamma) \to \psi'$.

From $\models \phi \land \gamma \to \psi$ it follows that $\models (\phi \to \gamma) \to (\phi \to \psi)$, and so we also have $\models (\forall\, \vec{x}.\ \phi \to \gamma) \to (\forall\, \vec{x}.\ \phi \to \psi)$. Consequently $\models \phi' \land (\forall\, \vec{x}.\ \phi \to \gamma) \to \phi' \land (\forall\, \vec{x}.\ \phi \to \psi)$, and thus $\phi' \land (\forall\, \vec{x}.\ \phi \to \gamma) \models \phi' \land (\forall\, \vec{x}.\ \phi \to \psi)$. On the other hand, as $\vec{x} \cap \mathsf{FV}(\phi') = \emptyset$, we have $\phi' \land (\forall\, \vec{x}.\ \phi \to \psi) \models \phi' \land (\phi \to \psi)$. Now, since $\models \{\phi\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\psi\}$ implies $\models \{\phi'\}\, \lfloor \mathcal{W}(C) \rfloor\, \{\psi'\}$, it follows by Lemma 3.7 that $\models \phi' \to (\phi \to \psi) \to \psi'$, and hence we get $\phi' \land (\forall\, \vec{x}.\ \phi \to \gamma) \models \psi'$. Now we can conclude that $\models \phi' \land (\forall\, \vec{x}.\ \phi \to \gamma) \to \psi'$. □

Consider again the example referred at the beginning of the section, now in ISA form as in Example 3.2. Let K be a positive constant; the Hoare triple $\{n = K\}\, \mathsf{Fact}^{\mathsf{isa}}\, \{f_{a0} = K!\}$ can now be derived from $\{n \geq 0 \land n_{aux} = n\}\, \mathsf{Fact}^{\mathsf{isa}}\, \{n \geq 0 \land n_{aux} = n \land f_{a0} = n_{aux}!\}$:

$$\frac{\{n \geq 0 \land n_{aux} = n\}\, \mathsf{Fact}^{\mathsf{isa}}\, \{n \geq 0 \land n_{aux} = n \land f_{a0} = n_{aux}!\}}{\{n = K\}\, \mathsf{Fact}^{\mathsf{isa}}\, \{n = K \land (\forall\, n_{aux}.\ n \geq 0 \land n_{aux} = n \to f_{a0} = n_{aux}!)\}}$$

since $\models n = K \land (\forall\, n_{aux}.\ n \geq 0 \land n_{aux} = n \to f_{a0} = n_{aux}!) \to f_{a0} = K!$. As to the second example, consider the following derivation, recalling that x is not assigned in P:

$$\frac{\{x > 0\}\, P\, \{x > 0 \land y = x\}}{\{x = K\}\, P\, \{x = K \land (x > 0 \to y = x)\}}$$

This proves $\{x = K\}\, P\, \{y = K\}$ holds since $\models x = K \land (x > 0 \to y = x) \to y = K$.

3.5 Iterating SA Translation

The definition of a valid ISA translation was introduced in Section 3.3, and the subsequent results were established with respect to it. Nonetheless, at that point nothing was said about a concrete translation, neither if such a translation existed. In this section we shown that it is possible to define such a translation by presenting a concrete one, and in this way we show that our results are not vacuous.

The translation will be given through a function that transforms annotated programs into ISA form, that is, it transforms **AComm** programs into $\mathbf{AComm}^{\text{ISA}}$ programs. In order to be able to create unique symbols for the introduced ISA variables some new notation must be introduced. Without loss of generality, it will be assumed that each ISA variable (a single variable of the universe of variables from ISA programs) will be divided into two parts: the *variable identifier* and the variable *version*. The identifier will be taken from the provided program and the version will consist in a non-empty list of positive integers. Using a list for the version part will be particularly advantageous when dealing with nested loops. The set of ISA variables will be given by $\mathbf{Var}^{\text{ISA}} = \mathbf{Var} \times \mathbb{N}^{+}$ and x_l will denote the ISA variable $(x, l) \in \mathbf{Var}^{\text{ISA}}$. The set of states over ISA variables will be denoted by $\Sigma^{\text{ISA}} = \mathbf{Var}^{\text{ISA}} \to D$, with D being the interpretation domain. Finally, the set of program expressions, boolean expressions, and assertions over ISA variables will be respectively given by $\mathbf{Exp}^{\text{ISA}}$, $\mathbf{Exp}^{\text{ISA}}_{\mathbf{bool}}$, $\mathbf{Assert}^{\text{ISA}}$.

For convenience purposes, along this section we will overload function names, but this should

not impose any additional challenge since it is implicit from the context which specific function to apply. The version function $\mathcal{V} : \mathbf{Var} \rightarrow \mathbb{N}^{+}$ assigns versions to variables and $\widehat{\mathcal{V}} : \mathbf{Var} \rightarrow \mathbf{Var}^{\text{ISA}}$ renames variables according to $\mathcal{V}$, such that $\widehat{\mathcal{V}}(x) = x_{\mathcal{V}(x)}$. The same function is lifted to **Exp**, $\mathbf{Exp}^{\mathbf{bool}}$, and **Assert** as shown below (we omit here the case of $\mathbf{Exp}^{\mathbf{bool}}$ due to its similarity to **Exp**).

$$\begin{array}{ll}
\widehat{\mathcal{V}} : \mathbf{Exp} \rightarrow \mathbf{Exp}^{\text{ISA}} & \widehat{\mathcal{V}} : \mathbf{Assert} \rightarrow \mathbf{Assert}^{\text{ISA}} \\
\widehat{\mathcal{V}}(e) = e[\widehat{\mathcal{V}}(x_1)/x_1, \ldots, \widehat{\mathcal{V}}(x_n)/x_n] & \widehat{\mathcal{V}}(\phi) = \phi[\widehat{\mathcal{V}}(x_1)/x_1, \ldots, \widehat{\mathcal{V}}(x_n)/x_n] \\
\qquad \text{for all } x_1, \ldots, x_n \in \mathsf{FV}(e) & \qquad \text{for all } x_1, \ldots, x_n \in \mathsf{FV}(\phi)
\end{array}$$

Lastly, if $s \in \Sigma$ and $\mathcal{V} : \mathbf{Var} \rightarrow \mathbb{N}^{+}$, then $\mathcal{V}(s) \in \mathbf{Var}^{\text{ISA}} \rightharpoonup D$ is the partial function $[\widehat{\mathcal{V}}(x) \mapsto s(x) \mid x \in \mathbf{Var}]$.

With the presented foundations, we are now ready to introduce a function that translates programs into ISA. The following definition does precisely this.

Definition 3.6 (Concrete ISA translation)**.** *The translation function for annotated While programs* $\mathfrak{T} : (\mathbf{Var} \rightarrow \mathbb{N}^{+}) \times \mathbf{AComm} \rightarrow (\mathbf{Var} \rightarrow \mathbb{N}^{+}) \times \mathbf{AComm}^{\text{ISA}}$ *is defined in Figure 3.2 (top). The translation function for Hoare triples with annotated While programs* $\mathfrak{T}$: $\mathbf{Assert} \times \mathbf{AComm} \times \mathbf{Assert} \rightarrow \mathbf{Assert}^{\text{ISA}} \times \mathbf{AComm}^{\text{ISA}} \times \mathbf{Assert}^{\text{ISA}}$ *is defined as* $\mathfrak{T}(\phi, C, \psi) = (\widehat{\mathcal{V}}(\phi), C', \widehat{\mathcal{V}'}(\psi))$*, where* $(\mathcal{V}', C') = \mathfrak{T}(\mathcal{V}, C)$ *and* $\mathcal{V}(x) = 0$*, for all* $x \in \mathbf{Var}$*.*

The function $\mathfrak{T}$ for annotated While programs receives the initial version of the variable identifier and the annotated program, and returns a pair with the final version of each variable and the ISA translated program. In particular note that the initial versions of the variables are not fixed - any version will do. The definition of $\mathfrak{T}$ relies on various auxiliary functions that are defined using Haskell-like syntax in Figure 3.2 (bottom). The main goal of each function is as follows:

- next increments the head of a variable version.
- new appends a new element to the variable version list.
- jump removes the head of the variable version list and increments the head of the resulting list – can only be used with a list of at least length two.
- sup creates a new version function using the highest version for each variable from the two version functions received – variable versions are compared using $\prec$.
- mrg creates a **Rnm** (see Definition 2.19) to advance the variables from the first version function to the second.
- upd uses the previous mentioned function jump to create assignments to propagate ISA variables.

$$\mathfrak{T} : (\mathbf{Var} \to \mathbb{N}^+) \times \mathbf{AComm} \to (\mathbf{Var} \to \mathbb{N}^+) \times \mathbf{AComm}^{\text{ISA}}$$

$$\begin{aligned}
\mathfrak{T}(\mathcal{V}, \mathbf{skip}) &= (\, \mathcal{V}, \mathbf{skip}\,) \\
\mathfrak{T}(\mathcal{V}, x := e) &= (\, \mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))], x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e)\,) \\
\mathfrak{T}(\mathcal{V}, C_1\,;\, C_2) &= (\, \mathcal{V}'', C_1'\,;\, C_2'\,), \\
&\quad \text{where } (\mathcal{V}', C_1') = \mathfrak{T}(\mathcal{V}, C_1) \text{ and } (\mathcal{V}'', C_2') = \mathfrak{T}(\mathcal{V}', C_2) \\
\mathfrak{T}(\mathcal{V}, \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \mathbf{fi}) &= (\, \mathsf{sup}(\mathcal{V}', \mathcal{V}''),\ \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then}\ \{C_t'\,;\, \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\} \\
&\qquad \mathbf{else}\ \{C_f'\,;\, \mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\}\ \mathbf{fi}\,), \\
&\quad \text{where } (\mathcal{V}', C_t') = \mathfrak{T}(\mathcal{V}, C_t) \text{ and } (\mathcal{V}'', C_f') = \mathfrak{T}(\mathcal{V}, C_f) \\
\mathfrak{T}(\mathcal{V}, \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \mathbf{od}) &= (\, \mathcal{V}''',\ \mathbf{for}\ (\mathcal{I}, \widehat{\mathcal{V}'}(b), \mathcal{U})\ \mathbf{do}\, \{\widehat{\mathcal{V}'}(\theta)\}\, C'\ \mathbf{od}\,;\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))\,), \\
&\quad \text{where} \quad \mathcal{I} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}(x)} \mid x \in \mathsf{Asgn}(C)], \\
&\qquad\qquad \mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)], \\
&\qquad\qquad (\mathcal{V}'', C') = \mathfrak{T}(\mathcal{V}', C), \\
&\qquad\qquad \mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)], \\
&\quad \textit{and} \quad \mathcal{V}''' = \mathcal{V}''[x \mapsto \mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})]
\end{aligned}$$

$\mathsf{next} : \mathbb{N}^+ \to \mathbb{N}^+$
$\mathsf{next}\ (h : t) = (h + 1) : t$

$\mathsf{new} : \mathbb{N}^+ \to \mathbb{N}^+$
$\mathsf{new}\ l = 1 : l$

$\mathsf{jump} : \mathbb{N}^+ \to \mathbb{N}^+$
$\mathsf{jump}\ (i : j : t) = (j + 1) : t$

$(h : t) \prec (h' : t') = h < h'$

$\mathsf{sup} : (\mathbf{Var} \to \mathbb{N}^+)^2 \to (\mathbf{Var} \to \mathbb{N}^+)$

$$\mathsf{sup}\ (\mathcal{V}, \mathcal{V}')(x) = \begin{cases} \mathcal{V}(x) & \text{if } \mathcal{V}'(x) \prec \mathcal{V}(x) \\ \mathcal{V}'(x) & \text{otherwise} \end{cases}$$

$\mathsf{mrg} : (\mathbf{Var} \to \mathbb{N}^+)^2 \to \mathbf{Rnm}$
$\mathsf{mrg}\ (\mathcal{V}, \mathcal{V}') = [x_{\mathcal{V}'(x)} := x_{\mathcal{V}(x)} \mid x \in \mathbf{Var} \land \mathcal{V}(x) \prec \mathcal{V}'(x)]$

$\mathsf{upd} : \mathcal{P}(\mathbf{Var}^{\text{ISA}}) \to \mathbf{Rnm}$
$\mathsf{upd}\ (X) = [x_{\mathsf{jump}(l)} := x_l \mid x_l \in X]$

Figure 3.2: ISA translation function

For the sake of simplicity, it will be assumed that the renaming sequences $\mathcal{I}$ and $\mathcal{U}$, defined in the case of while commands, follow some predefined order established over **Var** (any order will do).

Example 3.6. *To illustrate the application of $\mathfrak{T}$ to a particular example, let us consider a different factorial program that only uses sum arithmetic expressions. The following triple is based on the Turing notes 'Checking a Large Routine' [101], and in addition to its historical significance, is particularly interesting here because it allows us to demonstrate how to rename variables inside nested loops. For simplification purposes this program will be referred to as* $\mathsf{FACT_T}$.

$$
\begin{array}{l}
f := 1\,; \\
i := 1\,; \\
\textbf{while } i \leq n \textbf{ do } \{f = (i-1)! \wedge i \leq n+1\} \\
\quad j := 1\,; \\
\quad r := 0\,; \\
\quad \textbf{while } j \leq i \textbf{ do } \{j \leq i+1 \wedge r = f * (j-1)\} \\
\quad\quad r := r + f\,; \\
\quad\quad j := j + 1 \\
\quad \textbf{od}\,; \\
\quad f := r\,; \\
\quad i := i + 1 \\
\textbf{od}
\end{array}
$$

Let $\mathcal{V} : (\mathbf{Var} \to \mathbb{N}^+)$ be the version function that maps every variable to the list containing the sole element 0. The result of applying $\mathfrak{T}$ to $\mathcal{V}$ and $\mathsf{FACT_T}$ is the version function $\mathcal{V}[f \mapsto 2, i \mapsto 2, j \mapsto 1, r \mapsto 1]$ and the program below. For the sake of presentation, index lists are depicted using '.' as a separator and omitting the empty list constructor.

$$
\begin{array}{l}
f_1 := 1\,; \\
i_1 := 1\,; \\
\textbf{for } (\{j_{1.0} := j_0\,;\, r_{1.0} := r_0\,;\, f_{1.1} := f_1\,;\, i_{1.1} := i_1\}, \\
\quad\quad i_{1.1} \leq n_0, \\
\quad\quad \{j_{1.0} := j_{3.0}\,;\, r_{1.0} := r_{3.0}\,;\, f_{1.1} := f_{2.1}\,;\, i_{1.1} := i_{2.1}\}) \textbf{ do } \{f_{1.1} = (i_{1.1} - 1)! \wedge i_{1.1} \leq n_0 + 1\} \\
\quad j_{2.0} := 1\,; \\
\quad r_{2.0} := 0\,; \\
\quad \textbf{for } (\{r_{1.2.0} := r_{2.0}\,;\, j_{1.2.0} := j_{2.0}\}, j_{1.2.0} \leq i_{1.1}, \{r_{1.2.0} := r_{2.2.0}\,;\, j_{1.2.0} := j_{2.2.0}\}) \textbf{ do } \{ \\
\quad\quad\quad j_{1.2.0} \leq i_{1.1} + 1 \wedge r_{1.2.0} = f_{1.1} * (j_{1.2.0} - 1)\} \\
\quad\quad r_{2.2.0} := r_{1.2.0} + f_{1.1}\,; \\
\quad\quad j_{2.2.0} := j_{1.2.0} + 1
\end{array}
$$

od ;
$r_{3.0} := r_{1.2.0}$;
$j_{3.0} := j_{1.2.0}$;
$f_{2.1} := r_{3.0}$;
$i_{2.1} := i_{1.1} + 1$
od ;
$j_1 := j_{1.0}$;
$r_1 := r_{1.0}$;
$f_2 := f_{1.1}$;
$i_2 := i_{1.1}$

This program will be referred to as $\mathsf{FACT}_\mathsf{T}^\mathsf{isa}$. *The initial and final version functions can be used to translate a Hoare triple whose program is* FACT_T. *For instance, the Hoare triple* $\{n \geq 0 \wedge aux = n\}\, \mathsf{FACT}_\mathsf{T}\, \{f = aux!\}$ *can be translated into the ISA Hoare triple* $\{n_0 \geq 0 \wedge aux_0 = n_0\}\, \mathsf{FACT}_\mathsf{T}^\mathsf{isa}\, \{f_2 = aux_0!\}$. *As expected, the program does not assign free variables from the preconditions, that is,* $\{n_0 \geq 0 \wedge aux_0 = n_0\} \# \mathsf{FACT}_\mathsf{T}^\mathsf{isa}$ *holds.*

Let us now focus on the proof that $\mathfrak{T}$ is indeed an ISA translation. For that we start by considering some lemmas. In particular, we start by establishing some results relating version functions, program states and the evaluation of expressions and asserts. This is particularly useful to relate ISA variables with non-ISA variables.

Lemma 3.13. *Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $s \in \Sigma$ *and* $s' \in \Sigma^{\text{ISA}}$. *If* $\forall\, x \in \mathbf{Var}.\ s(x) = s'(\widehat{\mathcal{V}}(x))$, *then* $s' = s'_0 \oplus \mathcal{V}(s)$ *for some* $s'_0 \in \Sigma^{\text{ISA}}$.

Proof. Follows directly from the definitions. □

The lemma states basically that if some $s \in \Sigma$ agrees with $s' \in \Sigma^{\text{ISA}}$ for all variables in some version given by $\mathcal{V}$, then s' can be defined by resorting to $\mathcal{V}(s)$. Since $\mathcal{V}(s)$ is a partial function that assigns values only to the ISA variables whose version agree with $\mathcal{V}$, and the functions from Σ^{ISA} should be total, some $s'_0 \in \Sigma^{\text{ISA}}$ must exist that renames all other versions. Note however that nothing is known about s'_0.

With the previous result it is obvious to conclude that if all variables from some expression or assert are renamed with a particular version function $\mathcal{V}$, then the evaluation of that expression or assert in some state $s' \oplus \mathcal{V}(s)$ only depends on s.

Lemma 3.14. *Let* $e \in \mathbf{Exp}$, $b \in \mathbf{Exp^{bool}}$, $\phi \in \mathbf{Assert}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $s \in \Sigma$ *and* $s' \in \Sigma^{\text{ISA}}$.

1. $[\![\widehat{\mathcal{V}}(e)]\!](s' \oplus \mathcal{V}(s)) = [\![e]\!](s)$.
2. $[\![\widehat{\mathcal{V}}(b)]\!](s' \oplus \mathcal{V}(s)) = [\![b]\!](s)$.
3. $[\![\widehat{\mathcal{V}}(\phi)]\!](s' \oplus \mathcal{V}(s)) = [\![\phi]\!](s)$.

Proof. By induction on the structure of e (resp. b or ϕ). □

It is guaranteed that the function $\mathfrak{T}$ never touches the version of variables that are not assigned in the program, therefore the initial and final version functions will always agree on the version of those variables.

Lemma 3.15. *Let $C \in \mathbf{AComm}$ and $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$. If $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', C')$, then for every $x \in \mathbf{Var} \setminus \mathsf{Asgn}(C)$, $\mathcal{V}(x) = \mathcal{V}'(x)$.*

Proof. By induction on the structure of C. □

The following lemma plays a central role in the proof of the while command case in the subsequent proposition. It captures the fact that the translation of the loop preserves the operational semantics and that the final value of the variables in the original program correspond to the version of the variables in $\mathcal{V}'$.

Lemma 3.16. *Let $C_t \in \mathbf{AComm}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $s_i, s_f \in \Sigma$, $s', s'_f \in \Sigma^{\text{ISA}}$ and*

$$\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C_t)]$$
$$\mathfrak{T}(\mathcal{V}', C_t) = (\mathcal{V}'', C'_t)$$
$$\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C_t)]$$

If $\langle \mathbf{while}\ b\ \mathbf{do}\ \lfloor C_t \rfloor\ \mathbf{od}, s_i\rangle \rightsquigarrow s_f$ and $\langle \mathbf{while}\ \widehat{\mathcal{V}'(b)}\ \mathbf{do}\ \lfloor \mathcal{W}(C'_t)\rfloor ; \mathcal{U}\ \mathbf{od}, s' \oplus \mathcal{V}(s_i)\rangle \rightsquigarrow s'_f$ then $\forall\, x \in \mathbf{Var}.\ s_f(x) = s'_f(\widehat{\mathcal{V}}'(x))$.

Proof. By induction on the derivation of the evaluation relation $\rightsquigarrow$. Assume $\langle \mathbf{while}\ b\ \mathbf{do}\ \lfloor C_t \rfloor\ \mathbf{od}, s_i\rangle \rightsquigarrow s_f$ and $\langle \mathbf{while}\ \widehat{\mathcal{V}'(b)}\ \mathbf{do}\ \lfloor \mathcal{W}(C'_t)\rfloor ; \mathcal{U}\ \mathbf{od}, s' \oplus \mathcal{V}(s_i)\rangle \rightsquigarrow s'_f$. From the definition of $\rightsquigarrow$, two cases can occur:

- Case $[\![b]\!](s_i) = \bot$, if and only if (from Lemma 3.14), $[\![\widehat{\mathcal{V}}'(b)]\!](s' \oplus \mathcal{V}'(s_i)) = [\![b]\!](s_i) = \bot$. In this case, from definition of $\rightsquigarrow$, $s_f = s_i$ and $s'_f = s' \oplus \mathcal{V}'(s_i)$. Hence, $\forall\, x \in \mathbf{Var}.\ s'_f(\widehat{\mathcal{V}}'(x)) = s_i(x) = s_f(x)$.
- Case $[\![b]\!](s_i) = \top$, if and only if (again, from Lemma 3.14), $[\![\widehat{\mathcal{V}}'(b)]\!](s' \oplus \mathcal{V}'(s_i)) = [\![b]\!](s_i) = \top$. In this case we must have, for some $s_1 \in \Sigma$,

$$\langle \lfloor C_t \rfloor, s_i\rangle \rightsquigarrow s_1 \tag{3.1}$$
$$\langle \mathbf{while}\ b\ \mathbf{do}\ \lfloor C_t \rfloor\ \mathbf{od}, s_1\rangle \rightsquigarrow s_f \tag{3.2}$$

and also, for some $s'_0, s'_1 \in \Sigma^{\text{ISA}}$,

$$\langle \lfloor \mathcal{W}(C'_t)\rfloor, s' \oplus \mathcal{V}'(s_i)\rangle \rightsquigarrow s'_0 \tag{3.3}$$
$$s'_0 = s'_2 \oplus \mathcal{V}''(s_i) \tag{3.4}$$
$$\langle \mathcal{U}, s'_0\rangle \rightsquigarrow s'_1 \tag{3.5}$$
$$s'_1 = s'_0[x_{\mathsf{new}(x_{\mathcal{V}(x)})} \mapsto [\![x_{\mathcal{V}''(x)}]\!](s'_0) \mid x \in \mathsf{Asgn}(C_t)] = s'_0 \oplus \mathcal{V}'(s_1) \tag{3.6}$$
$$\langle \mathbf{while}\ \widehat{\mathcal{V}}'(b)\ \mathbf{do}\ \lfloor \mathcal{W}(C'_t)\rfloor ; \mathcal{U}\ \mathbf{od}, s'_1\rangle \rightsquigarrow s'_f \tag{3.7}$$

Note that (3.4) follows from (3.3) by Lemma 3.15, and that justifies (3.6). From (3.2), (3.7) and (3.6), by IH, we get $\forall x \in \mathbf{Var}.\ s_f(x) = s'_f(\widehat{\mathcal{V}'}(x))$.

□

It can now be proved that the $\mathfrak{T}$ translation preserves the operational semantics of the original program.

Proposition 3.17. *Let* $C \in \mathbf{AComm}$*,* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$*,* $s_i, s_f \in \Sigma$*,* $s', s'_f \in \Sigma^{\mathrm{ISA}}$ *and* $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', C')$*. If* $\langle \lfloor C \rfloor, s_i \rangle \rightsquigarrow s_f$ *and* $\langle \lfloor \mathcal{W}(C') \rfloor, s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f$*, then* $\forall x \in \mathbf{Var}.\ s_f(x) = s'_f(\widehat{\mathcal{V}'}(x))$*.*

Proof. By induction on the structure of C.

- Case $C \equiv \mathbf{skip}$: $\mathfrak{T}(\mathcal{V}, \mathbf{skip}) = (\mathcal{V}, \mathbf{skip})$, $\langle \lfloor \mathbf{skip} \rfloor, s_i \rangle \rightsquigarrow s_i$, and $\langle \lfloor \mathcal{W}(\mathbf{skip}) \rfloor, s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s' \oplus \mathcal{V}(s_i)$. As for every $x \in \mathbf{Var}$, $(s' \oplus \mathcal{V}(s_i))(\widehat{\mathcal{V}}(x)) = s_i(x)$ we are done.

- Case $C \equiv x := e$. The hypotheses are:

$$\begin{aligned}
&\mathfrak{T}(\mathcal{V}, x := e) = (\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))], x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e)) \\
&\langle \lfloor x := e \rfloor, s_i \rangle \rightsquigarrow s_f \quad \text{with} \quad s_f = s_i[x \mapsto [\![e]\!](s_i)] \\
&\langle \lfloor \mathcal{W}(x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e)) \rfloor, s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f \quad \text{with} \\
&s'_f = (s' \oplus \mathcal{V}(s_i))[x_{\mathsf{next}(\mathcal{V}(x))} \mapsto [\![\widehat{\mathcal{V}}(e)]\!](s' \oplus \mathcal{V}(s_i))]
\end{aligned}$$

  We want to prove that $\forall y \in \mathbf{Var}.\ s_f(y) = s'_f(\widehat{\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))]}(y))$. Two cases can occur:

  - If $y = x$, we have $s_f(y) = s_f(x) = [\![e]\!](s_i)$ and $s'_f(\widehat{\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))]}(x)) = s'_f(x_{\mathsf{next}(\mathcal{V}(x))}) = [\![\widehat{\mathcal{V}}(e)]\!](s' \oplus \mathcal{V}(s_i)) = [\![e]\!](s_i)$.
  - If $y \neq x$, we have $s_f(y) = s_i(y)$ and $s'_f(\widehat{\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))]}(y)) = s'_f(\widehat{\mathcal{V}}(y)) = (s' \oplus \mathcal{V}(s_i))(\widehat{\mathcal{V}}(y)) = s_i(y)$.

- Case $C \equiv C_1; C_2$. The hypotheses are:

$$\begin{aligned}
&\mathfrak{T}(\mathcal{V}, C_1; C_2) = (\mathcal{V}'', C'_1; C'_2) \text{ with } \mathfrak{T}(\mathcal{V}, C_1) = (\mathcal{V}', C'_1) \text{ and } \mathfrak{T}(\mathcal{V}', C_2) = (\mathcal{V}'', C'_2) \\
&\langle \lfloor C_1; C_2 \rfloor, s_i \rangle \rightsquigarrow s_f \\
&\langle \lfloor \mathcal{W}(C'_1; C'_2) \rfloor, s \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f
\end{aligned}$$

  We must have, for some $s_0 \in \Sigma$ and $s'_0 \in \Sigma^{\mathrm{ISA}}$

$$\langle \lfloor C_1 \rfloor, s_i \rangle \rightsquigarrow s_0 \tag{3.8}$$

$$\langle \lfloor C_2 \rfloor, s_0 \rangle \rightsquigarrow s_f \tag{3.9}$$

$$\langle \lfloor \mathcal{W}(C'_1) \rfloor, s \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_0 \tag{3.10}$$

$$\langle \lfloor \mathcal{W}(C'_2) \rfloor, s'_0 \rangle \rightsquigarrow s'_f \tag{3.11}$$

From (3.8) and (3.10), by IG, we have $\forall\, x \in \mathbf{Var}.\ s_0(x) = s'_0(\widehat{\mathcal{V}'}(x))$. Therefore, by Lemma 3.13, we have $s'_0 = s'_1 \oplus \mathcal{V}'(s_0)$ for some $s'_1 \in \Sigma^{\mathrm{ISA}}$. Consequently, from (3.9) and (3.11), by IH, we conclude that $\forall\, x \in \mathbf{Var}.\ s_f(x) = s'_f(\widehat{\mathcal{V}''}(x))$.

- Case $C \equiv \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \mathbf{fi}$. The hypotheses are:

$$\mathfrak{T}(\mathcal{V}, C) = (\mathsf{sup}(\mathcal{V}', \mathcal{V}''), \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then}\ \{C'_t; \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\}\ \mathbf{else}\ \{C'_f; \mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\}\ \mathbf{fi}\ )$$
$$\text{with } \mathfrak{T}(\mathcal{V}, C_t) = (\mathcal{V}', C'_t) \text{ and } \mathfrak{T}(\mathcal{V}, C_f) = (\mathcal{V}'', C'_f)$$
$$\langle \lfloor \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \mathbf{fi} \rfloor, s_i \rangle \leadsto s_f$$
$$\langle \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then}\ \{C'_t; \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\}\ \mathbf{else}\ \{C'_f; \mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\}\ \mathbf{fi}, s' \oplus \mathcal{V}(s_i) \rangle \leadsto s'_f$$

  - Case $[\![b]\!](s_i) = \top$, then by Lemma 3.14 $[\![\widehat{\mathcal{V}}(b)]\!](s' \oplus \mathcal{V}(s_i)) = \top$. Therefore one must have, for some $s'_0 \in \Sigma^{\mathrm{ISA}}$,

$$\langle \lfloor C_t \rfloor, s_i \rangle \leadsto s_f \tag{3.12}$$
$$\langle \lfloor \mathcal{W}(C'_t) \rfloor, s' \oplus \mathcal{V}(s_i) \rangle \leadsto s'_0 \tag{3.13}$$
$$\langle \mathsf{mrg}(\mathcal{V}', \mathcal{V}''), s'_0 \rangle \leadsto s'_f \tag{3.14}$$

    From (3.12) and (3.13), by IH we have that

$$\forall\, x \in \mathbf{Var}.\ s_f(x) = s'_0(\widehat{\mathcal{V}'}(x)) \tag{3.15}$$

    We also have that $\mathsf{mrg}\ (\mathcal{V}', \mathcal{V}'') = [x_{\mathcal{V}''(x)} := x_{\mathcal{V}'(x)} \mid x \in \mathbf{Var} \wedge \mathcal{V}'(x) \prec \mathcal{V}''(x)]$ so, $s'_f = s'_0[x_{\mathcal{V}''(x)} \mapsto [\![x_{\mathcal{V}'(x)}]\!](s'_0) \mid x \in \mathbf{Var} \wedge \mathcal{V}'(x) \prec \mathcal{V}''(x)]$. Moreover, $\mathsf{sup}\ (\mathcal{V}', \mathcal{V}'')(x)$ is $\mathcal{V}'(x)$ if $\mathcal{V}''(x) \prec \mathcal{V}'(x)$ or $\mathcal{V}''(x)$ otherwise.

    We will now prove that $\forall\, x \in \mathbf{Var}.\ s_f(x) = s'_f(\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(x))$. Let $x \in \mathbf{Var}$.

    * If $\mathcal{V}'(x) \prec \mathcal{V}''(x)$, then $s'_f(\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(x)) = s'_f(\widehat{\mathcal{V}''}(x)) = [\![x_{\mathcal{V}'(x)}]\!](s'_0) = s'_0(\widehat{\mathcal{V}'}(x)) = s_f(x)$ by (3.15).
    * If $\mathcal{V}'(x) \not\prec \mathcal{V}''(x)$, then $s'_f(\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(x)) = s'_f(\widehat{\mathcal{V}'}(x)) = s'_0(\widehat{\mathcal{V}'}(x)) = s_f(x)$ by (3.15).

  - Case $[\![b]\!](s_i) = \bot$. Analogous to the previous case.

- Case $C \equiv \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C_t\ \mathbf{od}$. The hypotheses are:

$$\mathfrak{T}(\mathcal{V}, \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C_y\ \mathbf{od}) = (\mathcal{V}''', \mathbf{for}\ (\mathcal{I}, \widehat{\mathcal{V}'}(b), \mathcal{U})\ \mathbf{do}\ \{\widehat{\mathcal{V}'}(\theta)\}\ C'_t\ \mathbf{od}) \tag{3.16}$$
$$\text{with } \mathcal{I} = [x_{\mathsf{new}(\mathcal{V}(x))} | x \in \mathsf{Asgn}(C_t)] \tag{3.17}$$
$$\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C_t)] \tag{3.18}$$
$$(\mathcal{V}'', C'_t) = \mathfrak{T}(\mathcal{V}', C_t) \tag{3.19}$$
$$\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C_t)] \tag{3.20}$$
$$\mathcal{V}''' = \mathcal{V}''[x \mapsto \mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})] \tag{3.21}$$

$$\langle \textbf{while } b \textbf{ do } \lfloor C_t \rfloor \textbf{ od}, s_i \rangle \rightsquigarrow s_f \tag{3.22}$$

$$\langle \mathcal{I}; \textbf{while } \widehat{\mathcal{V}'(b)} \textbf{ do } \lfloor \mathcal{W}(C_t') \rfloor; \mathcal{U} \textbf{ od}; \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s_f' \tag{3.23}$$

From (3.17), (3.18) and (3.23), we must have for some $s_0', s_1' \in \Sigma^{\text{ISA}}$ that:

$$\langle \mathcal{I}, s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s_1'$$
$$\langle \textbf{while } \widehat{\mathcal{V}'(b)} \textbf{ do } \lfloor \mathcal{W}(C_t') \rfloor; \mathcal{U} \textbf{ od}; \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s_1' \rangle \rightsquigarrow s_f' \tag{3.24}$$
$$s_1' = s_0' \oplus \mathcal{V}'(s_i)$$

There are the following cases to consider:

- Case $[\![b]\!](s_i) = \bot$, then $s_f = s_i$ and $[\![\widehat{\mathcal{V}}'(b)]\!](s' \oplus \mathcal{V}'(s_i)) = \bot$, by Lemma 3.14. Therefore one must have $\langle \textbf{while } b \textbf{ do } \lfloor C_t \rfloor \textbf{ od}, s_i \rangle \rightsquigarrow s_i$, $\langle \textbf{while } \mathcal{V}'(b) \textbf{ do } \lfloor \mathcal{W}(C_t') \rfloor; \mathcal{U} \textbf{ od}, s_1' \rangle \rightsquigarrow s_1'$, and $\langle \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s_1' \rangle \rightsquigarrow s_f'$. Moreover, we know that $\mathsf{upd}(\mathsf{dom}(\mathcal{U})) = [x_{\mathsf{jump}(l)} := x_l \mid x_l \in \mathsf{dom}(\mathcal{U})]$ so:

$$\langle \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s_0' \oplus \mathcal{V}'(s_i) \rangle \rightsquigarrow s_f' \tag{3.25}$$
$$s_f' = (s_0' \oplus \mathcal{V}'(s_i))[x_{\mathsf{jump}(l)} \mapsto [\![x_l]\!](s_0' \oplus \mathcal{V}'(s_i)) \mid x_l \in \mathsf{dom}(\mathcal{U})] \tag{3.26}$$

  We now prove that $\forall\, x \in \textbf{Var}.\ s_f(x) = s_f'(\widehat{\mathcal{V}}'(x))$.

  * If $y \notin \mathsf{Asgn}(C_t)$, then $s_f'(\widehat{\mathcal{V}'''}(y)) = s_f'(\widehat{\mathcal{V}''}(y)) = s_f'(\widehat{\mathcal{V}}'(y)) = s_i(y)$, using Lemma 3.15.
  * If $x \in \mathsf{Asgn}(C_t)$, then $s_f'(\widehat{\mathcal{V}'''}(x)) = s_f'(x_{\mathsf{jump}(l)})$ for some $x_l \in \mathsf{dom}(\mathcal{U})$ and $s_f'(x_{\mathsf{jump}(l)}) = [\![x_l]\!](s_0' \oplus \mathcal{V}'(s_i)) = [\![\widehat{\mathcal{V}}'(x)]\!](s_0' \oplus \mathcal{V}'(s_i)) = [\![x]\!](s_i) = s_i(x)$, using Lemma 3.13.

- Case $[\![b]\!](s_i) = \top$, then $[\![\widehat{\mathcal{V}}'(b)]\!](s_0' \oplus \mathcal{V}'(s_i)) = \top$, by Lemma 3.14. From (3.22), we must have, for some $s_2 \in \Sigma$,

$$\langle \lfloor C_t \rfloor, s_i \rangle \rightsquigarrow s_2 \tag{3.27}$$
$$\langle \textbf{while } b \textbf{ do } \lfloor C_t \rfloor \textbf{ od}, s_2 \rangle \rightsquigarrow s_f \tag{3.28}$$

  and also, from (3.24), for some $s_2', s_3', s_4' \in \Sigma^{\text{ISA}}$,

$$\langle \lfloor \mathcal{W}(C_t') \rfloor, s_0' \oplus \mathcal{V}'(s_i) \rangle \rightsquigarrow s_4' \tag{3.29}$$
$$\langle \mathcal{U}, s_4' \rangle \rightsquigarrow s_2' \tag{3.30}$$
$$s_2' = s_4'[x_{\mathsf{new}(x_{\mathcal{V}(x)})} \mapsto [\![x_{\mathcal{V}''(x)}]\!](s_4') \mid x \in \mathsf{Asgn}(C_t)] \tag{3.31}$$
$$\langle \textbf{while } \widehat{\mathcal{V}}'(b) \textbf{ do } \lfloor \mathcal{W}(C_t') \rfloor; \mathcal{U} \textbf{ od}, s_2' \rangle \rightsquigarrow s_3' \tag{3.32}$$
$$\langle \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s_3' \rangle \rightsquigarrow s_f' \tag{3.33}$$

  From (3.27), (3.29) and (3.19), by IH, we have that $\forall\, x \in \textbf{Var}.\ s_2(x) = s_4'(\widehat{\mathcal{V}''}(x))$.

Using this fact, (3.31) and (3.18), we can conclude that $s'_2 = s'_4 \oplus \mathcal{V}'(s_2)$. Because of this, (3.28) and (3.32), we get by Lemma 3.16

$$\forall x \in \mathbf{Var}.\ s_f(x) = s'_3(\widehat{\mathcal{V}'}(x)) \tag{3.34}$$

$\mathsf{upd}(\mathsf{dom}(\mathcal{U})) = [x_{\mathsf{jump}(l)} := x_l \mid x_l \in \mathsf{dom}(\mathcal{U})]$ and $\langle \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s'_3\rangle \rightsquigarrow s'_f$, so $s'_f = s'_3[x_{\mathsf{jump}(l)} \mapsto [\![x_l]\!](s'_3) \mid x_l \in \mathsf{dom}(\mathcal{U})]$. We will now prove that $\forall x \in \mathbf{Var}.\ s_f(x) = s'_f(\widehat{\mathcal{V}'''}(x))$.

* If $y \notin \mathsf{Asgn}(C_t)$, then $s'_f(\widehat{\mathcal{V}'''}(y)) = s'_f(\widehat{\mathcal{V}''}(y)) = s'_3(\widehat{\mathcal{V}'}(y)) = s_f(y)$, using Lemma 3.15 and (3.34).
* If $x \in \mathsf{Asgn}(C_t)$, then $s'_f(\widehat{\mathcal{V}'''}(x)) = s'_f(x_{\mathsf{jump}(l)})$ for some $x_l = x_{\mathsf{new}(\mathcal{V}(c)}\in \mathsf{dom}(\mathcal{U})$, and $s'_f(x_{\mathsf{jump}(l)}) = [\![x_l]\!](s'_3) = s_f(x)$, using (3.34).

□

The previous lemma establishes basically that if a program starts executing in a certain state, for instance s_i, and its translation starts in any state where the initial versions of the variables agree with s_i, then both programs will terminate and the final states will agree on the final version of the variables. Before establishing that the translation is sound for ISA triples an additional lemma that is similar to Lemma 3.13 must be considered. It basically says that the final version of the variables can be isolated in a partial function.

Lemma 3.18. *Let $C \in \mathbf{AComm}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, and $s_i, s_f \in \Sigma$. If $\langle \lfloor C \rfloor, s_i\rangle \rightsquigarrow s_f$ and $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', C')$, then $\langle \lfloor \mathcal{W}(C')\rfloor, s'_1 \oplus \mathcal{V}(s_i)\rangle \rightsquigarrow s'_2 \oplus \mathcal{V}'(s_f)$, for some $s'_1, s'_2 \in \Sigma^{\mathrm{ISA}}$.*

Proof. By induction on the structure of C using Proposition 3.17. □

It can now be proved that the translation function is sound, in the sense that if the translated triple is valid then the original triple is also valid.

Proposition 3.19. *Let $C \in \mathbf{AComm}$, $\phi, \psi \in \mathbf{Assert}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$ and $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', C')$. If $\models \{\widehat{\mathcal{V}}(\phi)\}\, \lfloor \mathcal{W}(C')\rfloor\, \{\widehat{\mathcal{V}'}(\psi)\}$, then $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$.*

Proof. Let $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', C')$ and assume that $\models \{\widehat{\mathcal{V}}(\phi)\}\, \lfloor \mathcal{W}(C')\rfloor\, \{\widehat{\mathcal{V}'}(\psi)\}$ holds. Assume also that for some $s_i, s_f \in \Sigma$, $[\![\phi]\!](s_i) = \top$ and $\langle \lfloor C \rfloor, s_i\rangle \rightsquigarrow s_f$. By Lemma 3.18, we have for some $s'_1, s'_2 \in \Sigma^{\mathrm{ISA}}$, that $\langle \lfloor \mathcal{W}(C')\rfloor, s'_1 \oplus \mathcal{V}(s_i)\rangle \rightsquigarrow s'_2 \oplus \mathcal{V}'(s_f)$. By Lemma 3.14, we have $[\![\widehat{\mathcal{V}}(\phi)]\!](s'_1 \oplus \mathcal{V}(s_i)) = \top$ and thus $[\![\widehat{\mathcal{V}'}(\psi)]\!](s'_2 \oplus \mathcal{V}'(s_f)) = \top$. Again, by Lemma 3.14, it follows that $[\![\psi]\!](s_f) = \top$. Hence $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$ holds. □

Finally it will be shown that the translation $\mathfrak{T}$ preserves Hg-derivations, in the sense that if a Hoare triple is derivable in Hg, then the translated triple is also derivable in Hg. Some additional lemmas will be useful to help us reason about the auxiliary functions and the renamings.

Lemma 3.20. *Let $\mathcal{V}, \mathcal{V}' \in \mathbf{Var} \to \mathbb{N}^+$, $\psi \in \mathbf{Assert}$, and $\mathcal{I} \in \mathbf{Rnm}$. The following derivations hold:*

1. $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\psi)\}\,\mathsf{mrg}(\mathcal{V},\mathcal{V}')\,\{\widehat{\mathsf{sup}(\mathcal{V},\mathcal{V}')}(\psi)\}$.

2. $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\psi)\}\,\mathsf{mrg}(\mathcal{V}',\mathcal{V})\,\{\widehat{\mathsf{sup}(\mathcal{V},\mathcal{V}')}(\psi)\}$.

3. $\vdash_{\mathsf{Hg}} \{\psi\}\,\mathcal{I}\,\{\mathcal{I}^{-1}(\psi)\}$.

Proof. 1. We have $\mathsf{mrg}\ (\mathcal{V},\mathcal{V}') = [x_{\mathcal{V}'(x)} := x_{\mathcal{V}(x)} \mid x \in \mathbf{Var} \wedge \mathcal{V}(x) \prec \mathcal{V}'(x)]$ and $\mathsf{sup}(\mathcal{V},\mathcal{V}')(x) = \mathcal{V}(x)$ if $\mathcal{V}'(x) \prec \mathcal{V}(x)$ or $\mathcal{V}'(x)$ otherwise. The validity of the triple $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\psi)\}\,\mathsf{mrg}(\mathcal{V},\mathcal{V}')\,\{\widehat{\mathsf{sup}(\mathcal{V},\mathcal{V}')}(\psi)\}$ follows from successively applying the (assign) and (seq) rules, using as precondition the postcondition with the substitution $[x_{\mathcal{V}(x)}/x_{\mathcal{V}'(x)}]$ for each assigment $x_{\mathcal{V}'(x)} := x_{\mathcal{V}(x)}$ of the renaming sequence, since $\widehat{\mathsf{sup}(\mathcal{V},\mathcal{V}')}(\psi)[x_{\mathcal{V}(x)}/x_{\mathcal{V}'(x)} \mid x \in \mathbf{Var} \wedge \mathcal{V}(x) \prec \mathcal{V}'(x)] = \widehat{\mathcal{V}}(\psi)$.

2. Analogous to 1.

3. Follows directly from Lemma 2.13. □

Lemma 3.21. *Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $C \in \mathbf{AComm}$, $\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)]$, $\mathfrak{T}(\mathcal{V}',C) = (\mathcal{V}'',C')$ *and* $\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)]$. *Then* $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}''}(\theta)\}\,\mathcal{U}\,\{\widehat{\mathcal{V}'}(\theta)\}$.

Proof. The derivability of the triple $\{\widehat{\mathcal{V}''}(\theta)\}\,\mathcal{U}\,\{\widehat{\mathcal{V}'}(\theta)\}$ in Hg follows from successively applying the (assign) and (seq) rules, using as precondition the postcondition with the substitution $[x_{\mathcal{V}''(x)}/x_{\mathsf{new}(\mathcal{V}(x))}]$ for each assigment $x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)}$ of the renaming sequence, since

$$\begin{aligned}&(\widehat{\mathcal{V}'}(\theta))[x_{\mathcal{V}''(x)}/x_{\mathsf{new}(\mathcal{V}(x))} \mid x \in \mathsf{Asgn}(C)]\\ &= (\mathcal{V}[x \mapsto \widehat{\mathsf{new}(\mathcal{V}(x)) \mid} x \in \mathsf{Asgn}(C)](\theta))[x_{\mathcal{V}''(x)}/x_{\mathsf{new}(\mathcal{V}(x))} \mid x \in \mathsf{Asgn}(C)]\\ &= \widehat{\mathcal{V}''}(\theta)\end{aligned}$$

□

Lemma 3.22. *Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $C \in \mathbf{AComm}$, $\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)]$, $\mathfrak{T}(\mathcal{V}',C) = (\mathcal{V}'',C')$, $\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)]$ *and* $\mathcal{V}''' = \mathcal{V}''[x \mapsto \mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})]$. *Then* $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\psi)\}\,\mathsf{upd}(\mathsf{dom}(\mathcal{U}))\,\{\widehat{\mathcal{V}'''}(\psi)\}$.

Proof. Recall that $\mathsf{upd}\ (\mathsf{dom}(\mathcal{U})) = [x_{\mathsf{jump}(l)} := x_l \mid x_l \in \mathsf{dom}(\mathcal{U})]$. Then, the validity of $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\psi)\}\,\mathsf{upd}(\mathsf{dom}(\mathcal{U}))\,\{\widehat{\mathcal{V}'''}(\psi)\}$ follows from successively applying the (assign) and (seq) rules, using as precondition the postcondition with the substitution $[x_l/x_{\mathsf{jump}(l)}]$ for each assigment $x_{\mathsf{jump}(l)} := x_l$ of the renaming sequence, since $(\widehat{\mathcal{V}'''}(\psi)[x_l/x_{\mathsf{jump}(l)} \mid x_l \in \mathsf{dom}(\mathcal{U})]) = (\mathcal{V}''[x \mapsto \widehat{\mathsf{jump}(l) \mid} x_l \in \mathsf{dom}(\mathcal{U})](\psi)[x_l/x_{\mathsf{jump}(l)} \mid x_l \in \mathsf{dom}(\mathcal{U})]) = \mathcal{V}'(\psi)$. □

It can now be proved that the translation is complete, in the sense that if a triple is derivable in system Hg then its translation is also derivable in system Hg.

Proposition 3.23. *Let* $C \in \mathbf{AComm}$, $\phi,\psi \in \mathbf{Assert}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$ *and* $\mathfrak{T}(\mathcal{V},C) = (\mathcal{V}',C')$. *If* $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$, *then* $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\,\mathcal{W}(C')\,\{\widehat{\mathcal{V}'}(\psi)\}$.

Proof. By induction on the structure of $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$. The proof is straightforward if the last step is (skip) and (seq).

- Assume the last step is $\dfrac{}{\{\phi\}\, x := e\, \{\psi\}}$ with $\phi \to \psi[e/x]$, and $\mathfrak{T}(\mathcal{V}, x := e) = (\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))], x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e))$. Since $\models \phi \to \psi[e/x]$, it follows that $\models \widehat{\mathcal{V}}(\phi \to \psi[e/x])$. Moreover,

$$\begin{aligned}
&\widehat{\mathcal{V}}(\phi) \to \widehat{\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))]}(\psi)[\widehat{\mathcal{V}}(e)/x_{\mathsf{next}(\mathcal{V}(x))}] \\
&= \widehat{\mathcal{V}}(\phi) \to \widehat{\mathcal{V}}(\psi[x_{\mathsf{next}(\mathcal{V}(x))}/x])[\widehat{\mathcal{V}}(e)/x_{\mathsf{next}(\mathcal{V}(x))}] \\
&= \widehat{\mathcal{V}}(\phi) \to \widehat{\mathcal{V}}(\psi[x_{\mathsf{next}(\mathcal{V}(x))}/x][e/x_{\mathsf{next}(\mathcal{V}(x))}]) \quad \text{, because } x_{\mathsf{next}(\mathcal{V}(x))} \notin \mathsf{FV}(\psi) \\
&= \widehat{\mathcal{V}}(\phi) \to \widehat{\mathcal{V}}(\psi[e/x]) \\
&= \widehat{\mathcal{V}}(\phi \to \psi[e/x])
\end{aligned}$$

Hence $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e)\, \{\widehat{\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))]}(\psi)\}$ follows from the (assign) rule.

- Assume the last step is $\dfrac{\{\phi \wedge b\}\, C_t\, \{\psi\} \qquad \{\phi \wedge \neg b\}\, C_f\, \{\psi\}}{\{\phi\}\, \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \mathbf{fi}\, \{\psi\}}$, and $\mathfrak{T}(\mathcal{V}, C) = (\mathsf{sup}(\mathcal{V}', \mathcal{V}''),$ $\mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then}\ \{C_t'; \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\}\ \mathbf{else}\ \{C_f'; \mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\}\ \mathbf{fi}\,)$ with $\mathfrak{T}(\mathcal{V}, C_t) = (\mathcal{V}', C_t')$ and $\mathfrak{T}(\mathcal{V}, C_f) = (\mathcal{V}'', C_f')$.

  By induction hypothesis we have $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi \wedge b)\}\, \mathcal{W}(C_t')\, \{\widehat{\mathcal{V}'}(\psi)\}$. By Lemma 3.20 we have $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\psi)\}\, \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\, \{\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(\psi)\}$. So, applying the (seq) rule, we get $\vdash_{\mathsf{Hg}}$ $\{\widehat{\mathcal{V}}(\phi \wedge b)\}\, \mathcal{W}(C_t')\, ;\, \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\, \{\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(\psi)\}$. Similarly we can obtain $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi \wedge \neg b)\}\, \mathcal{W}(C_f')\, ;\, \mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\, \{\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(\psi)\}$.

  From the above, using (if) rule we get $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then}\ \{\mathcal{W}(C_t'); \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\}$ $\mathbf{else}\ \{\mathcal{W}(C_f'); \mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\}\, \{\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(\psi))\}$.

- Assume the last step is $\dfrac{\{\theta \wedge b\}\, C\, \{\theta\}}{\{\phi\}\, \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \mathbf{od}\, \{\psi\}}$ with $\phi \to \theta$ and $\theta \wedge \neg b \to \psi$, and also that

$$\begin{aligned}
\mathfrak{T}(\mathcal{V}, \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \mathbf{od}) = (\mathcal{V}''', \mathbf{for}\ (\mathcal{I}, \widehat{\mathcal{V}'}(b), \mathcal{U})\ \mathbf{do}\, \{\widehat{\mathcal{V}'}(\theta)\}\, C'\ \mathbf{od}; \mathsf{upd}(\mathsf{dom}(\mathcal{U})))& \\
\text{with } \mathcal{I} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}(x)} \mid x \in \mathsf{Asgn}(C)]& \\
\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)]& \\
(\mathcal{V}'', C') = \mathfrak{T}(\mathcal{V}', C)& \\
\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)]& \\
\mathcal{V}''' = \mathcal{V}''[x \mapsto \mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})]&
\end{aligned}$$

$$\begin{aligned}
\text{and}\quad &\mathcal{W}(\mathbf{for}\ (\mathcal{I}, \widehat{\mathcal{V}'}(b), \mathcal{U})\ \mathbf{do}\, \{\widehat{\mathcal{V}'}(\theta)\}\, C'\ \mathbf{od}\, ;\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))) \\
&= \mathcal{I}\, ;\, \mathbf{while}\ \widehat{\mathcal{V}'}(b)\ \mathbf{do}\ \{\widehat{\mathcal{V}'}(\theta)\}\ \mathcal{W}(C')\, ;\, \mathcal{U}\ \mathbf{od}\, ;\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))
\end{aligned}$$

  We must prove that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathcal{I}\, ;\, \mathbf{while}\ \widehat{\mathcal{V}'}(b)\ \mathbf{do}\ \{\widehat{\mathcal{V}'}(\theta)\}\ \mathcal{W}(C')\, ;\, \mathcal{U}\ \mathbf{od}\, ;\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))$

$\{\widehat{\mathcal{V}'''}(\psi)\}$, which follows from applying rule (seq) twice, to the following premisses:

$$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathcal{I}\, \{\mathcal{I}^{-1}(\widehat{\mathcal{V}}(\phi))\} \tag{3.35}$$

$$\vdash_{\mathsf{Hg}} \{\mathcal{I}^{-1}(\widehat{\mathcal{V}}(\phi))\}\, \textbf{while}\ \widehat{\mathcal{V}'}(b)\ \textbf{do}\ \{\widehat{\mathcal{V}'}(\theta)\}\ \mathcal{W}(C')\,;\mathcal{U}\ \textbf{od}\, \{\widehat{\mathcal{V}'}(\psi)\} \tag{3.36}$$

$$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\psi)\}\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))\, \{\widehat{\mathcal{V}'''}(\psi)\} \tag{3.37}$$

The derivation of the triple in (3.35) follows from Lemma 3.20, and the triple in (3.37) from Lemma 3.22. By IH, we have $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\theta) \wedge \widehat{\mathcal{V}'}(b)\}\, \mathcal{W}(C')\, \{\widehat{\mathcal{V}''}(\theta)\}$ and by Lemma 3.21 we also have $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}''}(\theta)\}\, \mathcal{U}\, \{\widehat{\mathcal{V}'}(\theta)\}$. Thus, by rule (seq), $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\theta) \wedge \widehat{\mathcal{V}'}(b)\}\, \mathcal{W}(C')\,;\mathcal{U}\, \{\widehat{\mathcal{V}'}(\theta)\}$. Since $(\mathcal{I}^{-1}(\widehat{\mathcal{V}}(\phi)) \rightarrow \widehat{\mathcal{V}'}(\theta))$ and $(\widehat{\mathcal{V}'}(\theta) \wedge \neg\widehat{\mathcal{V}'}(b) \rightarrow \widehat{\mathcal{V}'}(\psi))$ both hold, we can now apply rule (while), and obtain $\vdash_{\mathsf{Hg}} \{\mathcal{I}^{-1}(\widehat{\mathcal{V}}(\phi))\}\, \textbf{while}\ \widehat{\mathcal{V}'}(b)\ \textbf{do}\ \{\widehat{\mathcal{V}'}(\theta)\}\ \mathcal{W}(C')\,;\mathcal{U}\ \textbf{od}\, \{\widehat{\mathcal{V}'}(\psi)\}$.

□

Proposition 3.24. *The $\mathfrak{T}$ function from Definition 3.6 is an ISA translation in the sense of Definition 3.5.*

Proof. Let $C \in \textbf{AComm}$, $\phi, \psi \in \textbf{Assert}$, $\mathcal{V} \in \textbf{Var} \rightarrow \mathbb{N}^+$, and $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', C')$. It follows from Proposition 3.19 that if $\models \{\widehat{\mathcal{V}}(\phi)\}\, \lfloor \mathcal{W}(C') \rfloor\, \{\widehat{\mathcal{V}'}(\psi)\}$ then $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$, and from Proposition 3.23 that if $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$ then $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathcal{W}(C')\, \{\widehat{\mathcal{V}'}(\psi)\}$. □

3.6 Related Work

As explained in Chapter 2, the original notion of static single-assignment (SSA) form [40] limits the syntactic occurrence of each variable as L-value of a single-assignment instruction. On the other hand in dynamic single-assignment (DSA) form [102] variables may occur as L-value in multiple assignments in different execution paths. Abstracting from the fact that assignments are replaced by assume statements, the original notion of *passive form* of [52] can be seen as a kind of dynamic SA where assignment instructions are replaced by *assume* commands, but similarly to the static notion, variable synchronization is achieved by introducing fresh variables assigned in both branches. Adapting this to standard imperative syntax, the fragment **if** $x > 0$ **then** $x := x + 10$ **else** $x := x + 20$ **fi** would be translated as **if** $x_0 > 0$ **then** $x_1 := x_0 + 10\,; x_3 := x_1$ **else** $x_2 := x_0 + 20\,; x_3 := x_2$ **fi**. Our translation resembles the *passify* function introduced in [52], but there are significant differences: *passify* generates fresh variables abstractly, whereas we provide a concrete mechanism for this purpose; while *passify* only handles loop-free programs, our translation considers programs with loops annotated with invariants; *passify* is proved to be sound in the sense that it preserves the weakest precondition interpretation of programs, while our translation is proved to be sound with respect to the validity of Hoare triples, and moreover it is shown to be complete since it preserves derivability guided by the invariants. This is a crucial issue from the point of view of the completeness of using an intermediate ISA form for verification.

Finally, *passify* does not generate version-optimal programs; the notion of *version-optimal* passive form, which uses the minimum number of version variables, is defined for unstructured programs in [60], together with a translation algorithm. In this form the above fragment becomes simply **if** $x_0 > 0$ **then** $x_1 := x_0 + 10$ **else** $x_1 := x_0 + 20$ **fi**. The algorithm differs from the translation of Section 3.5 in that it does not contemplate annotated loops, and no proof of soundness is given. However, they are similar in the use of variables: for loop-free programs, the DSA form produced by our translation is version-optimal.

In terms of verification single-assignment form has played an important role in two different families of efficient program verification techniques: in the line of work that was originated by Flanagan et. al. [52], and that had a great impact on the efficiency of deductive verification methods; and in bounded model checking as an intermediate step towards reaching a logical formula.

With respect to the former, where the idea is to use predicate transformers to reason about programs, there exists a single semantics of guarded commands, given by the definition of the predicate transformers, from which a VCGen is directly derived. This stands in contrast to our approach: soundness and relative completeness of the logic and VCGen are established with respect to a standard operational semantics of *While* programs. A second important difference is the treatment of iteration. The fixpoint definition of predicate transformers for iterating commands are of no use in the verification of programs annotated with loop invariants, and on the other hand iteration is not directly supported in a dynamic SA setting, since it is not allowed for the same variable to be assigned more than once. The approach used in ESC/Java and Boogie has been to convert each program into an iteration-free program (see the end of Section 2.4), such that the verification conditions generated for the latter guarantee the soundness of the initial program. We should start by noting that a soundness proof of this approach is given for the first time later in this thesis (see Chapter 6). Our work in the current chapter differs from this in that loops are part of the intermediate SA language and are translated into this language together with their invariants. The soundness and completeness properties are established based on the standard semantics of iteration. Annotated programs are treated explicitly, and the notion of correctly-annotated program is introduced, which allows us to distinguish between incorrect programs and correct programs containing 'wrong' invariants.

Chapter 4

Verification Conditions for SA Programs with Assume, Assert, and Exceptions

In the previous chapter we addressed program verification from a traditional point of view. Knowing that SA has a clear benefit on the generation of VCs, we proposed the notion of iterating SA language and a translation of While programs into such a language. A framework to verify this new category of programs was presented, and the complete workflow was shown to be sound and relatively complete. It was also shown that considering such a language has multiple benefits in terms of program verification.

In the second part of the thesis, starting with the present chapter, we address program verification from a different perspective. Instead of proposing a new workflow with a justified theoretical background, we capture what verification tools based on the generation of verification conditions typically do, and establish theoretical results from that. In particular we try to bring together the notions from deductive verification and bounded model checking of software and to close the gap between the theoretical foundations of program verification and the state of the art verification tools. This chapter explores the algorithms that are typically used by tools to generate verification conditions. We leave all the theoretical results to the next chapter, where we will establish a bridge to the theoretical foundations of program verification, and focus for now on the details of the VCGens. In particular, we identify interesting trends on the generation of verification conditions, mix them together to create new VCGens, and organize them into a *cube of VCGens*.

Aiming at capturing the essence of verification tools, as a source programming language, we will initially consider a While language extended with assume and asserts as typically found in program verification tools and described in Sections 2.4 and 2.6. As an intermediate language, a notion of SA language (without iteration) will be used, and trying to mimic the tools' workflow, we will assume that loops are captured in this intermediate SA language as described at the end of Section 2.4. Although While programs simply extended with assumes and asserts are

already enough to motivate the creation of a cube of VCGens, we later extend the language with exceptions and study how they influence the previous analysis.

The chapter is organized as follows. Section 4.1 motivates our study of VCGens. It analyses the state of the art VCGens for branching SA programs with assume and assert commands, and identifies three aspects in which the methods differ: logical encoding of control flow, use of contexts and the semantics of assert statements. These aspects are then combined to produce what we call a *VCGen cube* in Section 4.2. The VCGens from the cube are compared asymptotically and then optimized in Section 4.3. The chapter proceeds by extending the VCGens for programs with exceptions (Section 4.4) and then by presenting a unified VCGen that agglomerates all the VCGens previously presented. Finally, the last section presents related work.

4.1 Verification Conditions for SA Programs with Asserts

Similarly to the beginning of Chapter 3 the programs we will consider in this chapter will be assumed to be already in SA form, not containing iteration. The SA program can be written directly by a programmer, or else, obtained by a translation that takes a non SA program and generates an SA program. Recall that an SA program is any program that does not assign a variable more than once during an execution, and that once a variable is read it cannot be assigned.

As in the guarded commands, the idea of the statement $\textbf{assert}\,\theta$ as a specification mechanism is that it produces an error if θ is false in the present state. A program is correct if *no error may occur in any execution*. For example, the fragment $x_1 := x_0 + 10\,;\,\textbf{assert}\,x_1 > 10$ admits erroneous executions (for initial values $x_0 \leq 0$) and normal executions (for $x_0 > 0$). The fragment $x_1 := x_0 + 10\,;\,\textbf{assert}\,x_1 > x_0$ on the other hand is correct, since the assert statement may never fail (assuming no overflow occurs).

Example 4.1. *The program below containing a sequence of three conditionals, each followed by an* assert, *will be used as a running example. This program is simple enough for us to explore the generation of VCs and identify differences in each VCGen method.*

$$\begin{array}{l}
\textbf{if}\ x_0 > 0\ \textbf{then}\ y_1 := 1\ \textbf{else}\ y_1 := 0\ \textbf{fi}\,;\\
\textbf{assert}\,y_1 = 0 \vee y_1 = 1\,;\\
\textbf{if}\ x_0 > 0\ \textbf{then}\ y_2 := 1\ \textbf{else}\ y_2 := 0\ \textbf{fi}\,;\\
\textbf{assert}\,y_2 = y_1\,;\\
\textbf{if}\ x_0 > 0\ \textbf{then}\ y_3 := 1\ \textbf{else}\ y_3 := 0\ \textbf{fi}\,;\\
\textbf{assert}\,y_3 = y_1\,;
\end{array}$$

One way, maybe the simplest, to approach the problem of generating logical formulas that guarantee the correctness of programs, is by *symbolic execution*, i.e. by enumerating execution paths: it suffices to produce, for each assert statement in the program, a different formula for

each path that reaches the statement (which is straightforward for programs in SA form). The validity of these formulas ensures that every execution that reaches that assert is safe (i.e. does not violate any assert). Doing this for every execution path covers the entire space of executions of the program.

Example 4.2. *The generated VCs based on symbolic execution for the program in Example 4.1 are as follows:*

1. $x_0 > 0 \land y_1 = 1 \rightarrow y_1 = 0 \lor y_1 = 1$.
2. $\neg x_0 > 0 \land y_1 = 0 \rightarrow y_1 = 0 \lor y_1 = 1$.
3. $x_0 > 0 \land y_1 = 1 \land (y_1 = 0 \lor y_1 = 1) \land x_0 > 0 \land y_2 = 1 \rightarrow y_2 = y_1$.
4. $\neg x_0 > 0 \land y_1 = 0 \land (y_1 = 0 \lor y_1 = 1) \land x_0 > 0 \land y_2 = 1 \rightarrow y_2 = y_1$.
5. $x_0 > 0 \land y_1 = 1 \land (y_1 = 0 \lor y_1 = 1) \land \neg x_0 > 0 \land y_2 = 0 \rightarrow y_2 = y_1$.
6. $\neg x_0 > 0 \land y_1 = 0 \land (y_1 = 0 \lor y_1 = 1) \land \neg x_0 > 0 \land y_2 = 0 \rightarrow y_2 = y_1$.
7. $x_0 > 0 \land y_1 = 1 \land (y_1 = 0 \lor y_1 = 1) \land x_0 > 0 \land y_2 = 1 \land y_2 = y_1 \land x_0 > 0 \land y_3 = 1 \rightarrow y_3 = y_1$.
8. $\neg x_0 > 0 \land y_1 = 0 \land (y_1 = 0 \lor y_1 = 1) \land x_0 > 0 \land y_2 = 1 \land y_2 = y_1 \land x_0 > 0 \land y_3 = 1 \rightarrow y_3 = y_1$.
9. $x_0 > 0 \land y_1 = 1 \land (y_1 = 0 \lor y_1 = 1) \land \neg x_0 > 0 \land y_2 = 0 \land y_2 = y_1 \land x_0 > 0 \land y_3 = 1 \rightarrow y_3 = y_1$.
10. $\neg x_0 > 0 \land y_1 = 0 \land (y_1 = 0 \lor y_1 = 1) \land \neg x_0 > 0 \land y_2 = 0 \land y_2 = y_1 \land x_0 > 0 \land y_3 = 1 \rightarrow y_3 = y_1$.
11. $x_0 > 0 \land y_1 = 1 \land (y_1 = 0 \lor y_1 = 1) \land x_0 > 0 \land y_2 = 1 \land y_2 = y_1 \land \neg x_0 > 0 \land y_3 = 0 \rightarrow y_3 = y_1$.
12. $\neg x_0 > 0 \land y_1 = 0 \land (y_1 = 0 \lor y_1 = 1) \land x_0 > 0 \land y_2 = 1 \land y_2 = y_1 \land \neg x_0 > 0 \land y_3 = 0 \rightarrow y_3 = y_1$.
13. $x_0 > 0 \land y_1 = 1 \land (y_1 = 0 \lor y_1 = 1) \land \neg x_0 > 0 \land y_2 = 0 \land y_2 = y_1 \land \neg x_0 > 0 \land y_3 = 0 \rightarrow y_3 = y_1$.
14. $\neg x_0 > 0 \land y_1 = 0 \land (y_1 = 0 \lor y_1 = 1) \land \neg x_0 > 0 \land y_2 = 0 \land y_2 = y_1 \land \neg x_0 > 0 \land y_3 = 0 \rightarrow y_3 = y_1$.

A VC is generated for each execution path and for each assert. Recall that the number of execution paths is in the worst case exponential in the length of the program, and consequently so is the size of the VCs. The third assert alone generates eight VCs, since it is preceded by a sequence of three conditionals, each of which duplicates the number of paths. Overall, path enumeration generates a total number of $2 + 4 + 8 = 14$ VCs for this program. If we were to add one more branching command and an assert the number of VCs would duplicate.

We remark that the well-know formalisms supporting deductive program verification techniques – *program logics* (Section 2.2 and 2.3) and *predicate transformers* (Section 2.4) – do not require programs to be converted to SA form. Instead, they rely on *variable substitutions* to encode assignments. Even though calculating predicate transformers in a naive way still causes

exponential explosion of VC size, in the context of SA, such an explosion can be avoided, following the discussion of Section 2.4 describing the work by Flanagan et al. [52]. In this formulation each assert generates a verification condition of linear size, which globally results in a set of VCs of quadratic-size in the worst case. The compact VCs for the program of Example 4.1 (which is in fact a worst-case example) are given in the following example.

Example 4.3. *The generated VCs based on the strongest postcondition predicate transformer for the program in Example 4.1 are as follows:*

1. $(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0) \rightarrow y_1 = 0 \vee y_1 = 1.$
2. $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \rightarrow y_2 = y_1.$
3. $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \rightarrow y_3 = y_1.$

A different VC is generated for each assert statement. The right hand side of each VC is the property to be checked, and the left hand side corresponds to the compact operational encoding of the program that precedes it. As opposed to symbolic execution, it encodes all possible execution paths into a single formula.

Shortly after Flanagan and Saxe's study of compact verification conditions, an alternative VC generation method was proposed by Clarke et al. [31] in the context of the development of the CBMC bounded model checker (Section 2.6). The VCs are generated from the Conditional Normal Form of the program which is obtained after a number of transformations on the initial program. Even though this method can generate quadratic size VCs in the worst case (as we will see below), for this example it generates a single VC of size linear on the size of the program.

Example 4.4. *The generated VC based on CNF for the program in Example 4.1 is as follows:*

$$\begin{aligned}(x_0 > 0 \rightarrow y_1 = 1)\ &\wedge (\neg x_0 > 0 \rightarrow y_1 = 0)\ \wedge (x_0 > 0 \rightarrow y_2 = 1)\ \wedge \\ &(\neg x_0 > 0 \rightarrow y_2 = 0)\ \wedge (x_0 > 0 \rightarrow y_3 = 1)\ \wedge (\neg x_0 > 0 \rightarrow y_3 = 0) \\ &\qquad \rightarrow (\top \rightarrow y_1 = 0 \vee y_1 = 1)\ \wedge (\top \rightarrow y_2 = y_1)\ \wedge (\top \rightarrow y_3 = y_1)\end{aligned}$$

The single generated VC contains the complete operational encoding of the program on the left hand side of the outer implication, and the properties to be checked on the right hand side. Each assignment is composed by an implication, where the right hand side corresponds to the encoding of an assignment and the left hand side the path condition for the respective assignment to be reached. The properties to be checked are composed in a similar way, but the right hand side is the result of encoding assert statements.

The fact that both the predicate transformers and CNF method (used in BMC of software) generate VCs of quadratic size in the worst case, motivates the discussion of three dimensions in which the methods differ. Let us analyze each one individually:

- The first dimension in which the methods differ is in the **logical encoding** of the program. The VCGens based on predicate transformers, encode programs following their branching structure and conditionals are captured using disjunctions of the form $(b \wedge \ldots) \vee (\neg b \wedge \ldots)$. On the other hand, VCGens based on CNF encode programs through a conjunction of implicative formulas, with path conditions guarding atomic statements of the form $(b \rightarrow \ldots) \wedge (\neg b \rightarrow \ldots)$.

- The second dimension is the use of **contexts** for each VC. The predicate transformers method produces one VC for each *assert*; each VC is an implicative formula whose antecedent is a *partial context* that encodes only the part of the program that is relevant for that *assert*. The CNF method produces a *global context* that encodes the behavior of the entire program. This context is used for all the asserted properties.

- Finally, the VCGens differ in the **semantics of the assert** command. The traditional interpretation of guarded commands implies that an execution stops whenever $\textbf{assert}\,\theta$ produces an error. In other words, one cannot rely on the validity of VCs that follow a failed assert because that assert will introduce a contradiction in the context. In the VCs based on predicate transformers generated from the program in Example 4.1, assert conditions are introduced in the context to be used in subsequent VCs, which is consistent with this interpretation. The CNF method treats *asserts* as commands that check the value of assertions, but are not used in the context of subsequent VCs; when an *assert* is reached, it is not known whether previous asserts have been passed successfully or not. In the VCs obtained from Example 4.1, using the CNF method, the global context $\mathcal{C}$ does not contain assert conditions.

Example 4.5. *The last dimension is maybe the least intuitive. Consider for instance the program* $x := 1; \textbf{assert}\, x = 2; \textbf{assert}\, x = 3$. *Then with predicate transformers we would have the VC* (i) $x = 1 \rightarrow x = 2$ *and* (ii) $x = 1 \wedge x = 2 \rightarrow x = 3$. *Obviously* (i) *is not valid, but* (ii) *is, because it has as antecedent a contradiction. On the other hand, the CNF method would generate the VC* $x = 1 \rightarrow x = 2 \wedge x = 3$*, which deals with each assert separately. Even if the VC is split into* (i) $x = 1 \rightarrow x = 2$ *and* (ii) $x = 1 \rightarrow x = 3$*, both* (i)*, and* (ii) *would fail.*

Let us then compare these dimensions with respect to the relevant aspects for both the traceability and efficiency of the verification method. The term traceability will be used to refer to the degree of connection between the program and the VCs. On the other hand, the term efficiency will be used to refer to the solving time, that is, the time a solver takes to give a verdict about a certain VC.

Traceability. With a global context, if a VC is invalid then the only way to locate the error is to *interpret the counter-example* (if it exists), which will give us a concrete execution that

violates a particular property (a common approach in software model checking). This may not be required with a partial context encoding, since there is a clear association between invalid VCs and violated properties. In fact, the degree of traceability would be even greater if a different VC were generated for each execution path reaching an assert as in symbolic execution.

Economy of Contexts. From the strict viewpoint of checking each *assert* statement, partial contexts are preferable to a global context, which may contain unnecessary information. Consider a big program with a single *assert* placed at the beginning: with partial contexts a small VC will be generated, but with a global context the encoding of the entire program will be part of the VC.

Redundancy. When verifying a whole program, the use of partial contexts has the disadvantage of replicating operational and axiomatic information in the different VCs, potentially increasing the overall size.

Lemmas. Including assertions in the context, as in the predicate transformers VCGen, implies that intermediate assertions play the role of *lemmas*: once they are proved they can be used to prove subsequent assertions, which may be advantageous for the automated proofs.

In summary, partial contexts seem to be preferable regarding traceability and the verification of each individual assert, whereas global contexts reduce the overall redundancy of generated VCs.

Dealing with Assumes in CNF: axiomatic context vs. operational context. The presence of assume commands brings an additional layer of discussion, which requires a careful consideration of the semantics of assume statements. In short, *assume statements cannot go into the global context!* Consider for instance the following sequence of statements:

$$\textbf{assume}\, x_0 > 0\,;\, x_1 := x_0 + 1\,;\, \textbf{assert}\, y_0 = x_1\,;\, \textbf{assume}\, y_0 = x_1\,;\, \textbf{assert}\, y_0 = x_1$$

It is obvious that when considering the semantics of the assume statements as in Section 2.4, the first assert does not hold for every execution (e.g. the execution where $x_0 = 1$ and $y_0 = 0$ will fail). Nonetheless, if we take the assume statement into the global context, the VC will be trivially valid: $x_0 > 0 \wedge x_1 = x_0 + 1 \wedge \mathbf{y_0} = \mathbf{x_1} \rightarrow \mathbf{y_0} = \mathbf{x_1} \wedge y_0 = x_1$. This happens because the encoding of the first assert is implied by an assume that occurs after it in the program.

It should be noted that we just consider this to be an erroneous behavior, because we had in mind the semantics discussed in Section 2.4. If another interpretation was given to the assume statement, for instance that it influences all the asserts in the code, then we would not consider the previous logical formula to be a bad description of the program correctness. Nonetheless, since the current state of the art verification tools are based on the former semantics, we should find a way to generate VCs without introducing inconsistencies. Therefore, we will introduce a new and particular context, in addition to the global context, containing all the

assumed properties that lead to a certain asserted property. This particular context will be called *axiomatic context*, and the global one will be from now on *operational context*. For the previous example the generated VC will be as follows: $x_1 = x_0 + 1 \rightarrow (\mathbf{x_0} > \mathbf{0} \rightarrow y_0 = x_1) \wedge (\mathbf{x_0} > \mathbf{0} \wedge \mathbf{y_0} = \mathbf{x_1} \rightarrow y_0 = x_1)$. Note that the axiomatic encoding of the first asserted property does not contain the second assumed property.

4.2 A Cube of VCGens

The three VCGen dimensions identified in the previous section are orthogonal, and can now be freely combined to produce hybrid VCGens. The idea is to use some particular dimension to modify a VCGen such that it will generate VCs that are different in shape, but still logically equivalent to the original.

Let us see how the VCs of Examples 4.3 and 4.4 would be modified if hybrid VCGens were used. It is straightforward to see the effect of not including assertions in the context of predicate transformers VCs.

Example 4.6. *Changing the VCs of Example 4.3 to not include asserts as lemmas is trivial. Basically one just needs to remove them from the context in every VC.*

1. $(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0) \longrightarrow y_1 = 0 \vee y_1 = 1.$

2. $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \longrightarrow y_2 = y_1.$

3. $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \longrightarrow y_3 = y_1.$

Note that this transformation will in fact reduce the size of the VC. Moving forward, it is equally easy to change the logical encoding of predicate transformers VCs to CNF, while keeping the partial contexts that are typical of predicate transformers.

Example 4.7. *The context of the VCs in the previous example can be changed, so it is written using the CNF method of capturing the the operational encoding.*

1. $(x_0 > 0 \rightarrow y_1 = 1) \wedge (\neg x_0 > 0 \rightarrow y_1 = 0) \longrightarrow y_1 = 0 \vee y_1 = 1.$

2. $(x_0 > 0 \rightarrow y_1 = 1) \wedge (\neg x_0 > 0 \rightarrow y_1 = 0) \wedge (x_0 > 0 \rightarrow y_2 = 1) \wedge (\neg x_0 > 0 \rightarrow y_2 = 0) \longrightarrow y_2 = y_1.$

3. $(x_0 > 0 \rightarrow y_1 = 1) \wedge (\neg x_0 > 0 \rightarrow y_1 = 0) \wedge (x_0 > 0 \rightarrow y_2 = 1) \wedge (\neg x_0 > 0 \rightarrow y_2 = 0) \wedge (x_0 > 0 \rightarrow y_3 = 1) \wedge (\neg x_0 > 0 \rightarrow y_3 = 0) \longrightarrow y_3 = y_1.$

The context of each VC is now a conjunction of implications: each of one of these implications is composed by the encoding of an assignment as consequent and the respective path condition for it to be reached as antecedent.

It seems less trivial to include assertions in the context in the CNF VCGen, while keeping a single global context. This is in fact directly linked to the discussion regarding assume commands, at the end of the previous section. Each asserted property has to go into the axiomatic context of the subsequent assert statements in the form of a lemma. Once again, if the property was inserted instead into the operational context, all the VCs would be trivially and erroneously valid.

Example 4.8. *Changing the VC of Example 4.4 to include asserts as lemmas results in the following VC.*

$$(x_0 > 0 \to y_1 = 1) \wedge (\neg x_0 > 0 \to y_1 = 0) \wedge (x_0 > 0 \to y_2 = 1) \wedge$$
$$(\neg x_0 > 0 \to y_2 = 0) \wedge (x_0 > 0 \to y_3 = 1) \wedge (\neg x_0 > 0 \to y_3 = 0) \to$$
$$(y_1 = 0 \vee y_1 = 1) \wedge (\mathbf{y_1 = 0 \vee y_1 = 1} \to y_2 = y_1) \wedge ((\mathbf{y_1 = 0 \vee y_1 = 1}) \wedge \mathbf{y_2 = y_1} \to y_3 = y_1)$$

Note that the use of these axiomatic partial contexts reintroduces redundancy in the encoding of axiomatic information, since it will replicate each lemma in the VCs referring to the subsequent asserts, but not of operational behavior.

Finally, we can also create a single VC with a global context based on predicate transformers.

Example 4.9. *The VC of the previous example can be changed to use predicate transformers.*

$$((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee$$
$$(\neg x_0 > 0 \wedge y_2 = 0)) \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0))$$
$$\to (y_1 = 0 \vee y_1 = 1) \wedge (y_1 = 0 \vee y_1 = 1 \to y_2 = y_1) \wedge ((y_1 = 0 \vee y_1 = 1 \wedge y_2 = y_1) \to y_3 = y_1)$$

The pair of original VCGens and the hybrid VCGens that result from combining aspects from each one can be depicted visually in what we call *the VCGen cube*, Figure 4.1. Each vertex of the cube represents a VCGen and the edges represent transformations over those VCGens. The original VCGens, are shown in shaded boxes, and all their derivations are shown in regular boxes. Moreover, each face of the cube explores a dimension:

Left/right: SP versus CNF encoding of the control flow. Each VCGen that belongs to the left hand side of the cube, uses the predicate transformers encoding of branches, while all the VCGens that are on the right hand side of the cube use the CNF method of encoding commands.

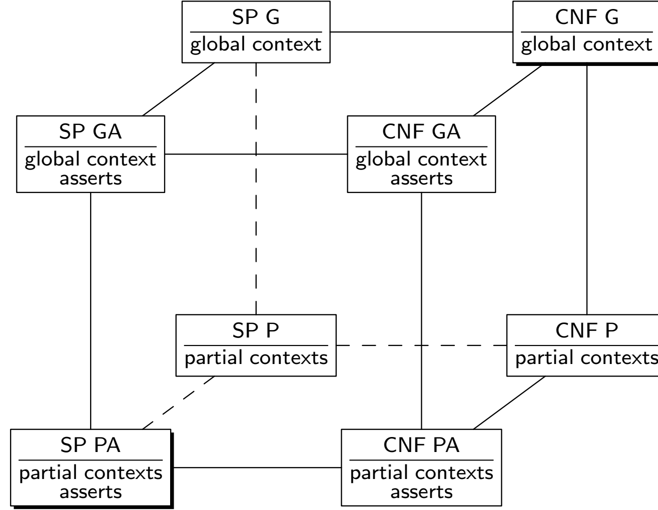


Figure 4.1: A Cube of VCGens

Top/bottom: Use of global versus partial contexts. VCGens on the top of the cube use global context while VCGens on the bottom use partial contexts.

Front/back: Asserts are included in the context versus asserts are not included in the context. The VCGens on the front face of the cube include asserts in the context, and the VCGens on the back do not include them in the context.

The VCGens in the cube will be defined by two generic algorithms. Basically, each generic algorithm corresponds to the left and right faces of the cube, and a parameter can be passed to indicate which kind of VC is to be generated. We will start by considering programs without exceptions in order to explore the dimensions discussed above, and then expand them to programs with exceptions. As suggested in Figure 4.1, the parameters G, and P will be used to refer respectively to global and partial context. The optional parameter A will be used to generate VCs with asserts in the context.

The Generic Strongest Postcondition VCGen. The generic VCGen based on predicate transformers, defined in Figure 4.2, comprises the four concrete versions of VCSP: $\mathsf{VCSP}^{\mathsf{P}}$, $\mathsf{VCSP}^{\mathsf{PA}}$, $\mathsf{VCSP}^{\mathsf{G}}$, and $\mathsf{VCSP}^{\mathsf{GA}}$. As discussed before, to standardize the VCGen definition while allowing for the use of either partial or a global context, incoming executions need to be encoded by two different formulas ϕ and ρ capturing the operational (state changing) and axiomatic (*assume* and *assert*) aspects of executions. Note however that this separation only happens to make possible writing all the versions in a single definition, therefore, if one wants to implement a VCGen this separation is not required in the partial context variants (which will reduce redundancy). The type of the function is merely informative. The $\mathbf{AComm}^{\mathrm{SA}}$ is the class of

$$\begin{array}{l}
\mathsf{VCSP}^i : \textbf{Assert} \times \textbf{Assert} \times \textbf{AComm}^{\text{SA}} \to \textbf{Assert} \times \textbf{Assert} \times \mathcal{P}(\textbf{Assert}) \\
\mathsf{VCSP}^i(\phi, \rho, \textbf{skip}) = (\top, \top, \emptyset) \\
\mathsf{VCSP}^i(\phi, \rho, x := e) = (x = e, \top, \emptyset) \\
\mathsf{VCSP}^i(\phi, \rho, \textbf{assume}\, \theta) = (\top, \theta, \emptyset) \\
\mathsf{VCSP}^i(\phi, \rho, \textbf{assert}\, \theta) = \begin{cases} (\top, \top, \{\phi \wedge \rho \to \theta\}) & \text{if } i = \mathsf{P} \\ (\top, \theta, \{\phi \wedge \rho \to \theta\}) & \text{if } i = \mathsf{PA} \\ (\top, \top, \{\rho \to \theta\}) & \text{if } i = \mathsf{G} \\ (\top, \theta, \{\rho \to \theta\}) & \text{if } i = \mathsf{GA} \end{cases} \\
\mathsf{VCSP}^i(\phi, \rho, C_1\, ;\, C_2) = (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \Gamma_1 \cup \Gamma_2) \\
\qquad \text{where } (\psi_1, \gamma_1, \Gamma_1) = \mathsf{VCSP}^i(\phi, \rho, C_1) \text{ and } (\psi_2, \gamma_2, \Gamma_2) = \mathsf{VCSP}^i(\phi \wedge \psi_1, \rho \wedge \gamma_1, C_2) \\
\mathsf{VCSP}^i(\phi, \rho, \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}) = ((b \wedge \psi_1) \vee (\neg b \wedge \psi_2), (b \wedge \gamma_1) \vee (\neg b \wedge \gamma_2), \Gamma_1 \cup \Gamma_2) \\
\qquad \text{where } (\psi_1, \gamma_1, \Gamma_1) = \mathsf{VCSP}^i(\phi \wedge b, \rho \wedge b, C_1) \text{ and } (\psi_2, \gamma_2, \Gamma_2) = \mathsf{VCSP}^i(\phi \wedge \neg b, \rho \wedge \neg b, C_2)
\end{array}$$

Figure 4.2: Predicate transformer VCGens for $i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$

single-assignment programs (the notation will be made clear in the next chapter).

The result of $\mathsf{VCSP}^i(\phi, \rho, C)$ is a tuple (ψ, γ, Γ) where Γ is a set of logical formulas, and ψ, γ encode the operational and axiomatic aspects of the behavior of C. The set Γ can represent different kind of formulas depending on the parameter that was used to generate VCs. When the intention is to generate VCs with a global context, then the formulas in Γ will correspond to the properties to be checked. Therefore, if $i \in \{\mathsf{G}, \mathsf{GA}\}$, the global context VC is $\{\psi \to \bigwedge \Gamma\}$, that is, the VC is formed by adding the operational context to the formulas in Γ (note that the axiomatic context is already in the formulas of Γ). On the other hand, if the intention is to generate VCs with partial context, that is, when $i \in \{\mathsf{P}, \mathsf{PA}\}$, then the formulas in Γ will correspond to the final VCs.

The Generalized Conditional Normal Form VCGen. We now turn our focus to the CNF method. The generic CNF VCGen, defined in Figure 4.3, differs from the previous one in that the encoding of both the operational and axiomatic parts of programs are based on conditional normal form. This requires separating path conditions from the accumulator corresponding to the axiomatic component of executions. Thus, a new parameter is used to carry the path condition that enables commands to be executed. In the call $\mathsf{VCCNF}^i(\pi, \phi, \rho, C)$ the formula π is the path condition enabling the execution of C, and as before ϕ and ρ encode respectively the operational and axiomatic contents of incoming executions. Once again, the concrete VCGens only diverge from each other in the assert clause, depending on the variant to be used (P,PA,G,GA).

The VCs are generated the following way: let $(\psi, \gamma, \Gamma) = \mathsf{VCCNF}^i(\top, \top, \top, C)$. If $i \in \{\mathsf{P}, \mathsf{PA}\}$,

$$\mathsf{VCCNF}^i : \textbf{Assert} \times \textbf{Assert} \times \textbf{Assert} \times \textbf{AComm}^{\text{SA}} \to \textbf{Assert} \times \textbf{Assert} \times \mathcal{P}(\textbf{Assert})$$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, \textbf{skip}) = (\top, \top, \emptyset)$$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, x := e) = (\pi \to x = e, \top, \emptyset)$$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, \textbf{assume}\,\theta) = (\top, \pi \to \theta, \emptyset)$$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, \textbf{assert}\,\theta) = \begin{cases} (\top, \top, \{\phi \wedge \rho \to \pi \to \theta\}) & \text{if } i = \mathsf{P} \\ (\top, \pi \to \theta, \{\phi \wedge \rho \to \pi \to \theta\}) & \text{if } i = \mathsf{PA} \\ (\top, \top, \{\rho \to \pi \to \theta\}) & \text{if } i = \mathsf{G} \\ (\top, \pi \to \theta, \{\rho \to \pi \to \theta\}) & \text{if } i = \mathsf{GA} \end{cases}$$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, C_1 \,;\, C_2) = (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \Gamma_1 \cup \Gamma_2)$$

where $(\psi_1, \gamma_1, \Gamma_1) = \mathsf{VCCNF}^i(\pi, \phi, \rho, C_1)$ and $(\psi_2, \gamma_2, \Gamma_2) = \mathsf{VCCNF}^i(\pi, \phi \wedge \psi_1, \rho \wedge \gamma_1, C_2)$

$$\mathsf{VCCNF}^i(\pi, \phi, \rho, \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi}) = (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \Gamma_1 \cup \Gamma_2)$$

where $(\psi_1, \gamma_1, \Gamma_1) = \mathsf{VCCNF}^i(\pi \wedge b, \phi, \rho, C_1)$ and $(\psi_2, \gamma_2, \Gamma_2) = \mathsf{VCCNF}^i(\pi \wedge \neg b, \phi, \rho, C_2)$

Figure 4.3: Conditional normal form VCGens for $i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$

the partial contexts VC are given by Γ. Otherwise, if $i \in \{\mathsf{G}, \mathsf{GA}\}$, the global context VC is $\{\psi \to \bigwedge \Gamma\}$.

Asymptotic Analysis of the VCGens. Let us now study the size of the VCs generated by the previous VCGens. We have stated before that the original VCGens based on SP and CNF, both generate VCs whose size can be in the worst case quadratic w.r.t. the size of the program. In this section we investigate each different variant of the VCGen, using a concrete example, and analyze how the size of the VCs can vary.

Our case study example is representative of programs with a dense presence of *assert* statements. This will allow us to investigate whether including asserts in the context or having a global context has in fact influence on the size of the VCs. We will consider a program of size N containing a number of asserts that is linear in N, such that executions may go through all the asserts. An easy way to generate such a program is by expanding iterations of a loop containing asserts, as the one shown in Example 4.10 (left). If the program is unwound N times then the resulting loop-free program will have $2N$ assert conditions. Note however, that the discussion in this section is not, in any way, related to unwinding loops: we use loop unwinding to obtain programs that have interesting properties and hence, our sole interest is in the program obtained after the loop expansion.

Example 4.10. *The program below (left) contains a loop that is unwound twice (middle) and then converted to SA (right).*

assume $x \geq 0 \wedge x \leq 50$;
assume $y < x$;
while $x < 100$ **do**
  assert $y < 100$;
  $x := x + 1$;
  $y := y + 1$;
  assert $y \leq 100$
od

assume $x \geq 0 \wedge x \leq 50$;
assume $y < x$;
if $x < 100$ **then**
  assert $y < 100$;
  $x := x + 1$;
  $y := y + 1$;
  assert $y \leq 100$;
  if $x < 100$ **then**
    assert $y < 100$;
    $x := x + 1$;
    $y := y + 1$;
    assert $y \leq 100$;
    if $x < 100$ **then**
      assume $\perp$ **fi**
  fi
fi

assume $x_0 \geq 0 \wedge x_0 \leq 50$;
assume $y_0 < x_0$;
if $x_0 < 100$ **then**
  assert $y_0 < 100$;
  $x_1 := x_0 + 1$;
  $y_1 := y_0 + 1$;
  assert $y_1 \leq 100$;
  if $x_1 < 100$ **then**
    assert $y_1 < 100$;
    $x_2 := x_1 + 1$;
    $y_2 := y_1 + 1$;
    assert $y_2 \leq 100$;
    if $x_2 < 100$ **then**
      assume $\perp$ **fi**
  else
    $x_2 := x_1$; $y_2 := y_1$
  fi
else
  $x_2 := x_0$; $y_2 := y_0$
fi

We will now study how the VCGens behave when generating VCs for this example. The VCs that will be presented in this section will be the result of unwinding the program once or twice, but we will draw our conclusions over an arbitrary number N for unwinding loops. In what follows the axiomatic context will be shown in gray. Let us start with $\mathsf{VCSP}^{\mathsf{PA}}$ and consider the case in which the loop is unwound just once. The generated VCs are as follows:

1. $x_0 < 100 \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \rightarrow y_0 < 100$.
2. $x_0 < 100 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge y_0 < 100 \rightarrow y_1 \leq 100$.

Both VCs have size that is linear on the size of the initial program, which for this case is the same as the unwound program. It becomes more interesting if we unwind the loop further. When the loop is unwound twice ($N = 2$) the generated VCs will be as follows:

1. $x_0 < 100 \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \rightarrow y_0 < 100$.

2. $x_0 < 100 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge y_0 < 100 \rightarrow y_1 \leq 100$.

3. $x_0 < 100 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge x_1 < 100 \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge y_0 < 100 \wedge y_1 \leq 100 \wedge x_1 < 100 \rightarrow y_1 < 100$.

4. $x_0 < 100 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge x_1 < 100 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1 \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge y_0 < 100 \wedge y_1 \leq 100 \wedge x_1 < 100 \wedge y_1 < 100 \rightarrow y_2 \leq 100$.

The first two VCs are exactly the same as before, but the two new VCs, corresponding to the second iteration of the loop, have linear size on the number of times that the loop was unwound. Therefore, each VC is still of linear size w.r.t. the program being verified, that is, the program that was obtained by unwinding the loop twice. This pattern is observed as the value N is incremented: each VC will be of size $\theta(1), \ldots, \theta(N)$, and thus the total size will be $\theta(N^2)$.

If asserts are not to be used in the context, as in $\mathsf{VCSP}^{\mathsf{P}}$, the overall size of the VCs will still be of $\theta(N^2)$. The reason for this is that both the operational and axiomatic context of each VC are of linear size and therefore adding them together or not does not change the overall size. The quadratic size comes in fact from the duplication of context (either operational or axiomatic) in each VC.

Let us now turn to the global context variant. Consider the VCs generated by $\mathsf{VCSP}^{\mathsf{GA}}$ for the program that results from unwinding the loop once.

$$(x_0 < 100 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1) \vee (\neg x_0 < 100 \wedge x_1 = x_0 \wedge y_1 = y_0) \rightarrow (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge y_0 < 100 \rightarrow y_1 \leq 100)$$

Note that the operational context (left hand side of the implication) is still linear and is not replicated. Nonetheless, the axiomatic context (shown in gray) is replicated for each assert. This becomes even clearer when the VCs are generated for the program that results from unwinding the loop twice.

$$(x_0 < 100 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge ((x_1 < 100 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1) \vee (\neg x_1 < 100 \wedge x_2 = x_1 \wedge y_2 = y_1))) \vee (\neg x_0 < 100 \wedge x_2 = x_0 \wedge y_2 = y_0) \rightarrow (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge y_0 < 100 \rightarrow y_1 \leq 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge y_0 < 100 \wedge y_1 \leq 100 \wedge x_1 < 100 \rightarrow y_1 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge y_0 < 100 \wedge y_1 \leq 100 \wedge x_1 < 100 \wedge y_1 < 100 \rightarrow y_2 \leq 100)$$

Now it becomes clear that the axiomatic context for each assert will grow in the form of $\theta(1), \ldots, \theta(N)$. For instance the axiomatic context of the condition of the first assert is replicated through all the conditions of the subsequent asserts. If assert conditions are not to be used as lemmas, as in $\mathsf{VCSP}^{\mathsf{G}}$, the overall size of the VC is actually not affected. This is due to the assume conditions before the if statements and also to the if conditions (which correspond to

the loop condition in the initial program) that are still replicated through all the properties. For instance, if lemmas are removed from the previous VCs, we would obtain the VC below. Note how the assume conditions and the branching conditions are replicated before each property coming from the assert statements.

$$(x_0 < 100 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge ((x_1 < 100 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1) \vee (\neg x_1 < 100 \wedge x_2 = x_1 \wedge y_2 = y_1))) \vee (\neg x_0 < 100 \wedge x_2 = x_0 \wedge y_2 = y_0) \rightarrow (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \rightarrow y_1 \leq 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 \leq 100)$$

So, for the verification of the program that results from unwinding the loop in Example 4.10, in terms of VC size it is irrelevant which concrete variant of VCSP is used. The overall size for the example is always $\theta(N^2)$.

This is however not the case when a VCGen from the VCCNF family is used. Consider for instance the VC that is generated by $\mathsf{VCCNF}^{\mathsf{G}}$ for the program that results from unwinding the loop once.

$$(x_0 < 100 \rightarrow x_1 = x_0 + 1) \wedge (x_0 < 100 \rightarrow y_1 = y_0 + 1) \wedge (\neg x_0 < 100 \rightarrow x_1 = x_0) \wedge (\neg x_0 < 100 \rightarrow y_1 = y_0) \rightarrow (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \rightarrow y_1 \leq 100)$$

Note that the loop condition is replicated through all the implications in the operational context. More precisely, in this case, it is replicated twice because there are two assignments, and its negation is also replicated twice to synchronize variables. The conditions referring to the assumes in the program are also replicated in the axiomatic context of the two properties to be checked. This replication grows as the size of the program grows. Consider now that the loop is unwound twice:

$$(x_0 < 100 \rightarrow x_1 = x_0 + 1) \wedge (x_0 < 100 \rightarrow y_1 = y_0 + 1) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow x_2 = x_1 + 1) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 = y_1 + 1) \wedge (x_0 < 100 \wedge \neg x_1 < 100 \rightarrow x_2 = x_1) \wedge (x_0 < 100 \wedge \neg x_1 < 100 \rightarrow y_2 = y_1) \wedge (\neg x_0 < 100 \rightarrow x_2 = x_0) \wedge (\neg x_0 < 100) \rightarrow y_2 = y_0) \rightarrow (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \rightarrow y_1 \leq 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 \leq 100)$$

This confirms the pattern described above. So, if the loop is unwound N times, the loop condition will be replicated N times. When transforming the program into CNF, since the unwound program consists in a set of nested if commands, the first if condition, corresponding to the first iteration, will be replicated $N \times 2$ times (it will be replicated through all the inner atomic commands), the second $(N-1) \times 2$ times and so on, such that the condition referring to the last iteration will be replicated 1×2 times. This pattern leads to an operational context of size $\theta(N^2)$, which is visible in the VC above.

With respect to the right hand side of the VC, note that each of the properties to be verified is guarded by $1, \ldots, N$ conditions. Since we have $2N$ asserts, the size of the properties to be checked will be $2N \times N$, which leads to a size of N^2. In total, the size of the VC is $\theta(N^2)$.

The overall size of the VC becomes N^3 when asserts are used as lemmas, as in $\mathsf{VCCNF}^{\mathsf{GA}}$. Consider the VC below obtained by expanding the program twice.

$$(x_0 < 100 \rightarrow x_1 = x_0 + 1) \wedge (x_0 < 100 \rightarrow y_1 = y_0 + 1) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow x_2 = x_1 + 1) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 = y_1 + 1) \wedge (x_0 < 100 \wedge \neg x_1 < 100 \rightarrow x_2 = x_1) \wedge (x_0 < 100 \wedge \neg x_1 < 100 \rightarrow y_2 = y_1) \wedge (\neg x_0 < 100 \rightarrow x_2 = x_0) \wedge (\neg x_0 < 100 \rightarrow y_2 = y_0) \rightarrow (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge (x_0 < 100 \rightarrow y_0 < 100) \rightarrow x_0 < 100 \rightarrow y_1 \leq 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge (x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 < 100 \rightarrow y_1 \leq 100) \rightarrow x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge (x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 < 100 \rightarrow y_1 \leq 100) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100) \rightarrow x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 \leq 100)$$

Note now that each assert condition that is used as lemma is guarded by $1, \ldots, N$ conditions and since it is replicated through all the subsequent assert conditions, it will contribute to a $\theta(N^3)$ overall size in the right hand side of the VC. The same overall size is obtained when using $\mathsf{VCSP}^{\mathsf{G}}$ with a program that includes assumes in the loop body.

Changing now to partial contexts, the $\mathsf{VCCNF}^{\mathsf{P}}$ variant generates the following VCs when the loop is unwound twice.

1. $x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow (x_0 < 100 \rightarrow y_0 < 100)$.
2. $(x_0 < 100 \rightarrow x_1 = x_0 + 1) \wedge (x_0 < 100 \rightarrow y_1 = y_0 + 1) \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow (x_0 < 100 \rightarrow y_1 \leq 100)$.
3. $(x_0 < 100 \rightarrow x_1 = x_0 + 1) \wedge (x_0 < 100 \rightarrow y_1 = y_0 + 1) \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow (x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100)$.
4. $(x_0 < 100 \rightarrow x_1 = x_0 + 1) \wedge (x_0 < 100 \rightarrow y_1 = y_0 + 1) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow x_2 = x_1 + 1) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 = y_1 + 1) \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow (x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 \leq 100)$.

First of all note that a total of $2N$ VCs will be generated because there are two asserts in the loop. Each VC is of size $\theta(1^2), \theta(2^2), \ldots, \theta(N^2)$, resulting in an overall size of $\theta(N^3)$.

If asserts are included in the context, the global context VC also becomes of size $\theta(N^3)$, because each lemma will be guarded by a formula of size $\theta(1^2)$, $\ldots$, $\theta(N^2)$. In this case, the resulting VCs are as follows:

1. $x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow (x_0 < 100 \rightarrow y_0 < 100)$.
2. $(x_0 < 100 \rightarrow x_1 = x_0 + 1) \wedge (x_0 < 100 \rightarrow y_1 = y_0 + 1) \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge (x_0 < 100 \rightarrow y_0 < 100) \rightarrow (x_0 < 100 \rightarrow y_1 \leq 100)$.

3. $(x_0 < 100 \rightarrow x_1 = x_0 + 1) \wedge (x_0 < 100 \rightarrow y_1 = y_0 + 1) \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge (x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 < 100 \rightarrow y_1 \leq 100) \rightarrow (x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100)$.

4. $(x_0 < 100 \rightarrow x_1 = x_0 + 1) \wedge (x_0 < 100 \rightarrow y_1 = y_0 + 1) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow x_2 = x_1 + 1) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 = y_1 + 1) \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge (x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 < 100 \rightarrow y_1 \leq 100) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100) \rightarrow (x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 \leq 100)$.

In summary, the size of the generated VCs for the program that results from unwinding the program from Example 4.10 (left) N times, are given as follows:

	G	GA	P	PA
VCCNF	$\theta(N^2)$	$\theta(N^3)$	$\theta(N^3)$	$\theta(N^3)$
VCSP	$\theta(N^2)$	$\theta(N^2)$	$\theta(N^2)$	$\theta(N^2)$

4.3 VCGens Optimizations

The previous section shows a concrete example that led the presented VCGens to generate VCs with redundancy of conditions. Identifying the source of these redundancies motivates the introduction of optimizations that may reduce the overall size of the VCs. Moreover, the foremost deductive verification and bounded model checking tools are usable in practice because of their advanced optimization techniques. These techniques are usually tied to specific VCGen algorithms, and thus it is not clear whether they can be applicable to others. This section addresses this issue by showing how two important practical optimization techniques [11, 31] are applicable to different VCGens. The VCGen cube helps clarifying the intricacies between the optimization techniques and VCGens.

To start with, it is possible to optimize Flanagan and Saxe's predicate transformers VCGen to produce VCs [11] that reduce the amount of redundancy. The idea is fairly simple: even though the size of each VC is linear, the overall quadratic size comes from the possible existence of a linear number of such VCs, each replicating partially the encoding of the program. It is however possible to transform this set of VCs into a single VC, applying the following equivalence

$$(\phi \rightarrow \theta) \wedge (\phi \wedge \theta \wedge \gamma \rightarrow \psi) \equiv \phi \rightarrow \theta \wedge (\theta \wedge \gamma \rightarrow \psi)$$

the required number of times (with θ and ψ being the encoding of assert conditions). The idea is that if we have two VCs, $\phi \rightarrow \theta$ and $\phi \wedge \theta \wedge \gamma \rightarrow \psi$ and the context of the second expands the context of the first, then, this context should not be replicated, but instead placed such that it will serve as context for both assert conditions. In this case, the first assert condition is used as a lemma for the second condition, and therefore, it must be replicated.

Consider again the three VCs of Example 4.3. Applying the equivalence described before we obtain the following VC, where the properties to be checked are in bold, and the axiomatic context in gray:

$$(x_0 > 0 \land y_1 = 1) \lor (\neg x_0 > 0 \land y_1 = 0) \to \boldsymbol{(y_1 = 0 \lor y_1 = 1)}$$
$$\land (y_1 = 0 \lor y_1 = 1 \to (x_0 > 0 \land y_2 = 1) \lor (\neg x_0 > 0 \land y_2 = 0) \to \boldsymbol{y_2 = y_1}$$
$$\land (y_2 = y_1 \to (x_0 > 0 \land y_3 = 1) \lor (\neg x_0 > 0 \land y_3 = 0) \to \boldsymbol{y_3 = y_1}))$$

If asserts are not to be introduced in the context we simply note that $\phi \to \theta \land (\theta \land \gamma \to \psi) \equiv \phi \to \theta \land (\gamma \to \psi)$, and thus it can be simply removed from the context of the subsequent assert conditions as follows:

$$(x_0 > 0 \land y_1 = 1) \lor (\neg x_0 > 0 \land y_1 = 0) \to \boldsymbol{(y_1 = 0 \lor y_1 = 1)}$$
$$\land ((x_0 > 0 \land y_2 = 1) \lor (\neg x_0 > 0 \land y_2 = 0) \to \boldsymbol{y_2 = y_1}$$
$$\land ((x_0 > 0 \land y_3 = 1) \lor (\neg x_0 > 0 \land y_3 = 0) \to \boldsymbol{y_3 = y_1}))$$

The same method can also be used with global contexts, applying the equivalence described before to the operational and axiomatic context. This optimization, can actually be applied to the entire left-hand face of the VCGens cube.

$$((x_0 > 0 \land y_1 = 1) \lor (\neg x_0 > 0 \land y_1 = 0)) \land ((x_0 > 0 \land y_2 = 1) \lor (\neg x_0 > 0 \land y_2 = 0)) \land ((x_0 > 0 \land y_3 = 1) \lor (\neg x_0 > 0 \land y_3 = 0)) \to \boldsymbol{(y_1 = 0 \lor y_1 = 1)} \land (y_1 = 0 \lor y_1 = 1 \to \boldsymbol{y_2 = y_1} \land (y_2 = y_1 \to \boldsymbol{y_3 = y_1}))$$

If asserts are not to be used as lemmas they can simply be removed from the axiomatic context.

$$((x_0 > 0 \land y_1 = 1) \lor (\neg x_0 > 0 \land y_1 = 0)) \land ((x_0 > 0 \land y_2 = 1) \lor (\neg x_0 > 0 \land y_2 = 0)) \land ((x_0 > 0 \land y_3 = 1) \lor (\neg x_0 > 0 \land y_3 = 0)) \to \boldsymbol{(y_1 = 0 \lor y_1 = 1)} \land \boldsymbol{y_2 = y_1} \land \boldsymbol{y_3 = y_1}$$

The VCLin^i generic VCGen is defined in Figure 4.4. Note that, it does not use the accumulator parameters and the result of $\mathsf{VCLin}^i(C)$ is a tuple (ψ, γ, δ) where the first two components are the same as with VCSP^i, and δ is a single formula. Again, if $i \in \{\mathsf{P}, \mathsf{PA}\}$, the partial context VC is δ, otherwise, if $i \in \{\mathsf{G}, \mathsf{GA}\}$, the global context VC is $\psi \to \delta$.

Considering the program from Example 4.10 (right), $\mathsf{VCLin}^{\mathsf{PA}}$ would generate the following VC:

$$x_0 \geq 0 \land x_0 \leq 50 \to y_0 < x_0 \to x_0 < 100 \to y_0 < 100 \land (y_0 < 100 \to x_1 = x_0 + 1 \to y_1 = y_0 + 1 \to y_1 \leq 100 \land (y_1 \leq 100 \to x_1 < 100 \to y_1 < 100 \land (y_1 < 100 \to x_2 = x_1 + 1 \to y_2 = y_1 + 1 \to y_2 \leq 100)))$$

The only part of the program that is duplicated in VC above corresponds to the asserted conditions. Nonetheless, the duplication only happens once and just because asserts are to be used as lemmas. The $\mathsf{VCLin}^{\mathsf{G}}$ would generate a similar VC but without the duplication of the

$$\mathsf{VCLin}^i : \mathbf{AComm}^{\text{SA}} \to \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert}$$

$$\mathsf{VCLin}^i(\mathbf{skip}) = (\top, \top, \top)$$

$$\mathsf{VCLin}^i(x := e) = (x = e, \top, \top)$$

$$\mathsf{VCLin}^i(\mathbf{assume}\,\theta) = (\top, \theta, \top)$$

$$\mathsf{VCLin}^i(\mathbf{assert}\,\theta) = \begin{cases} (\top, \top, \theta) & \text{if } i \in \{\mathsf{P}, \mathsf{G}\} \\ (\top, \theta, \theta) & \text{if } i \in \{\mathsf{PA}, \mathsf{GA}\} \end{cases}$$

$$\mathsf{VCLin}^i(C_1 \,;\, C_2) = \begin{cases} (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \delta_1 \wedge (\psi_1 \wedge \gamma_1 \to \delta_2)) & \text{if } i \in \{\mathsf{P}, \mathsf{PA}\} \\ (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \delta_1 \wedge (\gamma_1 \to \delta_2)) & \text{if } i \in \{\mathsf{G}, \mathsf{GA}\} \end{cases}$$

where $(\psi_1, \gamma_1, \delta_1) = \mathsf{VCLin}^i(C_1)$ and $(\psi_2, \gamma_2, \delta_2) = \mathsf{VCLin}^i(C_2)$

$$\mathsf{VCLin}^i(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}) = ((b \wedge \psi_1) \vee (\neg b \wedge \psi_2), (b \wedge \gamma_1) \vee (\neg b \wedge \gamma_2), (b \to \delta_1) \wedge (\neg b \to \delta_2))$$

where $(\psi_1, \gamma_1, \delta_1) = \mathsf{VCLin}^i(C_1)$, and $(\psi_2, \gamma_2, \delta_2) = \mathsf{VCLin}^i(C_2)$

Figure 4.4: Lin optimization VCGen, for $i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$

asserted conditions. Turning to the global context variants, the $\mathsf{VCLin}^{\mathsf{GA}}$ generates the following VC:

$$(x_0 < 100 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge ((x_1 < 100 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1) \vee (\neg x_1 < 100 \wedge x_2 = x_1 \wedge y_2 = y_1))) \vee (\neg x_0 < 100 \wedge x_2 = x_0 \wedge y_2 = y_0) \to (x_0 \geq 0 \wedge x_0 \leq 50 \to y_0 < x_0 \to x_0 < 100 \to y_0 < 100 \wedge (y_0 < 100 \to y_1 \leq 100 \wedge (y_1 \leq 100 \to x_1 < 100 \to y_1 < 100 \wedge (y_1 < 100 \to y_2 \leq 100))))$$

Now the operational context is separated from the axiomatic context and the properties to be verified. This imposes that the loop condition is duplicated in the operational and axiomatic context, but it does not change the overall size of the VC. For this example, all VCGens from the VCLin family generate VCs of size $\theta(N)$.

An entirely different optimization, integrated in the CBMC tool [31], can be applied to the right-hand side face of the cube. This optimization requires converting programs to a *static* single-assignment (SSA) form, where variables may occur in the code (syntactically) at most once as an l-value (the conversion of programs to SSA was covered in Section 2.6).

The basis of the optimization is the observation that path conditions introduced in the VCCNF variants for assignment statements are not required. The idea is that only the values of the variables that are effectively assigned are propagated by the ϕ-function (in our setting ϕ-function will be captured by conditional expressions, as in Section 2.6). For instance, without the optimization the program $\mathbf{if}\ b\ \mathbf{then}\ x_1 := e_1\ \mathbf{else}\ x_2 := e_2\ \mathbf{fi}\,;\, x_3 := b\,?\,x_1 : x_2$ would be encoded as the formula $(b \to x_1 = e_1) \wedge (\neg b \to x_2 = e_2) \wedge (b \to x_3 = x_1) \wedge (\neg b \to x_3 = x_2)$. Note that, if the condition b is true then only the value of the variable x_1 is propagated,

since we have $b \to x_3 = x_1$. In this case, the value of the variable x_2 will not be used in subsequent conditions and therefore it can be assigned anyway. The inverse argument can be made for when b does not hold. With the optimization the encoding of the program becomes $x_1 = e_1 \wedge x_2 = e_2 \wedge (b \to x_3 = x_1) \wedge (\neg b \to x_3 = x_2)$. The guards for encoding the assignment to x_1 and x_2 are removed, but only the value of the correct variable will be assigned.

Example 4.11. *Below is a version of the program of Example 4.1 in SSA. Note that in order to convert the program into SSA one has to increase the number of variables.*

$$
\begin{array}{l}
\textbf{if } x_0 > 0 \textbf{ then } y_1 := 1 \textbf{ else } y_2 := 0 \textbf{ fi}\,; \\
y_3 := x_0 > 0 \;?\; y_1 \;:\; y_2\,; \\
\textbf{assert } y_3 = 0 \vee y_3 = 1\,; \\
\textbf{if } x_0 > 0 \textbf{ then } y_4 := 1 \textbf{ else } y_5 := 0 \textbf{ fi}\,; \\
y_6 := x_0 > 0 \;?\; y_4 \;:\; y_5\,; \\
\textbf{assert } y_6 = y_3\,; \\
\textbf{if } x_0 > 0 \textbf{ then } y_7 := 1 \textbf{ else } y_8 := 0 \textbf{ fi}\,; \\
y_9 := x_0 > 0 \;?\; y_7 \;:\; y_8\,; \\
\textbf{assert } y_9 = y_3\,;
\end{array}
$$

The $\mathsf{VCCNF}^{\mathsf{G}}$ VCGen can be optimized to generate the VC below, where the duplication of the loop condition is avoided in the operational encoding:

$$y_1 = 1 \wedge y_2 = 0 \wedge (x_0 > 0 \to y_3 = y_1) \wedge (\neg x_0 > 0 \to y_3 = y_2) \wedge y_4 = 1 \wedge y_5 = 0 \wedge (x_0 > 0 \to y_6 = y_4) \wedge (\neg x_0 > 0 \to y_6 = y_5) \wedge y_7 = 1 \wedge y_8 = 0 \wedge (x_0 > 0 \to y_9 = y_7) \wedge (\neg x_0 > 0 \to y_9 = y_8) \to (y_3 = 0 \vee y_3 = 1) \wedge y_6 = y_3 \wedge y_9 = y_3$$

Note how in the operational context, the if conditions only appear to synchronize variables.[1]

In this case, having asserts as lemmas would not bring any additional complications (either the context is global or partial). The partial context VCs with the optimization would be as follows:

1. $y_1 = 1 \wedge y_2 = 0 \wedge (x_0 > 0 \to y_3 = y_1) \wedge (\neg x_0 > 0 \to y_3 = y_2) \to (y_3 = 0 \vee y_3 = 1)$.

2. $y_1 = 1 \wedge y_2 = 0 \wedge (x_0 > 0 \to y_3 = y_1) \wedge (\neg x_0 > 0 \to y_3 = y_2) \wedge y_4 = 1 \wedge y_5 = 0 \wedge (x_0 > 0 \to y_6 = y_4) \wedge (\neg x_0 > 0 \to y_6 = y_5) \to y_6 = y_3$.

3. $y_1 = 1 \wedge y_2 = 0 \wedge (x_0 > 0 \to y_3 = y_1) \wedge (\neg x_0 > 0 \to y_3 = y_2) \wedge y_4 = 1 \wedge y_5 = 0 \wedge (x_0 > 0 \to y_6 = y_4) \wedge (\neg x_0 > 0 \to y_6 = y_5) \wedge y_7 = 1 \wedge y_8 = 0 \wedge (x_0 > 0 \to y_9 = y_7) \wedge (\neg x_0 > 0 \to y_9 = y_8) \to y_9 = y_3$.

[1] We remark that, if the logic language allows for *ite*-like operators as found for instance in the SMT-LIB [14], then the if conditions do not even need to be duplicated. The encoding of the first if statement would be of the form $y_3 = \textbf{ite}\ (x_0 > 0)\ y_1\ y_2$.

The generic VCGen based on this optimization will be denoted by VCSSA^i and differs from VCCNF^i only in the assignment case. In the special case of assignments with conditionals expressions it becomes $\mathsf{VCSSA}^i(\pi, \phi, \rho, x := b\ ?\ y\ :\ z) = (\,(b \to x = y) \wedge (\neg b \to x = z),\ \top,\ \emptyset\,)$ and for other cases it becomes $\mathsf{VCSSA}^i(\pi, \phi, \rho, x := e) = (\,x = e,\ \top,\ \emptyset\,)$.

Example 4.12. *The program below is the conversion into SSA of the one in Example 4.10 (center). Note that the only difference in this case w.r.t. the program in Example 4.10 (right) is in the last version of the variables. While in the former case, the variables x_2 and y_2 are to be used by subsequent expressions, here it is the new variables introduced to synchronize branches that should be used.*

$$
\begin{array}{l}
\textbf{assume}\, x_0 \geq 0 \wedge x_0 \leq 50\,;\\
\textbf{assume}\, y_0 < x_0\,;\\
\textbf{if}\ x_0 < 100\ \textbf{then}\\
\quad \textbf{assert}\, y_0 < 100\,;\\
\quad x_1 := x_0 + 1\,;\\
\quad y_1 := y_0 + 1\,;\\
\quad \textbf{assert}\, y_1 \leq 100\,;\\
\quad \textbf{if}\ x_1 < 100\ \textbf{then}\\
\quad\quad \textbf{assert}\, y_1 < 100\,;\\
\quad\quad x_2 := x_1 + 1\,;\\
\quad\quad y_2 := y_1 + 1\,;\\
\quad\quad \textbf{assert}\, y_2 \leq 100\,;\\
\quad\quad \textbf{if}\ x_2 < 100\ \textbf{then}\\
\quad\quad\quad \textbf{assume}\, \bot\ \textbf{fi}\\
\quad \textbf{fi}\\
\quad x_3 := x_1 < 100\ ?\, x_2\ :\ x_1\,;\\
\quad y_3 := x_1 < 100\ ?\, y_2\ :\ y_1\\
\textbf{fi}\\
x_4 := x_0 < 100\ ?\, x_3\ :\ x_0\,;\\
y_4 := x_0 < 100\ ?\, y_3\ :\ y_0
\end{array}
$$

For the example above, the $\mathsf{VCSSA}^\mathsf{G}$ will generate the following VC, that avoids the duplication of path conditions in the operational encoding. Note however that they are still present in the axiomatic context.

$x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1 \wedge (x_1 < 100 \to x_3 = x_2) \wedge (\neg x_1 < 100 \to x_3 = x_1) \wedge (x_1 < 100 \to y_3 = y_2) \wedge (\neg x_1 < 100 \to y_3 = y_1) \wedge (x_0 < 100 \to x_4 = x_3) \wedge (\neg x_1 < 100 \to x_4 = x_0) \wedge (x_0 < 100 \to y_4 = y_3) \wedge (\neg x_1 < 100 \to y_4 = y_0) \to (x_0 \geq$

$0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \rightarrow y_1 \leq 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 \leq 100)$

The operational encoding becomes of size $\theta(N)$ because the optimization dispenses entirely with the accumulated path conditions. Nonetheless, since they cannot be avoided in the assumed and asserted properties, the consequent of the VC is still of size $\theta(N^2)$, because it contains $2N$ assert conditions, each guarded by a path condition of size in $\theta(N)$. If asserts are to be introduced in the context, the following VC would be generated:

$x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1 \wedge (x_1 < 100 \rightarrow x_3 = x_2) \wedge (\neg x_1 < 100 \rightarrow x_3 = x_1) \wedge (x_1 < 100 \rightarrow y_3 = y_2) \wedge (\neg x_1 < 100 \rightarrow y_3 = y_1) \wedge (x_0 < 100 \rightarrow x_4 = x_3) \wedge (\neg x_1 < 100 \rightarrow x_4 = x_0) \wedge (x_0 < 100 \rightarrow y_4 = y_3) \wedge (\neg x_1 < 100 \rightarrow y_4 = y_0) \rightarrow (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge (x_0 < 100 \rightarrow y_0 < 100) \rightarrow x_0 < 100 \rightarrow y_1 \leq 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge (x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 < 100 \rightarrow y_1 \leq 100) \rightarrow x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100) \wedge (x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \wedge x_0 < 100 \wedge (x_0 < 100 \rightarrow y_0 < 100) \wedge (x_0 < 100 \rightarrow y_1 \leq 100) \wedge (x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100) \rightarrow x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 \leq 100)$

Due to the duplication of conditions and asserted properties in the axiomatic context the size of the VC using asserts as lemmas becomes $\theta(N^3)$, since it contains $2N$ assert conditions, each guarded by a path condition of size in $\theta(N^2)$.

It is in the partial contexts version (with no asserts) that the optimization becomes more interesting, since it will now generate $2N$ VCs, of size $\theta(1)$ to $\theta(N)$ (each having a single assert condition as consequent), with overall size in $\theta(N^2)$ as follows:

1. $x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \rightarrow y_0 < 100$.

2. $x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \rightarrow y_1 \leq 100$.

3. $x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \wedge x_1 < 100 \rightarrow y_1 < 100$.

4. $x_1 = x_0 + 1 \wedge y_1 = y_0 + 1 \wedge x_2 = x_1 + 1 \wedge y_2 = y_1 + 1 \wedge x_0 \geq 0 \wedge x_0 \leq 50 \wedge y_0 < x_0 \rightarrow x_0 < 100 \wedge x_1 < 100 \rightarrow y_2 \leq 100$.

The overall results corresponding to Example 4.10 can now be summarized in Table 4.1 (left), which includes the optimizations presented in this section. The worst-case results are on the right. In the worst case, the VCCNF and VCSSA family of VCGens will all generate VCs of size in $O(N^3)$. A trivial way of creating a program that generates such a growth on the size of the VC for all the VCGens, is by considering the program from Example 4.10 and inserting an assume statement after each assert (with the same condition, for instance). The variants of VCSP generate VCs of size $O(N^2)$ and the variants of VCLin of size $O(N^2)$. The latter can actually come as a surprise, but we should recall that the function VCLin returns the VC $\delta_1 \wedge (\psi_1 \wedge \gamma_1 \rightarrow \delta_2)$ (resp. $\delta_1 \wedge (\gamma_1 \rightarrow \delta_2)$) for the partial context variants (resp. global context

	G	GA	P	PA
VCSP	$\theta(N^2)$	$\theta(N^2)$	$\theta(N^2)$	$\theta(N^2)$
VCLin	$\theta(N)$	$\theta(N)$	$\theta(N)$	$\theta(N)$
VCCNF	$\theta(N^2)$	$\theta(N^3)$	$\theta(N^3)$	$\theta(N^3)$
VCSSA	$\theta(N^2)$	$\theta(N^3)$	$\theta(N^2)$	$\theta(N^3)$

	G	GA	P	PA
VCSP	$\theta(N^2)$	$\theta(N^2)$	$\theta(N^2)$	$\theta(N^2)$
VCLin	$\theta(N^2)$	$\theta(N^2)$	$\theta(N^2)$	$\theta(N^2)$
VCCNF	$\theta(N^3)$	$\theta(N^3)$	$\theta(N^3)$	$\theta(N^3)$
VCSSA	$\theta(N^3)$	$\theta(N^3)$	$\theta(N^3)$	$\theta(N^3)$

Table 4.1: Overall results for the running example (left), and worst-case VC size (right)

variants). The duplication of ψ_1 and γ_1 may originate VCs of quadratic size w.r.t. the program size. An example of this is shown below.

Example 4.13. *Consider the following program (left) consisting in a sequence of asserts. As shown below, the sequence of asserts can be associated to the right or to the left.*

assert θ_1 ;	**assert** θ_1 ;	((**assert** θ_1 ;
assert θ_2 ;	(**assert** θ_2 ;	**assert** θ_2) ;
assert θ_3 ;	(**assert** θ_3 ;	**assert** θ_3) ;
assert θ_4	**assert** θ_4))	**assert** θ_4

The VC for the program that associates parentheses to the right is $\theta_1 \wedge (\theta_1 \rightarrow \theta_2 \wedge (\theta_2 \rightarrow \theta_3 \wedge (\theta_3 \rightarrow \theta_4)))$*, which is clearly of linear size w.r.t. the size of the program. Nonetheless, when parentheses are associated to the left, the situation is different: the VC is now* $\theta_1 \wedge (\theta_1 \rightarrow \theta_2) \wedge (\theta_1 \wedge \theta_2 \rightarrow \theta_3) \wedge (\theta_1 \wedge \theta_2 \wedge \theta_3 \rightarrow \theta_4)$*, which constantly duplicates conditions, originating a VC that is quadratic w.r.t. the size of the program.*

If the problem was just a matter of association, it could easily be solved by associating all the commands to the right. However, the problem is not with the association, but else with nested 'blocks' of code containing asserts and followed by other asserts. Such blocks can be avoided in the previous example, but they cannot be avoided when those 'blocks' are branches of an if command. We postpone further details about this topic to Section 7.2.

It is clear from the above that each of the two original optimizations generates four new hybrid VCGens that can be added to our cube as in Figure 4.5. One new face is added on the left, representing an optimization of the VCSP family of VCGens, and another on the right, representing an optimization of the VCCNF family of VCGens.

4.4 Generic VCGens for Programs with Exceptions

The language constructs presented in Section 2.1 and their respective semantics allow for a single flow of execution. Abstracting from the fact that an assume or an assert can fail and make the program terminate immediately in a special state where the value of the variables is

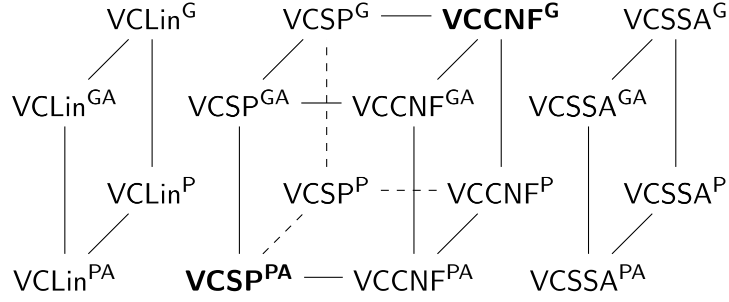


Figure 4.5: An expansion of the Cube of VCGens

unknown, the execution of a program may terminate normally or else it may never terminate at all. Note also that it is not possible to recover once an assert or an assume fails. However, it is sometimes desirable to consider alternative flows of execution for unexpected behaviors (unpredictable or erroneous) or just for the sake of structuring the program in a different way. *Exceptions* are commonly used for this [52, 63]. Whenever an exception is raised, the program continues to execute, but through a different flow. It is possible to go back into the normal flow, by catching the exception that was raised, and recover from it. To support exceptions handling in the while language, we extend the syntax of **Comm** (and **AComm**) as follows:

$$C \;::=\; \ldots \;\mid\; \textbf{throw} \;\mid\; \textbf{try}\, C \,\textbf{catch}\, C\, \textbf{hc}$$

The command **throw** will raise an exception, and the command $\textbf{try}\, C_1\, \textbf{catch}\, C_2\, \textbf{hc}$ will execute C_1 and if it terminates exceptionally it will then execute C_2, otherwise C_2 is just not executed. For the sequence statement $C_1 \,;\, C_2$, the command C_2 is only executed if C_1 terminates normally. With this, program can now terminate normally or exceptionally.

Example 4.14. *The program on the left divides y by x and places the result in x whenever x is not zero. In this case, it also adds 2 to x and then terminates normally. If x is equal to zero an exception is raised and the assignment $x := x + 2$ is* ***not*** *executed: the program terminates exceptionally. The program on the right uses a* try-catch *block, therefore when x is equal to zero an exception is raised and the flow continues in the* **catch** *block. Since the* **skip** *terminates normally, the execution of the whole program terminates always normally (once again, assuming that the evaluation of an expression does not raise an exception).*

if $x = 0$ **then throw else** $x := y/x$ **fi** ;
$x := x + 2$
	try
	  if $x = 0$ **then throw else** $x := y/x$ **fi** ;
	  $x := x + 2$
	catch
	  skip
	hc

As stated before, exceptions may be used for recovering from unexpected behaviours or just as a choice of implementation. It is sometimes useful to consider multiple kinds of exceptions for distinguishing the different reasons for an exception to occur. For instance, if an exception is raised due to erroneous behaviour, the recovery may be different depending on the error that occurred. Moreover, from the implementation point of view one may want to use different kinds of exceptions just for structuring the code with multiple execution flows. Nonetheless, in this thesis, for the sake of clarity, we will consider a single kind of exception. The presented results can however be easily extended when considering different types of exceptions.

The formal semantics of the language with exceptions will be given in Chapter 5. For now let us see how the VCGens presented in the previous section must be adapted adequately for programs with exceptions. The difference is that in the presence of exceptions a program can terminate in two different and disjoint sets of states: it can terminate normally or it can terminate exceptionally. In terms of VCGens two additional formulas will be generated to capture the encoding of the program when it terminates exceptionally: one will encode the operational encoding and the other will capture the axiomatic encoding.

Example 4.15. *The annotated program below calculates the greatest common divisor between u and v, using the Euclidean algorithm (adapted from the Why3 gallery[2]). We consider that the modulo operator % belongs to the language of expressions and* gcd *to the language of assertions. The auxiliary variables u_{aux} and v_{aux} contain respectively the initial value of u and v and the result is returned in u.*

try
  while $\top$ **do** $\{u \geq 0 \wedge v \geq 0 \wedge \mathsf{gcd}\,(u, v) = \mathsf{gcd}\,(u_{aux}, v_{aux})\}$
    if $v = 0$ **then**
      throw
    else
      $t := v\,;$
      $v := u \,\%\, v\,;$
      $u := t$
    fi
  od
catch
  skip
hc

Let us focus on the generic predicate transformers VCGen presented in Figure 4.6. Note that the formulas received as parameters are the same as before. These formulas capture all the executions that reach the current command (the first is the operational encoding and the

[2] http://toccata.lri.fr/gallery/gcd.en.html

$$\mathsf{VCSP}^i : \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{AComm}^{\mathrm{SA}} \to \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert} \times \mathcal{P}(\mathbf{Assert})$$

$$\mathsf{VCSP}^i(\phi, \rho, \mathbf{skip}) = (\top, \top, \bot, \bot, \emptyset)$$

$$\mathsf{VCSP}^i(\phi, \rho, \mathbf{throw}) = (\bot, \bot, \top, \top, \emptyset)$$

$$\mathsf{VCSP}^i(\phi, \rho, x := e) = (x = e, \top, \bot, \bot, \emptyset)$$

$$\mathsf{VCSP}^i(\phi, \rho, \mathbf{assume}\,\theta) = (\top, \theta, \bot, \bot, \emptyset)$$

$$\mathsf{VCSP}^i(\phi, \rho, \mathbf{assert}\,\theta) = \begin{cases} (\top, \top, \bot, \bot, \{\phi \wedge \rho \to \theta\}) & \text{if } i = \mathsf{P} \\ (\top, \theta, \bot, \bot, \{\phi \wedge \rho \to \theta\}) & \text{if } i = \mathsf{PA} \\ (\top, \top, \bot, \bot, \{\rho \to \theta\}) & \text{if } i = \mathsf{G} \\ (\top, \theta, \bot, \bot, \{\rho \to \theta\}) & \text{if } i = \mathsf{GA} \end{cases}$$

$$\mathsf{VCSP}^i(\phi, \rho, C_1\,;\,C_2) = (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \epsilon_1 \vee (\psi_1 \wedge \epsilon_2), \mu_1 \vee (\gamma_1 \wedge \mu_2), \Gamma_1 \cup \Gamma_2)$$

where $(\psi_1, \gamma_1, \epsilon_1, \mu_1, \Gamma_1) = \mathsf{VCSP}^i(\phi, \rho, C_1)$

and $(\psi_2, \gamma_2, \epsilon_2, \mu_2, \Gamma_2) = \mathsf{VCSP}^i(\phi \wedge \psi_1, \rho \wedge \gamma_1, C_2)$

$$\mathsf{VCSP}^i(\phi, \rho, \mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}) = (\psi_1 \vee (\epsilon_1 \wedge \psi_2), \gamma_1 \vee (\mu_1 \wedge \gamma_2), \epsilon_1 \wedge \epsilon_2, \mu_1 \wedge \mu_2, \Gamma_1 \cup \Gamma_2)$$

where $(\psi_1, \gamma_1, \epsilon_1, \mu_1, \Gamma_1) = \mathsf{VCSP}^i(\phi, \rho, C_1)$

and $(\psi_2, \gamma_2, \epsilon_2, \mu_2, \Gamma_2) = \mathsf{VCSP}^i(\phi \wedge \epsilon_1, \rho \wedge \mu_1, C_2)$

$$\mathsf{VCSP}^i(\phi, \rho, \mathbf{if}\,b\,\mathbf{then}\,C_1\,\mathbf{else}\,C_2\,\mathbf{fi}) = ((b \wedge \psi_1) \vee (\neg b \wedge \psi_2), (b \wedge \gamma_1) \vee (\neg b \wedge \gamma_2), (b \wedge \epsilon_1) \vee (\neg b \wedge \epsilon_2), (b \wedge \mu_1) \vee (\neg b \wedge \mu_2), \Gamma_1 \cup \Gamma_2)$$

where $(\psi_1, \gamma_1, \epsilon_1, \mu_1, \Gamma_1) = \mathsf{VCSP}^i(\phi \wedge b, \rho \wedge b, C_1)$

and $(\psi_2, \gamma_2, \epsilon_2, \mu_2, \Gamma_2) = \mathsf{VCSP}^i(\phi \wedge \neg b, \rho \wedge \neg b, C_2)$

Figure 4.6: Predicate transformer VCGens for programs with exceptions ($i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$)

second the axiomatic encoding). Regarding the result of the VCGen note that a tuple of five elements is now returned. The first two elements correspond respectively to the operational and axiomatic encoding of the current command when the program terminates normally, and the next two correspond to the operational and axiomatic encoding of the current command when the program terminates exceptionally. Finally, the last element is the set of VCs that are generated for the current command. Focusing on the **skip** command, it is possible to observe that the formulas corresponding to the normal (resp. exceptional) termination are simply $\top$ (resp. $\bot$), meaning that the **skip** command *always terminates normally* (resp. *never terminates exceptionally*). The opposite happens in the **throw** command: it never terminates normally and it always terminates exceptionally. In fact **throw** is the only atomic command that can terminate exceptionally.

$$\begin{array}{l}
\mathsf{VCCNF}^i \;:\; \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{AComm}^{\mathrm{SA}} \\
\qquad\qquad\qquad \to \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert} \times \mathcal{P}(\mathbf{Assert}) \\
\mathsf{VCCNF}^i(\pi, \phi, \rho, \mathbf{skip}) = (\top, \top, \pi \to \bot, \pi \to \bot, \emptyset) \\
\mathsf{VCCNF}^i(\pi, \phi, \rho, \mathbf{throw}) = (\pi \to \bot, \pi \to \bot, \top, \top, \emptyset) \\
\mathsf{VCCNF}^i(\pi, \phi, \rho, x := e) = (\pi \to x = e, \top, \pi \to \bot, \pi \to \bot, \emptyset) \\
\mathsf{VCCNF}^i(\pi, \phi, \rho, \mathbf{assume}\,\theta) = (\top, \pi \to \theta, \pi \to \bot, \pi \to \bot, \emptyset) \\
\mathsf{VCCNF}^i(\pi, \phi, \rho, \mathbf{assert}\,\theta) = \begin{cases} (\top, \top, \pi \to \bot, \pi \to \bot, \{\phi \wedge \rho \to \pi \to \theta\}) & \text{if } \; i = \mathsf{P} \\ (\top, \pi \to \theta, \pi \to \bot, \pi \to \bot, \{\phi \wedge \rho \to \pi \to \theta\}) & \text{if } \; i = \mathsf{PA} \\ (\top, \top, \pi \to \bot, \pi \to \bot, \{\rho \to \pi \to \theta\}) & \text{if } \; i = \mathsf{G} \\ (\top, \pi \to \theta, \pi \to \bot, \pi \to \bot, \{\rho \to \pi \to \theta\}) & \text{if } \; i = \mathsf{GA} \end{cases} \\
\mathsf{VCCNF}^i(\pi, \phi, \rho, C_1\,;\,C_2) = (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \epsilon_1 \vee (\psi_1 \wedge \epsilon_2), \mu_1 \vee (\gamma_1 \wedge \mu_2), \Gamma_1 \cup \Gamma_2) \\
\qquad \text{where } (\psi_1, \gamma_1, \epsilon_1, \mu_1, \Gamma_1) = \mathsf{VCCNF}^i(\pi, \phi, \rho, C_1) \\
\qquad \text{and } (\psi_2, \gamma_2, \epsilon_2, \mu_2, \Gamma_2) = \mathsf{VCCNF}^i(\pi, \phi \wedge \psi_1, \rho \wedge \gamma_1, C_2) \\
\mathsf{VCCNF}^i(\pi, \phi, \rho, \mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}) = (\psi_1 \vee (\epsilon_1 \wedge \psi_2), \gamma_1 \vee (\mu_1 \wedge \gamma_2), \epsilon_1 \wedge \epsilon_2, \mu_1 \wedge \mu_2, \Gamma_1 \cup \Gamma_2) \\
\qquad \text{where } (\psi_1, \gamma_1, \epsilon_1, \mu_1, \Gamma_1) = \mathsf{VCCNF}^i(\pi, \phi, \rho, C_1) \\
\qquad \text{and } (\psi_2, \gamma_2, \epsilon_2, \mu_2, \Gamma_2) = \mathsf{VCCNF}^i(\pi, \phi \wedge \epsilon_1, \rho \wedge \mu_1, C_2) \\
\mathsf{VCCNF}^i(\pi, \phi, \rho, \mathbf{if}\;b\;\mathbf{then}\;C_1\;\mathbf{else}\;C_2\;\mathbf{fi}) = (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \epsilon_1 \wedge \epsilon_2, \mu_1 \wedge \mu_2, \Gamma_1 \cup \Gamma_2) \\
\qquad \text{where } (\psi_1, \gamma_1, \epsilon_1, \mu_1, \Gamma_1) = \mathsf{VCCNF}^i(\pi \wedge b, \phi, \rho, C_1) \\
\qquad \text{and } (\psi_2, \gamma_2, \epsilon_2, \mu_2, \Gamma_2) = \mathsf{VCCNF}^i(\pi \wedge \neg b, \phi, \rho, C_2)
\end{array}$$

Figure 4.7: Conditional normal form VCGens for programs with exceptions ($i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$)

The analysis of the VCGen becomes more interesting in the composite commands. In these cases, both sub-commands can terminate normally and exceptionally, therefore, the results of applying the VCGen recursively to these sub-commands must be merged together. Basically the sequence command only terminates normally if both C_1 and C_2 terminate normally. Hence the operational (resp. axiomatic) encoding is $\psi_1 \wedge \psi_2$ (resp. $\gamma_1 \wedge \gamma_2$). On the other hand, it can terminate exceptionally if either C_1 terminates exceptionally or C_1 terminates normally but then C_2 terminates exceptionally. Thus the exceptional operational (resp. axiomatic) encoding is $\epsilon_1 \vee (\psi_1 \wedge \epsilon_2)$ (resp. $\mu_1 \vee (\gamma_1 \wedge \mu_2)$). Note how in the try-catch command the opposite happens: the program terminates normally if either C_1 terminates normally or, if C_1 terminates exceptionally and C_2 normally; on the contrary it terminates exceptionally only if C_1 and C_2 terminate exceptionally. The if command does not bring any additional complication: it terminates normally if b is true and C_1 terminates normally, or else, if b is false and C_2 terminates normally; analogously for exceptional termination.

The verification conditions of the program C are obtained as follows. Let $(\,\psi,\,\gamma,\,\epsilon,\,\mu,\,\Gamma\,) =$

$$\mathsf{VCLin}^i : \mathbf{AComm}^{\mathrm{SA}} \to \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert} \times \mathbf{Assert}$$

$$\mathsf{VCLin}^i(\mathbf{skip}) = (\top, \top, \bot, \bot, \top)$$

$$\mathsf{VCLin}^i(\mathbf{throw}) = (\bot, \bot, \top, \top, \top)$$

$$\mathsf{VCLin}^i(x := e) = (x = e, \top, \bot, \bot, \top)$$

$$\mathsf{VCLin}^i(\mathbf{assume}\,\theta) = (\top, \theta, \bot, \bot, \top)$$

$$\mathsf{VCLin}^i(\mathbf{assert}\,\theta) = \begin{cases} (\top, \top, \bot, \bot, \theta) & \text{if } i \in \{\mathsf{P}, \mathsf{G}\} \\ (\top, \theta, \bot, \bot, \theta) & \text{if } i \in \{\mathsf{PA}, \mathsf{GA}\} \end{cases}$$

$$\mathsf{VCLin}^i(C_1 \,;\, C_2) = \begin{cases} (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \epsilon_1 \vee (\psi_1 \wedge \epsilon_2), & \text{if } i \in \{\mathsf{P}, \mathsf{PA}\} \\ \quad \mu_1 \vee (\gamma_1 \wedge \mu_2), \delta_1 \wedge (\psi_1 \wedge \gamma_1 \to \delta_2)) & \\ (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \epsilon_1 \vee (\psi_1 \wedge \epsilon_2), & \text{if } i \in \{\mathsf{G}, \mathsf{GA}\} \\ \quad \mu_1 \vee (\gamma_1 \wedge \mu_2), \delta_1 \wedge (\gamma_1 \to \delta_2)) & \end{cases}$$

where $(\psi_1, \gamma_1, \epsilon_1, \mu_1, \delta_1) = \mathsf{VCLin}^i(C_1)$ and $(\psi_2, \gamma_2, \epsilon_2, \mu_2, \delta_2) = \mathsf{VCLin}^i(C_2)$

$$\mathsf{VCLin}^i(\mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}) = \begin{cases} (\psi_1 \vee (\epsilon_1 \wedge \psi_2), \gamma_1 \vee (\mu_1 \wedge \gamma_2), & \text{if } i \in \{\mathsf{P}, \mathsf{PA}\} \\ \quad \epsilon_1 \wedge \epsilon_2, \mu_1 \wedge \mu_2, \delta_1 \wedge (\epsilon_1 \wedge \mu_1 \to \delta_2)) & \\ (\psi_1 \vee (\epsilon_1 \wedge \psi_2), \gamma_1 \vee (\mu_1 \wedge \gamma_2), & \text{if } i \in \{\mathsf{G}, \mathsf{GA}\} \\ \quad \epsilon_1 \wedge \epsilon_2, \mu_1 \wedge \mu_2, \delta_1 \wedge (\mu_1 \to \delta_2)) & \end{cases}$$

where $(\psi_1, \gamma_1, \epsilon_1, \mu_1, \delta_1) = \mathsf{VCLin}^i(C_1)$ and $(\psi_2, \gamma_2, \epsilon_2, \mu_2, \delta_2) = \mathsf{VCLin}^i(C_2)$

$$\begin{aligned} \mathsf{VCLin}^i(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}) = ((b \wedge \psi_1) \vee (\neg b \wedge \psi_2), (b \wedge \gamma_1) \vee (\neg b \wedge \gamma_2), \\ (b \wedge \epsilon_1) \vee (\neg b \wedge \epsilon_2), (b \wedge \mu_1) \vee (\neg b \wedge \mu_2), \\ (b \to \delta_1) \wedge (\neg b \to \delta_2)) \end{aligned}$$

where $(\psi_1, \gamma_1, \epsilon_1, \mu_1, \delta_1) = \mathsf{VCLin}^i(C_1)$ and $(\psi_2, \gamma_2, \epsilon_2, \mu_2, \delta_2) = \mathsf{VCLin}^i(C_2)$

Figure 4.8: Lin VCGens ($i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$)

$\mathsf{VCSP}^i(\top, \top, C)$. If $i \in \{\mathsf{P}, \mathsf{PA}\}$, the partial contexts VC are given by Γ. Otherwise, if $i \in \{\mathsf{G}, \mathsf{GA}\}$, the global context VCs are $\{\psi \to \bigwedge \Gamma, \epsilon \to \bigwedge \Gamma\}$. With a global context the VCs are formed by adding the context to the conditions in Γ. However, in a language with exceptions, there might be as many contexts as the number of possible terminations (i.e. kinds of exceptions): in the simple case we are considering here, we have the normal operational context ψ and the exceptional operational context ϵ.

The previous discussion about extending VCSP to programs with exceptions also applies to the rest of the generic VCGens. The generic VCGen based on conditional normal form is presented in Figure 4.7. Again, the result of the algorithm is a tuple of five elements as in VCSP for capturing the normal and exceptional program encoding.

Let $(\,\psi,\, \gamma,\, \epsilon,\, \mu,\, \Gamma\,) = \mathsf{VCCNF}^i(\top, \top, \top, C)$. If $i \in \{\mathsf{P}, \mathsf{PA}\}$, the partial contexts VC are given

by Γ. Otherwise, if $i \in \{\mathsf{G}, \mathsf{GA}\}$, the global context VCs are $\{\psi \to \bigwedge\Gamma, \epsilon \to \bigwedge\Gamma\}$. Again, the VCSSA^i differs from VCCNF^i only in the assignment case. In the special case of assignments with conditionals expressions it becomes $\mathsf{VCSSA}^i(\pi, \phi, \rho, x := b\,?\,y\ :\ z) = (\,(b \to x = y) \wedge (\neg b \to x = z),\ \top,\ \bot,\ \bot,\ \emptyset\,)$ and for other cases it becomes $\mathsf{VCSSA}^i(\pi, \phi, \rho, x := e) = (\,x = e,\ \top,\ \bot,\ \bot,\ \emptyset\,)$.

For the VCLin presented in Figure 4.8 the VCs are generated as follows. If $(\,\psi,\ \gamma,\ \epsilon,\ \mu,\ \delta\,) = \mathsf{VCLin}^i(C)$, and $i \in \{\mathsf{G}, \mathsf{GA}\}$ then the VC is δ, otherwise, if $i \in \{\mathsf{G}, \mathsf{GA}\}$ then the global context VCs are $\{\psi \to \delta, \epsilon \to \delta\}$.

Asymptotic Analysis of VCLin and VCSP for Programs with Exceptions. The inclusion of exceptions in the language changes completely the overall size of the VCs in the worst case. In fact, it is well known that the presence of exceptions can lead to VCs of exponential size with respect to the program being verified [52, 11] but the intuition for this not clear in the cited works. We describe below how exceptions influence the size of the VCs.

Let us start with the VCLin and VCSP family of VCGens. For this discussion we will only consider the normal and exceptional operational encoding of programs, and thus it is indifferent which variant of VCLin or VCSP is used: for the sake of consistency with [11], we will use $\mathsf{VCLin}^{\mathsf{PA}}$, but the discussion extends to the others. Assume that $C_0, C_1, C_2, \ldots$ are arbitrary programs that can terminate either normally or exceptionally, and that $\mathsf{VCLin}^{\mathsf{PA}}(C_i) = (\psi_i, _, \epsilon_i, _, _)$, for each $i \in \{0, 1, 2, \ldots\}$. Therefore the operational encoding of C_i is captured by ψ_i in case of normal termination and by ϵ_i in case of exceptional termination. Having this in mind, it follows directly from the definition of $\mathsf{VCLin}^{\mathsf{PA}}$ that $\mathsf{VCLin}^{\mathsf{PA}}(C_0\,;\,C_1) = (\psi_0 \wedge \psi_1, _, \epsilon_0 \vee (\psi_0 \wedge \epsilon_1), _, _)$, and thus $\mathsf{VCLin}^{\mathsf{PA}}(\mathbf{try}\,C_0\,;\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}) = (\boldsymbol{\psi_0} \wedge \psi_1 \vee ((\epsilon_0 \vee (\boldsymbol{\psi_0} \wedge \epsilon_1)) \wedge \psi_2), _, (\epsilon_0 \vee (\psi_0 \wedge \epsilon_1)) \wedge \epsilon_2, _, _)$. Note that, the encoding of the program C_0 was replicated once in the operational encoding corresponding to normal termination. In order to keep things understandable, in what follows we will consider that $\psi_{0,1,2} = \boldsymbol{\psi_0} \wedge \psi_1 \vee ((\epsilon_0 \vee (\boldsymbol{\psi_0} \wedge \epsilon_1)) \wedge \psi_2)$ and $\epsilon_{0,1,2} = (\epsilon_0 \vee (\psi_0 \wedge \epsilon_1)) \wedge \epsilon_2$.

Let us now consider an additional command C_3. Then $\mathsf{VCLin}^{\mathsf{PA}}(\mathbf{try}\,C_0\,;\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}\,;\,C_3)$ $= (\psi_{0,1,2} \wedge \psi_3, _, \epsilon_{0,1,2} \vee (\psi_{0,1,2} \wedge \epsilon_3), _, _)$. In the formula corresponding to normal termination, there was no change in the formulas being replicated. Nonetheless, in the formula corresponding to exceptional termination, ψ_0 occurs now three times and ϵ_0, and ϵ_1 occur twice. Again, let $\psi_{0,1,2,3} = \psi_{0,1,2} \wedge \psi_3$ and $\epsilon_{0,1,2,3} = \epsilon_{0,1,2} \vee (\psi_{0,1,2} \wedge \epsilon_3)$. Then $\mathsf{VCLin}^{\mathsf{PA}}(\mathbf{try}\,\mathbf{try}\,C_0\,;\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}\,;\,C_3\,\mathbf{catch}\,C_4\,\mathbf{hc}) = (\psi_{0,1,2,3} \vee (\epsilon_{0,1,2,3} \wedge \psi_4), _, \epsilon_{0,1,2,3} \wedge \epsilon_4, _, _)$. Now, in terms of replication nothing changed w.r.t. exceptional termination, but in the formula capturing normal termination, ψ_0 occurs now five times, ψ_1 and ψ_2 twice, and ϵ_0 and ϵ_1 three times. Now if we consider the encoding of the command $\mathbf{try}\,\mathbf{try}\,\mathbf{try}\,C_0\,;\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}\,;\,C_3\,\mathbf{catch}\,C_4\,\mathbf{hc}\,;\,C_5\,\mathbf{catch}\,C_6\,\mathbf{hc}$ in case of normal termination, the number of occurrences of these formulas will actually duplicate which justifies the exponential explosion in the worst case.

Example 4.16. *The following program, let us call it C, originates the replication described above.*

try
  if b_1 **then** $x_1 := y_1$ **else** $x_1 := z_1$; **throw fi** ;
  if b_2 **then** $x_2 := y_2$ **else** $x_2 := z_2$; **throw fi**
catch
  if b_3 **then** $x_3 := y_3$ **else** $x_3 := z_3$; **throw fi**
hc

For this program we have $\mathsf{VCLin}^{\mathsf{PA}}(C) = ((\boldsymbol{b_1 \wedge x_1 = y_1} \wedge b_2 \wedge x_2 = y_2) \vee (((\neg b1 \wedge x_1 = z_1) \vee ((\boldsymbol{b_1 \wedge x_1 = y_1}) \wedge (\neg b_2 \wedge x_2 = z_2))) \wedge (b_3 \wedge x_3 = y_3)), _, ((\neg b_1 \wedge x_1 = y_1) \vee (b_1 \wedge x_1 = y_1 \wedge \neg b_2 \wedge x_2 = z_2)) \wedge (\neg b_3 \wedge x_3 = z_3), _, _)$*, and as shown in bold, the part of the encoding* $b_1 \wedge x_1 = y_1$ *is duplicated. If we increase the size of this program as described above the growth of the size of the VCs will be exponential.*

Towards Avoiding Explosion in CNF. The variants of VCCNF presented in Figure 4.7 also lead to exponential explosion due to the same reasons as explained above. Let C be a program that can terminate either normally or exceptionally. The difficulty of avoiding the duplication of formulas in the encoding of C, comes from the fact that we have one formula to capture the normal termination and another to capture the exceptional termination, and the only way to know when C terminates normally or exceptionally is by inspecting these formulas. As such, in the sequence and in the try-catch case, they need to be merged together to capture the correct flow of execution.

Note that in CNF each atomic command is guarded by a path condition, but such a path condition in the VCGens of Figure 4.7 only take into account the branching conditions. For instance, assume that we have a program of the form $C_1 ; C_2$, and that C_1 can terminate either normally or exceptionally. Moreover, assume that the path condition for the program $C_1 ; C_2$ to be executed is π. With this information we know that C_1 will be executed whenever π holds, and thus each sub-command of C_1 will have π in its path condition. But what about C_2? It is clear that π must hold, but it is also required that C_1 terminates normally. The problem is that the VCGen of Figure 4.7 does not have any condition that indicates when C_1 terminates normally or exceptionally.

This motivates the introduction of a new VCGen based on the one of Figure 4.7. The idea is that we will have only one formula that will capture both the normal and exceptional execution of a program, but additional formulas will capture the path conditions for a program to terminate normally or exceptionally. Such a generic VCGen is given in Figure 4.9 and incorporates the variants $\{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$ as in the previous cases. The function receives a program and the respective path condition for it to be reached, and it returns a tuple containing the following:

- the operational encoding of the program (covering both normal or exceptional termination).
- the axiomatic encoding of the program (covering both normal or exceptional termination).

$$\mathsf{VCCNF_e}^i : \textbf{Assert} \times \textbf{AComm}^{\mathrm{SA}} \to \textbf{Assert} \times \textbf{Assert} \times \textbf{Assert} \times \textbf{Assert} \times \textbf{Assert}$$

$$\mathsf{VCCNF_e}^i(\pi, \textbf{skip}) = (\top, \top, \top, \bot, \top)$$

$$\mathsf{VCCNF_e}^i(\pi, \textbf{throw}) = (\top, \top, \bot, \top, \top)$$

$$\mathsf{VCCNF_e}^i(\pi, x := e) = (\pi \to x = e, \top, \top, \bot, \top)$$

$$\mathsf{VCCNF_e}^i(\pi, \textbf{assume}\,\theta) = (\top, \pi \to \theta, \top, \bot, \top)$$

$$\mathsf{VCCNF_e}^i(\pi, \textbf{assert}\,\theta) = \begin{cases} (\top, \top, \top, \bot, \pi \to \theta) & \text{if } i \in \{\mathsf{P}, \mathsf{G}\} \\ (\top, \pi \to \theta, \top, \bot, \pi \to \theta) & \text{if } i \in \{\mathsf{PA}, \mathsf{GA}\} \end{cases}$$

$$\mathsf{VCCNF_e}^i(\pi, C_1\,;\,C_2) = \begin{cases} (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \lambda_1 \wedge \lambda_2, & \text{if } i \in \{\mathsf{P}, \mathsf{PA}\} \\ \quad \tau_1 \vee (\lambda_1 \wedge \tau_2), \delta_1 \wedge (\psi_1 \wedge \gamma_1 \to \delta_2)) & \\ (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \lambda_1 \wedge \lambda_2, & \text{if } i \in \{\mathsf{G}, \mathsf{GA}\} \\ \quad \tau_1 \vee (\lambda_1 \wedge \tau_2), \delta_1 \wedge (\gamma_1 \to \delta_2)) & \end{cases}$$

where $(\psi_1, \gamma_1, \lambda_1, \tau_1) = \mathsf{VCCNF_e}^i(\pi, C_1)$ and $(\psi_2, \gamma_2, \lambda_2, \tau_2) = \mathsf{VCCNF_e}^i(\pi \wedge \lambda_1, C_2)$

$$\mathsf{VCCNF_e}^i(\pi, \textbf{try}\,C_1\,\textbf{catch}\,C_2\,\textbf{hc}) = \begin{cases} (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \lambda_1 \vee (\tau_1 \wedge \lambda_2), & \text{if } i \in \{\mathsf{P}, \mathsf{PA}\} \\ \quad \tau_1 \wedge \tau_2, \delta_1 \wedge (\psi_1 \wedge \gamma_1 \to \delta_2)) & \\ (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, \lambda_1 \vee (\tau_1 \wedge \lambda_2), & \text{if } i \in \{\mathsf{G}, \mathsf{GA}\} \\ \quad \tau_1 \wedge \tau_2, \delta_1 \wedge (\gamma_1 \to \delta_2)) & \end{cases}$$

where $(\psi_1, \gamma_1, \lambda_1, \tau_1, \delta_1) = \mathsf{VCCNF_e}^i(\pi, C_1)$ and $(\psi_2, \gamma_2, \lambda_2, \tau_2, \delta_2) = \mathsf{VCCNF_e}^i(\pi \wedge \tau_1, C_2)$

$$\mathsf{VCCNF_e}^i(\pi, \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}) = (\psi_1 \wedge \psi_2, \gamma_1 \wedge \gamma_2, (b \wedge \lambda_1) \vee (\neg b \wedge \lambda_2), (b \wedge \tau_1) \vee (\neg b \wedge \tau_2), \delta_1 \wedge \delta_2)$$

where $(\psi_1, \gamma_1, \lambda_1, \tau_1, \delta_1) = \mathsf{VCCNF_e}^i(\pi \wedge b, C_1)$
and $(\psi_2, \gamma_2, \lambda_2, \tau_2, \delta_2) = \mathsf{VCCNF_e}^i(\pi \wedge \neg b, C_2)$

Figure 4.9: Conditional normal form VCGens avoiding replication of code ($i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$)

- the condition for the program to terminate normally.
- the condition for the program to terminate exceptionally.
- the property to be proved.

The skip and throw command do not produce any operational or axiomatic encoding and thus return $\top$. Moreover, skip, assume, and assert commands always terminate normally and never exceptionally, and vice-versa for the throw command. The assignment will originate a formula guarded with the received path condition. Note that if the code is in SSA, the optimization described in Section 4.3 can be used. Nonetheless, the path condition cannot be avoided in the axiomatic context produced by the assume and assert command (when using PA or GA) in the property to be proved. With respect to the if command, the condition for it

to terminate normally consists in a disjunction of the condition for each branch to terminate normally with its branching condition (analogously for the condition capturing exceptional termination).

Finally, in the sequence and in the try-catch command the operational and axiomatic encodings are both just a conjunction of formulas, which avoids the exponential explosion described above. However, the VCGen introduces another source for a possible exponential growth. The problems are on the path condition for a sequence to terminate exceptionally, and for a try-catch to terminate normally. For the former the condition is $\tau_1 \vee (\lambda_1 \wedge \tau_2)$, and for the latter $\lambda_1 \vee (\tau_1 \wedge \lambda_2)$. These conditions clearly mix the path conditions corresponding to normal and exceptional termination and may lead to an exponential explosion in the size of the path conditions.

Example 4.17. *Consider the program in Example 4.16 and a variant of* $\mathsf{VCCNF_e}$. *The condition for the program to terminate normally is* $\boldsymbol{b_1} \wedge b_2 \vee ((\neg b_1 \vee (\boldsymbol{b_1} \wedge \neg b_2)) \wedge b_3)$. *Note how the condition* b_1 *in bold is duplicated. Nonetheless, the encoding containing the atomic commands is not duplicated. In this case, the VCGen generates the following operational encoding.*

$$
\begin{aligned}
&(b_1 \rightarrow x_1 = y_1) \wedge (\neg b_1 \rightarrow x_1 = z_1) \wedge \\
&\quad (b_1 \wedge b_2 \rightarrow x_2 = y_2) \wedge (b_1 \wedge \neg b_2 \rightarrow x_1 = z_2) \wedge \\
&\quad\quad ((\neg b_1 \vee (b_1 \wedge \neg b_2)) \wedge b_3 \rightarrow x_3 = y_3) \wedge ((\neg b_1 \vee (b_1 \wedge \neg b_2)) \wedge \neg b_3 \rightarrow x_3 = z_3)
\end{aligned}
$$

In fact the duplication of path conditions is not the only limitation of the $\mathsf{VCCNF_e}$ family of VCGens. Note that the returned path conditions are never simplified.

Example 4.18. *Consider the following program:*

try
  if b_1 **then** $x_1 := y_1$ **else** $x_1 := z_1$; **throw fi**
catch
  if b_2 **then** $x_2 := y_2$ **else** $x_2 := z_2$ **fi**
hc

The condition capturing normal termination is $b_1 \vee (\neg b_1 \wedge (b_2 \vee \neg b_2))$, *which is equivalent to the condition* $\top$, *but the VCGen does not make this type of simplification. Therefore, if the program above, let us call it* C, *is part of a sequence of commands, such as for instance* $C\,;\, x_3 := y_3$ *then the operational encoding of* $x_3 := y_3$ *is* $b_1 \vee (\neg b_1 \wedge (b_2 \vee \neg b_2)) \rightarrow x_3 = y_3$.

The limitation referred in the example above motivates the use of some solver in order to keep path conditions in a simplified form. We postpone the discussion of a concrete method for dealing with this to Chapter 7, nonetheless it should be noted that such a solver will never

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathsf{VCG_T} : \mathbf{Assert} \times \mathbf{AComm}^{\text{SA}} \times \mathbf{Assert} \times \mathbf{Assert}} \\
\multicolumn{3}{r}{\times\{\mathsf{SP}, \mathsf{CNF}, \mathsf{LIN}, \mathsf{SSA}\} \times \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\} \rightarrow \mathcal{P}(\mathbf{Assert})} \\
\mathsf{VCG_T}(\phi', C, \psi', \epsilon', \mathsf{SP}, i) & = & \mathsf{let}\ (\psi, \gamma, \epsilon, \mu, \Gamma) = \mathsf{VCSP}^i(\top, \phi', C)\ \mathsf{in} \\
& & \mathsf{if}\ i \in \{\mathsf{P}, \mathsf{PA}\}\ \mathsf{then}\ \Gamma \cup \{\phi' \land \psi \land \gamma \rightarrow \psi', \phi' \land \epsilon \land \mu \rightarrow \epsilon'\} \\
& & \mathsf{else}\ \{\phi' \land (\psi \lor \epsilon) \rightarrow \bigwedge \Gamma, \phi' \land \psi \land \gamma \rightarrow \psi', \phi' \land \epsilon \land \mu \rightarrow \epsilon'\} \\
\mathsf{VCG_T}(\phi', C, \psi', \epsilon', \mathsf{CNF}, i) & = & \mathsf{let}\ (\psi, \gamma, \epsilon, \mu, \Gamma) = \mathsf{VCCNF}^i(\top, \top, \phi', C)\ \mathsf{in} \\
& & \mathsf{if}\ i \in \{\mathsf{P}, \mathsf{PA}\}\ \mathsf{then}\ \Gamma \cup \{\phi' \land \psi \land \gamma \rightarrow \psi', \phi' \land \epsilon \land \mu \rightarrow \epsilon'\} \\
& & \mathsf{else}\ \{\phi' \land (\psi \lor \epsilon) \rightarrow \bigwedge \Gamma, \phi' \land \psi \land \gamma \rightarrow \psi', \phi' \land \epsilon \land \mu \rightarrow \epsilon'\} \\
\mathsf{VCG_T}(\phi', C, \psi', \epsilon', \mathsf{LIN}, i) & = & \mathsf{let}\ (\psi, \gamma, \epsilon, \mu, \delta) = \mathsf{VCLin}^i(C)\ \mathsf{in} \\
& & \mathsf{if}\ i \in \{\mathsf{P}, \mathsf{PA}\}\ \mathsf{then}\ \{\phi' \rightarrow \delta, \phi' \land \psi \land \gamma \rightarrow \psi', \phi' \land \epsilon \land \mu \rightarrow \epsilon'\} \\
& & \mathsf{else}\ \{\phi' \land (\psi \lor \epsilon) \rightarrow \delta, \phi' \land \psi \land \gamma \rightarrow \psi', \phi' \land \epsilon \land \mu \rightarrow \epsilon'\} \\
\mathsf{VCG_T}(\phi', C, \psi', \epsilon', \mathsf{SSA}, i) & = & \mathsf{let}\ (\psi, \gamma, \epsilon, \mu, \Gamma) = \mathsf{VCSSA}^i(\top, \top, \phi', C)\ \mathsf{in} \\
& & \mathsf{if}\ i \in \{\mathsf{P}, \mathsf{PA}\}\ \mathsf{then}\ \Gamma \cup \{\phi' \land \psi \land \gamma \rightarrow \psi', \phi' \land \epsilon \land \mu \rightarrow \epsilon'\} \\
& & \mathsf{else}\ \{\phi' \land (\psi \lor \epsilon) \rightarrow \bigwedge \Gamma, \phi' \land \psi \land \gamma \rightarrow \psi', \phi' \land \epsilon \land \mu \rightarrow \epsilon'\}
\end{array}
$$

Figure 4.10: Unified VCGen for Hoare triples

eliminate the problem of exponential explosion, since the conditions originated by a program that grows with the pattern described in Example 4.16, cannot be simplified.

4.5 Unifying the Generation of Verification Conditions

All the VCGens can be integrated into a single unified generator that will receive a triple and return a set of verification conditions, that are valid if and only if the triple is valid. The function implementing such a unified VCGen is given in Figure 4.10 and it receives a precondition, an SA program, two postconditions (one for normal and another for exceptional termination), the generic VCGen to be used (SP, CNF, LIN, or SSA), and the respective variant (P, PA, G, or GA). With this, the function uses the generic VCGens from the previous section to generate the encoding of the program and the VCs, and returns a set of VCs to be proved. If the VCs are valid then so is the Hoare triple.

Note that in the VCSP, VCCNF, and VCSSA families of VCGens, the precondition from the received triple is passed to the concrete VCGen as axiomatic context, which will imply that it will appear in all the VCs referring to asserts in the program. In the VCLin family of VCGens, since only the program is passed as parameter, the precondition must be used to prove the single VC that is generated from all the asserts in the program. Moreover, in order to prove that the postconditions are met when the program terminates normally (resp. exceptionally),

two new VCs must be added to those generated from the program: these news VCs will ensure that the precondition, together with the operational and axiomatic contexts are sufficient to prove the normal (resp. exceptional) postcondition.

If one wants to check the validity of the asserts in a program C, then it is possible to use the function $\mathsf{VCG_T}$ to check the validity of the triple $\{\top\}\, C\, \{\top, \top\}$. A different approach for checking the validity of a triple would be to add the precondition and the postconditions as part of the program through assumes and asserts. So, for a triple $\{\phi\}\, C\, \{\psi, \epsilon\}$ we would create the program $C' = \mathbf{assume}\, \phi\, ;\, \mathbf{try}\, C\, ;\, \mathbf{assert}\, \psi\, \mathbf{catch}\, \mathbf{assert}\, \epsilon\, \mathbf{hc}$, and then use a different unified VCGen that would receive only a program and generate a set of VCs.

4.6 Related Work

Based on different families of verification tools, we have identified in this chapter a visual representation and a unified method for generating verification conditions. The VCGens from this chapter differ from the one of the previous chapter in that the input programs may contain assume, assert, and exceptions and do not contain an iterating construct. Even though VC_{ISA} from the previous section generates VCs that are similar to those generated by VCSP with partial contexts, the rest of the VCGens could easily be adapted for generating VCs for programs ranging over $\mathbf{AComm}^{\text{ISA}}$.

Some published works have already contributed towards a uniform formulation for verification condition generation. A first step taken by Gordon et al. [58] covers some ground on proving the correctness of Hoare triples based on forward computation of postconditions and comparing predicate transformers with software model checking techniques. Godefroid et al. [56] report on the techniques used by some tools for creating logical encodings of programs. The authors briefly mention the complexity of each method but no comparisons between the logical encodings are offered, neither empirically nor theoretically. Cruz et al. [41] present in a systematic way VCGen algorithms for code in *static single-assignment* (SSA) form. Although these works demonstrate the importance of such an approach, they fall short of both proposing a uniform framework at the theoretical level and presenting a suitable empirical evaluation of different VCGens. Moreover, the interplay between baseline methods and optimization techniques was not addressed.

Let us now consider how the existing verification tools fit in our extended cube. First of all, it should be said that no tool, as far as we know, uses global contexts: all existing tools are located in the bottom face of the cube. Although the pure $\mathsf{VCCNF}^{\mathsf{G}}$, using a global context, was originally introduced for CBMC [32], and described in [8, 37], the current version of the tool seems to use the CNF-encoding with partial contexts together with the SSA-based optimization. Therefore, CBMC is likely based on $\mathsf{VCSSA}^{\mathsf{P}}$.

The Boogie [10] and Why3 [50] deductive verifiers are located on the left-hand face. Both use partial contexts and while Boogie uses *assert* conditions as lemmas, Why3 has two different commands: one that uses assert as lemmas and another that does not (more on this at the

beginning of Section 7.2). Boogie incorporates $\mathsf{VCLin}^{\mathsf{PA}}$, and allows VCs to be split which originates VCs similar to those generated by VCSP. Why3 has traceability as an important feature. The method used by default to generate VCs is based on a potentially exponential path enumeration (see Section 7.2 for details), which has advantages from the point of view of traceability: the Why3 graphical interface is able to highlight execution paths corresponding to selected VCs. Nevertheless, Why3 also implements and makes available (through a command-line switch) the $\mathsf{VCLin}^{\mathsf{PA}}$ algorithm, and a splitting operation can then be used explicitly to separate the single VC into a set of 'single-goal' VCs.

We note that there exists other deductive verification tools that were not examined here because they do not rely on the generation of verification conditions in the sense considered in this thesis. For instance, VeriFast [69], which has support for *separation logic* [97], relies on *forward symbolic execution* [58]; KIV [47] also supports separation logic, but it is an interactive tool based on higher-order logic. KeY [1] is also an interactive tool and relies on *dynamic logic based on a sequent calculus* [16].

Finally, it should be mentioned that tools based on other families of software analysis techniques, such as model checkers based on existential abstraction and symbolic execution tools, often also integrate a VCGen as an auxiliary component. For instance, in the TRACER tool [70], a VCGen is used to determine when a given execution path subsumes another (the problem of exponential explosion is solved by resorting to an *interpolation* technique). Therefore we believe that our work here may also, indirectly, be of use to developers of verification tools outside the deductive and bounded model checking families.

Chapter 5

A Verification Workflow Based on SA Programs

The previous chapter explored multiple ways of generating verification conditions for single-assignment programs. This was done by providing different algorithms, based on different program verification techniques, that take a program and return a set of verification conditions to be checked by a logic solver. Supposedly, if the VCs are shown to be valid, then the program is correct with respect to the specification. Nonetheless, no formal relation was shown to exist between the VCGens introduced in the previous chapter, and no formal relation was established with the theoretical foundations of program verification.

In this chapter we do precisely this. We start by expanding the language semantics, initially presented in Chapter 2 for programs with *assumes*, *asserts*, and, *exceptions*, and then adapt the inference system for Hoare triples containing programs constructed over this new class. We then introduce a new class of SA programs that stands in contrast with the one from Chapter 3 because it does not contain an iterating construct. An inference system for Hoare triples containing SA programs is then proposed and shown to be sound and relatively complete with respect to the semantics of SA programs. This system will be particularly useful for showing that the VCGens from the previous sections are all sound and complete (besides generating equivalent VCs).

In a similar way to the approach of Chapter 3, and since our main goal is to check the correctness of non-SA programs, we also formalize the correctness of a verification framework's workflow based on the translation of programs into SA form. We present in this chapter the notion of SA translation and leave for the next chapter the proposal of a concrete translation together with the proof that it complies with the notion of SA translation.

The chapter is organized in the following way. The next section expands the While language with assume, assert and exceptions, and introduces an adequate operational and axiomatic programming semantics. Section 5.2 introduces the notion of single-assignment program, together with an inference system that is shown to be sound and complete. The soundness and completeness proof for the VCGen cube is given in Section 5.3 together with the equivalence result.

Finally Section 5.4 introduces a proved verification framework building on the previous results, and Section 5.5 presents related work.

5.1 Semantics for Programs with Assume, Assert, and Exceptions

The language for this and the next chapter will be based on a While language like the one presented in Sections 2.1 and 2.3 but extended with exceptions, assume and assert commands. The abstract syntax for $C \in \mathbf{AComm}$ is as follows:

$$
\begin{aligned}
C \ ::= & \ \mathbf{skip} \mid \mathbf{throw} \mid x := e \mid \mathbf{assume}\,\theta \mid \mathbf{assert}\,\theta \mid C\,;C \mid \mathbf{try}\,C\,\mathbf{catch}\,C\,\mathbf{hc} \\
& \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C\ \mathbf{fi} \mid \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \mathbf{od}
\end{aligned}
$$

Observe that the while command is annotated with an invariant, and thus, it should be referred that we are in fact extending the class of annotated programs **AComm** introduced in Section 2.3, and not the class **Comm** of programs introduced in Section 2.1. Nonetheless, since we did not provide an operational semantics for **AComm** (in Section 2.3 we had a function to transform programs from **AComm** into commands from **Comm**), we will adapt the operational semantics introduced Section 2.1 and simply ignore the loop invariants annotated in the program. We made this decision to avoid the introduction of multiple classes of program and causing obfuscation in the formulation of results. The results that will be presented can be easily adapted to an approach based on two classes of programs, like the one used in Chapters 2 and 3, where two classes of programs are considered (one where loops are not annotated and another with annotated loops). Still with respect to this matter, in what follows and when reasoning exclusively about the semantics of programs we will omit the annotations in loops, meaning that the result holds for every annotation. This means that we will write **while** b **do** C **od** instead of **while** b **do** $\{\theta\}$ C **od**, whenever θ is not relevant for the result being established.

The above syntax declaration also has some degree of redundancy, in the sense that in terms of verification, loops can be encoded through assume and asserts statements as explained at the end of Section 2.4. We should note two facts: first of all, the referred approach for removing loops has never, as long as we know, been shown to be sound and/or complete prior to our work (we do prove its soundness and completeness in the next chapter); the second justification is that following this approach we allow for a verification workflow where the source language contains while loops and does not contain assume and asserts commands, and a different intermediate language, intended for verification, contains assume and assert commands and does not contain while loops.

As was said in the previous chapter, a program with exceptions may terminate in different kinds of states and thus, the set of states representing normal termination should be distinct from the set of states representing exceptional termination. A termination state representing normal termination will be represented by $\mathbf{n}(s)$, and a termination state representing exceptional

termination will be represented by $\mathbf{e}(s)$. What is more, since our language has assumes and asserts, the program can terminate in other kinds of states as well. When an asserted property does not hold in the current state, the execution of the program should fail in some way and terminate immediately. On the other hand, when an assumed property does not hold in the current state, the execution of the program blocks, since the state in which the subsequent commands would be executed is seen as inconsistent. Having this in mind, when an assert fails, the program terminates immediately in a special termination state error denoted by $\bullet$; when an assume does not hold in the current state, the program goes into a blocked state, denoted by $\blacksquare$. It will be relevant to distinguish between the set of states that contain the $\blacksquare$ state and the set that does not contain it. The set of termination states containing normal and exceptional termination, as well as the error state, will be denominated by $\Sigma_\bullet$. The syntax for $\sigma \in \Sigma_\bullet$ is as follows:

$$\sigma ::= \ \mathbf{n}(s) \ \mid \ \mathbf{e}(s) \ \mid \bullet$$

The set of states containing the previous and the blocked state will be denoted by $\Sigma_\bullet^\blacksquare$ and the syntax for $\sigma \in \Sigma_\bullet^\blacksquare$ is given as follows:

$$\sigma ::= \ \mathbf{n}(s) \ \mid \ \mathbf{e}(s) \ \mid \bullet \ \mid \blacksquare$$

We are now ready to introduce a big-step semantics for programs with exceptions, assume and assert commands. Note that the relation has now a different set of initial states Σ, and termination states $\Sigma_\bullet^\blacksquare$. Programs always execute from some $s \in \Sigma$ and terminate in some state $\sigma \in \Sigma_\bullet^\blacksquare$.

Definition 5.1 (Natural semantics)**.** *The evaluation relation for* **AComm** *is defined as the smallest relation* $\rightsquigarrow \subseteq$ **AComm** $\times \Sigma \times \Sigma_\bullet^\blacksquare$ *satisfying the following set of rules:*

1. $\langle \mathbf{skip}, s\rangle \rightsquigarrow \boldsymbol{n}(s)$.
2. $\langle \mathbf{throw}, s\rangle \rightsquigarrow \boldsymbol{e}(s)$.
3. $\langle x := e, s\rangle \rightsquigarrow \boldsymbol{n}(s[x \mapsto [\![e]\!](s)])$.
4. *if* $s \models \theta$ *then* $\langle \mathbf{assume}\,\theta, s\rangle \rightsquigarrow \boldsymbol{n}(s)$.
5. *if* $s \not\models \theta$ *then* $\langle \mathbf{assume}\,\theta, s\rangle \rightsquigarrow \blacksquare$.
6. *if* $s \models \theta$ *then* $\langle \mathbf{assert}\,\theta, s\rangle \rightsquigarrow \boldsymbol{n}(s)$.
7. *if* $s \not\models \theta$ *then* $\langle \mathbf{assert}\,\theta, s\rangle \rightsquigarrow \bullet$.
8. *if* $\langle C_1, s\rangle \rightsquigarrow \blacksquare$ *then* $\langle C_1 \,;\, C_2, s\rangle \rightsquigarrow \blacksquare$.
9. *if* $\langle C_1, s\rangle \rightsquigarrow \bullet$ *then* $\langle C_1 \,;\, C_2, s\rangle \rightsquigarrow \bullet$.
10. *if* $\langle C_1, s\rangle \rightsquigarrow \boldsymbol{e}(s')$ *then* $\langle C_1 \,;\, C_2, s\rangle \rightsquigarrow \boldsymbol{e}(s')$.

11. *if* $\langle C_1, s\rangle \rightsquigarrow \boldsymbol{n}(s')$ *and* $\langle C_2, s'\rangle \rightsquigarrow \sigma$ *then* $\langle C_1\,;\, C_2, s\rangle \rightsquigarrow \sigma$.

12. *if* $\langle C_1, s\rangle \rightsquigarrow \blacksquare$ *then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \rightsquigarrow \blacksquare$.

13. *if* $\langle C_1, s\rangle \rightsquigarrow \bullet$ *then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \rightsquigarrow \bullet$.

14. *if* $\langle C_1, s\rangle \rightsquigarrow \boldsymbol{n}(s')$ *then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \rightsquigarrow \boldsymbol{n}(s')$.

15. *if* $\langle C_1, s\rangle \rightsquigarrow \boldsymbol{e}(s')$ *and* $\langle C_2, s'\rangle \rightsquigarrow \sigma$ *then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \rightsquigarrow \sigma$.

16. *if* $s \models b$ *and* $\langle C_1, s\rangle \rightsquigarrow \sigma$, *then* $\langle \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s\rangle \rightsquigarrow \sigma$.

17. *if* $s \not\models b$ *and* $\langle C_2, s\rangle \rightsquigarrow \sigma$, *then* $\langle \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s\rangle \rightsquigarrow \sigma$.

18. *if* $s \models b$, $\langle C, s\rangle \rightsquigarrow \blacksquare$ *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, s\rangle \rightsquigarrow \blacksquare$.

19. *if* $s \models b$, $\langle C, s\rangle \rightsquigarrow \bullet$ *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, s\rangle \rightsquigarrow \bullet$.

20. *if* $s \models b$, $\langle C, s\rangle \rightsquigarrow \boldsymbol{e}(s')$ *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, s\rangle \rightsquigarrow \boldsymbol{e}(s')$.

21. *if* $s \models b$, $\langle C, s\rangle \rightsquigarrow \boldsymbol{n}(s')$ *and* $\langle \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, s'\rangle \rightsquigarrow \sigma$, *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, s\rangle \rightsquigarrow \sigma$.

22. *if* $s \not\models b$, *then* $\langle \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}, s\rangle \rightsquigarrow \boldsymbol{n}(s)$.

In what follows we extend the notion of validity of a Hoare triple to programs containing exceptions, assert and assume commands. These are programs that already contain their own built-in specification, conferred by the assume and assert commands. Our interpretation of Hoare triples must handle this 'internal' specification in addition to the 'external' specification given by the precondition and the postcondition. In particular, the postcondition will now be divided into two conditions, one that must hold when the program terminates normally and another when the program terminates exceptionally. Having these aspects in mind, for assessing the validity of a Hoare triple it suffices to state that executions that do terminate in the $\blacksquare$ state, do not enter the $\bullet$ state (because of a failed assert), and also satisfy the normal or exceptional postcondition, depending on the termination. Note that this is a partial notion of correctness, since it does not require termination.

Definition 5.2 (Validity of a Hoare triple)**.** *The Hoare triple* $\{\phi\}\, C\, \{\psi, \epsilon\}$ *is said to be* valid, *denoted* $\models \{\phi\}\, C\, \{\psi, \epsilon\}$, *whenever for every* $s \in \Sigma$ *and* $\sigma \in \Sigma^{\blacksquare}_{\bullet}$, *if* $s \models \phi$ *and* $\langle C, s\rangle \rightsquigarrow \sigma$ *then:*

1. $\sigma \neq \bullet$.

2. *if* $\sigma = \boldsymbol{n}(s')$, *for some* $s' \in \Sigma$, *then* $s' \models \psi$.

3. *if* $\sigma = \boldsymbol{e}(s')$, *for some* $s' \in \Sigma$, *then* $s' \models \epsilon$.

If $\models \{\phi\}\, C\, \{\psi, \epsilon\}$ holds, we say that C is *correct* w.r.t. the specification (ϕ, ψ, ϵ).

Hoare logic's inference system H can be extended for Hoare triples with exceptions, assumes and asserts. This system is shown in Figure 5.1 and, similarly to the original system, contains

$$\text{(skip)}\ \ \frac{}{\{\phi\}\,\mathbf{skip}\,\{\phi,\bot\}} \qquad \text{(throw)}\ \ \frac{}{\{\phi\}\,\mathbf{throw}\,\{\bot,\phi\}}$$

$$\text{(assign)}\ \ \frac{}{\{\psi[e/x]\}\,x := e\,\{\psi,\bot\}}$$

$$\text{(assert)}\ \ \frac{}{\{\theta\wedge\psi\}\,\mathbf{assert}\,\theta\,\{\psi,\bot\}} \qquad \text{(assume)}\ \ \frac{}{\{\theta\rightarrow\psi\}\,\mathbf{assume}\,\theta\,\{\psi,\bot\}}$$

$$\text{(seq)}\ \ \frac{\{\phi\}\,C_1\,\{\theta,\epsilon\}\quad\{\theta\}\,C_2\,\{\psi,\epsilon\}}{\{\phi\}\,C_1\,;\,C_2\,\{\psi,\epsilon\}} \qquad \text{(try-catch)}\ \ \frac{\{\phi\}\,C_1\,\{\psi,\theta\}\quad\{\theta\}\,C_2\,\{\psi,\epsilon\}}{\{\phi\}\,\mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}\,\{\psi,\epsilon\}}$$

$$\text{(if)}\ \ \frac{\{\phi\wedge b\}\,C_1\,\{\psi,\epsilon\}\quad\{\phi\wedge\neg b\}\,C_2\,\{\psi,\epsilon\}}{\{\phi\}\,\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}\,\{\psi,\epsilon\}}$$

$$\text{(while)}\ \ \frac{\{\theta\wedge b\}\,C\,\{\theta,\epsilon\wedge\neg b\}}{\{\theta\}\,\mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}\,\{\theta\wedge\neg b,\epsilon\wedge\neg b\}}$$

$$\text{(conseq)}\ \ \frac{\{\phi\}\,C\,\{\psi,\epsilon\}}{\{\phi'\}\,C\,\{\psi',\epsilon'\}}\ \text{if}\ \begin{array}{l}\phi'\rightarrow\phi\ \text{ and}\\ \psi\rightarrow\psi'\ \text{ and } \epsilon\rightarrow\epsilon'\end{array}$$

Figure 5.1: System H

the rule (conseq) that is guarded by first-order side conditions, whose validity must be checked when constructing derivations. Note also that if while loops are annotated with invariants, those will not be used in the derivation.

The new (assume) and (assert) rules follow the assignment rule in this respect: they propagate the normal postcondition ψ backward, according to the definition of the weakest precondition predicate transformer for the guarded commands language [45]. A derivation using system H of the program Example 4.1 is given in Appendix A.

Proposition 5.1 (Soundness of system H)**.** *If* $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi,\epsilon\}$*, then* $\models \{\phi\}\,C\,\{\psi,\epsilon\}$.

Proof. By induction on the derivation of $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi,\epsilon\}$. We only include here some cases. The remaining cases are proved in a similar manner.

Case (assert). Assume $\vdash_{\mathsf{H}} \{\theta\wedge\psi\}\,\mathbf{assert}\,\theta\,\{\psi,\bot\}$. If $s\models\theta\wedge\psi$, then $s\models\theta$ and $s\models\psi$. Hence, $\langle\mathbf{assert}\,\theta,s\rangle\rightsquigarrow\mathbf{n}(s)$ and $\models\{\theta\wedge\psi\}\,\mathbf{assert}\,\theta\,\{\psi,\bot\}$.

Case (assume). Assume $\vdash_{\mathsf{H}} \{\theta\rightarrow\psi\}\,\mathbf{assume}\,\theta\,\{\psi,\bot\}$. If $s\models\theta\rightarrow\psi$, then we have two cases: 1. if $s\not\models\theta$, then $\langle\mathbf{assume}\,\theta,s\rangle\rightsquigarrow\blacksquare$. 2. if $s\models\theta$ and $s\models\psi$, then $\langle\mathbf{assume}\,\theta,s\rangle\rightsquigarrow\mathbf{n}(s)$. Hence $\models\{\theta\rightarrow\psi\}\,\mathbf{assert}\,\theta\,\{\psi,\bot\}$.

Case (try-catch). Assume $\vdash_{\mathsf{H}} \{\phi\}\,\mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}\,\{\psi,\epsilon\}$ follows from $\vdash_{\mathsf{H}} \{\phi\}\,C_1\,\{\psi,\theta\}$ and $\vdash_{\mathsf{H}} \{\theta\}\,C_2\,\{\psi,\epsilon\}$. We want to prove that $\models\{\phi\}\,\mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}\,\{\psi,\epsilon\}$ so, assume $s\models\phi$. By IH $\models\{\phi\}\,C_1\,\{\psi,\theta\}$, so we have three cases:

1. $\langle C_1,s\rangle\rightsquigarrow\blacksquare$ and we are done.

2. $\langle C_1, s\rangle \rightsquigarrow \mathbf{n}(s')$ and $s' \models \psi$. Therefore $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \rightsquigarrow \mathbf{n}(s')$ and we are done.

3. $\langle C_1, s\rangle \rightsquigarrow \mathbf{e}(s')$ and $s' \models \theta$. Therefore $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \rightsquigarrow \sigma$ where $\langle C_2, s'\rangle \rightsquigarrow \sigma$. By IH $\models \{\theta\}\, C_2\, \{\psi, \epsilon\}$ so, we have three subcases: (a) $\sigma = \blacksquare$. (b) $\sigma = \mathbf{n}(s'')$ and $s'' \models \psi$. (c) $\sigma = \mathbf{e}(s'')$ and $s'' \models \epsilon$.

Hence $\models \{\phi\}\, \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}\, \{\psi, \epsilon\}$. □

The notion of the assertions language must be adapted for triples with exceptions. Naturally and since each program can terminate normally or exceptionally, two sets of intermediate states must be taken into account. For $C \in \mathbf{AComm}$ and $\phi \in \mathbf{Assert}$, we denote by $\mathrm{postN}(\phi, \mathrm{C})$ the set of states $\{s' \in \Sigma \mid \langle C, s\rangle \rightsquigarrow \mathbf{n}(s')$ for some $s \in \Sigma$ such that $s \models \phi\}$ and by $\mathrm{postE}(\phi, \mathrm{C})$ the set of states $\{s' \in \Sigma \mid \langle C, s\rangle \rightsquigarrow \mathbf{e}(s')$ for some $s \in \Sigma$ such that $s \models \phi\}$. The notion of expressiveness is given by the following definition.

Definition 5.3 (Expressiveness)**.** *The assertion language* **Assert** *is said to be* expressive *with respect to the command language* **AComm** *and interpretation structure* $\mathcal{M}$*, if for every* $\phi \in$ **Assert** *and* $C \in$ **AComm** *there exist* $\psi, \epsilon \in$ **Assert** *such that for any* $s \in \Sigma$*, (i)* $s \models \psi$ *iff* $s \in \mathrm{postN}(\phi, \mathrm{C})$ *and (ii)* $s \models \epsilon$ *iff* $s \in \mathrm{postE}(\phi, \mathrm{C})$.

With the notion of expressiveness, system H can be shown to be complete w.r.t. the semantics of the programming language.

Proposition 5.2 (Completeness of system H in the sense of Cook)**.** *Let* $C \in$ **AComm***,* $\phi, \psi, \epsilon \in$ **Assert***, and* $\mathcal{M}$ *be an interpretation structure such that* **Assert** *is expressive w.r.t.* **AComm** *and* $\mathcal{M}$*. If* $\models \{\phi\}\, C\, \{\psi, \epsilon\}$*, then* $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$.

Proof. By induction on the structure of the program C. We show here the proof for the cases of the new constructs of the present language. The **throw** case is trivial.

Assume $\models \{\phi\}\, \mathbf{assume}\, \theta\, \{\psi, \epsilon\}$, for some $\phi, \psi, \epsilon \in$ **Assert**. Let $s \in \Sigma$ and assume that $s \models \phi$. Then either $s \not\models \theta$ or $s \models \theta$, $\langle \mathbf{assume}\, \theta, s\rangle \rightsquigarrow \mathbf{n}(s)$, and $s \models \psi$. So, from the axiom $\vdash_{\mathsf{Hg}} \{\theta \to \psi\}\, \mathbf{assume}\, \theta\, \{\psi, \bot\}$, applying (conseq) rule, we conclude that $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathbf{assume}\, \theta\, \{\psi, \epsilon\}$, because $\models \phi \to (\theta \to \psi)$ and $\models \bot \to \epsilon$.

Assume $\models \{\phi\}\, \mathbf{assert}\, \theta\, \{\psi, \epsilon\}$, for some $\phi, \psi, \epsilon \in$ **Assert**. Let $s \in \Sigma$ and assume that $s \models \phi$. Then it must be the case that $s \models \theta$, $\langle \mathbf{assert}\, \theta, s\rangle \rightsquigarrow \mathbf{n}(s)$ and $s \models \psi$. So, from the axiom $\vdash_{\mathsf{Hg}} \{\theta \wedge \psi\}\, \mathbf{assert}\, \theta\, \{\psi, \bot\}$, applying (conseq) rule, we conclude that $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathbf{assert}\, \theta\, \{\psi, \epsilon\}$, because $\models \phi \to \theta \wedge \psi$ and $\bot \to \epsilon$.

Assume $\models \{\phi\}\, \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}\, \{\psi, \epsilon\}$, for some $\phi, \psi, \epsilon \in$ **Assert**. Let $s \in \Sigma$ such that $s \models \phi$. If $\langle C_1, s\rangle \rightsquigarrow \mathbf{n}(s')$ then $s' \models \psi$. Also, by Definition 5.3 there exist some $\theta \in$ **Assert** such that $s'' \models \theta$ for any $s'' \in \mathrm{postE}(\phi, \mathrm{C}_1)$. Then we have that $\models \{\phi\}\, C_1\, \{\psi, \theta\}$, and since $\models \{\phi\}\, C_1\, ;\, C_2\, \{\psi, \epsilon\}$, also that $\models \{\theta\}\, C_2\, \{\psi, \epsilon\}$. Therefore, using both IH, we have that $\vdash_{\mathsf{Hg}} \{\phi\}\, C_1\, \{\psi, \theta\}$ and $\vdash_{\mathsf{Hg}} \{\theta\}\, C_2\, \{\psi, \epsilon\}$, and applying the (try-catch) rule we conclude $\vdash_{\mathsf{Hg}} \{\phi\}\, C_1\, ;\, C_2\, \{\psi, \epsilon\}$.

$$\text{(skip)}\quad \frac{}{\{\phi\}\,\mathbf{skip}\,\{\psi,\epsilon\}}\ \text{if } \phi\to\psi \qquad\qquad \text{(throw)}\quad \frac{}{\{\phi\}\,\mathbf{throw}\,\{\psi,\epsilon\}}\ \text{if } \phi\to\epsilon$$

$$\text{(assign)}\quad \frac{}{\{\phi\}\,x:=e\,\{\psi,\epsilon\}}\ \text{if } \phi\to\psi[e/x]$$

$$\text{(assert)}\quad \frac{}{\{\phi\}\,\mathbf{assert}\,\theta\,\{\psi,\epsilon\}}\ \text{if } \phi\to\theta\wedge\psi \qquad \text{(assume)}\quad \frac{}{\{\phi\}\,\mathbf{assume}\,\theta\,\{\psi,\epsilon\}}\ \text{if } \phi\wedge\theta\to\psi$$

$$\text{(seq)}\quad \frac{\{\phi\}\,C_1\,\{\theta,\epsilon\}\quad \{\theta\}\,C_2\,\{\psi,\epsilon\}}{\{\phi\}\,C_1\,;\,C_2\,\{\psi,\epsilon\}} \qquad \text{(try-catch)}\quad \frac{\{\phi\}\,C_1\,\{\psi,\theta\}\quad \{\theta\}\,C_2\,\{\psi,\epsilon\}}{\{\phi\}\,\mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}\,\{\psi,\epsilon\}}$$

$$\text{(while)}\quad \frac{\{\theta\wedge b\}\,C\,\{\theta,\epsilon\}}{\{\phi\}\,\mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C\ \mathbf{od}\,\{\psi,\epsilon\}}\ \text{if}\ \begin{array}{l}\phi\to\theta \text{ and}\\ \theta\wedge\neg b\to\psi\end{array}$$

$$\text{(if)}\quad \frac{\{\phi\wedge b\}\,C_1\,\{\psi,\epsilon\}\quad \{\phi\wedge\neg b\}\,C_2\,\{\psi,\epsilon\}}{\{\phi\}\,\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}\,\{\psi,\epsilon\}}$$

Figure 5.2: System Hg

Of the remaining cases, the sequence case also makes use of the expressiveness of the language. □

Let us now turn our focus to annotated programs. Up to this point, although loops were annotated with invariants, the latter were not being taken into account by the semantics of the programming language neither by system H. Similarly to what was done in Section 2.3, we propose now a system to reason about annotated programs that is deterministic on the rule to apply, and as such does not contain a consequence rule. The *goal-directed* system Hg for annotated programs containing exceptions, assumes, and asserts is shown in Figure 5.2. It is straightforward to show that the system is sound w.r.t. system H.

Proposition 5.3 (Soundness of Hg)**.** *If* $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi,\epsilon\}$*, then* $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi,\epsilon\}$.

Proof. By induction on the derivation of $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi,\epsilon\}$. □

As explained in Section 2.3, the reverse implication does not hold, since there may well be valid triples that are impossible to derive since they contain wrong annotations. We note that if a triple $\{\phi\}\,C\,\{\psi,\epsilon\}$ is derivable in H, then it is certain that *there exists an annotated version* C' of C such that $\{\phi\}\,C'\,\{\psi,\epsilon\}$ is derivable in Hg (it suffices to use the invariants that were used in the derivation of the non-annotated triple). So instead of working with a completeness result, we extend the previous notion of correctly-annotated program to the language of this section.

Definition 5.4. *Let* $C \in \mathbf{AComm}$ *and* $\phi,\psi \in \mathbf{Assert}$*. The program* C *is* correctly-annotated *with respect to* (ϕ,ψ,ϵ) *if* $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi,\epsilon\}$ *implies* $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi,\epsilon\}$.

The following lemma states the admissibility of the (conseq) rule in system Hg.

Lemma 5.4. *If* $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$, $\models \phi' \to \phi$, $\models \psi \to \psi'$, *and* $\models \epsilon \to \epsilon'$, *then* $\vdash_{\mathsf{Hg}} \{\phi'\}\, C\, \{\psi', \epsilon'\}$.

Proof. By induction on the derivation of $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$. □

A derivation using system Hg based on the weakest precondition of the program of Example 4.1 is given in Appendix A. Note that, as described at the beginning of Section 3.2, the system Hg may produce side conditions of exponential size in the length of the program.

In the next section we will focus on a subclass of single-assignment programs, introduce a program logic specifically for these programs, and show that such a logic avoids the exponential explosion referred above. We will also show how the logic is related to VCGens. System Hg will play an important role in the proof of completeness of that logic.

5.2 SA Program with Assume, Assert, and Exceptions

Let us now introduce a subclass of programs. We will be focusing on a dynamic notion of single-assignment programs, and as such, we will introduce syntactic restrictions on the language introduced in the previous section. In terms of commands, the language will preserve the assume and assert commands but will not contain any iterating construct. A clear consequence of this is that programs over this SA language will always terminate. We will then define a suitable notion of Hoare triple for these programs, and introduce a specific inference system for them. The system will be shown to be suitable for program verification because (i) it is goal-directed (no consequence rule is present), and moreover (unlike system Hg) it admits a single derivation for each valid triple; and (ii) it dispenses with variable substitutions and produces derivations that are, in the absence of try-catch statements, free from the problem of exponential explosion of the size of formulas.

We start with the notion of single-assignment program, that can be defined inductively, based on the sets of variables occurring in, and assigned by, a program. The following definition expands the previous notions of Vars and Asgn to the language of this chapter.

Definition 5.5. *Let* $C \in \mathbf{AComm}$*. The sets* $\mathsf{Vars}(C)$ *and* $\mathsf{Asgn}(C)$ *of variables occurring and assigned in* $C \in \mathbf{AComm}$*, which were initially introduced in Definition 2.3, are extended to programs containing exceptions, assumes and asserts as follows:*

$$\begin{aligned}
\mathsf{Vars}(\mathbf{throw}) &= \emptyset \\
\mathsf{Vars}(\mathbf{assume}\,\theta) &= \mathsf{FV}(\theta) \\
\mathsf{Vars}(\mathbf{assert}\,\theta) &= \mathsf{FV}(\theta) \\
\mathsf{Vars}(\mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}) &= \mathsf{Vars}(C_1) \cup \mathsf{Vars}(C_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Asgn}(\mathbf{throw}) &= \emptyset \\
\mathsf{Asgn}(\mathbf{assume}\,\theta) &= \emptyset \\
\mathsf{Asgn}(\mathbf{assert}\,\theta) &= \emptyset \\
\mathsf{Asgn}(\mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}) &= \mathsf{Asgn}(C_1) \cup \mathsf{Asgn}(C_2)
\end{aligned}$$

Using the previous definition it is now possible to capture the class of single-assignment programs, which is a subclass of the programs constructed over **AComm**. The following notion of SA program guarantees that variables are assigned at most once in any execution and never after they have been read.

Definition 5.6 (SA program)**.** *The set* $\mathbf{AComm}^{\text{SA}} \subset \mathbf{AComm}$ *of* single-assignment programs *is defined inductively as follows:*

- $\mathbf{skip}, \mathbf{assert}\,\theta, \mathbf{assume}\,\theta, \mathbf{throw} \in \mathbf{AComm}^{\text{SA}}$.
- $x := e \in \mathbf{AComm}^{\text{SA}}$ *if* $x \notin \mathsf{Vars}(e)$.
- $C_1\,;\,C_2 \in \mathbf{AComm}^{\text{SA}}$ *if* $C_1, C_2 \in \mathbf{AComm}^{\text{SA}}$*, and* $\mathsf{Vars}(C_1) \cap \mathsf{Asgn}(C_2) = \emptyset$.
- $\mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc} \in \mathbf{AComm}^{\text{SA}}$ *if* $C_1, C_2 \in \mathbf{AComm}^{\text{SA}}$*, and* $\mathsf{Vars}(C_1) \cap \mathsf{Asgn}(C_2) = \emptyset$.
- $\mathbf{if}\,b\,\mathbf{then}\,C_1\,\mathbf{else}\,C_2\ \mathbf{fi} \in \mathbf{AComm}^{\text{SA}}$ *if* $C_1, C_1 \in \mathbf{AComm}^{\text{SA}}$*, and* $\mathsf{Vars}(b) \cap (\mathsf{Asgn}(C_1) \cup \mathsf{Asgn}(C_2)) = \emptyset$.

The previous definition is similar to Definition 3.1 for the commands that are present in both. The commands that are not present in the previous definition, $\mathbf{assert}\,\theta$, $\mathbf{assume}\,\theta$, and **throw** are always in SA form when considered alone. Moreover, the try-catch command has a restriction that is very similar to the sequence command. The second part of the command cannot assign variables that occur in the first part.

Focusing on the verification of SA programs using Hoare logic, it will be convenient to forbid programs from assigning variables occurring free in the precondition. We note that there is not much use for free occurrences of variables that are assigned in the program, since in the paths containing these assignments the initial values of the variables cannot be read. Remember that $\phi\#C$ is used to denote that the program C does not assign free variables of ϕ.

Definition 5.7 (SA triple)**.** *Let* $C \in \mathbf{AComm}^{\text{SA}}$ *and* $\phi, \psi, \epsilon \in \mathbf{Assert}$*. A Hoare triple* $\{\phi\}\,C\,\{\psi, \epsilon\}$ *is said to be* single-assignment *if* $\phi\#C$*.*

Figure 5.3 contains the rules of the goal-directed system Hsa, which is based on *forward propagation* of assertions. It derives triples of the form $\{\phi\}\,C\,\{\phi \wedge \psi, \phi \wedge \epsilon\}$, where ϕ encodes logically all incoming executions, ψ and ϵ encode the normal and exceptional executions of C, and $\phi \wedge \psi$, $\phi \wedge \epsilon$ are respectively the normal and exceptional strongest postconditions of C with respect to ϕ.

We will prove the soundness of this system with respect to system H by first noting the following:

Lemma 5.5. *Let* $C \in \mathbf{AComm}^{\text{SA}}$ *and* ϕ*,* ψ*,* $\psi', \epsilon, \epsilon' \in \mathbf{Assert}$ *such that* $\phi\#C$*, and* $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\psi, \epsilon\}$*. Then:*

1. $\mathsf{FV}(\psi) \cup \mathsf{FV}(\epsilon) \subseteq \mathsf{FV}(\phi) \cup \mathsf{Vars}(C)$.

$$\text{(skip)}\quad \overline{\{\phi\}\,\textbf{skip}\,\{\phi\wedge\top,\phi\wedge\bot\}} \qquad \text{(throw)}\quad \overline{\{\phi\}\,\textbf{throw}\,\{\phi\wedge\bot,\phi\wedge\top\}}$$

$$\text{(assign)}\quad \overline{\{\phi\}\,x:=e\,\{\phi\wedge x=e,\phi\wedge\bot\}}$$

$$\text{(assert)}\quad \overline{\{\phi\}\,\textbf{assert}\,\theta\,\{\phi\wedge\theta,\phi\wedge\bot\}}\ \text{if } \phi\to\theta \qquad \text{(assume)}\quad \overline{\{\phi\}\,\textbf{assume}\,\theta\,\{\phi\wedge\theta,\phi\wedge\bot\}}$$

$$\text{(seq)}\quad \frac{\{\phi\}\,C_1\,\{\phi\wedge\psi_1,\phi\wedge\epsilon_1\}\quad \{\phi\wedge\psi_1\}\,C_2\,\{\phi\wedge\psi_1\wedge\psi_2,\phi\wedge\psi_1\wedge\epsilon_2\}}{\{\phi\}\,C_1\,;\,C_2\,\{\phi\wedge(\psi_1\wedge\psi_2),\phi\wedge(\epsilon_1\vee(\psi_1\wedge\epsilon_2))\}}$$

$$\text{(try-catch)}\quad \frac{\{\phi\}\,C_1\,\{\phi\wedge\psi_1,\phi\wedge\epsilon_1\}\quad \{\phi\wedge\epsilon_1\}\,C_2\,\{\phi\wedge\epsilon_1\wedge\psi_2,\phi\wedge\epsilon_1\wedge\epsilon_2\}}{\{\phi\}\,\textbf{try}\,C_1\,\textbf{catch}\,C_2\,\textbf{hc}\,\{\phi\wedge(\psi_1\vee(\epsilon_1\wedge\psi_2)),\phi\wedge(\epsilon_1\wedge\epsilon_2)\}}$$

$$\text{(if)}\quad \frac{\{\phi\wedge b\}\,C_1\,\{\phi\wedge b\wedge\psi_1,\phi\wedge b\wedge\epsilon_1\}\quad \{\phi\wedge\neg b\}\,C_2\,\{\phi\wedge\neg b\wedge\psi_2,\phi\wedge\neg b\wedge\epsilon_2\}}{\{\phi\}\,\textbf{if}\,b\,\textbf{then}\,C_1\,\textbf{else}\,C_2\,\textbf{fi}\,\{\phi\wedge((b\wedge\psi_1)\vee(\neg b\wedge\psi_2)),\phi\wedge((b\wedge\epsilon_1)\vee(\neg b\wedge\epsilon_2))\}}$$

Figure 5.3: System Hsa

2. If $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\psi',\epsilon'\}$*, then* $\psi'=\psi$ *and* $\epsilon'=\epsilon$*.*

Proof. 1. By induction on the derivation of $\vdash_{\mathsf{Hsa}} \{\phi\}\,C\,\{\psi,\epsilon\}$. We write here the case where the last step is the rule (seq). For the remaining cases the proofs are either trivial or similar to this case. Assume $C_1\,;\,C_2 \in \mathbf{AComm}^{\mathrm{SA}}$ and $\phi\#(C_1\,;\,C_2)$. Then we have $\mathsf{Vars}(C_1)\cap\mathsf{Asgn}(C_2)=\emptyset$ and $\phi\#C_1$. Assume also that $\vdash_{\mathsf{Hsa}} \{\phi\}\,C_1\,;\,C_2\,\{\phi\wedge(\psi_1\wedge\psi_2),\phi\wedge(\epsilon_1\vee(\psi_1\wedge\epsilon_2))\}$ follows from $\vdash_{\mathsf{Hsa}} \{\phi\}\,C_1\,\{\phi\wedge\psi_1,\phi\wedge\epsilon_1\}$ and $\vdash_{\mathsf{Hsa}} \{\phi\wedge\psi_1\}\,C_2\,\{\phi\wedge\psi_1\wedge\psi_2,\phi\wedge\psi_1\wedge\epsilon_2\}$. Then, by IH, $\mathsf{FV}(\phi\wedge\psi_1)\cup\mathsf{FV}(\phi\wedge\epsilon_1)\subseteq\mathsf{FV}(\phi)\cup\mathsf{Vars}(C_1)$. Moreover, it follows that $(\phi\wedge\psi_1)\#C_2$. Therefore, by IH, $\mathsf{FV}(\phi\wedge\psi_1\wedge\psi_2)\cup\mathsf{FV}(\phi\wedge\psi_1\wedge\epsilon_2)\subseteq\mathsf{FV}(\phi\wedge\psi_1)\cup\mathsf{Vars}(C_2)$. Hence $\mathsf{FV}(\phi\wedge\psi_1\wedge\psi_2)\cup\mathsf{FV}(\phi\wedge(\epsilon_1\vee(\psi_1\wedge\epsilon_2)))\subseteq\mathsf{FV}(\phi)\cup\mathsf{Vars}(C_1)\cup\mathsf{Vars}(C_2)$.

2. By induction on C. The proof is trivial for the base cases because the derivation is unique. Next follows the proof for the case that C is $\textbf{try}\,C_1\,\textbf{catch}\,C_2\,\textbf{hc}$, and the proof of the missing cases are similar. Assume $\vdash_{\mathsf{Hsa}} \{\phi\}\,\textbf{try}\,C_1\,\textbf{catch}\,C_2\,\textbf{hc}\,\{\phi\wedge(\psi_1\vee(\epsilon_1\wedge\psi_2)),\phi\wedge(\epsilon_1\wedge\epsilon_2)\}$. Then we must have that $\vdash_{\mathsf{Hsa}} \{\phi\}\,C_1\,\{\phi\wedge\psi_1,\phi\wedge\epsilon_1\}$ and $\vdash_{\mathsf{Hsa}} \{\phi\wedge\epsilon_1\}\,C_2\,\{\phi\wedge\epsilon_1\wedge\psi_2,\phi\wedge\epsilon_1\wedge\epsilon_2\}$, for some $\theta\in\mathbf{Assert}$. Assume now that $\vdash_{\mathsf{Hsa}} \{\phi\}\,\textbf{try}\,C_1\,\textbf{catch}\,C_2\,\textbf{hc}\,\{\phi\wedge(\psi_1'\vee(\epsilon_1'\wedge\psi_2')),\phi\wedge(\epsilon_1'\wedge\epsilon_2')\}$. Then it must be the case that $\vdash_{\mathsf{Hsa}} \{\phi\}\,C_1\,\{\phi\wedge\psi_1',\phi\wedge\epsilon_1'\}$, and since $\phi\#C_1$ because $\phi\#(C_1\,;\,C_2)$, by IH we have that $\psi_1'=\psi_1$ and $\epsilon_1'=\epsilon_1$. It must also be the case that $\vdash_{\mathsf{Hsa}} \{\phi\wedge\epsilon_1'\}\,C_2\,\{\phi\wedge\epsilon_1'\wedge\psi_2',\phi\wedge\epsilon_1'\wedge\epsilon_2'\}$ and since $(\phi\wedge\epsilon_1')\#C_2$, because from 1. $\mathsf{FV}(\phi\wedge\epsilon_1')\subseteq\mathsf{FV}(\phi)\cup\mathsf{Vars}(C_1)$ and $\mathsf{Vars}(C_1)\cap\mathsf{Asgn}(C_2)=\emptyset$, by IH we have that $\psi_2'=\psi_2$ and $\epsilon_2'=\epsilon_2$. □

The first part of the lemma above states that the free variables that occur in the normal and exceptional postcondition are not invented: they come either from the precondition or from the program. On the other hand, the second part, states that system Hsa is deterministic, in the

sense that for a certain Hoare triple, there is only one possible derivable pair of postconditions.

Note that the soundness result does not hold for $C \in \mathbf{AComm}$ since system Hsa only derives SA triples. For instance, system Hsa does not provides a rule for while loops. Therefore, the result must be established for $C \in \mathbf{AComm}^{\text{SA}}$ only.

Proposition 5.6 (Soundness of Hsa). *Let $C \in \mathbf{AComm}^{\text{SA}}$ and $\phi, \psi, \epsilon \in \mathbf{Assert}$ such that $\phi \# C$. If $\vdash_{\mathsf{Hsa}} \{\phi\}\, C\, \{\psi, \epsilon\}$, then $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$.*

Proof. By induction on the derivation of $\vdash_{\mathsf{Hsa}} \{\phi\}\, C\, \{\psi, \epsilon\}$, using Lemma 5.5. We show here the cases of the (assign) and the (try-catch) rules. For the remaining cases the proofs are similar.

Case (assign). Assume $\phi \#(x := e)$ and $\vdash_{\mathsf{Hsa}} \{\phi\}\, x := e\, \{\phi \wedge x = e, \phi \wedge \bot\}$. We know that $\vdash_{\mathsf{H}} \{(\phi \wedge x = e)[e/x]\}\, x := e\, \{\phi \wedge x = e, \bot\}$. Moreover, as $\phi \#(x := e)$, $x \notin \mathsf{FV}(\phi) \cup \mathsf{Vars}(e)$, we have $\phi \to (\phi \wedge x = e)[e/x]$. Also $\phi \wedge \bot \to \bot$. So, by the (conseq) rule, $\vdash_{\mathsf{H}} \{\phi\}\, x := e\, \{\phi \wedge x = e, \phi \wedge \bot\}$.

Case (try-catch). Assume $\phi \#(\mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc})$ and that $\vdash_{\mathsf{Hsa}} \{\phi\}\, \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}\, \{\phi \wedge (\psi_1 \vee (\epsilon_1 \wedge \psi_2)), \phi \wedge (\epsilon_1 \wedge \epsilon_2)\}$ follows from $\vdash_{\mathsf{Hsa}} \{\phi\}\, C_1\, \{\phi \wedge \psi_1, \phi \wedge \epsilon_1\}$ and $\vdash_{\mathsf{Hsa}} \{\phi \wedge \epsilon_1\}\, C_2\, \{\phi \wedge \epsilon_1 \wedge \psi_2, \phi \wedge \epsilon_1 \wedge \epsilon_2\}$. Since $\phi \# C_1$, we get by IH $\vdash_{\mathsf{H}} \{\phi\}\, C_1\, \{\phi \wedge \psi_1, \phi \wedge \epsilon_1\}$ and, as $\phi \wedge \psi_1 \to \phi \wedge (\psi_1 \vee (\epsilon_1 \wedge \psi_2))$, by (conseq) we get $\vdash_{\mathsf{H}} \{\phi\}\, C_1\, \{\phi \wedge (\psi_1 \vee (\epsilon_1 \wedge \psi_2)), \phi \wedge \epsilon_1\}$. Using Lemma 5.5, we have that $(\phi \wedge \epsilon_1) \# C_2$, so we get by IH $\vdash_{\mathsf{H}} \{\phi \wedge \epsilon_1\}\, C_2\, \{\phi \wedge \epsilon_1 \wedge \psi_2, \phi \wedge \epsilon_1 \wedge \epsilon_2\}$ and, since $\phi \wedge \epsilon_1 \wedge \psi_2 \to \phi \wedge (\psi_1 \vee (\epsilon_1 \wedge \psi_2))$, by (conseq) we get $\vdash_{\mathsf{H}} \{\phi \wedge \epsilon_1\}\, C_2\, \{\phi \wedge (\psi_1 \wedge (\epsilon_1 \wedge \psi_2), \phi \wedge \epsilon \wedge \epsilon_2\}$. Finally, we apply the (try-catch) rule to obtain $\vdash_{\mathsf{H}} \{\phi\}\, \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}\, \{\phi \wedge (\psi_1 \vee (\epsilon_1 \wedge \psi_2)), \phi \wedge (\epsilon_1 \wedge \epsilon_2)\}$. □

The completeness of Hsa on the other hand will be established with respect to the goal-directed system Hg. Observe that the Hsa system is not capable of deriving every valid triple, and the completeness result takes this into account.

Proposition 5.7 (Completeness of Hsa). *Let $C \in \mathbf{AComm}^{\text{SA}}$ and ϕ, $\psi, \epsilon \in \mathbf{Assert}$ such that $\phi \# C$ and $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$. Then $\vdash_{\mathsf{Hsa}} \{\phi\}\, C\, \{\phi \wedge \psi', \phi \wedge \epsilon'\}$ for some $\psi', \epsilon' \in \mathbf{Assert}$ such that $\models \phi \wedge \psi' \to \psi$ and $\models \phi \wedge \epsilon' \to \epsilon$.*

Proof. By induction on C, using lemmas 2.6 and 5.5. We write here the cases of assigment and the try-catch commands. The remaining cases are proved in a similar manner.

Case $C = x := e$. Assume $\phi \#(x := e)$ and $\vdash_{\mathsf{Hg}} \{\phi\}\, x := e\, \{\psi, \epsilon\}$ with $\phi \to \psi[e/x]$. We have $\vdash_{\mathsf{Hsa}} \{\phi\}\, x := e\, \{\phi \wedge x = e, \phi \wedge \bot\}$. As $\phi \#(x := e)$, whenever $\phi \to \psi[e/x]$ holds the assertion $\phi \wedge x = e \to \psi$ holds. Hence have $\phi \wedge x = e \to \psi$. Moreover $\phi \wedge \bot \to \epsilon$. So, we are done.

Case $C = \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}$. Assume $\phi \#(\mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc})$ and $\vdash_{\mathsf{Hg}} \{\phi\}\, \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}\, \{\psi, \epsilon\}$ follows from $\vdash_{\mathsf{Hg}} \{\phi\}\, C_1\, \{\psi, \theta\}$ and $\vdash_{\mathsf{Hg}} \{\theta\}\, C_2\, \{\psi, \epsilon\}$. From $\phi \#(\mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc})$ we get $\phi \# C_1$ and $\phi \# C_2$. From IH, we have for some ψ_1, ϵ_1 that $\vdash_{\mathsf{Hsa}} \{\phi\}\, C_1\, \{\phi \wedge \psi_1, \phi \wedge \epsilon_1\}$, $\models \phi \wedge \psi_1 \to \psi$ and $\models \phi \wedge \epsilon_1 \to \theta$. So, by Lemma 2.6, $\vdash_{\mathsf{Hg}} \{\theta \wedge \epsilon_1\}\, C_2\, \{\psi, \epsilon\}$. Moreover, using Lemma 5.5, we have $(\phi \wedge \epsilon_1) \# C_2$. By IH we have, for some ψ_2, ϵ_2, that $\vdash_{\mathsf{Hsa}} \{\phi \wedge \epsilon_1\}\, C_2\, \{\phi \wedge \epsilon_1 \wedge \psi_2, \phi \wedge \epsilon_1 \wedge \epsilon_2\}$, $\models \phi \wedge \epsilon_1 \wedge \psi_2 \to \psi$ and $\phi \wedge \epsilon_1 \wedge \epsilon_2 \to \epsilon$. We can now derive $\vdash_{\mathsf{Hsa}} \{\phi\}\, \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}\, \{\phi \wedge (\psi_1 \vee (\epsilon_1 \wedge \psi_2)), \phi \wedge (\epsilon_1 \wedge \epsilon_2)\}$ by applying the (try-catch) rule. Moreover we have $\models \psi \wedge (\psi_1 \vee (\epsilon_1 \wedge \psi_2) \to \psi$ and $\models \phi \wedge (\epsilon_1 \wedge \epsilon_2) \to \epsilon$. □

To check the validity of $\{\phi\}\, C\, \{\psi, \epsilon\}$ (where $\phi \# C$) one attempts to construct a Hsa derivation with root $\{\phi\}\, C\, \{\phi \wedge \psi', \phi \wedge \epsilon'\}$ for some formulas ψ', ϵ'. This may not be possible if there exists an execution in which an assert statement fails (in which case some side condition is not valid), but if all conditions are valid, then the derivation is unique (and so are ψ', ϵ', following Lemma 5.5). If additionally the formulas $\phi \wedge \psi' \to \psi$ and $\phi \wedge \epsilon' \to \epsilon$ are valid, then so is the initial triple. The following example uses system Hsa to do precisely this.

Example 5.1. *Consider the program from Example 4.1. We will refer to the statements of each line as* $C_1, \ldots, C_6$. *The derivation of the triple* $\{\top\}\, C_1 ; C_2 ; C_3 ; C_4 ; C_5 ; C_6\, \{\top, \bot\}$ *is below. Since the side conditions are all valid and* $\models \top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \wedge y_3 = y_1 \to \top$, *one can conclude that the program terminates always without error and never in an exceptional state.*

$\{\top\}\, C_1; C_2; C_3; C_4; C_5; C_6\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \wedge y_3 = y_1, \bot\}$
(seq)

1. $\{\top\}\, C_1; C_2; C_3\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)), \bot\}$
   (seq)
   1. $\{\top\}$ **if** $x_0 > 0$ **then** $y_1 := 1$ **else** $y_1 := 0$ **fi** $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)), \bot\}$
      (if)
      1. $\{\top \wedge x_0 > 0\}\, y_1 := 1\, \{\top \wedge x_0 > 0 \wedge y_1 = 1, \bot\}$
      2. $\{\top \wedge \neg x_0 > 0\}\, y_1 := 0\, \{\top \wedge \neg x_0 > 0 \wedge y_1 = 0, \bot\}$
   2. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0))\}\, C_2; C_3\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)), \bot\}$
      (seq)
      1. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0))\}$ **assert** $y_1 = 0 \vee y_1 = 1\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1), \bot\}$
      2. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1)\}$ **if** $x_0 > 0$ **then** $y_2 := 1$ **else** $y_2 := 0$ **fi** $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)), \bot\}$
         (if)
         1. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge x_0 > 0\}\, y_2 := 1\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge x_0 > 0 \wedge y_2 = 1, \bot\}$
         2. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge \wedge \neg x_0 > 0\}\, y_2 := 0\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge \wedge \neg x_0 > 0 \wedge y_2 = 0, \bot\}$
2. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0))\}\, C_4; C_5; C_6\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \wedge y_3 = y_1, \bot\}$
   (seq)
   1. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0))\}$ **assert** $y_2 = y_1\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1, \bot\}$

2. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1\}\, C_5; C_6\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \wedge y_3 = y_1, \bot\}$
   (seq)
   1. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1\}$ **if** $x_0 > 0$ **then** $y_3 := 1$ **else** $y_3 := 0$ **fi** $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)), \bot\}$
      (if)
      1. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge x_0 > 0\}\, y_3 := 1\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge x_0 > 0 \wedge y_3 = 1, \bot\}$
      2. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge \neg x_0 > 0\}\, y_3 := 0\, \{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge \neg x_0 > 0 \wedge y_3 = 0, \bot\}$
   2. $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0))\}$ **assert** $y_3 = y_1$ $\{\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \wedge y_3 = y_1, \bot\}$

Side conditions for application of the (assert) rules:

- $\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \to (y_1 = 0 \vee y_1 = 1)$.
- $\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \to y_2 = y_1$.
- $\top \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)) \wedge y_2 = y_1 \wedge ((x_0 > 0 \wedge y_3 = 1) \vee (\neg x_0 > 0 \wedge y_3 = 0)) \to y_3 = y_1$.

We remark that, in the absence of try-catch statements, derivations in system Hsa avoid the exponential explosion of the size of the formulas illustrated at the end of the previous section, since in the (if) rule, the postconditions in the conclusion triple contain each a single copy of the precondition ϕ. Avoiding exponential explosion is not however possible in this system when exception handling is considered: the observations that were first made in the context of weakest precondition computations for guarded commands in ESC/Java [52], and explored in the previous chapter, apply to our forward-propagation inference system.

Finally, we note that, from the point of view of a program logic, the idea behind a verification conditions generator is that it is possible to write a function that takes a Hoare triple and produces a set of assertions, whose validity implies that the triple is derivable in the inference system. The properties of system Hsa make it straightforward to relate it with the VCSP family of VCGens from the previous chapter, more precisely with the $\mathsf{VCSP}^{\mathsf{PA}}$ VCGen.

5.3 Correctness of the Cube of VCGens

System Hsa introduced in the previous section can now be used to show that the VCGens introduced in the previous chapter are sound and complete w.r.t. the semantics of the programming language introduced in the previous section. The first part of this section shows that the VCGens from the VCSP family are all sound and complete and the second part expands these results to the VCCNF family. As in the previous chapter, in what follows, we will use the term *variant* of a VCGen family to refer to a concrete VCGen, that can be of the form, VCSP^i or VCCNF^i, where i can be one of the following: P, PA, G, GA.

The Generic Strongest Postcondition VCGen Family. The VCGens of Figure 4.6 are sound and complete with respect to system Hsa. The proof follows by showing that $\mathsf{VCSP}^{\mathsf{PA}}$ is sound and complete w.r.t. system Hsa, and then proving that all other VCGens generate VCs that are equivalent to those generated by $\mathsf{VCSP}^{\mathsf{PA}}$. For a matter of simplification, in what follows we will be consistent regarding the symbols to use for the meta-variables. The operational and axiomatic context with which a certain command is reached (the first two parameters of the VCSP family of functions), will be denoted by $\phi \in \mathbf{Assert}$ and $\rho \in \mathbf{Assert}$ respectively. The returned operational context will be denoted by $\psi \in \mathbf{Assert}$ in case of normal termination and $\epsilon \in \mathbf{Assert}$ in case of exceptional termination, and the returned axiomatic context will be denoted by $\gamma \in \mathbf{Assert}$ in case of normal termination and $\mu \in \mathbf{Assert}$ in case of exceptional termination. Finally the set of returned VCs, or properties to be checked, will be represented by $\Gamma \in \mathcal{P}(\mathbf{Assert})$. Moreover, along this section we only consider SA programs and will use normally $C, C_1, C_2 \in \mathbf{AComm}^{\mathrm{SA}}$ to refer to them.

Before presenting soundness and completeness results for $\mathsf{VCSP}^{\mathsf{PA}}$, consider the following lemma that reflects the fact that a program terminates either normally or exceptionally. It does so by stating that the formulas that correspond to normal and exceptional termination are contradictory.

Lemma 5.8. *Let* $i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$. *If* $\mathsf{VCSP}^i(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$, *then* $\models \psi \wedge \epsilon \to \bot$, $\models \psi \wedge \mu \to \bot$, $\models \gamma \wedge \epsilon \to \bot$, *and* $\models \gamma \wedge \mu \to \bot$.

Proof. By induction on C. All cases are straightforward. □

It is now possible to formulate the correctness result for the $\mathsf{VCSP}^{\mathsf{PA}}$ through the following proposition. In particular, it captures the fact that the VCGen makes the distinction between the operational and axiomatic context, in contrast with system Hsa.

Proposition 5.9 (Hsa and $\mathsf{VCSP}^{\mathsf{PA}}$). *Let* $\mathsf{VCSP}^{\mathsf{PA}}(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$ *and* $\phi' \equiv \phi \wedge \rho$. *Then the following holds:*

1. *If* $\models \Gamma$ *then* $\vdash_{\mathsf{Hsa}} \{\phi'\}\, C\, \{\phi' \wedge \psi', \phi' \wedge \epsilon'\}$, *for some* $\psi', \epsilon' \in \mathbf{Assert}$, *such that* $\psi' \equiv \psi \wedge \gamma$ *and* $\epsilon' \equiv \epsilon \wedge \mu$.
2. *If* $\vdash_{\mathsf{Hsa}} \{\phi'\}\, C\, \{\phi' \wedge \psi', \phi' \wedge \epsilon'\}$, *for some* $\psi', \epsilon' \in \mathbf{Assert}$, *then* $\models \Gamma$, $\psi' \equiv \psi \wedge \gamma$ *and* $\epsilon' \equiv \epsilon \wedge \mu$.

Proof. 1. By induction on C. We show here the proof for the case that C is $\textbf{assert}\,\theta$ or $C_1\,;\,C_2$. The other cases are trivial or analogous.

Case $C = \textbf{assert}\,\theta$. We have $\mathsf{VCSP}^{\mathsf{PA}}(\phi,\rho,\textbf{assert}\,\theta) = (\top,\theta,\bot,\bot,\{\phi\wedge\rho\rightarrow\theta\})$, Assume that $\models \phi\wedge\rho\rightarrow\theta$. Then $\vdash_{\mathsf{Hsa}} \{\phi'\}\,\textbf{assert}\,\theta\,\{\phi'\wedge\theta,\phi'\wedge\bot\}$ because $\models \phi'\rightarrow\theta$ holds since $\phi'\equiv\phi\wedge\rho$. Moreover, $\theta\equiv\top\wedge\theta$ and $\bot\equiv\bot\wedge\bot$.

Case $C = C_1\,;\,C_2$. We have $\mathsf{VCSP}^{\mathsf{PA}}(\phi,\rho,C_1\,;\,C_2) = (\,\psi_1\wedge\psi_2,\ \gamma_1\wedge\gamma_2,\ \epsilon_1\vee(\psi_1\wedge\epsilon_2),\ \mu_1\vee(\gamma_1\wedge\mu_2),\ \Gamma_1\cup\Gamma_2\,)$ with

$$\mathsf{VCSP}^{\mathsf{PA}}(\phi,\rho,C_1) = (\,\psi_1,\ \gamma_1,\ \epsilon_1,\ \mu_1,\ \Gamma_1\,) \tag{5.1}$$

$$\mathsf{VCSP}^{\mathsf{PA}}(\phi\wedge\psi_1,\rho\wedge\gamma_1,C_2) = (\,\psi_2,\ \gamma_2,\ \epsilon_2,\ \mu_2,\ \Gamma_2\,) \tag{5.2}$$

Assume $\models \Gamma_1\cup\Gamma_2$. So, we have in particular $\models\Gamma_1$ and thus from (5.1), by IH $\vdash_{\mathsf{Hsa}} \{\phi'\}\,C_1\,\{\phi'\wedge\psi_1',\phi'\wedge\epsilon_1'\}$, for some ψ_1',ϵ_1' such that $\psi_1'\equiv\psi_1\wedge\gamma_1$ and $\epsilon_1'\equiv\epsilon_1\wedge\mu_1$. Since $(\phi\wedge\psi_1)\wedge(\rho\wedge\gamma_1)\equiv\phi'\wedge\psi_1'$, from (5.2), by IH $\vdash_{\mathsf{Hsa}} \{\phi'\wedge\psi_1'\}\,C_2\,\{\phi'\wedge\psi_1'\wedge\psi_2',\phi'\wedge\psi_1'\wedge\epsilon_2'\}$, for some ψ_2',ϵ_2', such that $\psi_2'\equiv\psi_2\wedge\gamma_2$ and $\epsilon_2'\equiv\epsilon_2\wedge\mu_2$. Therefore, we have $\vdash_{\mathsf{Hsa}} \{\phi'\}\,C_1\,;\,C_2\,\{\phi'\wedge(\psi_1'\wedge\psi_2'),\phi'\wedge(\epsilon_1'\vee(\psi_1'\wedge\epsilon_2'))\}$. Moreover, $\psi_1'\wedge\psi_2'\equiv(\psi_1\wedge\psi_2)\wedge(\gamma_1\wedge\gamma_2)$ and $\epsilon_1'\wedge\epsilon_2'\equiv\epsilon_1'\vee(\psi_1'\wedge\epsilon_2')$, because $\epsilon_1'\vee(\psi_1'\wedge\epsilon_2')\equiv(\epsilon_1\wedge\mu_1)\vee(\psi_1\wedge\epsilon_2\wedge\mu_1)\vee(\epsilon_1\wedge\gamma_1\wedge\mu_2)\vee(\psi_1\wedge\epsilon_2\wedge\gamma_1\wedge\mu_2)$ and, by Lemma 5.8, $\psi_1\wedge\mu_1\rightarrow\bot$ and $\epsilon_1\wedge\gamma_1\rightarrow\bot$.

Of the remaining cases, the try-catch case also uses Lemma 5.8.

2. By induction on the derivation of $\vdash_{\mathsf{Hsa}} \{\phi'\}\,C\,\{\phi'\wedge\psi,\phi'\wedge\epsilon'\}$. We only show here the case of the try-catch rule. Assume $\phi'\equiv\phi\wedge\rho$ and that $\vdash_{\mathsf{Hsa}} \{\phi'\}\,\textbf{try}\,C_1\,\textbf{catch}\,C_2\,\textbf{hc}\,\{\phi'\wedge(\psi_1\vee(\epsilon_1\wedge\psi_2)),\phi'\wedge(\epsilon_1\wedge\epsilon_2)\}$ follows from $\vdash_{\mathsf{Hsa}} \{\phi'\}\,C\,\{\phi'\wedge\psi_1,\phi'\wedge\epsilon_1\}$ and $\vdash_{\mathsf{Hsa}} \{\phi'\wedge\epsilon_1\}\,C_2\,\{\phi'\wedge\epsilon_1\wedge\psi_2,\phi'\wedge\epsilon_1\wedge\epsilon_2\}$. We have $\mathsf{VCSP}^{\mathsf{PA}}(\phi,\rho,\textbf{try}\,C_1\,\textbf{catch}\,C_2\,\textbf{hc}) = (\,\psi_1\vee(\epsilon_1\wedge\psi_2),\ \gamma_1\vee(\mu_1\wedge\gamma_2),\ \epsilon_1\wedge\epsilon_2,\ \mu_1\wedge\mu_2,\ \Gamma_1\cup\Gamma_2\,)$ with

$$\mathsf{VCSP}^{\mathsf{PA}}(\phi',\rho,C_1) = (\,\psi_1,\ \gamma_1,\ \epsilon_1,\ \mu_1,\ \Gamma_1\,) \tag{5.3}$$

$$\mathsf{VCSP}^{\mathsf{PA}}(\phi\wedge\epsilon_1,\rho\wedge\mu_1,C_2) = (\,\psi_2,\ \gamma_2,\ \epsilon_2,\ \mu_1,\ \Gamma_2\,) \tag{5.4}$$

So, from (5.3), by IH, we obtain $\models\Gamma_1$, $\psi_1'\equiv\psi_1\wedge\gamma_1$ and $\epsilon_1'\equiv\epsilon_1\wedge\mu_1$. Hence $(\phi_0\wedge\epsilon_1)\wedge(\rho\wedge\mu_1)\equiv\phi'\wedge\epsilon_1$. So, from (5.4), by IH, we obtain $\models\Gamma_2$, $\psi_2'\equiv\psi_2\wedge\gamma_2$ and $\epsilon_2'\equiv\epsilon_2\wedge\mu_2$. Hence we have $\models\Gamma_1\cup\Gamma_2$, and $\psi_1'\vee(\epsilon_1'\wedge\psi_2')\equiv(\psi_1\vee(\epsilon_1\wedge\psi_2))\wedge(\gamma_1\vee(\mu_1\wedge\gamma_2))$ because $(\psi_1\vee(\epsilon_1\wedge\psi_2))\wedge(\gamma_1\vee(\mu_1\wedge\gamma_2))\equiv(\psi_1\wedge\gamma_1)\vee(\epsilon_1\wedge\psi_2\wedge\gamma_1)\vee(\psi_1\wedge\mu_1\wedge\gamma_2)\vee(\epsilon_1\wedge\psi_2\wedge\mu_1\wedge\gamma_2)$ and, by Lemma 5.8, $\psi_1\wedge\gamma_1\rightarrow\bot$ and $\psi_1\wedge\mu_1\rightarrow\bot$. Moreover, $\epsilon_1'\wedge\epsilon_2'\equiv(\epsilon_1\wedge\epsilon_2)\wedge(\mu_1\wedge\mu_2)$.

Of the remaining cases, the sequence case also uses Lemma 5.8. $\square$

When one wants to generate a set of VCs for a program C, we will have $\mathsf{VCSP}^{\mathsf{PA}}(\top,\top,C) = (\psi,\gamma,\epsilon,\mu,\Gamma)$ and it is trivial that $\top=\top\wedge\top$. It is also possible to feed a precondition to the VCGen in the parameter ρ, as long as, the previous mentioned conditions are met.

To prove that the triple $\{\phi\}\,C\,\{\psi,\epsilon\}$, where $\phi\#C$, is valid, we use $\mathsf{VCSP}^{\mathsf{PA}}$ to obtain the program encoding and set of VCs as $\mathsf{VCSP}^{\mathsf{PA}}(\top,\phi,C) = (\psi',\gamma',\epsilon',\mu',\Gamma)$, and prove that $\models\Gamma$, $\models\psi'\wedge\gamma'\rightarrow\psi$, and $\models\epsilon'\wedge\mu'\rightarrow\epsilon$. From the previous proposition we have that there exists some

$\psi'', \epsilon \in$ **Assert** such that $\psi'' \equiv \psi' \wedge \gamma'$, $\epsilon'' \equiv \epsilon' \wedge \mu'$, and $\vdash_{\mathsf{Hsa}} \{\phi \wedge \top\}\, C\, \{\phi \wedge \top \wedge \psi'', \phi \wedge \top \wedge \epsilon''\}$. Then from Propositions 5.1, 5.3 and 5.6 it holds that $\models \{\phi \wedge \top\}\, C\, \{\phi \wedge \top \wedge \psi'', \phi \wedge \top \wedge \epsilon''\}$.

Let us now prove that all other variants generate VCs that are equivalent to VCSP and also that the encoding of the programs returned by the different variants are in some way related. The lemma below establishes that for all the variants, the returned encoding of the program, both operational and axiomatic, only depends on the input program.

Lemma 5.10. *Let* $i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$. *If* $\mathsf{VCSP}^i(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$ *and* $\mathsf{VCSP}^i(\phi', \rho', C) = (\psi', \gamma', \epsilon', \mu', \Gamma')$, *then* $\psi = \psi'$, $\gamma = \gamma'$, $\epsilon = \epsilon'$, *and* $\mu = \mu'$.

Proof. By induction on C. All cases are straightforward. □

With respect to the encoding of the program, observe that the generated operational encoding is the same for all the variants and that the axiomatic encoding varies depending on whether asserts are used as lemmas or not, as stated in the lemma below.

Lemma 5.11. *If* $\mathsf{VCSP}^{\mathsf{P}}(\phi_1, \rho_1, C) = (\psi_1, \gamma_1, \epsilon_1, \mu_1, \Gamma_1)$, $\mathsf{VCSP}^{\mathsf{PA}}(\phi_2, \rho_2, C) = (\psi_2, \gamma_2, \epsilon_2, \mu_2, \Gamma_2)$, $\mathsf{VCSP}^{\mathsf{G}}(\phi_3, \rho_3, C) = (\psi_3, \gamma_3, \epsilon_3, \mu_3, \Gamma_3)$, *and* $\mathsf{VCSP}^{\mathsf{GA}}(\phi_4, \rho_4, C) = (\psi_4, \gamma_4, \epsilon_4, \mu_4, \Gamma_4)$, *then* $\psi_1 = \psi_2 = \psi_3 = \psi_4$, $\epsilon_1 = \epsilon_2 = \epsilon_3 = \epsilon_4$, $\gamma_1 = \gamma_3$, $\gamma_2 = \gamma_4$, $\mu_1 = \mu_3$, *and* $\mu_2 = \mu_4$.

Proof. By induction on C. All cases are straightforward. □

From this point onward, we will use this result without mentioning it. Moreover, we also need to rely on the fact that all variables occurring in the generated tuple come only from the received arguments, that is, the VCGens do not produce new variables.

Lemma 5.12. *Let* $i \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$. *If* $\mathsf{VCSP}^i(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$, *then* $\mathsf{FV}(\psi) \cup \mathsf{FV}(\gamma) \cup \mathsf{FV}(\epsilon) \cup \mathsf{FV}(\mu) \subseteq \mathsf{Vars}(C)$, *and* $\mathsf{FV}(\bigwedge \Gamma) \subseteq \mathsf{FV}(\phi) \cup \mathsf{FV}(\rho) \cup \mathsf{Vars}(C)$.

Proof. By induction on C. All cases are straightforward. □

We will now relate the variants that use asserts as lemmas and those that do not. Basically, even if asserts are not used as lemmas, they can be obtained by putting together all the other components.

Lemma 5.13. *Let* $(i, j) \in \{(\mathsf{P}, \mathsf{PA}), (\mathsf{G}, \mathsf{GA})\}$. *If* $\phi \wedge \rho \to \rho'$, $\rho' \to \rho$, $\mathsf{VCSP}^i(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$, $\mathsf{VCSP}^j(\phi, \rho', C) = (\psi', \gamma', \epsilon', \mu', \Gamma')$, *and either* $\models \Gamma$ *or* $\models \Gamma'$, *then the following hold:*

1. $\models \phi \wedge \psi \wedge \rho \wedge \gamma \to \gamma'$ *and* $\models \gamma' \to \gamma$.

2. $\models \phi \wedge \epsilon \wedge \rho \wedge \mu \to \mu'$ *and* $\models \mu' \to \mu$.

Proof. The proof proceeds by induction on C for the case in which $(i, j) = (\mathsf{P}, \mathsf{PA})$. For $(i, j) = (\mathsf{G}, \mathsf{GA})$ the proof is analogous.

Case $C = \mathbf{assert}\, \theta$. Since $\mu = \mu' = \bot$ we have (2). If $\models \{\phi \wedge \rho \to \theta\}$ we have (1). Otherwise, we must have that $\models \{\phi \wedge \rho' \to \theta\}$ and since we have $\phi \wedge \rho \to \rho'$ we can conclude (1).

Case $C = C_1\,;\,C_2$. We have

$$\begin{aligned}\mathsf{VCSP}^{\mathsf{P}}(\phi,\rho,C_1\,;\,C_2) = (\,\psi_1\wedge\psi_2,\,\gamma_1\wedge\gamma_2,\,\epsilon_1\vee(\psi_1\wedge\epsilon_2),\,\mu_1\vee(\gamma_1\wedge\mu_2),\,\Gamma_1\cup\Gamma_2\,)\\ \text{with}\;\;(\,\psi_1,\,\gamma_1,\,\epsilon_1,\,\mu_1,\,\Gamma_1\,) = \mathsf{VCSP}^{\mathsf{P}}(\phi,\rho,C_1) \qquad (5.5)\\ (\,\psi_2,\,\gamma_2,\,\epsilon_2,\,\mu_2,\,\Gamma_2\,) = \mathsf{VCSP}^{\mathsf{P}}(\phi\wedge\psi_1,\rho\wedge\gamma_1,C_2) \qquad (5.6)\end{aligned}$$

$$\begin{aligned}\mathsf{VCSP}^{\mathsf{PA}}(\phi,\rho',C_1\,;\,C_2) = (\,\psi_1\wedge\psi_2,\,\gamma_1'\wedge\gamma_2',\,\epsilon_1\vee(\psi_1\wedge\epsilon_2),\,\mu_1'\vee(\gamma_1'\wedge\mu_2'),\,\Gamma_1'\cup\Gamma_2'\,)\\ \text{with}\;\;(\,\psi_1,\,\gamma_1',\,\epsilon_1,\,\mu_1',\,\Gamma_1'\,) = \mathsf{VCSP}^{\mathsf{PA}}(\phi,\rho',C_1) \qquad (5.7)\\ (\,\psi_2,\,\gamma_2',\,\epsilon_2,\,\mu_2',\,\Gamma_2'\,) = \mathsf{VCSP}^{\mathsf{PA}}(\phi\wedge\psi_1,\rho'\wedge\gamma_1',C_2) \qquad (5.8)\end{aligned}$$

and $\models \Gamma_1\cup\Gamma_2$ holds or $\models \Gamma_1'\cup\Gamma_2'$ holds.
From $\phi\wedge\rho\rightarrow\rho'$, $\rho'\rightarrow\rho$, and the assumption that $\models\Gamma_1\cup\Gamma_2$ or $\models\Gamma_1'\cup\Gamma_2'$ by IH we obtain

$$\phi\wedge\psi_1\wedge\rho\wedge\gamma_1\rightarrow\gamma_1',\quad \gamma_1'\rightarrow\gamma_1,\quad \phi\wedge\epsilon_1\wedge\rho\wedge\mu_1\rightarrow\mu_1',\quad \mu_1'\rightarrow\mu_1 \qquad (5.9)$$

and with this result by IH we also get:

$$\phi\wedge\psi_1\wedge\psi_2\wedge\rho\wedge\gamma_1\wedge\gamma_2\rightarrow\gamma_2',\quad \gamma_2'\rightarrow\gamma_2,\quad \phi\wedge\psi_1\wedge\epsilon_2\wedge\rho\wedge\gamma_1\wedge\mu_2\rightarrow\mu_2',\quad \mu_2'\rightarrow\mu_2 \quad (5.10)$$

From (5.9) and (5.10) it follows that $\phi\wedge\psi_1\wedge\psi_2\wedge\rho\wedge\gamma_1\wedge\gamma_2\rightarrow\gamma_1'\wedge\gamma_2'$, $\gamma_1'\wedge\gamma_2'\rightarrow\gamma_1\wedge\gamma_2$, and $(\mu_1'\vee(\gamma_1'\wedge\mu_2'))\rightarrow\mu_1\vee(\gamma_1\wedge\mu_2)$. We are then left to prove $\phi\wedge(\epsilon_1\vee(\psi_1\wedge\epsilon_2))\wedge\gamma\wedge(\mu_1\vee(\gamma_1\wedge\mu_2))\rightarrow \mu_1'\vee(\gamma_1'\wedge\mu_2')$. Using Lemma 5.8 it follows that $\phi\wedge(\epsilon_1\vee(\psi_1\wedge\epsilon_2))\wedge\gamma\wedge(\mu_1\vee(\gamma_1\wedge\mu_2))\equiv (\phi\wedge\rho\wedge\epsilon_1\wedge\mu_1)\vee(\phi\wedge\rho\wedge\psi_1\wedge\epsilon_2\wedge\gamma_1\wedge\mu_2)$, and therefore, from (5.9) and (5.10) we get $\mu_1'\vee(\gamma_1'\wedge\mu_2')$.

Case $C =$ **if** b **then** C_1 **else** C_2 **fi**. We have

$$\begin{aligned}\mathsf{VCSP}^{\mathsf{P}}(\phi,\rho,\textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}) = ((b\wedge\psi_1)\vee(\neg b\wedge\psi_2),(b\wedge\gamma_1)\vee(\neg b\wedge\gamma_2),\\ (b\wedge\epsilon_1)\vee(\neg b\vee\epsilon_2),(b\wedge\mu_1)\vee(\neg b\wedge\mu_2),\Gamma_1\cup\Gamma_2)\\ \text{with}\;\;(\,\psi_1,\,\gamma_1,\,\epsilon_1,\,\mu_1,\,\Gamma_1\,) = \mathsf{VCSP}^{\mathsf{P}}(\phi\wedge b,\rho\wedge b,C_1) \qquad (5.11)\\ (\,\psi_2,\,\gamma_2,\,\epsilon_2,\,\mu_2,\,\Gamma_2\,) = \mathsf{VCSP}^{\mathsf{P}}(\phi\wedge\neg b,\rho\wedge\neg b,C_2) \qquad (5.12)\end{aligned}$$

$$\begin{aligned}\mathsf{VCSP}^{\mathsf{PA}}(\phi,\rho',\textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}) = ((b\wedge\psi_1)\vee(\neg b\wedge\psi_2),(b\wedge\gamma_1')\vee(\neg b\wedge\gamma_2'),\\ (b\wedge\epsilon_1)\vee(\neg b\wedge\epsilon_2),(b\wedge\mu_1')\vee(\neg b\wedge\mu_2'),\Gamma_1'\cup\Gamma_2')\\ \text{with}\;\;(\,\psi_1,\,\gamma_1',\,\epsilon_1,\,\mu_1',\,\Gamma_1'\,) = \mathsf{VCSP}^{\mathsf{PA}}(\phi,\rho',C_1) \qquad (5.13)\\ (\,\psi_2,\,\gamma_2',\,\epsilon_2,\,\mu_2',\,\Gamma_2'\,) = \mathsf{VCSP}^{\mathsf{PA}}(\phi\wedge\psi_1,\rho'\wedge\gamma_1',C_2) \qquad (5.14)\end{aligned}$$

and $\models\Gamma_1\cup\Gamma_2$ holds or $\models\Gamma_1'\cup\Gamma_2'$ holds.

From $\phi \wedge \rho \rightarrow \rho'$, $\rho' \rightarrow \rho$, and the assumption that $\models \Gamma_1 \cup \Gamma_2$ or $\models \Gamma_1' \cup \Gamma_2'$, by IH we obtain:

$$\phi \wedge b \wedge \psi_1 \wedge \rho \wedge \gamma_1 \rightarrow \gamma_1', \quad \gamma_1' \rightarrow \gamma_1, \quad \phi \wedge b \wedge \epsilon_1 \wedge \gamma \wedge \mu_1 \rightarrow \mu_1', \quad \mu_1' \rightarrow \mu_1 \tag{5.15}$$

$$\phi \wedge \neg b \wedge \psi_2 \wedge \rho \wedge \gamma_2 \rightarrow \gamma_2', \quad \gamma_2' \rightarrow \gamma_2, \quad \phi \wedge \neg b \wedge \epsilon_2 \wedge \gamma \wedge \mu_2 \rightarrow \mu_2', \quad \mu_2' \rightarrow \mu_2 \tag{5.16}$$

The following proposition equivalence holds $\phi \wedge ((b \wedge \psi_1) \vee (\neg b \wedge \psi_2)) \wedge \rho \wedge ((b \wedge \gamma_1) \vee (\neg b \wedge \gamma_2)) \equiv (\phi \wedge \rho \wedge b \wedge \psi_1 \wedge \gamma_1) \vee (\phi \wedge \rho \wedge \neg b \wedge \psi_2 \wedge \gamma_2)$, and therefore, from (5.15) and (5.16) we get $(b \wedge \gamma_1') \vee (\neg b \wedge \gamma_2')$, and $(b \wedge \gamma_1') \vee (\neg b \wedge \gamma_2') \rightarrow (b \wedge \gamma_1) \vee (\neg b \wedge \gamma_2)$. The rest of the proof is analogous.

The other cases are analogous. □

It is now possible to prove that, in terms of validity of the generated VCs, it is indifferent whether asserted conditions are added to contexts or not. In particular for the partial context variants we have the following:

Proposition 5.14 ($\mathsf{VCSP}^{\mathsf{P}}$ and $\mathsf{VCSP}^{\mathsf{PA}}$)**.** *If* $\phi \wedge \rho \rightarrow \rho'$, $\rho' \rightarrow \rho$, $\mathsf{VCSP}^{\mathsf{P}}(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$, *and* $\mathsf{VCSP}^{\mathsf{PA}}(\phi, \rho', C) = (\psi, \gamma', \epsilon, \mu', \Gamma')$, *then* $\models \Gamma$ *iff* $\models \Gamma'$.

Proof. The proof follows by induction on C.

Case $C = \mathbf{assert}\,\theta$. We have that $\phi \wedge \rho \rightarrow \rho'$ and $\rho' \rightarrow \rho$. Assume that $\models \phi \wedge \rho \rightarrow \theta$ holds. Then $\models \phi \wedge \rho' \rightarrow \phi \wedge \rho \rightarrow \theta$. Otherwise $\models \phi \wedge \rho' \rightarrow \theta$ holds and therefore $\models \phi \wedge \rho \rightarrow \phi \wedge \rho' \rightarrow \theta$.

Case $C = C_1; C_2$. We have

$$\begin{aligned} \mathsf{VCSP}^{\mathsf{P}}(\phi, \rho, C_1\,;\, C_2) = (\, & \psi_1 \wedge \psi_2,\, \gamma_1 \wedge \gamma_2,\, \epsilon_1 \vee (\psi_1 \wedge \epsilon_2),\, \mu_1 \vee (\gamma_1 \wedge \mu_2),\, \Gamma_1 \cup \Gamma_2\,) \\ \text{with } & (\,\psi_1,\, \gamma_1,\, \epsilon_1,\, \mu_1,\, \Gamma_1\,) = \mathsf{VCSP}^{\mathsf{P}}(\phi, \rho, C_1) \qquad (5.17) \\ & (\,\psi_2,\, \gamma_2,\, \epsilon_2,\, \mu_2,\, \Gamma_2\,) = \mathsf{VCSP}^{\mathsf{P}}(\phi \wedge \psi_1, \rho \wedge \gamma_1, C_2) \qquad (5.18) \end{aligned}$$

$$\begin{aligned} \mathsf{VCSP}^{\mathsf{PA}}(\phi, \rho', C_1\,;\, C_2) = (\, & \psi_1 \wedge \psi_2,\, \gamma_1' \wedge \gamma_2',\, \epsilon_1 \vee (\psi_1 \wedge \epsilon_2),\, \mu_1' \vee (\gamma_1' \wedge \mu_2'),\, \Gamma_1' \cup \Gamma_2'\,) \\ \text{with } & (\,\psi_1,\, \gamma_1',\, \epsilon_1,\, \mu_1',\, \Gamma_1'\,) = \mathsf{VCSP}^{\mathsf{PA}}(\phi, \rho', C_1) \qquad (5.19) \\ & (\,\psi_2,\, \gamma_2',\, \epsilon_2,\, \mu_2',\, \Gamma_2'\,) = \mathsf{VCSP}^{\mathsf{PA}}(\phi \wedge \psi_1, \rho' \wedge \gamma_1', C_2) \qquad (5.20) \end{aligned}$$

Assume $\models \Gamma_1 \cup \Gamma_2$. From Lemma 5.13 we have that $\phi \wedge \psi_1 \wedge \rho \wedge \gamma_1 \rightarrow \gamma_1'$ and $\gamma_1' \rightarrow \gamma_1$. Therefore, by IH (twice) we have $\models \Gamma_1' \cup \Gamma_2'$. The inverse implication is analogous.

The other cases are analogous. □

In order to make the bridge between the partial and global context variants, we will need a few more lemmas. The following states that if the axiomatic context fed to the VCGen is contradictory with some other condition then, using that other condition to prove the set of VCs originates an inconsistency, and therefore, the set of VCs becomes trivially valid. The intuition behind this is that the condition $\rho \wedge \delta$ will appear in the left hand side of all the VCs, and appending to it the condition θ creates the inconsistency.

Lemma 5.15. *Let* $i \in \{\mathsf{G}, \mathsf{GA}\}$*. If* $\delta \wedge \theta \to \perp$ *and* $\mathsf{VCSP}^i(\phi, \rho \wedge \delta, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$*, then* $\models \theta \to \bigwedge \Gamma$.

Proof. By induction on C. All cases are straightforward. □

At this point, it becomes essential to state that the encoding of the program is never inconsistent, in the sense that, there is always some state that satisfies one of the operational encodings referring to normal or exceptional termination. This is captured by the following lemma, and is crucial to prove the subsequent one.

Lemma 5.16. *Let* $s \in \Sigma$*, and* $i \in \{\mathsf{G}, \mathsf{GA}\}$*, such that* $s \models \phi$ *and* $\phi \# C$*. If* $\mathsf{VCSP}^i(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$*, then there is some* $s' \in \Sigma$ *such that for all* $x \in \mathbf{Var} \setminus \mathsf{Asgn}(C)$*,* $s'(x) = s(x)$*, and either* $s' \models \phi \wedge \psi$ *or* $s' \models \phi \wedge \epsilon$*.*

Proof. The proof follows by induction on C for $i = \mathsf{G}$. For $i = \mathsf{GA}$ the proof is analogous.

Case $C = \mathbf{skip}$, $C = \mathbf{assume}\,\theta$, $C = \mathbf{assert}\,\theta$ or $C = \mathbf{throw}$. Trivial, since $\mathsf{Asgn}(C) = \emptyset$, and either ψ is a tautology or ϵ is a tautology.

Case $C = x := e$. We can take $s' = s[x \mapsto [\![e]\!](s)]$. Clearly for all $y \in \mathbf{Var} \setminus \{x\}$, $s'(y) = s(y)$ and since $x := e \in \mathbf{AComm}^{\mathrm{SA}}$, we have $x \notin \mathsf{FV}(e)$ and therefore $s' \models x = e$.

Case $C = C_1 \,;\, C_2$. Since we have $s \models \phi$ and $\phi \# C$, by IH we have that there some $s' \in \Sigma$ such that for all $x \in \mathbf{Var} \setminus \mathsf{Asgn}(C_1)$, $s'(x) = s(x)$, and either $s' \models \phi \wedge \psi_1$ or $s' \models \phi \wedge \epsilon_1$.

- If $s' \models \phi \wedge \epsilon_1$ we are done, because $\mathsf{Asgn}(C_1) \subseteq \mathsf{Asgn}(C_1 \,;\, C_2)$ and $s' \models \phi \wedge (\epsilon_1 \vee (\psi_1 \wedge \epsilon_2))$.
- If $s' \models \phi \wedge \psi_1$ then from Lemma 5.12 and Definition 5.6 we have $(\phi \wedge \psi_1) \# C_2$ and therefore, by IH we have that exists some $s'' \in \Sigma$, such that for all $x \in \mathbf{Var} \setminus \mathsf{Asgn}(C_2)$, $s''(x) = s'(x)$ and either $s'' \models \phi \wedge \psi_1 \wedge \psi_2$ or $s'' \models \phi \wedge \psi_1 \wedge \epsilon_2$. We have that for all $x \in \mathbf{Var} \setminus \mathsf{Asgn}(C_1 \,;\, C_2)$, $s''(x) = s(x)$ because $x \notin \mathsf{Asgn}(C_2)$ and $x \notin \mathsf{Asgn}(C_1)$ we therefore $s''(x) = s'(x) = s(x)$. Finally we have that either $s'' \models \phi \wedge \psi_1 \wedge \psi_2$ or $s'' \models \phi \wedge (\epsilon_1 \vee (\psi_1 \wedge \epsilon_2))$.

Case $C = \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}$. Analogous to the sequence case.

Case $C = \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}$. Since $s \models \phi$, we clearly have that $s \models \phi \wedge b$ or $s \models \phi \wedge \neg b$.

- If $s \models \phi \wedge b$, since $(\phi \wedge b) \# C_1$, by IH there exists some $s' \in \Sigma$, such that for all $x \in \mathbf{Var} \setminus \mathsf{Asgn}(C_1)$, $s'(x) = s(x)$, and $s' \models \phi \wedge b \wedge \psi_1$ or $s' \models \phi \wedge b \wedge \epsilon_1$. We have that for all $x \in \mathbf{Var} \setminus \mathsf{Asgn}(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi})$, $s'(x) = s(x)$, because $\mathsf{Asgn}(C_1) \subseteq \mathsf{Asgn}(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi})$.
  - Now, if $s' \models \phi \wedge b \wedge \psi_1$ then we are done because $s' \models \phi \wedge ((b \wedge \psi_1) \vee (\neg b \wedge \psi_2))$.
  - Otherwise, $s' \models \phi \wedge b \wedge \epsilon_1$ and we are done because $s' \models \phi \wedge ((b \wedge \epsilon_1) \vee (\neg b \wedge \epsilon_2))$.
- For the case $s \models \phi \wedge \neg b$ the proof is analogous.

□

Using the lemma above, we can now prove that if we have two programs, for instance C_1 and C_2, and if C_2 does not assign variables from C_1 then the encoding coming from C_2 does not influence the proof of the properties generated for C_1. The idea here is to isolate the part of the context that is relevant to prove the set of properties and exclude the rest.

Lemma 5.17. *Let* $i \in \{\mathsf{G}, \mathsf{GA}\}$. *If* $\mathsf{Vars}(C_1) \cap \mathsf{Asgn}(C_2) = \emptyset$, $\phi \# C_1$, $\phi \# C_2$, $\rho \# C_1$, $\rho \# C_2$, $\mathsf{VCSP}^i(\phi, \rho, C_1) = (\psi, \gamma, \epsilon, \mu, \Gamma)$, *and* $\mathsf{VCSP}^i(\phi', \rho', C_2) = (\psi', \gamma', \epsilon', \mu', \Gamma')$, *then the following hold:*

1. *If* $\models \phi \wedge \psi \wedge \psi' \to \bigwedge \Gamma$ *and* $\models \phi \wedge \psi \wedge \epsilon' \to \bigwedge \Gamma$ *then* $\models \phi \wedge \psi \to \bigwedge \Gamma$.
2. *If* $\models \phi \wedge \epsilon \wedge \psi' \to \bigwedge \Gamma$ *and* $\models \phi \wedge \epsilon \wedge \epsilon' \to \bigwedge \Gamma$ *then* $\models \phi \wedge \epsilon \to \bigwedge \Gamma$.

Proof. (1) The proof follows by contradiction. Assume that

$$\models \phi \wedge \psi \wedge \psi' \to \bigwedge \Gamma \text{ and} \models \phi \wedge \psi \wedge \epsilon' \to \bigwedge \Gamma \tag{5.21}$$

Assume also that $\not\models \phi \wedge \psi \to \bigwedge \Gamma$, that is, there exists some $s \in \Sigma$ such that $s \models \phi \wedge \psi$ and $s \not\models \bigwedge \Gamma$. From $\mathsf{Vars}(C_1) \cap \mathsf{Asgn}(C_2) = \emptyset$, $\phi \# C_2$ and Lemma 5.12 we have that $(\phi \wedge \psi) \# C_2$. Therefore, by Lemma 5.16 we have that there exists some $s' \in \Sigma$, such that for all $x \in \mathbf{Var} \setminus \mathsf{Asgn}(C_2)$, $s'(x) = s(x)$, and either $s' \models \phi \wedge \psi \wedge \psi'$ or $s' \models \phi \wedge \psi \wedge \epsilon'$. Since $\mathsf{FV}(\bigwedge \Gamma) \cap \mathsf{Asgn}(C_2) = \emptyset$ and $s \not\models \bigwedge \Gamma$ we have that $s' \not\models \bigwedge \Gamma$, which is a contradiction of (5.21).

(2) Analogous to the previous. □

Relying on the previous lemmas, it is now possible to prove that the VCs generated by VCSP^P are equivalent to those generated by VCSP^G.

Proposition 5.18 (VCSP^P and VCSP^G). *Let* $\phi \# C$, $\mathsf{VCSP}^\mathsf{P}(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$, *and* $\mathsf{VCSP}^\mathsf{G}(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma')$. *Then* $\models \Gamma$ *iff* $\models \phi \wedge \psi \to \bigwedge \Gamma'$ *and* $\models \phi \wedge \epsilon \to \bigwedge \Gamma'$.

Proof. The proof follows by induction on C.

Cases $C = \mathbf{skip}$, $x := e$, $C = \mathbf{assume}\, \theta$, and $C = \mathbf{throw}$ are trivial, since $\Gamma = \Gamma' = \emptyset$.

Case $C = \mathbf{assert}\, \theta$. In this case we have $\mathsf{VCSP}^\mathsf{P}(\phi, \rho, \mathbf{assert}\, \theta) = (\top, \top, \bot, \bot, \{\phi \wedge \rho \to \theta\})$ and $\mathsf{VCSP}^\mathsf{G}(\phi, \rho, \mathbf{assert}\, \theta) = (\top, \top, \bot, \bot, \{\rho \to \theta\})$. Therefore, it holds that $\models \phi \wedge \rho \to \theta$ iff $\models \phi \wedge \top \to \rho \to \theta$ and $\models \phi \wedge \bot \to \rho \to \theta$.

Case $C = C_1 \,;\, C_2$. We have

$$\begin{aligned} \mathsf{VCSP}^\mathsf{P}(\phi, \rho, C_1 \,;\, C_2) = (\, & \psi_1 \wedge \psi_2,\ \gamma_1 \wedge \gamma_2,\ \epsilon_1 \vee (\psi_1 \wedge \epsilon_2),\ \mu_1 \vee (\gamma_1 \wedge \mu_2),\ \Gamma_1 \cup \Gamma_2 \,) \\ \text{with } & (\, \psi_1,\ \gamma_1,\ \epsilon_1,\ \mu_1,\ \Gamma_1 \,) = \mathsf{VCSP}^\mathsf{P}(\phi, \rho, C_1) && (5.22) \\ & (\, \psi_2,\ \gamma_2,\ \epsilon_2,\ \mu_2,\ \Gamma_2 \,) = \mathsf{VCSP}^\mathsf{P}(\phi \wedge \psi_1, \rho \wedge \gamma_1, C_2) && (5.23) \end{aligned}$$

$$\begin{aligned} \mathsf{VCSP}^\mathsf{G}(\phi, \rho, C_1 \,;\, C_2) = (\, & \psi_1 \wedge \psi_2,\ \gamma_1 \wedge \gamma_2,\ \epsilon_1 \vee (\psi_1 \wedge \epsilon_2),\ \mu_1 \vee (\gamma_1 \wedge \mu_2),\ \Gamma'_1 \cup \Gamma'_2 \,) \\ \text{with } & (\, \psi_1,\ \gamma_1,\ \epsilon_1,\ \mu_1,\ \Gamma'_1 \,) = \mathsf{VCSP}^\mathsf{G}(\phi, \rho, C_1) && (5.24) \\ & (\, \psi_2,\ \gamma_2,\ \epsilon_2,\ \mu_2,\ \Gamma'_2 \,) = \mathsf{VCSP}^\mathsf{G}(\phi \wedge \psi_1, \rho \wedge \gamma_1, C_2) && (5.25) \end{aligned}$$

From $\phi\#C$ and $\rho\#C$ it follows $\phi\#C_1$ and $\rho\#C_1$. So, from (5.22) and (5.24), by IH, we have that

$$\models \Gamma_1 \quad \text{iff} \quad \models \phi \wedge \psi_1 \rightarrow \bigwedge \Gamma'_1 \quad \text{and} \quad \models \phi \wedge \epsilon_1 \rightarrow \bigwedge \Gamma'_1 \tag{5.26}$$

Using Lemma 5.12 we also have $(\phi \wedge \psi_1)\#C_2$ and $(\rho \wedge \gamma_1)\#C_2$. Therefore, from (5.23) and (5.25), by IH, it follows that

$$\models \Gamma_2 \quad \text{iff} \quad \models \phi \wedge \psi_1 \wedge \psi_2 \rightarrow \bigwedge \Gamma'_2 \quad \text{and} \quad \models \phi \wedge \psi_1 \wedge \epsilon_2 \rightarrow \bigwedge \Gamma'_2 \tag{5.27}$$

Let us now prove that

$$\models \Gamma_1 \cup \Gamma_2 \quad \text{iff} \quad \models \phi \wedge \psi_1 \wedge \psi_2 \rightarrow \bigwedge(\Gamma'_1 \cup \Gamma'_2) \quad \text{and} \quad \models \phi \wedge (\epsilon_1 \vee (\psi_1 \wedge \epsilon_2)) \rightarrow \bigwedge(\Gamma'_1 \cup \Gamma'_2)$$

Assume $\models \Gamma_1 \cup \Gamma_2$. From (5.24), by Lemma 5.8, we have $\models \gamma_1 \wedge \epsilon_1 \rightarrow \bot$, and from (5.25), using Lemma 5.15 we have $\models \epsilon_1 \rightarrow \bigwedge \Gamma'_2$. Hence, by (5.26) and (5.27), we can conclude that $\models \phi \wedge \psi_1 \wedge \psi_2 \rightarrow \bigwedge(\Gamma'_1 \cup \Gamma'_2)$ and $\models \phi \wedge (\epsilon_1 \vee (\psi_1 \wedge \epsilon_2)) \rightarrow \bigwedge(\Gamma'_1 \cup \Gamma'_2)$. On the other hand, assume $\models \phi \wedge \psi_1 \wedge \psi_2 \rightarrow \bigwedge(\Gamma'_1 \cup \Gamma'_2)$ and $\models \phi \wedge (\epsilon_1 \vee (\psi_1 \wedge \epsilon_2)) \rightarrow \bigwedge(\Gamma'_1 \cup \Gamma'_2)$. From the last statement we get $\models \phi \wedge \epsilon_1 \rightarrow \bigwedge(\Gamma'_1 \cup \Gamma'_2)$ and $\models \phi \wedge \psi_1 \wedge \epsilon_2 \rightarrow \bigwedge(\Gamma'_1 \cup \Gamma'_2)$. By Lemma 5.17 we get $\models \phi \wedge \psi_1 \rightarrow \bigwedge \Gamma'_1$ and, using (5.26) and (5.27), we can now conclude that $\models \Gamma_1 \cup \Gamma_2$.

Case $C = \textbf{try}\, C_1\, \textbf{catch}\, C_2\, \textbf{hc}$. Analogous to the previous case.

Case $C = \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}$. We have

$$\begin{aligned}
\mathsf{VCSP}^{\mathsf{P}}(\phi, \rho, \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}) = (\,&(b \wedge \psi_1) \vee (\neg b \wedge \psi_2),\ (b \wedge \gamma_1) \vee (\neg b \wedge \gamma_2),\\
&(b \wedge \epsilon_1) \vee (\neg b \vee \epsilon_2),\ (b \wedge \mu_1) \vee (\neg b \wedge \mu_2),\ \Gamma_1 \cup \Gamma_2\,)\\
\text{with}\ \ (\,\psi_1,\ \gamma_1,\ \epsilon_1,\ \mu_1,\ \Gamma_1\,) &= \mathsf{VCSP}^{\mathsf{P}}(\phi \wedge b, \rho \wedge b, C_1) \qquad (5.28)\\
(\,\psi_2,\ \gamma_2,\ \epsilon_2,\ \mu_2,\ \Gamma_2\,) &= \mathsf{VCSP}^{\mathsf{P}}(\phi \wedge \neg b, \rho \wedge \neg b, C_2) \qquad (5.29)
\end{aligned}$$

$$\begin{aligned}
\mathsf{VCSP}^{\mathsf{G}}(\phi, \rho, \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}) = (\,&(b \wedge \psi_1) \vee (\neg b \wedge \psi_2),\ (b \wedge \gamma_1) \vee (\neg b \wedge \gamma_2),\\
&(b \wedge \epsilon_1) \vee (\neg b \wedge \epsilon_2),\ (b \wedge \mu_1) \vee (\neg b \wedge \mu_2),\ \Gamma'_1 \cup \Gamma'_2\,)\\
\text{with}\ \ (\,\psi_1,\ \gamma_1,\ \epsilon_1,\ \mu_1,\ \Gamma'_1\,) &= \mathsf{VCSP}^{\mathsf{G}}(\phi \wedge b, \rho \wedge b, C_1) \qquad (5.30)\\
(\,\psi_2,\ \gamma_2,\ \epsilon_2,\ \mu_2,\ \Gamma'_2\,) &= \mathsf{VCSP}^{\mathsf{G}}(\phi \wedge \neg b, \rho \wedge \neg b, C_2) \qquad (5.31)
\end{aligned}$$

From $\phi\#C$ and $\rho\#C$ it follows $(\phi \wedge b)\#C_1$ and $(\rho \wedge b)\#C_1$. So, from (5.28) and (5.30), by IH, we have that

$$\models \Gamma_1 \quad \text{iff} \quad \models \phi \wedge b \wedge \psi_1 \rightarrow \bigwedge \Gamma'_1 \quad \text{and} \quad \models \phi \wedge b \wedge \epsilon_1 \rightarrow \bigwedge \Gamma'_1 \tag{5.32}$$

We also have $(\phi \wedge \neg b)\#C_2$ and $(\rho \wedge \neg b)\#C_2$. Therefore, from (5.29) and (5.31), by IH, it follows

that

$$\models \Gamma_1 \quad \text{iff} \quad \models \phi \wedge \neg b \wedge \psi_2 \to \bigwedge \Gamma'_2 \quad \text{and} \quad \models \phi \wedge \neg b \wedge \epsilon_2 \to \bigwedge \Gamma'_2 \tag{5.33}$$

From (5.30) and (5.31), by Lemma 5.15, we know that for every θ it holds that $b \wedge \theta \to \bigwedge \Gamma'_1$ and $\neg b \wedge \theta \to \bigwedge \Gamma'_2$. So, using (5.32) and (5.33) we can conclude that: $\models \Gamma_1 \cup \Gamma_2$ iff

$$\models \phi \wedge ((b \wedge \psi_1) \vee (\neg \wedge \psi_2)) \to \bigwedge(\Gamma'_1 \cup \Gamma'_2) \quad \text{and} \quad \models \phi \wedge ((b \wedge \epsilon_1) \vee (\neg \wedge \epsilon_2)) \to \bigwedge(\Gamma'_1 \cup \Gamma'_2)$$

□

Finally, regarding VCSP variants we can conclude that the VCs generated by $\mathsf{VCSP}^{\mathsf{PA}}$ are equivalent to those generated by $\mathsf{VCSP}^{\mathsf{GA}}$. The proof is analogous to the proof of the previous lemma, and it relies on the lemmas that were presented before.

Proposition 5.19 ($\mathsf{VCSP}^{\mathsf{PA}}$ and $\mathsf{VCSP}^{\mathsf{GA}}$). *Let* $\mathsf{VCSP}^{\mathsf{PA}}(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$ *and* $\mathsf{VCSP}^{\mathsf{GA}}(\phi, \rho, C) = (\psi', \gamma', \epsilon', \mu', \Gamma')$. *Then* $\models \Gamma$ *iff* $\models \phi \wedge \psi' \to \bigwedge \Gamma'$ *and* $\models \phi \wedge \epsilon' \to \bigwedge \Gamma$.

Proof. By induction on C. All cases are analogous to the proof of Proposition 5.18. □

We conclude that in terms of validity, it is indifferent to use any of the concrete VCGens from the VCSP family in order to generate VCs. With this, we move our discussion to the VCGens from the VCCNF family.

The Generalized Conditional Normal Form VCGen. Although this family of VCGens is used extensively in bounded model checking of software, none of its variants have, at least as far as we know, been proved sound nor complete w.r.t. a programming semantics. We will now proceed with the correspondence between this family of VCGens and the $\mathsf{VCSP}^{\mathsf{G}}$, which will allow us to establish a relation with the program semantics from the previous section.

Note that the main difference between these two families of VCGens is the way they encode path conditions, therefore, the following lemma makes a correspondence between the path conditions used by $\mathsf{VCCNF}^{\mathsf{G}}$ and the operational and axiomatic context of $\mathsf{VCSP}^{\mathsf{G}}$.

Lemma 5.20. *Let* $C \in \mathbf{AComm}^{\mathrm{SA}}$ *and* $\phi, \phi', \rho, \rho', \pi \in \mathbf{Assert}$. *If* $\mathsf{VCSP}^{\mathsf{G}}(\phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$ *and* $\mathsf{VCCNF}^{\mathsf{G}}(\pi, \phi', \rho', C) = (\psi', \gamma', \epsilon', \mu', \Gamma')$, *then* $(\pi \to \psi) \equiv \psi'$, $(\pi \to \gamma) \equiv \gamma'$, $(\pi \to \epsilon) \equiv \epsilon'$, *and* $(\pi \to \mu) \equiv \mu'$.

Proof. By induction on C. All cases are straightforward. □

With this, we are now ready to establish the relation between the two VCGens. First we show that the set of properties generated by $\mathsf{VCCNF}^{\mathsf{G}}$ is valid, if and only if, the set generated by $\mathsf{VCSP}^{\mathsf{G}}$ is valid. Since this is not enough to establish the relation between the two VCGens, because it does not say anything about the operational encoding, we postpone the final result for the subsequent proposition.

Lemma 5.21. *Let* $C \in \mathbf{AComm}^{\text{SA}}$ *and* $\rho' \wedge \pi \equiv \rho$. *If* $\mathsf{VCSP}^{\mathsf{G}}(\phi,\rho,C) = (\psi,\gamma,\epsilon,\mu,\Gamma)$ *and* $\mathsf{VCCNF}^{\mathsf{G}}(\pi,\phi',\rho',C) = (\psi',\gamma',\epsilon',\mu',\Gamma)$, *then* $\models \Gamma$ *iff* $\models \Gamma'$.

Proof. By induction on C.

Case $C = \mathbf{assert}\,\theta$. Since we have $\rho' \wedge \pi \equiv \rho$, the following holds $\bigwedge\{\rho \to \theta\} \equiv \bigwedge\{\rho' \to \pi \to \theta\}$

Case $C = C_1\,;\,C_2$. We have

$$\mathsf{VCSP}^{\mathsf{G}}(\phi,\rho,C_1\,;\,C_2) = (\,\psi_1 \wedge \psi_2,\, \gamma_1 \wedge \gamma_2,\, \epsilon_1 \vee (\psi_1 \wedge \epsilon_2),\, \mu_1 \vee (\gamma_1 \wedge \mu_2),\, \Gamma_1 \cup \Gamma_2\,)$$

$$\text{with } \;(\,\psi_1,\, \gamma_1,\, \epsilon_1,\, \mu_1,\, \Gamma_1\,) = \mathsf{VCSP}^{\mathsf{G}}(\phi,\rho,C_1) \tag{5.34}$$

$$(\,\psi_2,\, \gamma_2,\, \epsilon_2,\, \mu_2,\, \Gamma_2\,) = \mathsf{VCSP}^{\mathsf{G}}(\phi \wedge \psi_1, \rho \wedge \gamma_1, C_2) \tag{5.35}$$

$$\mathsf{VCCNF}^{\mathsf{G}}(\pi,\phi',\rho',C_1\,;\,C_2) = (\,\psi_1' \wedge \psi_2',\, \gamma_1' \wedge \gamma_2',\, \epsilon_1' \vee (\psi_1' \wedge \epsilon_2'),\, \mu_1' \vee (\gamma_1' \wedge \mu_2'),\, \Gamma_1' \cup \Gamma_2'\,)$$

$$\text{with } \;(\,\psi_1',\, \gamma_1',\, \epsilon_1',\, \mu_1',\, \Gamma_1'\,) = \mathsf{VCCNF}^{\mathsf{G}}(\pi,\phi',\rho',C_1) \tag{5.36}$$

$$(\,\psi_2',\, \gamma_2',\, \epsilon_2',\, \mu_2',\, \Gamma_2'\,) = \mathsf{VCCNF}^{\mathsf{G}}(\pi,\phi' \wedge \psi_1', \rho' \wedge \gamma_1', C_2) \tag{5.37}$$

From Lemma 5.20 we have $\pi \to \gamma_1 \equiv \gamma_1'$, therefore, the following holds $\rho' \wedge \gamma_1' \wedge \pi \equiv \rho' \wedge (\pi \wedge \gamma_1) \wedge \pi \equiv \rho' \wedge \gamma_1 \wedge \pi \equiv \rho \wedge \gamma_1$ (last step comes from $\rho' \wedge \pi \equiv \rho$). By IH we have $\models (\Gamma_1 \cup \Gamma_2)$ iff $\models (\Gamma_1' \cup \Gamma_2')$.

Other cases are analogous. □

Now, putting the last two lemmas together, we can conclude that $\mathsf{VCSP}^{\mathsf{G}}$ is indeed equivalent to $\mathsf{VCCNF}^{\mathsf{G}}$.

Proposition 5.22 ($\mathsf{VCSP}^{\mathsf{G}}$ and $\mathsf{VCCNF}^{\mathsf{G}}$)**.** *If* $\phi' \wedge \pi \equiv \phi$, $\rho' \wedge \pi \equiv \rho$, $\mathsf{VCSP}^{\mathsf{G}}(\phi,\rho,C) = (\psi,\gamma,\epsilon,\mu,\Gamma)$, *and* $\mathsf{VCCNF}^{\mathsf{G}}(\pi,\phi',\rho',C) = (\psi',\gamma',\epsilon',\mu',\Gamma')$ *then,* $\models \phi \wedge \psi \to \bigwedge\Gamma$ *iff* $\models \phi' \wedge \pi \wedge \psi' \to \bigwedge\Gamma'$, *and* $\models \phi \wedge \epsilon \to \bigwedge\Gamma$ *iff* $\models \phi' \wedge \pi \wedge \epsilon' \to \bigwedge\Gamma'$.

Proof. By Lemma 5.21 we have that $\models \Gamma$ iff $\models \Gamma'$. Using Lemma 5.20 we have that $\phi \wedge \psi \equiv \pi \wedge \phi' \wedge \psi'$ and $\phi \wedge \epsilon \equiv \pi \wedge \phi' \wedge \epsilon$. □

It is now possible to show that $\mathsf{VCCNF}^{\mathsf{G}}$ is sound and relatively complete (in the sense of Cook) w.r.t. the program semantics. Since traditionally this VCGen is used to check properties asserted in the code, we state the following corollary with respect to a program and not a Hoare triple. Nonetheless, the corollary is easily adapted for the former.

Corollary 5.23. *If* $\mathsf{VCCNF}^{\mathsf{G}}(\top,\top,\top,C) = (\psi,\gamma,\epsilon,\mu,\Gamma)$, *then* $\models \psi \to \Gamma$ *and* $\models \epsilon \to \Gamma$, *iff for all* $s \in \Sigma$, $\langle C, s\rangle \rightsquigarrow \sigma$ *implies* $\sigma \neq \bullet$.

Proof. Follows from Propositions 5.22, 5.18, 5.14, 5.9, 5.6, 5.3, and 5.1. □

In order to expand the previous corollary for the other variants of the VCCNF family, we need to create lemmas that are analogous to those used in the proofs of equivalence of the

VCGens from the VCSP family. Since this would be tedious and pretty straightforward, we will omit here the proofs for the rest of the cases and state only the equivalence results in the following propositions.

Proposition 5.24 ($\mathsf{VCCNF}^{\mathsf{P}}$ and $\mathsf{VCCNF}^{\mathsf{PA}}$). *If* $\phi \wedge \rho \rightarrow \rho'$, $\rho' \rightarrow \rho$, $\mathsf{VCCNF}^{\mathsf{P}}(\pi, \phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$, *and* $\mathsf{VCCNF}^{\mathsf{PA}}(\pi, \phi, \rho', C) = (\psi, \gamma', \epsilon, \mu', \Gamma')$, *then* $\models \Gamma$ *iff* $\models \Gamma'$.

Proof. Analogous to the proof of Proposition 5.14. □

Proposition 5.25 ($\mathsf{VCCNF}^{\mathsf{P}}$ and $\mathsf{VCCNF}^{\mathsf{G}}$). *If* $\phi \# C$, $\mathsf{VCCNF}^{\mathsf{P}}(\pi, \phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$, *and* $\mathsf{VCCNF}^{\mathsf{G}}(\pi, \phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma')$, *then* $\models \Gamma$ *iff* $\models \phi \wedge \pi \wedge \psi \rightarrow \bigwedge \Gamma'$ *and* $\models \phi \wedge \pi \wedge \epsilon \rightarrow \bigwedge \Gamma'$.

Proof. Analogous to the proof of Proposition 5.18. □

Proposition 5.26 ($\mathsf{VCCNF}^{\mathsf{PA}}$ and $\mathsf{VCCNF}^{\mathsf{GA}}$). *If* $\mathsf{VCCNF}^{\mathsf{PA}}(\pi, \phi, \rho, C) = (\psi, \gamma, \epsilon, \mu, \Gamma)$ *and* $\mathsf{VCCNF}^{\mathsf{GA}}(\pi, \phi, \rho, C) = (\psi', \gamma', \epsilon', \mu', \Gamma')$ *then* $\models \Gamma$ *iff* $\models \phi \wedge \pi \wedge \psi' \rightarrow \bigwedge \Gamma'$ *and* $\models \phi \wedge \pi \wedge \epsilon' \rightarrow \bigwedge \Gamma$.

Proof. Analogous to the proof of Proposition 5.19. □

Using the propositions presented in this section, it is now possible to show that all VCGens generate VCs that are equivalent. This result will be particularly useful when proving the correction of the verification framework in the next section.

Corollary 5.27. *Let* $\mathsf{v}, \mathsf{v}' \in \{\mathsf{SP}, \mathsf{CNF}\}$, $\mathsf{d}, \mathsf{d}' \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$, $\mathsf{VCG_T}(\phi, C, \psi, \epsilon, \mathsf{v}, \mathsf{d}) = \Gamma$, *and* $\mathsf{VCG_T}(\phi, C, \psi, \epsilon, \mathsf{v}', \mathsf{d}') = \Gamma'$. *Then* $\models \Gamma$ *iff* $\models \Gamma'$.

Proof. The proof follows by expanding the function $\mathsf{VCG_T}$ and by analysis of cases using Propositions 5.26, 5.25, 5.24, 5.22, 5.19, 5.18, and 5.14. □

5.4 A Proved Verification Framework

Recall that our goal is to have a completely proved verification workflow based on the translation of programs into SA. At this point, we have a set of frameworks to reason about non-SA programs and also a set of optimized frameworks to reason about SA programs (the system Hsa and the VCGens that were shown to be sound and complete in the previous section) . However, our main goal is not to reason about SA programs, but else to reason about non-SA programs by translating them into SA. For us to have a completely proved verification framework, such a translation of non-SA programs into SA programs needs to be formalized. In particular, a translation must preserve the validity of Hoare triples and must not translate invalid triples into valid ones.

Similarly to Section 3.1, in this section instead of proposing a concrete SA translation and showing that it fits in our verified framework, we introduce formally the notion of SA translation and then prove that any translation that complies with such a notion can be plugged into the verification framework without compromising the soundness and completeness of the verification method. Let us start by the defining what is an SA translation. The formulation of the following

definition follows closely Definition 3.1, but it uses the tools and concepts about programs presented in this chapter.

Definition 5.8 (SA translation)**.** *Let* $C \in \mathbf{AComm}$*,* $\phi, \psi, \epsilon \in \mathbf{Assert}$*, and* $\mathfrak{T} : \mathbf{Assert} \times \mathbf{AComm} \times \mathbf{Assert} \times \mathbf{Assert} \hookrightarrow \mathbf{Assert} \times \mathbf{AComm}^{\text{SA}} \times \mathbf{Assert} \times \mathbf{Assert}$. *The function* $\mathfrak{T}$ *is said to be a* SA translation *if when* $\mathfrak{T}(\phi, C, \psi, \epsilon) = (\phi', C', \psi', \epsilon')$*, we have that* $\phi' \# C'$*, and both the following hold:*

1. *If* $\models \{\phi'\}\, C'\, \{\psi', \epsilon'\}$*, then* $\models \{\phi\}\, C\, \{\psi, \epsilon\}$*.*
2. *If* $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$*, then* $\vdash_{\mathsf{Hg}} \{\phi'\}\, C'\, \{\psi', \epsilon'\}$*.*

With this, it is now possible to reason about the translation of programs into SA form without actually defining a concrete translation. In particular, this allows us to consider the complete verification workflow, based on the use of SA form, and the generation of VCs using the VCGens from the cube. We start by showing that when all the VCs generated by $\mathsf{VCG_T}$ (defined in Figure 4.10) are valid, then the original triple is semantically valid, that is, the verification workflow is sound.

Proposition 5.28. *Let* $C \in \mathbf{AComm}$*,* $\phi, \psi, \epsilon \in \mathbf{Assert}$*,* $\mathsf{v} \in \{\mathsf{SP}, \mathsf{CNF}\}$*,* $\mathsf{d} \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$*, and* $\mathfrak{T} : \mathbf{Assert} \times \mathbf{AComm} \times \mathbf{Assert} \times \mathbf{Assert} \hookrightarrow \mathbf{Assert} \times \mathbf{AComm}^{\text{SA}} \times \mathbf{Assert} \times \mathbf{Assert}$ *such that* $\mathfrak{T}$ *is a valid SA translation. Then, if* $\mathfrak{T}(\phi, C, \psi, \epsilon) = (\phi', C', \psi', \epsilon')$ *and* $\models \mathsf{VCG_T}(\phi', C', \psi', \epsilon', \mathsf{v}, \mathsf{d})$*, we have that* $\models \{\phi\}\, C\, \{\psi, \epsilon\}$*.*

Proof. Assume $\mathfrak{T}(\phi, C, \psi, \epsilon) = (\phi', C', \psi', \epsilon')$, $\mathsf{VCG_T}(\phi', C', \psi', \epsilon', \mathsf{v}, \mathsf{d}) = \Gamma$, and $\models \Gamma$. From Corollary 5.27, we have that $\mathsf{VCG_T}(\phi', C', \psi', \epsilon', \mathsf{SP}, \mathsf{PA}) = \Gamma'$ and $\models \Gamma'$, and by inspection of function $\mathsf{VCG_T}$, it must be that $\mathsf{VCSP}^{\mathsf{PA}}(\top, \phi', C') = (\psi_1, \gamma_1, \epsilon_1, \mu_1, \Gamma_1)$, $\models \Gamma_1$, $\models \phi' \wedge \psi_1 \wedge \gamma_1 \rightarrow \psi'$, and $\models \phi' \wedge \epsilon_1 \wedge \mu_1 \rightarrow \epsilon'$. By Proposition 5.9 we have that $\vdash_{\mathsf{Hsa}} \{\phi'\}\, C'\, \{\phi' \wedge \psi_2, \phi' \wedge \epsilon_2\}$, for some $\psi_2, \epsilon_2 \in \mathbf{Assert}$ such that $\psi_2 \equiv \psi_1 \wedge \gamma_1$ and $\epsilon_2 \equiv \epsilon_1 \wedge \mu_1$. Since $\phi' \# C'$, by Proposition 5.6 we obtain $\vdash_{\mathsf{H}} \{\phi'\}\, C'\, \{\phi' \wedge \psi_2, \phi' \wedge \epsilon_2\}$ and applying the (conseq) rule we have that $\vdash_{\mathsf{H}} \{\phi'\}\, C'\, \{\psi', \epsilon'\}$, because $\psi_2 \equiv \psi_1 \wedge \gamma_1$ and $\models \phi' \wedge \psi_1 \wedge \gamma_1 \rightarrow \psi'$, and also because $\epsilon_2 \equiv \epsilon_1 \wedge \mu_1$ and $\models \phi' \wedge \epsilon_1 \wedge \mu_1 \rightarrow \epsilon'$. Finally, by Proposition 5.1, we have $\models \{\phi'\}\, C'\, \{\psi', \epsilon'\}$, and from Definition 5.8, we conclude that $\models \{\phi\}\, C\, \{\psi, \epsilon\}$. □

We also need to express that whenever a triple is originally valid and the program correctly-annotated w.r.t. the specification, then the VCs that will be obtained by converting it into SA, and generating a set of VCs using one of the VCGens from the cube, will be valid. This shows basically that the verification workflow is complete (in the sense of Cook) for correctly annotated programs.

Proposition 5.29. *Let* $C \in \mathbf{AComm}$*,* $\phi, \psi, \epsilon \in \mathbf{Assert}$*,* $\mathsf{v} \in \{\mathsf{SP}, \mathsf{CNF}\}$*,* $\mathsf{d} \in \{\mathsf{P}, \mathsf{PA}, \mathsf{G}, \mathsf{GA}\}$ *and* $\mathfrak{T} : \mathbf{Assert} \times \mathbf{AComm} \times \mathbf{Assert} \times \mathbf{Assert} \hookrightarrow \mathbf{Assert} \times \mathbf{AComm}^{\text{SA}} \times \mathbf{Assert} \times \mathbf{Assert}$ *such that* $\mathfrak{T}$ *is an SA translation and* $\mathfrak{T}(\phi, C, \psi, \epsilon) = (\phi', C', \psi', \epsilon')$*. If* $\models \{\phi\}\, C\, \{\psi, \epsilon\}$ *and* C *is correctly-annotated w.r.t.* (ϕ, ψ, ϵ)*, then* $\mathsf{VCG_T}(\phi', C', \psi', \epsilon', \mathsf{v}, \mathsf{d}) = \Gamma$*, for some* Γ*, and* $\models \Gamma$*.*

Proof. Assume that $\mathfrak{T}(\phi, C, \psi, \epsilon) = (\phi', C', \psi', \epsilon')$, $\models \{\phi\}\, C\, \{\psi, \epsilon\}$ and C is correctly-annotated w.r.t. (ϕ, ψ, ϵ). From Proposition 5.2, we have that $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi, \epsilon\}$ and from Definition 5.4, $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$. Now, from Definition 5.8 we have that $\vdash_{\mathsf{Hg}} \{\phi'\}\, C\, \{\psi', \epsilon'\}$, and since $\phi' \# C'$, by Proposition 5.7 it is possible to obtain $\vdash_{\mathsf{Hsa}} \{\phi'\}\, C\, \{\phi' \wedge \psi_1, \phi' \wedge \epsilon_1\}$, $\models \phi' \wedge \psi_1 \rightarrow \psi'$, $\models \phi' \wedge \epsilon_1 \rightarrow \epsilon'$. It follows from Proposition 5.9 that if $\mathsf{VCSP}^{\mathsf{PA}}(\top, \phi', C') = (\psi_2, \gamma_2, \epsilon_2, \mu_2, \Gamma_2)$, then $\models \Gamma_2$, $\psi_1 \equiv \psi_2 \wedge \gamma_2$, and $\epsilon_1 \equiv \epsilon_2 \wedge \mu_2$. Finally, by definition $\mathsf{VCG}_{\mathsf{T}}(\phi', C', \psi_1, \epsilon_1, \mathsf{SP}, \mathsf{PA}) = \Gamma_2 \cup \{\phi' \wedge \psi_2 \wedge \gamma_2 \rightarrow \psi_1, \phi' \wedge \epsilon_2 \wedge \mu_2 \rightarrow \epsilon_1\}$. Since we have Γ_2, and $\phi' \wedge \psi_2 \wedge \gamma_2 \rightarrow \psi_1$ (resp. $\phi' \wedge \epsilon_2 \wedge \mu_2 \rightarrow \epsilon_1$) follows from $\psi_1 \equiv \psi_2 \wedge \gamma_2$ and $\phi' \wedge \psi_1 \rightarrow \psi'$ (resp.$\epsilon_1 \equiv \epsilon_2 \wedge \mu_2$ and $\phi' \wedge \epsilon_1 \rightarrow \epsilon'$), we conclude from Corollary 5.27 that $\mathsf{VCG}_{\mathsf{T}}(\phi', C', \psi', \epsilon', \mathsf{v}, \mathsf{d}) = \Gamma$, for some Γ, and $\models \Gamma$. □

5.5 Related Work

The verification framework of this chapter stands in contrast with the one of Chapter 3 in that it is based on the translation of programs into a non-iterating single-assignment form. Moreover it also considers a richer language by allowing assume, assert, and exceptions at both the source and intermediate language. Even though these commands can be omitted in the source language without affecting the overall formalization, they are essential for capturing loops in the SA translation.

A first attempt to verify programs with exception handling was proposed by Cristian [39]. The author formally defines a programming language with support for exceptions, and then proposes a deductive system Hoare triples. Along the years research was made towards the verification of programs with exceptions, in particular in the context of Java [67, 94]. It is common for the state of the art deductive verification tools to support exception handling. For instance, Why3 [50] provides a programming language called WhyML that contains exceptions. The KeY [1] verification tools, also deals with exceptions since it addresses verification of Java programs annotated with Java Modeling Language (JML) [81].

The formalization of verification condition generation is not new. It has been addressed in different contexts (e.g. [58, 54]), but they fall short of both unifying the different methods and considering an SA intermediate form. Verification condition generation based on Hoare logic has also been formalized using Higher-Order Logic, see for instance [66, 98], but not in the SA setting.

In bounded model checking of software, loops are not converted to SA form, but instead eliminated by a bounded expansion before conversion (see Section 2.6). A number of transformations are then performed on the SA form, and the resulting program is easily encoded as a satisfiability problem. The transformations avoid exponential explosion, although they are not based on the observations that led to the definition of efficient predicate transformers. To the best of our knowledge, no proofs of soundness or completeness are available for bounded model checking of software techniques.

Chapter 6

A Translation of Iterating Programs into SA Form

This chapter is completely dedicated to a concrete translation of programs into SA form that will be shown to be indeed an SA translation in the sense of Definition 5.8. The translation to be presented is based on the one discussed in Section 2.4 and, as far as we know, this is the first time that a translation of iterating programs into loop-free programs is proved correct. The results from this chapter together with the results from the previous one allow us to have a completely proved verification technique based on the translation of programs into (non-iterating) SA form with assumes and asserts.

The function implementing the translation receives a program $C \in \mathbf{AComm}$, that may in particular contain annotated loops, and returns an SA (loop-free) program $C' \in \mathbf{AComm}^{\mathrm{SA}}$. Basically, the translation is responsible for renaming the variables in order to generate an SA program, but also for replacing loops with code that checks that the annotated invariants are valid and preserved during the iterations.

In what follows we will start by presenting a set of theoretical artifacts and results that will allow us to present an SA translation as described above. In particular we will resort to a small-step semantics, instead of the big-step style that was used in the previous chapters. We will show that both styles are equivalent for our source language (and thus also for the SA language). The SA translation that transforms loops into assumes and asserts is proposed and the proof that it is indeed an SA translation w.r.t. Definition 5.8 is shown. This will allow us to conclude that the translation is sound, in the sense that, *if a translated triple is valid then so is the original one*, and complete w.r.t. system Hg, in the sense that, *if the original triple is derivable in system* Hg*, then so is the translated triple.*

It should be mentioned that in general, for the sake of legibility, we use the same notation for the same concepts as used in the translation of Chapter 3; however, those concepts can be slightly different here. For instance, we will refer to versioned variables through the class named $\mathbf{Var}^{\mathrm{SA}}$, but the version will now be just a number, instead of a list of numbers.

The chapter is organized as follows: the next section presents a small-step semantics that is

shown to be equivalent to the big-step semantics of the previous chapters. The SA translation together with some initial results and auxiliary functions are proposed in Section 6.2. Sections 6.3 and 6.4 show that the translation is indeed an SA translation and Section 6.5 presents related work.

6.1 A Small-step Semantics

It was referred in the background that a small-step semantics is sometimes convenient to prove certain properties about certain programming languages. For reasons that will become clear along the chapter this is precisely one of these cases. Therefore, we use this section to propose a small-step semantics for programs with exceptions, assumes and asserts, and will prove that it is equivalent to the big-step semantic of the previous chapter.

We start by noting that we inherit the notation introduced in Section 2.1, in particular the notion of stuck configuration, derivation sequence and the respective symbols. The small-step program semantics for programs with exceptions, assumes and asserts is given by a deterministic transition relation $\Rightarrow \subseteq \mathbf{AComm} \times \Sigma \times (\Sigma_\bullet + \mathbf{AComm} \times \Sigma)$ which is defined below. A given configuration can evolve into another intermediate configuration; end its execution in a terminal state from $\Sigma_\bullet$; or else, get stuck. The last two alternatives are specially relevant here, since the current semantics allow for executions to get stuck: note that the transition relation is not defined for the configuration $\langle \mathbf{assume}\, \theta, s\rangle$ when $s \not\models \theta$. This is exactly the reason why the set of termination states does not need to include the blocked state: in a small-step setting, we can distinguish commands that block by observing that the derivation sequence evolves into a stuck configuration.

Definition 6.1 (Structural operational semantics)**.** *The* transition relation *for* **AComm** *is defined as the smallest relation* $\Rightarrow \subseteq \mathbf{AComm} \times \Sigma \times (\Sigma_\bullet + \mathbf{AComm} \times \Sigma)$ *satisfying the following set of rules:*

1. $\langle \mathbf{skip}, s\rangle \Rightarrow \boldsymbol{n}(s)$.

2. $\langle \mathbf{throw}, s\rangle \Rightarrow \boldsymbol{e}(s)$.

3. $\langle x := e, s\rangle \Rightarrow \boldsymbol{n}(s[x \mapsto [\![e]\!](s)])$.

4. *if* $s \models \theta$ *then* $\langle \mathbf{assume}\, \theta, s\rangle \Rightarrow \boldsymbol{n}(s)$.

5. *if* $s \models \theta$ *then* $\langle \mathbf{assert}\, \theta, s\rangle \Rightarrow \boldsymbol{n}(s)$.

6. *if* $s \not\models \theta$ *then* $\langle \mathbf{assert}\, \theta, s\rangle \Rightarrow \bullet$.

7. *if* $\langle C_1, s\rangle \Rightarrow \bullet$ *then* $\langle C_1\,;\, C_2, s\rangle \Rightarrow \bullet$.

8. *if* $\langle C_1, s\rangle \Rightarrow \boldsymbol{e}(s')$ *then* $\langle C_1\,;\, C_2, s\rangle \Rightarrow \boldsymbol{e}(s')$.

9. *if* $\langle C_1, s\rangle \Rightarrow \boldsymbol{n}(s')$ *then* $\langle C_1\,;\, C_2, s\rangle \Rightarrow \langle C_2, s'\rangle$.

10. if $\langle C_1, s\rangle \Rightarrow \langle C_1', s'\rangle$*, then* $\langle C_1\,;\,C_2, s\rangle \Rightarrow \langle C_1'\,;\,C_2, s'\rangle$.

11. if $\langle C_1, s\rangle \Rightarrow \bullet$ *then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \bullet$.

12. if $\langle C_1, s\rangle \Rightarrow \boldsymbol{e}(s')$ *then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \langle C_2, s'\rangle$.

13. if $\langle C_1, s\rangle \Rightarrow \boldsymbol{n}(s')$ *then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \boldsymbol{n}(s')$.

14. if $\langle C_1, s\rangle \Rightarrow \langle C_1', s'\rangle$*, then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \langle \mathbf{try}\, C_1'\, \mathbf{catch}\, C_2\, \mathbf{hc}, s'\rangle$.

15. if $s \models b$*, then* $\langle \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s\rangle \Rightarrow \langle C_1, s\rangle$.

16. if $s \not\models b$*, then* $\langle \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s\rangle \Rightarrow \langle C_2, s\rangle$.

17. $\langle \mathbf{while}\ b\ \mathbf{do}\ C_1\ \mathbf{od}, s\rangle \Rightarrow \langle \mathbf{if}\ b\ \mathbf{then}\ \{C_1\,;\,\mathbf{while}\ b\ \mathbf{do}\ C_1\ \mathbf{od}\}\ \mathbf{else\ skip\ fi}, s\rangle$.

For the sake of proving the correctness of our translation w.r.t. the framework from the previous section, the equivalence of both semantics should be established. For that, we start by introducing some lemmas about the execution of the sequence and try-catch commands in the small-step semantics. The following lemma observes that the second command (either from the sequence, or try-catch structured commands) does not influence the execution of the first. In particular for the sequence command, the second command is only executed if the first terminates normally; analogously, for the try-catch command, the second command is only executed if the first terminates exceptionally.

Lemma 6.1. *Let* $C_1, C_2 \in \mathbf{AComm}$ *and* $s, s' \in \Sigma$*. Then the following holds:*

1. If $\langle C_1, s\rangle \not\Rightarrow^n$ *, then* $\langle C_1\,;\,C_2, s\rangle \not\Rightarrow^n$.

2. If $\langle C_1, s\rangle \Rightarrow^n \bullet$*, then* $\langle C_1\,;\,C_2, s\rangle \Rightarrow^n \bullet$.

3. If $\langle C_1, s\rangle \Rightarrow^n \boldsymbol{e}(s')$*, then* $\langle C_1\,;\,C_2, s\rangle \Rightarrow^n \boldsymbol{e}(s')$.

4. If $\langle C_1, s\rangle \Rightarrow^n \boldsymbol{n}(s')$*, then* $\langle C_1\,;\,C_2, s\rangle \Rightarrow^n \langle C_2, s'\rangle$.

5. If $\langle C_1, s\rangle \not\Rightarrow^n$ *, then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \not\Rightarrow^n$.

6. If $\langle C_1, s\rangle \Rightarrow^n \bullet$*, then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow^n \bullet$

7. If $\langle C_1, s\rangle \Rightarrow^n \boldsymbol{n}(s')$*, then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \boldsymbol{n}(s')$.

8. If $\langle C_1, s\rangle \Rightarrow^n \boldsymbol{e}(s')$*, then* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow^n \langle C_2, s'\rangle$.

Proof. All proofs are by strong induction on the length k of the derivation sequence. Below the proofs for 1. and 4. are shown. The other proofs are analogous.

1. Base case: Assume $\langle C_1, s\rangle \not\Rightarrow^0$. So $\langle C_1, s\rangle \not\Rightarrow$ and thus, by analysis of the Definition 6.1, $\langle C_1\,;\,C_2, s\rangle \not\Rightarrow$. Therefore $\langle C_1\,;\,C_2, s\rangle \not\Rightarrow^0$.

Induction step: Assume that if $\langle C_1, s\rangle \not\Rightarrow^k$, then $\langle C_1\,;\,C_2, s\rangle \not\Rightarrow^k$, for all $k \leq k_0$. Assume also that $\langle C_1, s\rangle \not\Rightarrow^{k_0+1}$. We want to prove that $\langle C_1\,;\,C_2, s\rangle \not\Rightarrow^{k_0+1}$. Since $\langle C_1, s\rangle \not\Rightarrow^{k_0+1}$, it

must be the case that $\langle C_1, s\rangle \Rightarrow \langle C_1', s'\rangle \not\Rightarrow^{k_0}$, for some C_1' and s'. By IH and Definition 6.1, we have that $\langle C_1\,;\,C_2, s\rangle \Rightarrow \langle C_1', s'\rangle \not\Rightarrow^{k_0}$. Hence $\langle C_1\,;\,C_2, s\rangle \not\Rightarrow^{k_0+1}$.

4. Base case: Trivial, since there is no derivation sequence of length 0 from $\langle C_1, s\rangle$ to $\mathbf{n}(s')$.

Induction step: For all $k \leq k_0$, assume that if $\langle C_1, s\rangle \Rightarrow^k \mathbf{n}(s')$, for some $s' \in \Sigma$, then $\langle C_1\,;\,C_2, s\rangle \Rightarrow^k \langle C_2, s'\rangle$. Assume also that $\langle C_1, s\rangle \Rightarrow^{k_0+1} \mathbf{n}(s_1)$ for some $s_1 \in \Sigma$. Then it must be the case that $\langle C_1, s\rangle \Rightarrow \delta \Rightarrow^{k_0} \mathbf{n}(s_1)$, for some configuration δ. By analysis of the derivation sequence starting in $\langle C_1\,;\,C_2, s\rangle$ we have two cases:

- the first rule is $\langle C_1\,;\,C_2, s\rangle \Rightarrow \langle C_2, s_2\rangle$ because $\langle C_1, s\rangle \Rightarrow \mathbf{n}(s_2)$ for some $s_2 \in \Sigma$. Since the semantics is deterministic we are done because $s_2 = s_1$ and $k_0 = 0$.
- the first rule is $\langle C_1\,;\,C_2, s\rangle \Rightarrow \langle C_1'\,;\,C_2, s_2\rangle$ because $\langle C_1, s\rangle \Rightarrow \langle C_1', s_2\rangle$ for some $s_2 \in \Sigma$. Then it must be the case that $\langle C_1', s_2\rangle \Rightarrow^{k_0} \mathbf{n}(s_1)$, and by IH we have that $\langle C_1'\,;\,C_2, s\rangle \Rightarrow^{k_0} \langle C_2, s_1\rangle$. Therefore $\langle C_1\,;\,C_2, s\rangle \Rightarrow \langle C_1'\,;\,C_2, s_2\rangle \Rightarrow^{k_0} \langle C_2, s_1\rangle$. □

Whenever there exists a derivation sequence for a sequence or try-catch command, that derivation can possibly be broken as stated by the following lemma. For instance, if a sequence command gets stuck (resp. terminates in error state, or terminates in an exceptional state) then either the first command got stuck (resp. terminated in error state, or terminated in exceptional state), and the second never executed, or else the first terminated normally in a finite number of steps and the second got stuck (resp. terminated in error state, or in exceptional state). On the other hand, if a sequence command terminates normally, then it must be the case that both sub-commands terminated normally. The same applies for the try-catch command in case of stuck configuration and error termination. For normal termination and exceptional termination the interpretation is analogous.

Lemma 6.2. *Let $C_1, C_2 \in$ **AComm** and $s \in \Sigma$. Then the following holds:*

1. *If $\langle C_1\,;\,C_2, s\rangle \not\Rightarrow^n$, then $\langle C_1, s\rangle \not\Rightarrow^n$, or there exist $n_1, n_2 \in \mathbb{N}$ and $s' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{n_1} \boldsymbol{n}(s')$, $\langle C_2, s'\rangle \not\Rightarrow^{n_2}$ and $n = n_1 + n_2$.*
2. *If $\langle C_1\,;\,C_2, s\rangle \Rightarrow^n \bullet$, then $\langle C_1, s\rangle \Rightarrow^n \bullet$, or there exist $n_1, n_2 \in \mathbb{N}$ and $s' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{n_1} \boldsymbol{n}(s')$, $\langle C_2, s'\rangle \Rightarrow^{n_2} \bullet$ and $n = n_1 + n_2$.*
3. *If $\langle C_1\,;\,C_2, s\rangle \Rightarrow^n \boldsymbol{e}(s')$ for some $s' \in \Sigma$, then $\langle C_1, s\rangle \Rightarrow^n \boldsymbol{e}(s')$, or there exist $n_1, n_2 \in \mathbb{N}$ and $s'' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{n_1} \boldsymbol{n}(s'')$, $\langle C_2, s''\rangle \Rightarrow^{n_2} \boldsymbol{e}(s')$ and $n = n_1 + n_2$.*
4. *If $\langle C_1\,;\,C_2, s\rangle \Rightarrow^n \boldsymbol{n}(s')$ for some $s' \in \Sigma$, then there exist $n_1, n_2 \in \mathbb{N}$ and $s'' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{n_1} \boldsymbol{n}(s'')$, $\langle C_2, s''\rangle \Rightarrow^{n_2} \boldsymbol{n}(s')$ and $n = n_1 + n_2$.*
5. *If $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \not\Rightarrow^n$, then $\langle C_1, s\rangle \not\Rightarrow^n$, or there exist $n_1, n_2 \in \mathbb{N}$ and $s' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{n_1} \boldsymbol{e}(s')$, $\langle C_2, s'\rangle \not\Rightarrow^{n_2}$ and $n = n_1 + n_2$.*
6. *If $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow^n \bullet$, then $\langle C_1, s\rangle \Rightarrow^n \bullet$, or there exist $n_1, n_2 \in \mathbb{N}$ and $s' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{n_1} \boldsymbol{e}(s')$, $\langle C_2, s'\rangle \Rightarrow^{n_2} \bullet$ and $n = n_1 + n_2$.*

7. *If* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow^n \mathbf{n}(s')$ *for some* $s' \in \Sigma$*, then* $\langle C_1, s\rangle \Rightarrow^n \mathbf{n}(s')$*, or there exist* $n_1, n_2 \in \mathbb{N}$ *and* $s'' \in \Sigma$ *such that* $\langle C_1, s\rangle \Rightarrow^{n_1} \mathbf{e}(s'')$*,* $\langle C_2, s''\rangle \Rightarrow^{n_2} \mathbf{n}(s')$ *and* $n = n_1 + n_2$*.*

8. *If* $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow^n \mathbf{e}(s')$ *for some* $s' \in \Sigma$*, then there exist* $n_1, n_2 \in \mathbb{N}$ *and* $s'' \in \Sigma$ *such that* $\langle C_1, s\rangle \Rightarrow^{n_1} \mathbf{e}(s'')$*,* $\langle C_2, s''\rangle \Rightarrow^{n_2} \mathbf{e}(s')$ *and* $n = n_1 + n_2$*.*

Proof. Again, all the proofs are by strong induction on the length k of the derivation sequence. The proof for 7. is shown below and the other cases are analogous.

7. Base case: This case is trivial, since there is no derivation sequence of length 0 from $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle$ to $\mathbf{n}(s)$.

Induction step: Assume that the property holds for all $k \leq k_0$. Assume also that $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow^{k_0+1} \mathbf{n}(s_1)$. Then it must be the case that $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \delta \Rightarrow^{k_0} \mathbf{n}(s_1)$ for some δ. We proceed by analysis of the derivation $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \delta$. If the applied rule is 12. then we have $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \langle C_2, s_2\rangle$ because $\langle C_1, s\rangle \Rightarrow \mathbf{e}(s_2)$. In this case we are done because it must be the case that $\langle C_2, s_2\rangle \Rightarrow^{k_0} \mathbf{n}(s_1)$. If the applied rule is 14., then $\langle \mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}, s\rangle \Rightarrow \langle \mathbf{try}\, C_1'\, \mathbf{catch}\, C_2\, \mathbf{hc}, s_2\rangle$ because $\langle C_1, s\rangle \Rightarrow \langle C_1', s_2\rangle$. It must then be the case that $\langle \mathbf{try}\, C_1'\, \mathbf{catch}\, C_2\, \mathbf{hc}, s_2\rangle \Rightarrow^{k_0} \mathbf{n}(s_1)$, therefore by IH we have one of the following:

- $\langle C_1', s_2\rangle \Rightarrow^{k_0} \mathbf{n}(s_1)$. In this case we are done because $\langle C_1, s\rangle \Rightarrow \langle C_1', s_2\rangle \Rightarrow^{k_0} \mathbf{n}(s_1)$. Hence, $\langle C_1, s\rangle \Rightarrow^{k_0+1} \mathbf{n}(s_1)$.

- there exist some $k_1, k_2 \in \mathbb{N}$ and $s_3 \in \Sigma$ such that $\langle C_1', s_2\rangle \Rightarrow^{k_1} \mathbf{e}(s_3)$, $\langle C_2, s_3\rangle \Rightarrow^{k_2} \mathbf{n}(s_1)$ and $k_0 = k_1 + k_2$. Then $\langle C_1, s\rangle \Rightarrow \langle C_1', s_2\rangle \Rightarrow^{k_1} \mathbf{e}(s_3)$ and $\langle C_2, s_3\rangle \Rightarrow^{k_2} \mathbf{n}(s_1)$. Hence, $\langle C_1, s\rangle \Rightarrow^{k_1+1} \mathbf{e}(s_3)$ and $\langle C_2, s_3\rangle \Rightarrow^{k_2} \mathbf{n}(s_1)$. □

With the lemmas above, we can now establish the equivalence of both semantics. In particular if one of the semantics terminates in a state that is not blocked, the other should terminate exactly in the same state. Moreover, a derivation sequence gets into a stuck configuration in the small-step semantics, if and only if it terminates in the blocked state in the big-step semantics.

Proposition 6.3. *Let* $C \in \mathbf{AComm}$*,* $s \in \Sigma$*,* $\sigma \in \Sigma_\bullet^\blacksquare$*, and* $\sigma' \in \Sigma_\bullet$*. Then the following holds:*

1. *If* $\langle C, s\rangle \rightsquigarrow \sigma$ *and* $\sigma \neq \blacksquare$ *then* $\langle C, s\rangle \Rightarrow^* \sigma$*.*

2. *If* $\langle C, s\rangle \Rightarrow^k \sigma'$ *then* $\langle C, s\rangle \rightsquigarrow \sigma'$*.*

3. $\langle C, s\rangle \rightsquigarrow \blacksquare$ *if and only if* $\langle C, s\rangle \not\Rightarrow^*$ *.*

Proof. All proofs follow by induction. The proof of 3. uses 1. and 2.

1. By induction on the derivation of $\langle C, s\rangle \rightsquigarrow \sigma$. The proof is straightforward using Lemma 6.1.

2. By induction on the pair $(k, \sharp C)$ using lexicographic order, where $\sharp C$ denotes the number of constructs in C.

Base case: $(0, \sharp C')$. Trivial for any C', since there is no derivation from $\langle C, s\rangle$ to state σ in 0 steps.

Induction step: Assume that if $\langle C', s'\rangle \Rightarrow^k \sigma$ then $\langle C', s'\rangle \rightsquigarrow \sigma$, for all $(k, \sharp C') < (k_0, \sharp C)$. We want to prove that if $\langle C, s\rangle \Rightarrow^{k_0} \sigma$ then $\langle C, s\rangle \rightsquigarrow \sigma$. If $k_0 = 0$ then the proof is trivial for the same reason as the base case. If $k_0 > 0$, then assume that $\langle C, s\rangle \Rightarrow^{k_0} \sigma$. Then it must be the case that $\langle C, s\rangle \Rightarrow \delta \Rightarrow^{k_0-1} \sigma$. We proceed by analysis of the derivation $\langle C, s\rangle \Rightarrow \delta$ and show here the case where the rule 17 is the last one applied. The remaining cases are similar to this one or just trivial.

From $C = \textbf{while}\ b\ \textbf{do}\ C_1\ \textbf{od}$ we have that $\langle \textbf{while}\ b\ \textbf{do}\ C_1\ \textbf{od}, s\rangle \Rightarrow \langle \textbf{if}\ b\ \textbf{then}\ C_1 ; C$ $\textbf{else skip fi}, s\rangle$ and we have the following cases:

- if $s \models b$ then $\langle \textbf{if}\ b\ \textbf{then}\ C_1 ; C\ \textbf{else skip fi}, s\rangle \Rightarrow \langle C_1 ; C, s\rangle$. It must then be the case that $\langle C_1 ; C, s\rangle \Rightarrow^{k_0-2} \sigma$. Let us now inspect the shape of σ:
  - if $\sigma = \bullet$, then by Lemma 6.2 we have one of the following cases:
    - $\langle C_1, s\rangle \Rightarrow^{k_0-2} \bullet$. In this case, by IH we have $\langle C_1, s\rangle \rightsquigarrow \bullet$, and by Definition 5.1-19, $\langle \textbf{while}\ b\ \textbf{do}\ C_1\ \textbf{od}, s\rangle \rightsquigarrow \bullet$.
    - there are some $k_1, k_2 \in \mathbb{N}$ and $s' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{k_1} \mathbf{n}(s')$, $\langle C, s'\rangle \Rightarrow^{k_2}$ $\bullet$, and $k_0 - 2 = k_1 + k_2$.
  - if $\sigma = \mathbf{e}(s')$ for some $s' \in \Sigma$, then the proof is similar to the previous.
  - if $\sigma = \mathbf{n}(s')$, for some $s' \in \Sigma$, then by Lemma 6.2, there exists some $k_1, k_2 \in \mathbb{N}$ and $s'' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{k_1} \mathbf{n}(s'')$, $\langle C, s''\rangle \Rightarrow^{k_2} \mathbf{n}(s')$, and $k_0-2 = k_1+k_2$. By IH $\langle C_1, s\rangle \rightsquigarrow \mathbf{n}(s'')$ and $\langle C, s''\rangle \rightsquigarrow \mathbf{n}(s')$ and by Definition 5.1-21 $\langle C_1 ; C, s\rangle \rightsquigarrow \mathbf{n}(s')$.
- if $s \not\models b$ then the proof is trivial using Definition 5.1-22.

3. The proof that if $\langle C, s\rangle \rightsquigarrow \blacksquare$ then $\langle C, s\rangle \not\Rightarrow^*$ follows by induction on the shape of the derivation $\langle C, s\rangle \rightsquigarrow \blacksquare$. All cases are straightforward using Lemma 6.1 and 1.

   We now demonstrate that if $\langle C, s\rangle \not\Rightarrow^*$ then $\langle C, s\rangle \rightsquigarrow \blacksquare$. The proof follows by induction on the pair $(k, \sharp C)$, where $\sharp C$ denotes the number of constructs in C.

   Base case: $(0, 1)$. The proof is trivial when C is **skip**, **throw**, $x := e$, or $\textbf{assert}\,\theta$. When C is $\textbf{assume}\,\theta$ we must have that $s \not\models \theta$, and therefore it holds that $\langle \textbf{assume}\,\theta, s\rangle \rightsquigarrow \blacksquare$.

   Induction step: Assume that if $\langle C', s'\rangle \not\Rightarrow^k$ then $\langle C', s'\rangle \rightsquigarrow \blacksquare$, for all $(k, \sharp C') < (k_0, \sharp C)$. We want to prove that if $\langle C, s\rangle \not\Rightarrow^{k_0}$ then $\langle C, s\rangle \rightsquigarrow \blacksquare$. If $k_0 = 0$ then $\langle C, s\rangle \not\Rightarrow^0$ only if C is of the form $\textbf{assume}\,\theta ; C'$, for some $C' \in \textbf{AComm}$. In this case $s \not\models \theta$ and thus $\langle C, s\rangle \rightsquigarrow \blacksquare$. If $k_0 > 0$, then it must be the case that $\langle C, s\rangle \Rightarrow \delta \not\Rightarrow^{k_0-1}$. We proceed by analysis of the derivation $\langle C, s\rangle \Rightarrow \delta$.

   We show here the proof for the case that C is $C_1 ; C_2$. From Lemma 6.2, we have one of the following:

   - $\langle C_1, s\rangle \not\Rightarrow^{k_0}$. Since $(k_0, \sharp C_1) < (k_0, \sharp(C_1 ; C_2))$, by IH we have that $\langle C_1, s\rangle \rightsquigarrow \blacksquare$.

- there are some $k_1, k_2 \in \mathbb{N}$ and $s' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{k_1} \mathbf{n}(s')$, $\langle C_2, s'\rangle \not\Rightarrow^{k_2}$ and $k_0 = k_1 + k_2$. From 2. we have that $\langle C_1, s\rangle \rightsquigarrow \mathbf{n}(s')$ and since $(k_2, \sharp C_2) < (k_0, \sharp(C_1 \,;\, C_2))$, by IH we have that $\langle C_2, s'\rangle \rightsquigarrow \blacksquare$. Therefore, using Definition 5.1 we have that $\langle C_1 \,;\, C_2, s\rangle \rightsquigarrow \blacksquare$.

The other cases are similar to the previous and also analogous to the proof of 2. □

It should be noted that up to this point, whenever we were referring to the validity of a Hoare triple, written $\models \{\phi\}\, C\, \{\psi, \epsilon\}$, it was implicit that the validity was w.r.t. the big-step semantic style. Nonetheless, we can now equivalently define the validity of a Hoare triple w.r.t. a small-step semantic style as follows:

Definition 6.2. *The Hoare triple $\{\phi\}\, C\, \{\psi, \epsilon\}$ is said to be* valid*, denoted $\models \{\phi\}\, C\, \{\psi, \epsilon\}$, whenever for every $s \in \Sigma$ and $\sigma \in \Sigma_\bullet$, if $s \models \phi$ and $\langle C, s\rangle \Rightarrow^* \sigma$ then:*

1. *$\sigma \neq \bullet$.*
2. *if $\sigma = \boldsymbol{n}(s')$ for some $s' \in \Sigma$, then $s' \models \psi$.*
3. *if $\sigma = \boldsymbol{e}(s')$ for some $s' \in \Sigma$, then $s' \models \epsilon$.*

From now on, and since we have two semantic styles, we should indicate the definition we are referring to (Definition 6.2 or 5.2). Nonetheless, and since both semantics are equivalent as exposed by the previous lemma, we will continue writing $\models \{\phi\}\, C\, \{\psi, \epsilon\}$ without identifying the semantics, unless this is necessary.

Corollary 6.4. *A Hoare triple is valid w.r.t. the small-step semantics, if and only if, it is valid w.r.t. the big-step semantics.*

Proof. Follows, directly from Definitions 5.2 and 6.2, and Proposition 6.3. □

Having said this, in the remaining of this chapter we will use the small-step semantics, and so we should see validity as given by Definition 6.2.

6.2 SA Translation

The translation will transform programs containing annotated loops into single-assignment, loop-free, programs. Basically, it will transform **AComm** into $\mathbf{AComm}^{\mathrm{SA}}$. The variables in the SA program obtained by our translation will have an identifier and a version. The set of all SA variables will be given by $\mathbf{Var}^{\mathrm{SA}} = \mathbf{Var} \times \mathbb{N}$, and x_i will denote the SA variable $(x, i) \in \mathbf{Var}^{\mathrm{SA}}$. In particular, in the translation we will propose ahead, the variable identifier will be inherited from the original program and the version will be generated along the translation. The set of states over SA variables will be denoted by $\Sigma^{\mathrm{SA}} = \mathbf{Var}^{\mathrm{SA}} \to D$, where D is the domain of interpretation, and the sets of program expressions, boolean expressions, and assertions over SA variables will be respectively $\mathbf{Exp}^{\mathrm{SA}}$, $\mathbf{Exp}^{\mathrm{SA}}_{\mathbf{bool}}$, and $\mathbf{Assert}^{\mathrm{SA}}$.

Once more we will overload some function identifiers, but again, it will always be clear from the context which specific function to apply. Version functions with the form $\mathcal{V} : \mathbf{Var} \to \mathbb{N}$ will assign a version to variables and the following definition will introduce functions to rename variables, expressions, and assertions according to a given version function. Moreover, the partial function $\mathcal{V}(s) : \mathbf{Var} \hookrightarrow D$, will be defined as $[\widehat{\mathcal{V}}(x) \mapsto s(x) \mid x \in \mathbf{Var}]$, for $s \in \Sigma$.

Definition 6.3. *The function* $\widehat{\ } : (\mathbf{Var} \to \mathbb{N}) \times \mathbf{Var} \to \mathbf{Var}^{\text{SA}}$ *is defined as* $\widehat{\mathcal{V}}(x) = x_{\mathcal{V}(x)}$*, and is lifted to* $\mathbf{Exp}$*,* $\mathbf{Exp^{bool}}$ *and* $\mathbf{Assert}$ *as follows (the definition for* $\mathbf{Exp^{bool}}$ *is omitted because it is analogous to* $\mathbf{Exp}$*):*

$$\begin{aligned} \widehat{\ } &: (\mathbf{Var} \to \mathbb{N}) \times \mathbf{Exp} \to \mathbf{Exp}^{\text{SA}} \\ \widehat{\mathcal{V}}(e) &= e[\widehat{\mathcal{V}}(x_1)/x_1, \ldots, \widehat{\mathcal{V}}(x_n)/x_n], \\ &\quad \textit{for all } x_1, \ldots, x_n \in \mathsf{FV}(e) \end{aligned}$$

$$\begin{aligned} \widehat{\ } &: (\mathbf{Var} \to \mathbb{N})_\perp \times \mathbf{Assert} \to \mathbf{Assert}^{\text{SA}} \\ \widehat{\perp}(\phi) &= \perp \\ \widehat{\mathcal{V}}(\phi) &= \phi[\widehat{\mathcal{V}}(x_1)/x_1, \ldots, \widehat{\mathcal{V}}(x_n)/x_n], \\ &\quad \textit{for all } x_1, \ldots, x_n \in \mathsf{FV}(\phi) \end{aligned}$$

With the concepts introduced, it is already possible to relate states from Σ and Σ^{SA}. In the following lemma, the state $\mathcal{V}(s)$ only assigns SA variables whose versions are given by $\mathcal{V}$, therefore an additional state function must exist that assigns the other variables that do not belong to the domain of $\mathcal{V}(s)$.

Lemma 6.5. *Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}$*,* $s \in \Sigma$ *and* $s' \in \Sigma^{\text{SA}}$*. If for all* $x \in \mathbf{Var}$*,* $s(x) = s'(\widehat{\mathcal{V}}(x))$*, then* $s' = s'_0 \oplus \mathcal{V}(s)$ *for some* $s'_0 \in \Sigma^{\text{SA}}$*.*

Proof. Follows directly from the definition of $\mathcal{V}(s)$ and $\oplus$. □

Obviously, evaluating an expression (resp. assertion) that was renamed using a version function $\mathcal{V}$ in a state 'dominated' by $\mathcal{V}(s)$ results in the same value as evaluating the original expression (resp. assertion) in the state s.

Lemma 6.6. *Let* $e \in \mathbf{Exp}$*,* $b \in \mathbf{Exp^{bool}}$*,* $\phi \in \mathbf{Assert}$*,* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}$*,* $s \in \Sigma$ *and* $s' \in \Sigma^{\text{SA}}$*.*

1. $[\![\widehat{\mathcal{V}}(e)]\!](s' \oplus \mathcal{V}(s)) = [\![e]\!](s)$.

2. $[\![\widehat{\mathcal{V}}(b)]\!](s' \oplus \mathcal{V}(s)) = [\![b]\!](s)$.

3. $[\![\widehat{\mathcal{V}}(\phi)]\!](s' \oplus \mathcal{V}(s)) = [\![\phi]\!](s)$.

Proof. By induction on the structure of e (resp. b or ϕ). □

From this lemma it follows directly that a certain assertion is valid, if and only if its translation into SA is valid.

Lemma 6.7. *Let* $\phi \in \mathbf{Assert}$ *and* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}$*. Then* $\models \phi$ *iff* $\models \widehat{\mathcal{V}}(\phi)$*.*

Proof. Follows directly from Lemma 6.6. □

Recall from Lemma 2.11 that the operator $\oplus$ is associative. Even though it is not commutative for arbitrary states, for some partial states, the operands can be swapped as described in Lemma 2.12. Nonetheless, the restrictions that allow the lemma to be applied are too strong. Aiming at weakening these restrictions we start by defining below the notion of compatible states w.r.t. version functions. This notion will allow us to relax the cases in which the operands can be swapped.

Definition 6.4. *Let $s, s' \in \Sigma$, and $\mathcal{V}, \mathcal{V}' \in \mathbf{Var} \to \mathbb{N}$. The states s and s' are said to be* compatible *w.r.t. $\mathcal{V}$ and $\mathcal{V}'$, if for every $x \in \mathbf{Var}$ such that $\mathcal{V}(x) = \mathcal{V}'(x)$, it holds that $s(x) = s'(x)$.*

Compatible states can in fact be swapped without changing the meaning of the state function.

Lemma 6.8. *Let $s \in \Sigma^{\text{SA}}$, $s, s' \in \Sigma$ and $\mathcal{V}, \mathcal{V}' \in \mathbf{Var} \to \mathbb{N}$. If s, s' are compatible w.r.t. $\mathcal{V}, \mathcal{V}'$, then $s \oplus \mathcal{V}(s) \oplus \mathcal{V}'(s') = s \oplus \mathcal{V}'(s') \oplus \mathcal{V}(s)$.*

Proof. Straightforward by expanding $s \oplus \mathcal{V}(s_1) \oplus \mathcal{V}'(s_2)$ and $s \oplus \mathcal{V}'(s_2) \oplus \mathcal{V}(s_1)$ using Definition 2.18 and Definition 6.4. $\square$

Before presenting the single-assignment translation, let us introduce some auxiliary functions to operate over version functions and some lemmas over those functions. We recall that **Rnm** was introduced in Definition 2.19 and represents a sequence of assignments in which all the variables are distinct. The function mrg will be used to synchronize version functions and the function sup to join version functions.

Definition 6.5. *The functions mrg and sup are defined as follows:*

$$\begin{aligned}
&\mathsf{mrg} : (\mathbf{Var} \to \mathbb{N})^2_\perp \to \mathbf{Rnm} \\
&\mathsf{mrg}(\perp, \mathcal{V}) = [] \\
&\mathsf{mrg}(\mathcal{V}, \perp) = [] \\
&\mathsf{mrg}(\mathcal{V}, \mathcal{V}') = [\widehat{\mathcal{V}'}(x) := \widehat{\mathcal{V}}(x) \mid x \in \mathbf{Var} \wedge \mathcal{V}(x) < \mathcal{V}'(x)]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{sup} : (\mathbf{Var} \to \mathbb{N})^2_\perp \to (\mathbf{Var} \to \mathbb{N}) \\
&\mathsf{sup}(\perp, \mathcal{V}) = \mathcal{V} \\
&\mathsf{sup}(\mathcal{V}, \perp) = \mathcal{V} \\
&\mathsf{sup}(\mathcal{V}, \mathcal{V}')(x) = \begin{cases} \mathcal{V}(x) & \text{if } \mathcal{V}(x) > \mathcal{V}'(x) \\ \mathcal{V}'(x) & \text{otherwise} \end{cases}
\end{aligned}$$

First of all, note that the functions receive a pair of elements and each of those elements is a version function or $\perp$. As usual $(\mathbf{Var} \to \mathbb{N})_\perp$ denotes $(\mathbf{Var} \to \mathbb{N}) + \perp$. The $\perp$ will be used when no version function is provided by the translation.

Some basic results concerning these functions will now be given. As observed by the following lemma, the order in which the elements are passed to the function sup is irrelevant.

Lemma 6.9. *Let $\mathcal{V}, \mathcal{V}' \in (\mathbf{Var} \to \mathbb{N})_\perp$. Then $\mathsf{sup}(\mathcal{V}, \mathcal{V}') = \mathsf{sup}(\mathcal{V}', \mathcal{V})$.*

Proof. Follows directly from the definition. $\square$

The evaluation of renamings obtained by the function mrg can be predicted a priori and are captured by the next lemma.

Lemma 6.10. *Let* $\mathcal{V}, \mathcal{V}' \in \mathbf{Var} \to \mathbb{N}$*,* $s' \in \Sigma^{\mathrm{SA}}$*, and* $s \in \Sigma$*. Then the following hold:*

1. $\langle \mathsf{mrg}(\mathcal{V}, \mathcal{V}'), s' \oplus \mathcal{V}(s) \rangle \Rightarrow^* \boldsymbol{n}(s' \oplus \mathcal{V}(s) \oplus \mathsf{sup}(\mathcal{V}, \mathcal{V}')(s))$.
2. $\langle \mathsf{mrg}(\mathcal{V}, \mathcal{V}')\,;\,\mathbf{throw}, s' \oplus \mathcal{V}(s) \rangle \Rightarrow^* \boldsymbol{e}(s' \oplus \mathcal{V}(s) \oplus \mathsf{sup}(\mathcal{V}, \mathcal{V}')(s))$.

Proof. Follows from the definition of mrg, sup, and $\Rightarrow$, noting that $\mathsf{mrg}(\mathcal{V}, \mathcal{V}')$ only assigns variables $x_i \in \mathbf{Var}^{\mathrm{SA}}$ if $\mathcal{V}'(x) = i$ and $i < \mathcal{V}(x)$. □

In particular the evaluation of a renaming never gets into a stuck configuration, and never terminates in exception nor error. In the second item of the previous lemma an exception is raised for the execution to terminate in an exceptional state.

Some triples containing renamings can be assumed to be derivable in system Hg if they satisfy some restrictions as indicated by the following lemma.

Lemma 6.11. *Let* $\mathcal{V}, \mathcal{V}' \in \mathbf{Var} \to \mathbb{N}$ *and* $\psi, \epsilon \in \mathbf{Assert}$*. The following derivations hold:*

1. $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\psi)\}\, \mathsf{mrg}(\mathcal{V}, \mathcal{V}')\, \{\widehat{\mathsf{sup}(\mathcal{V}, \mathcal{V}')}(\psi), \epsilon\}$.
2. $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\epsilon)\}\, \mathsf{mrg}(\mathcal{V}, \mathcal{V}')\,;\,\mathbf{throw}\, \{\psi, \widehat{\mathsf{sup}(\mathcal{V}, \mathcal{V}')}(\epsilon)\}$.

Proof. Similar to the proof of Lemma 3.20. □

The function that translates a program into SA can now be proposed. It uses mrg and sup from Definition 6.5 as auxiliary functions to synchronize and to find the appropriate version of each variable when different branches of the program meet. The function receives a version function (any will do) and an annotated program and returns a triple, where the first element corresponds to the versions of the variables when the program terminates normally, the second to the versions of the variables when the program terminates exceptionally, and the final element is the translated program.

Definition 6.6 (Concrete SA translation)**.** *Let* $C \in \mathbf{AComm}$*,* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}$*. The SA translation function* $\mathfrak{T}$ *is defined in Figure 6.1. If* $(\mathcal{V}', \mathcal{V}'_e, C') = \mathfrak{T}(\mathcal{V}, C)$*, then* C' *is the translated SA program,* $\mathcal{V}$ *gives the* initial versions *of the variables and* $\mathcal{V}'$ *(resp.* $\mathcal{V}'_e$*) the* final versions *of the variables when the program terminates normally (resp. exceptionally). A triple* $\{\phi\}\, C\, \{\psi, \epsilon\}$ *can be translated into SA as* $\{\widehat{\mathcal{V}}(\phi)\}\, C'\, \{\widehat{\mathcal{V}'}(\psi), \widehat{\mathcal{V}'_e}(\epsilon)\}$

There are some aspects that deserve a deeper discussion. The type of the translation function indicates that the elements from the returned triple, corresponding to the version functions for normal and exceptional termination, can in fact be the $\perp$ element. The reason for this is that a program may actually never terminate normally (resp. exceptionally), and in these cases the version of the variables is not relevant. Note also from the type that the function is only partially defined (indicated by $\hookrightarrow$). Basically the function is not defined for programs that contain *syntactic dead code*, that is, code that can be syntatically seen as never executing. For instance, in the program $\mathbf{try}\,\mathbf{skip}\,\mathbf{catch}\,x := 2\,\mathbf{hc}$, the assignment $x := 2$ will never be

$$\mathfrak{T} : (\mathbf{Var} \to \mathbb{N}) \times \mathbf{AComm} \hookrightarrow (\mathbf{Var} \to \mathbb{N})_\bot \times (\mathbf{Var} \to \mathbb{N})_\bot \times \mathbf{AComm}^{\mathrm{SA}}$$

$$\begin{array}{rcl}
\mathfrak{T}(\mathcal{V}, \mathbf{skip}) & = & (\,\mathcal{V}, \bot, \mathbf{skip}\,) \\
\mathfrak{T}(\mathcal{V}, \mathbf{throw}) & = & (\,\bot, \mathcal{V}, \mathbf{throw}\,) \\
\mathfrak{T}(\mathcal{V}, \mathbf{assume}\,\theta) & = & (\,\mathcal{V}, \bot, \mathbf{assume}\,\widehat{\mathcal{V}}(\theta)\,) \\
\mathfrak{T}(\mathcal{V}, \mathbf{assert}\,\theta) & = & (\,\mathcal{V}, \bot, \mathbf{assert}\,\widehat{\mathcal{V}}(\theta)\,) \\
\mathfrak{T}(\mathcal{V}, x := e) & = & (\,\mathcal{V}', \bot, \widehat{\mathcal{V}'}(x) := \widehat{\mathcal{V}}(e)\,), \quad \text{where } \mathcal{V}' = \mathcal{V}[x \mapsto \mathcal{V}(x)+1] \\
\mathfrak{T}(\mathcal{V}, C_1\,;\,C_2) & = & (\,\mathcal{V}'', \mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e), \\
 & & \mathbf{try}\,C'_1\,\mathbf{catch}\,\mathsf{mrg}(\mathcal{V}'_e, \mathcal{V}''_e)\,;\,\mathbf{throw}\,\mathbf{hc}\,; \\
 & & \mathbf{try}\,C'_2\,\mathbf{catch}\,\mathsf{mrg}(\mathcal{V}''_e, \mathcal{V}'_e)\,;\,\mathbf{throw}\,\mathbf{hc}\,), \\
 & & \quad \text{where} \quad (\mathcal{V}', \mathcal{V}'_e, C'_1) \;=\; \mathfrak{T}(\mathcal{V}, C_1),\; \mathcal{V}' \neq \bot, \\
 & & \quad\quad \text{and} \quad (\mathcal{V}'', \mathcal{V}''_e, C'_2) \;=\; \mathfrak{T}(\mathcal{V}', C_2) \\
\mathfrak{T}(\mathcal{V}, \mathbf{try}\,C_1\,\mathbf{catch}\,C_2\,\mathbf{hc}) & = & (\,\mathsf{sup}(\mathcal{V}', \mathcal{V}''), \mathcal{V}''_e, \\
 & & \mathbf{try}\,C'_1\,;\,\mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\,\mathbf{catch}\,C'_2\,;\,\mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\,\mathbf{hc}\,), \\
 & & \quad \text{where} \quad (\mathcal{V}', \mathcal{V}'_e, C'_1) \;=\; \mathfrak{T}(\mathcal{V}, C_1),\; \mathcal{V}'_e \neq \bot, \\
 & & \quad\quad \text{and} \quad (\mathcal{V}'', \mathcal{V}''_e, C'_2) \;=\; \mathfrak{T}(\mathcal{V}'_e, C_2) \\
\mathfrak{T}(\mathcal{V}, \mathbf{if}\,b\,\mathbf{then}\,C_1\,\mathbf{else}\,C_2\,\mathbf{fi}) & = & (\,\mathsf{sup}(\mathcal{V}', \mathcal{V}''), \mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e), \\
 & & \mathbf{if}\,\widehat{\mathcal{V}}(b)\,\mathbf{then} \\
 & & \mathbf{try}\,C'_1\,;\,\mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\,\mathbf{catch}\,\mathsf{mrg}(\mathcal{V}'_e, \mathcal{V}''_e)\,;\,\mathbf{throw}\,\mathbf{hc}\,\mathbf{else} \\
 & & \mathbf{try}\,C'_2\,;\,\mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\,\mathbf{catch}\,\mathsf{mrg}(\mathcal{V}''_e, \mathcal{V}'_e)\,;\,\mathbf{throw}\,\mathbf{hc}\,\mathbf{fi}\,), \\
 & & \quad \text{where} \quad (\mathcal{V}', \mathcal{V}'_e, C'_1) \;=\; \mathfrak{T}(\mathcal{V}, C_1), \\
 & & \quad\quad \text{and} \quad (\mathcal{V}'', \mathcal{V}''_e, C'_2) \;=\; \mathfrak{T}(\mathcal{V}, C_2) \\
\mathfrak{T}(\mathcal{V}, \mathbf{while}\,b\,\mathbf{do}\,\{\theta\}\,C\,\mathbf{od}) & = & (\,\mathcal{V}'', \mathcal{V}''_e,\; \mathbf{assert}\,\widehat{\mathcal{V}}(\theta)\,;\,\mathbf{assume}\,\widehat{\mathcal{V}'}(\theta)\,; \\
 & & \qquad\quad \mathbf{if}\,\widehat{\mathcal{V}'}(b)\,\mathbf{then}\,C'\,;\,\mathbf{assert}\,\widehat{\mathcal{V}''}(\theta)\,;\,\mathbf{assume}\,\bot \\
 & & \qquad\quad \mathbf{else}\,\mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\,\mathbf{fi}\,), \\
 & & \text{where} \qquad\qquad \mathcal{V}' \;=\; \mathcal{V}[x \mapsto \mathcal{V}(x)+1 \mid x \in \mathsf{Asgn}(C)], \\
 & & \quad \text{and} \;\; (\mathcal{V}'', \mathcal{V}''_e, C') \;=\; \mathfrak{T}(\mathcal{V}', C)
\end{array}$$

Figure 6.1: SA translation function for programs with exceptions

executed no matter what; the same happens for the program **throw** ; $x := e$. The restriction that programs do not contain this syntatic dead code can be seen in the sequence command in which it is imposed that $\mathcal{V}' \neq \perp$, and in the try-catch command where it is imposed that $\mathcal{V}'_e \neq \perp$. This should not however be seen as a limitation of the translation function because if programs contain syntatic dead code, that code can be easily and automatically removed a priori without changing the semantics of the program.

Let us analyze the application of the function for the different commands. The translation of atomic commands is fairly simple. If the command contains expressions or assertions, then those will be renamed according to the received version, and in the case it is an assignment, the version of the variable that is assigned is incremented and the new value propagated through the returned version function. The throw command is the only atomic command that can terminate exceptionally; the skip, assume, assert, and assignment command always terminate normally.

Things get more complicated for composite commands because the version of the variables may require synchronization after each step. For instance, in the sequence $C_1 ; C_2$, the sub-command C_2 is only executed when C_1 terminates normally, and it can then terminate normally or exceptionally. On the other hand if C_1 terminates exceptionally, C_2 is not even executed. So, it is clear that there is only one way for the sequence to terminate normally (C_1 and C_2 terminate normally), but there are two ways for it to terminate exceptionally (C_1 terminates exceptionally and C_2 is not even executed, or C_1 terminates normally and then C_2 terminates exceptionally). The fact that there are two alternatives for the command to terminate exceptionally imposes that after the translation, independently of the way it terminates exceptionally there is a version function that contains the current version of the variables. Obviously, the translation of C_1 must use the current version of the variables, which is represented by $\mathcal{V}$ in our function. The translation will produce a version function for the case of normal and exceptional termination (represented respectively by $\mathcal{V}'$ and $\mathcal{V}'_e$), and the translated program C'_1. If $\mathcal{V}' = \perp$ the function is simply not defined (we are in the case that syntatic dead code exists), otherwise the sub-program C_2 is translated using $\mathcal{V}'$ and the triple $(\mathcal{V}'', \mathcal{V}''_e, C'_2)$ will be obtained. It is obvious from the previous discussion that the final version of the variables in case of normal termination will be $\mathcal{V}''$. However, it may not be immediate to obtain a version function that captures the version of the variables after the sequence terminates exceptionally: they can come from $\mathcal{V}'_e$ or from $\mathcal{V}''_e$. In the case that at least one of these version functions is $\perp$ then the final version function will be the other (in case both are $\perp$ the result will also be $\perp$). Nonetheless, when both are different from $\perp$ they must be combined to produce a version function that will propagate the adequate versions of the variables. This can be done by encapsulating each part of the sequence into a try-catch, such that when one of these parts terminates exceptionally, the exception will be caught, the variables synchronized according to the other part and then the exceptions raised again. After this, following this approach it becomes easy to capture the current version of the variables, which is given by the version function $\mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e)$.

For the try-catch command we have the inverted scenario. We simply return the version function of the second component for exceptional termination, and for normal termination we

use sup to select the most recent variables' version, and also append to each component the appropriate code to synchronize variables.

For the if command, things get even more complicated and both version functions (for normal and exceptional termination), must be merged together. So, for each component a renaming sequence is appended to synchronize variables in case of normal termination, and the result is encapsulated in a try-catch to synchronize variables in case of exceptional termination. Essentially, here we put together the method used in the sequence and in the try-catch command.

The application of the function to the while command generates a sequence of commands without any loop construct that will (hopefully) capture the axiomatic semantics of the annotated while loop. The approach is similar to the one explained at the end of Section 2.4. The first assert ensures that the invariant θ is initially satisfied. The assume uses fresh versions to rename the variables such that they become isolated from what comes before. Basically the idea here is to capture an arbitrary iteration of the loop. The if command is used to check if after such an iteration the loop condition still holds or not. If it does, that is, if $\widehat{\mathcal{V}'}(b)$ holds in the current state, then the translated loop body, C', should be executed. If C' terminates normally, then the loop invariant must remain valid, which is ensured by $\mathbf{assert}\,\widehat{\mathcal{V}''}(\theta)$. After this, if the invariant is in fact preserved, the execution should block because it is not certain that the current iteration was the last, which explains the $\mathbf{assume}\,\bot$. Note also that if the execution of C' terminates exceptionally then it is not required that the invariant is preserved and the final version of the variables is in fact $\mathcal{V}''_e$. If the $\widehat{\mathcal{V}'}(b)$ condition is not satisfied, it means that the execution corresponds to the last iteration and thus the variables are synchronized w.r.t. the other branch. In this case, all versions from $\mathcal{V}'$ will be advanced to the version in $\mathcal{V}''$.

Note that we are assuming that some kind of 'smart-constructors' are being used to create sequences and try-catch commands for the synchronization commands that avoid the introduction of redundant code. This is particularly important because most part of the steps will not require synchronization, since one of the version functions, if not both, will be $\bot$. For instance, in the case of the sequence, if $\mathcal{V}'_e = \bot$ the translation as it is would generate the command $\mathbf{try}\,C'_1\,\mathbf{catch}\,\mathbf{skip}\,;\,\mathbf{throw}\,\mathbf{hc}$ for synchronization, which is exactly the same as writing simply C'_1. So, we assume that these smart-constructors will detect these situations and apply these optimizations. The following examples would be much more complicated and full of redundant code if the translation was applied blindly without smart-constructors.

Example 6.1. *Consider the* $\mathsf{FACT_T}$ *program introduced in Example 3.6 that calculates the nth factorial number. Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}$ *such that* $\mathcal{V}(x) = 0$*, for all* $x \in \mathcal{V}$*. Then* $\mathfrak{T}(\mathcal{V}, \mathsf{FACT_T}) = (\mathcal{V}', \mathcal{V}'_e, \mathsf{FACT_{SA}})$*, where* $\mathcal{V}' = \mathcal{V}[f \mapsto 3, i \mapsto 3, r \mapsto 3, j \mapsto 3,]$*,* $\mathcal{V}'_e = \mathcal{V}$*, and* $\mathsf{FACT_{SA}}$ *is the program below. Hence, the triple* $\{n \geq 0 \wedge aux = n\}\,\mathsf{FACT_T}\,\{f = aux!, \bot\}$ *can be translated into the SA triple* $\{n_0 \geq 0 \wedge aux_0 = n_0\}\,\mathsf{FACT_{SA}}\,\{f_3 = aux_0!, \bot\}$*.*

$f_1 := 1\,;$

$i_1 := 1\,;$

$\mathbf{assert}\,f_1 = (i_1 - 1)! \wedge i_1 \leq n_0 + 1\,;$

```
assume f_2 = (i_2 − 1)! ∧ i_2 ≤ n_0 + 1 ;
if i_2 ≤ n_0 then
   j_1 := 1 ;
   r_1 := 0 ;
   assert j_1 ≤ i_2 + 1 ∧ r_1 = f_2 ∗ (j_1 − 1) ;
   assume j_2 ≤ i_2 + 1 ∧ r_2 = f_2 ∗ (j_2 − 1) ;
   if j_2 ≤ i_2 then
      r_3 := r_2 + f_2 ;
      j_3 := j_2 + 1 ;
      assert j_3 ≤ i_2 + 1 ∧ r_3 = f_2 ∗ (j_3 − 1) ;
      assume ⊥
   else r_3 := r_2 ; j_3 := j_2 fi
   f_3 := r_3 ;
   i_3 := i_2 + 1 ;
   assert f_3 = (i_3 − 1)! ∧ i_3 ≤ n_0 + 1 ;
   assume ⊥
else f_3 := f_2 ; i_3 := i_2 ; r_3 := r_0 ; j_3 := j_0 fi
```

Example 6.2. *Recall the program* GCD *from Example 4.15, and let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}$ *such that* $\mathcal{V}(x) = 0$, *for all* $x \in \mathbf{Var}$. *Then* $\mathfrak{T}(\mathcal{V}, \mathsf{GCD}) = (\mathcal{V}', \bot, \mathsf{GCD_{SA}})$, *where* $\mathcal{V}' = \mathcal{V}[u \mapsto 2, v \mapsto 2, t \mapsto 1]$ *and* $\mathsf{GCD_{SA}}$ *is shown below. Hence, the triple* $\{u_{aux} = u \wedge v_{aux} = v \wedge u \geq 0 \wedge v \geq 0\}\ \mathsf{GCD}\ \{u = \mathsf{gcd}\,(u_{aux}, v_{aux}), \bot\}$ *can be translated into the SA triple* $\{u_{aux} = u_0 \wedge v_{aux} = v_0 \wedge u_0 \geq 0 \wedge v_0 \geq 0\}\ \mathsf{GCD_{SA}}\ \{u_2 = \mathsf{gcd}\,(u_{aux}, v_{aux}), \bot\}$.

```
try
    assert u_0 ≥ 0 ∧ v_0 ≥ 0 ∧ gcd (u_0, v_0) = gcd (u_aux, v_aux) ;
    assume u_1 ≥ 0 ∧ v_1 ≥ 0 ∧ gcd (u_1, v_1) = gcd (u_aux, v_aux) ;
    if ⊤ then
        if v_1 = 0 then
            throw
        else
            t_1 := v_1 ;
            v_2 := u_1 % v_1 ;
            u_2 := t_1
        fi ;
        assert u_2 ≥ 0 ∧ v_2 ≥ 0 ∧ gcd (u_2, v_2) = gcd (u_aux, v_aux) ;
        assume ⊥
    else
```

```
            u2 := u1 ; v2 := v1
        fi
catch
        skip ;
        u2 := u1 ; v2 := v1
hc
```

The question that remains to be answered after a program is translated into SA using the translation above, is whether its correction forces the correction of the initial program and vice-versa. The following sections make this clear.

6.3 Soundness of SA Translation

The present section proves that whenever a translated triple is semantically valid then so is the original triple. If we have a triple $\{\phi\}\, C\, \{\psi, \epsilon\}$ and the result of applying the translation is as $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', \mathcal{V}'_e, C')$, for some $\mathcal{V} \in \textbf{Var} \to \mathbb{N}$, then whenever $\models \{\widehat{\mathcal{V}}(\phi)\}\, C'\, \{\widehat{\mathcal{V}'}(\psi), \widehat{\mathcal{V}'_e}(\epsilon)\}$ holds, it must also hold that $\models \{\phi\}\, C\, \{\psi, \epsilon\}$. The proof is based on the analysis of the execution of the translated and the original program. Note however that this is not immediate since the original program contains loops and the translated triple does not.

What is more, the translated program contains asserts that were originated from invariants of the original program: basically the invariants are not taken into account in the executions of the original program, but they are in the translated program. We should note here that although invariants do not interfere in the execution of the original program, they play a crucial role in the axiomatic semantics when proving the correctness of a program w.r.t. some specification. Note also that, we aim to show that if the translated triple is valid, then so it is the original. Thus if some invariant does not hold in some execution of the translated program we have that $\not\models \{\widehat{\mathcal{V}}(\phi)\}\, C'\, \{\widehat{\mathcal{V}'}(\psi), \widehat{\mathcal{V}'_e}(\epsilon)\}$, and thus we do not have to care about the correction of the original triple. On the other hand, if it does hold for every execution in the translated program it must be preserved in the original program. In order to formally capture this argument we will consider an intermediate transformation that gives operational meaning to the invariants by placing appropriate asserts in the code.

Definition 6.7. *The function* $\mathcal{A} : \textbf{AComm} \to \textbf{AComm}$ *that gives semantic meaning to loop*

invariants is defined as follows:

$$
\begin{array}{rcl}
\mathcal{A}(\textbf{while}\ b\ \textbf{do}\ \{\theta\}\ C_1\ \textbf{od}) & = & \textbf{assert}\,\theta\,;\,\textbf{while}\ b\ \textbf{do}\ \mathcal{A}(C_1)\,;\,\textbf{assert}\,\theta\ \textbf{od} \\
\mathcal{A}(\textbf{skip}) & = & \textbf{skip} \\
\mathcal{A}(\textbf{throw}) & = & \textbf{throw} \\
\mathcal{A}(\textbf{assume}\,\theta) & = & \textbf{assume}\,\theta \\
\mathcal{A}(\textbf{assert}\,\theta) & = & \textbf{assert}\,\theta \\
\mathcal{A}(x := e) & = & x := e \\
\mathcal{A}(C_1\,;\,C_2) & = & \mathcal{A}(C_1)\,;\,\mathcal{A}(C_2) \\
\mathcal{A}(\textbf{try}\,C_1\,\textbf{catch}\,C_2\,\textbf{hc}) & = & \textbf{try}\,\mathcal{A}(C_1)\,\textbf{catch}\,\mathcal{A}(C_2)\,\textbf{hc} \\
\mathcal{A}(\textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}) & = & \textbf{if}\ b\ \textbf{then}\ \mathcal{A}(C_1)\ \textbf{else}\ \mathcal{A}(C_2)\ \textbf{fi}
\end{array}
$$

With this translation we can start stating some results about the execution of programs containing annotated loops C and programs where those annotations were transformed into asserts $\mathcal{A}(C)$. The following lemma starts by capturing the fact that if $\mathcal{A}(C)$ terminates in a state that is not error then C will also terminate in that state. This is actually easy to observe when looking at the definition of $\mathcal{A}$: the translation only inserts additional assert conditions and those either fail, and the program terminates in the error state, or do not fail and in that case do not change the program state. The second and third property indicate that if C executes from some arbitrary state and terminates in a final number of steps then $\mathcal{A}(C)$ will certainly terminate, but *possibly* in a different state. In particular, the third property indicates that if the original program terminates in the error state, then so will the translated program (the same cannot be said if the original program does not terminate in the error state).

Lemma 6.12. *Let $C \in$ **AComm**, $s_i \in \Sigma$, and $\sigma \in \Sigma_\bullet$. Then, the following properties hold:*

1. *If $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \sigma$ and $\sigma \neq \bullet$, then $\langle C, s_i\rangle \Rightarrow^* \sigma$.*
2. *If $\langle C, s_i\rangle \Rightarrow^* \sigma$ and $\sigma \neq \bullet$, then $\exists\, \sigma' \in \Sigma_\bullet.\ \langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \sigma'$.*
3. *If $\langle C, s_i\rangle \Rightarrow^* \bullet$, then $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \bullet$.*

Proof. 1. Follows by induction on the pair $(k, \sharp C)$, for the derivation sequence $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^k \sigma$. The proof uses Lemmas 6.1 and 6.2 and, in the induction step, follows by analysis of the first step of the derivation sequence. All cases are straightforward.

2. Follows by induction on the pair $(k, \sharp C)$ for the derivation sequence $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^k \sigma$.

Base case. Trivial, since there is no derivation from $\langle C, s_i\rangle$ to σ in zero steps.

Induction step. Assume that the property holds for every pair $(k', \sharp C') < (k, \sharp C)$. We want to prove that it holds for the pair $(k, \sharp C)$. Since the derivation sequence has at least one step, we must have that $\langle C, s_i\rangle \Rightarrow \delta \Rightarrow^{k-1} \sigma$ and $\sigma \neq \bullet$. We proceed by analysis of cases on the first step of the derivation sequence.

We show here the cases when C is $C_1\,;\,C_2$ or **while** b **do** $\{\theta\}$ C_1 **od**. The other cases are analogous or trivial.

If C is $C_1\,;\,C_2$, then we can have two cases:

- $\sigma = \mathbf{e}(s')$ for some $s' \in \Sigma$. Then by Lemma 6.2 we will have one of two cases:
  - $\langle C_1, s\rangle \Rightarrow^k \mathbf{e}(s')$. By IH there exists $\sigma' \in \Sigma_\bullet$ such that $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^* \sigma'$. If $\sigma' = \bullet$ then we are done because $\langle \mathcal{A}(C_1) \,;\, \mathcal{A}(C_2), s\rangle \Rightarrow^* \bullet$. If $\sigma' \neq \bullet$, then we are also done because from 1. we have that $\sigma' = \mathbf{e}(s')$ and thus $\langle \mathcal{A}(C_1) \,;\, \mathcal{A}(C_2), s\rangle \Rightarrow^* \mathbf{e}(s')$.
  - there exist $k_1, k_2 \in \mathbb{N}$ and $s'' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{k_1} \mathbf{n}(s'')$, $\langle C_2, s''\rangle \Rightarrow^{k_2} \mathbf{e}(s')$ and $k = k_1 + k_2$. By IH, there exists $\sigma' \in \Sigma_\bullet$ such that $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^* \sigma'$. If $\sigma' = \bullet$ then we are done, otherwise from 1, we have that $\sigma' = \mathbf{n}(s'')$ and thus by IH $\langle \mathcal{A}(C_2), s''\rangle \Rightarrow^* \mathbf{e}(s')$. Using Lemma 6.1, we have $\langle \mathcal{A}(C_1) \,;\, \mathcal{A}(C_2), s\rangle \Rightarrow^* \langle \mathcal{A}(C_2), s''\rangle \Rightarrow^* \mathbf{e}(s')$.
- $\sigma = \mathbf{n}(s')$ for some $s' \in \Sigma$. The proof is analogous to the previous case, also using Lemmas 6.1 and 6.2.

If C is **while** b **do** $\{\theta\}$ C_1 **od**, we can have two cases:

- $s \not\models b$. Then, either $s \not\models \theta$ and $\langle \mathcal{A}(C), s\rangle \Rightarrow \bullet$, or $s \models \theta$ and $\langle \mathcal{A}(C), s\rangle \Rightarrow \mathbf{n}(s)$.
- $s \models b$. Then we must have that $\langle C, s\rangle \Rightarrow \langle \textbf{if } b \textbf{ then } C_1 \,;\, C \textbf{ else skip fi}, s\rangle \Rightarrow \langle C_1 \,;\, C, s\rangle$. From here, the proof is analogous to the sequence case.

3. Follows by induction on the pair $(k, \sharp C)$ for the derivation sequence $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^k \bullet$.
Base case. Trivial, since there is no derivation sequence from $\langle C, s_i\rangle$ to $\bullet$ in zero steps.

Induction step. Assume that the property holds for every pair $(k', \sharp C') < (k, \sharp C)$. We want to prove that it holds for the pair $(k, \sharp C)$. Since the derivation sequence has at least one step, we must have that $\langle C, s_i\rangle \Rightarrow \delta \Rightarrow^{k-1} \bullet$. We proceed by analysis of cases on the first step of the derivation sequence.

We show here the case when C is **try** C_1 **catch** C_2 **hc** or **while** b **do** $\{\theta\}$ C_1 **od**. The other cases are analogous or trivial.

If C is **try** C_1 **catch** C_2 **hc**, by Lemma 6.2, we will have one of the following cases:

- $\langle C_1, s\rangle \Rightarrow^k \bullet$. In this case, by IH $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^k \bullet$ and by lemma Lemma 6.1, $\langle \textbf{try}\, \mathcal{A}(C_1)$ $\textbf{catch}\, \mathcal{A}(C_2)\, \textbf{hc}, s\rangle \Rightarrow^k \bullet$.
- there exist $k_1, k_2 \in \mathbb{N}$ and $s' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{k_1} \mathbf{e}(s')$, $\langle C_2, s'\rangle \Rightarrow^{k_2} \bullet$, and $k = k_1 + k_2$. From 2. there exists some $\sigma \in \Sigma_\bullet$ such that $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^* \sigma$. If $\sigma = \bullet$, then $\langle \textbf{try}\, \mathcal{A}(C_1)\, \textbf{catch}\, \mathcal{A}(C_2)\, \textbf{hc}, s\rangle \Rightarrow^* \bullet$ and we are done. Otherwise, if $\sigma \neq \bullet$ then $\sigma = \mathbf{n}(s')$ because 1. and therefore by IH we have that $\langle \mathcal{A}(C_2), s'\rangle \Rightarrow^* \bullet$. Using Lemma 6.1, we conclude that $\langle \textbf{try}\, \mathcal{A}(C_1)\, \textbf{catch}\, \mathcal{A}(C_2)\, \textbf{hc}, s\rangle \Rightarrow^* \bullet$.

If C is **while** b **do** $\{\theta\}$ C_1 **od** then one of the following can happen:

- $s \not\models \theta$ and we are done because $\langle \mathcal{A}(C), s\rangle \Rightarrow \bullet$, or
- $s \not\models b$ and we are done because $\langle C, s\rangle \Rightarrow^3 \mathbf{n}(s)$ and $\mathbf{n}(s) \neq \bullet$, or
- $s \models b \wedge \theta$, $\langle C, s\rangle \Rightarrow \langle \textbf{if } b \textbf{ then } C_1 \,;\, C \textbf{ else skip fi}, s\rangle \Rightarrow \langle C_1 \,;\, C, s\rangle \Rightarrow^{k-2} \bullet$, and from Lemma 6.2, we have one of the following:

- $\langle C_1, s\rangle \Rightarrow^{k-2} \bullet$. In this case by IH $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^* \bullet$ and thus $\langle \mathcal{A}(C), s\rangle \Rightarrow^* \bullet$, because $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^* \bullet$.
- there exist $k_1, k_2 \in \mathbb{N}$, $s' \in \Sigma$ such that $\langle C_1, s\rangle \Rightarrow^{k_1} \mathbf{n}(s')$, $\langle C, s'\rangle \Rightarrow^{k_2} \bullet$ and $k - 2 = k_1 + k_2$. In this case, from 2. we have that there exists some $\sigma \in \Sigma_\bullet$ such that $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^* \sigma$. If $\sigma = \bullet$ then $\langle \mathcal{A}(C_1)\,;\, \mathcal{A}(C), s\rangle \Rightarrow^* \bullet$ and we are done. Otherwise, $\sigma \neq \bullet$ and from 1 we have that $\sigma = \mathbf{n}(s')$. By IH $\langle \mathcal{A}(C), s'\rangle \Rightarrow^* \bullet$ and thus by Lemma 6.1, $\langle \mathcal{A}(C_1)\,;\, \mathcal{A}(C), s\rangle \Rightarrow^* \bullet$. □

Using the previous lemmas it is now possible to relate C and $\mathcal{A}(C)$ in terms of Hoare triples.

Lemma 6.13. *If* $\models \{\phi\}\, \mathcal{A}(C)\, \{\psi, \epsilon\}$*, then* $\models \{\phi\}\, C\, \{\psi, \epsilon\}$.

Proof. Assume that $\models \{\phi\}\, \mathcal{A}(C)\, \{\psi, \epsilon\}$. Let $s \in \Sigma$ be such that $s \models \phi$. So, if $\langle C, s\rangle \not\Rightarrow^*$ we are done, otherwise, if $\langle C, s\rangle \Rightarrow^* \sigma$ then:

- $\sigma \neq \bullet$, because if $\sigma = \bullet$ then by Lemma 6.12 $\langle \mathcal{A}(C), s\rangle \Rightarrow^* \bullet$, which is a contradiction.
- if $\sigma = \mathbf{n}(s')$ by Lemma 6.12 we have that there exists some σ' such that $\langle \mathcal{A}(C), s\rangle \Rightarrow^* \sigma'$ and since from the hypothesis $\sigma' \neq \bullet$, again by Lemma 6.12 we have that $\sigma' = \sigma = \mathbf{n}(s')$ and thus $s' \models \psi$.
- if $\sigma = \mathbf{n}(s')$ we also have that $s' \models \epsilon$ for the same reason as in the previous case.

□

With the previous lemma it remains to prove that the validity of the translated triple $\{\widehat{\mathcal{V}}(\phi)\}\, C'\, \{\widehat{\mathcal{V}'}(\psi), \widehat{\mathcal{V}'_e}(\epsilon)\}$ implies the validity of the triple $\{\phi\}\, \mathcal{A}(C)\, \{\psi, \epsilon\}$. For that purpose, let us make a small remark about the execution of loops in $\mathcal{A}(C)$ that will possibly give some intuition for what is to come. Consider that C is the program **while** b **do** $\{\theta\}$ C_1 **od**. When evaluating the program **assert** θ ; **while** b **do** C_1 ; **assert** θ **od** (which is the result of $\mathcal{A}(C)$) from a state $s \in \Sigma$, if $s \models \theta \wedge b$ and $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^* \mathbf{n}(s')$ then $\langle \mathcal{A}(C), s\rangle \Rightarrow^+ \langle \mathcal{A}(C), s'\rangle$. If we continue the evaluation from this configuration, it is possible that $\langle \mathcal{A}(C), s'\rangle \Rightarrow^+ \langle \mathcal{A}(C), s''\rangle$. Basically, the initial program $\mathcal{A}(C)$ can possibly be obtained over and over again with (possibly) different states, while the loop condition holds.

Before going any further, let us make some remarks. The first is that the values of the variables that are not assigned in the loop remain constant. The second is that the intermediate states resulting from evaluating iterations are always compatible with version functions that are used to translate loops. These observations are captured by the following lemma.

Lemma 6.14. *Let* $\mathcal{V}, \mathcal{V}' \in \mathbf{Var} \to \mathbb{N}$ *be such that* $\mathcal{V}' = \mathcal{V}[x \mapsto \mathcal{V}(x) + 1 \mid x \in \mathsf{Asgn}(C_1)]$. *If* $\langle \mathbf{assert}\, \theta\,;\, \mathbf{while}\ b\ \mathbf{do}\ \mathcal{A}(C_1)\,;\, \mathbf{assert}\, \theta\ \mathbf{od}, s\rangle \Rightarrow^* \langle \mathbf{assert}\, \theta\,;\, \mathbf{while}\ b\ \mathbf{do}\ \mathcal{A}(C_1)\,;\, \mathbf{assert}\, \theta\ \mathbf{od}, s'\rangle$ *then the following holds:*

1. $\forall\, x \notin \mathsf{Asgn}(C_1).\ s(x) = s'(x)$.
2. s, s' *are compatible w.r.t.* $\mathcal{V}, \mathcal{V}'$.

Proof. Both proofs are straightforward. □

Still with respect to the previous lemma, note that it always holds that $\langle \mathcal{A}(C), s\rangle \Rightarrow^* \langle \mathcal{A}(C), s'\rangle$, since it is always possible to go in zero steps into the same configuration (in this case $s = s'$).

Let us now turn our focus to the execution of SA programs. Since the translation always uses the most recent version of the variables, changing the value of older versions in the initial state does not change the way the program behaves, neither the value of the final version of the variables in the final state (if it is not error). The lemma below might seem to be going a bit further due to the restriction $\mathcal{V}(x) \leq \mathcal{V}'(x)$, nonetheless note that for an arbitrary variable x such that $\mathcal{V}(x) = \mathcal{V}'(x)$, according to Definition 2.18 the value will be given by $\mathcal{V}'(s)$.

Lemma 6.15. *Let $\mathfrak{T}(\mathcal{V}', C) = (\mathcal{V}'', \mathcal{V}''_e, C')$ and $\mathcal{V}(x) \leq \mathcal{V}'(x)$, for every $x \in \mathbf{Var}$.*

1. *If $\langle C', s' \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \boldsymbol{n}(s_f)$ then, $\forall\, s'' \in \Sigma^{\text{SA}}.\ \langle C', s' \oplus \mathcal{V}(s'') \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \boldsymbol{n}(s'_f)$ and $\forall\, x \in \mathbf{Var}.\ s'_f(\widehat{\mathcal{V}''}(x)) = s_f(\widehat{\mathcal{V}''}(x))$.*
2. *If $\langle C', s' \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \boldsymbol{e}(s_f)$ then, $\forall\, s'' \in \Sigma^{\text{SA}}.\ \langle C', s' \oplus \mathcal{V}(s'') \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \boldsymbol{e}(s'_f)$ and $\forall\, x \in \mathbf{Var}.\ s'_f(\widehat{\mathcal{V}''_e}(x)) = s_f(\widehat{\mathcal{V}''_e}(x))$.*
3. *If $\langle C', s' \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \bullet$ then, $\forall\, s'' \in \Sigma^{\text{SA}}.\ \langle C', s' \oplus \mathcal{V}(s'') \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \bullet$.*

Proof. All the cases are proved by induction on the length of the derivation. □

We will need to confine certain states to assign values only to a part of the variables. The definition below introduces the notation for that.

Definition 6.8. *Let $s \in \Sigma^{\text{SA}}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}$, and $\odot \in \{<, \leq, \geq, >\}$. Then $s|^{\odot}_{\mathcal{V}}$ represents the partial function defined as $[x_i \mapsto s(x_i) \mid x_i \in \mathsf{dom}(s) \wedge i \odot \mathcal{V}(x)]$.*

Basically the state $s|^{\odot}_{\mathcal{V}}$ only assigns values to variables whose version satisfy a given constraint expressed by the operator $\odot$ and the version function $\mathcal{V}$. For instance $s|^{>}_{\mathcal{V}}$ will assign values to all $x_i \in \mathsf{dom}(s)$ such that $i > \mathcal{V}(x)$, in which case $s|^{>}_{\mathcal{V}}(x_i) = s(x_i)$.

Similarly to Lemma 6.15, the lemma below observes that the only part of the state that is relevant for the execution of the program is the part that assigns values to the variables whose version is equal or greater than the version given by the function that was used to translate the program.

Lemma 6.16. *The following holds:*

1. *If $\langle C', s' \oplus \mathcal{V}(s)\rangle \Rightarrow^* \bullet$, then $\forall\, s'' \in \Sigma^{\text{SA}}.\ \langle C', s'' \oplus s'|^{>}_{\mathcal{V}} \oplus \mathcal{V}(s)\rangle \Rightarrow^* \bullet$.*
2. *If $\langle C', s' \oplus \mathcal{V}(s)\rangle \Rightarrow^* \boldsymbol{n}(s_f)$, then $\forall\, s'' \in \Sigma^{\text{SA}}.\ \langle C', s'' \oplus s'|^{>}_{\mathcal{V}} \oplus \mathcal{V}(s)\rangle \Rightarrow^* \boldsymbol{n}(s'_f)$ and $\forall\, x \in \mathbf{Var}.\ s'_f(\widehat{\mathcal{V}'}(x)) = s_f(\widehat{\mathcal{V}'}(x))$.*
3. *If $\langle C', s' \oplus \mathcal{V}(s)\rangle \Rightarrow^* \boldsymbol{e}(s_f)$, then $\forall\, s'' \in \Sigma^{\text{SA}}.\ \langle C', s'' \oplus s'|^{>}_{\mathcal{V}} \oplus \mathcal{V}(s)\rangle \Rightarrow^* \boldsymbol{e}(s'_f)$ and $\forall\, x \in \mathbf{Var}.\ s'_f(\widehat{\mathcal{V}'_e}(x)) = s_f(\widehat{\mathcal{V}'_e}(x))$.*

Proof. By derivation on the structure of C. □

The following lemma states that the execution of an SA program does not change the value of the variables whose version is greater than the final version function.

Lemma 6.17. *Let $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', \mathcal{V}'_e, C')$. Then:*

1. *If $\langle C', s\rangle \Rightarrow^* \boldsymbol{n}(s_f)$, then $\forall\, s' \in \Sigma^{\mathrm{SA}}.\ \langle C', s \oplus s'|^{>}_{\mathcal{V}'}\rangle \Rightarrow^* \boldsymbol{n}(s_f \oplus s'|^{>}_{\mathcal{V}'})$.*
2. *If $\langle C', s\rangle \Rightarrow^* \boldsymbol{e}(s_f)$, then $\forall\, s' \in \Sigma^{\mathrm{SA}}.\ \langle C', s \oplus s'|^{>}_{\mathcal{V}'_e}\rangle \Rightarrow^* \boldsymbol{e}(s_f \oplus s'|^{>}_{\mathcal{V}'_e})$.*

Proof. By derivation on the structure of C. □

We are now in a position to establish a relation between the execution of programs obtained by $\mathcal{A}$ and $\mathfrak{T}$. For a program C, the next proposition states that if the execution of $\mathcal{A}(C)$ from $s_i \in \Sigma$ terminates, then it is certain that there exists some state $s' \in \Sigma^{\mathrm{SA}}$ that when combined with $\mathcal{V}(s_i)$ will make the SA program terminate. For the case that $\mathcal{A}(C)$ terminates normally or exceptionally, the lemma goes even further and relates the final states.

Proposition 6.18. *Let $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', \mathcal{V}'_e, C')$. The following hold:*

1. *If $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \boldsymbol{n}(s_f)$, then $\exists\, s', s'_f \in \Sigma^{\mathrm{SA}}.\ \langle C', s' \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \boldsymbol{n}(s'_f)$ and $\forall\, x.\ s_f(x) = s'_f(\widehat{\mathcal{V}'}(x))$.*
2. *If $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \boldsymbol{e}(s_f)$, then $\exists\, s', s'_f \in \Sigma^{\mathrm{SA}}.\ \langle C', s' \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \boldsymbol{e}(s'_f)$ and $\forall\, x.\ s_f(x) = s'_f(\widehat{\mathcal{V}'_e}(x))$.*
3. *If $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \bullet$, then $\exists\, s' \in \Sigma^{\mathrm{SA}}.\ \langle C', s' \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \bullet$.*

Proof. The proof follows by simultaneous induction on the structure of C.

Case C is **skip**, **throw**, $x := e$, **assume** θ, or **assert** θ the proof is trivial.

Case C is **while** b **do** $\{\theta\}$ C_1 **od**. Then we have:

$$\begin{aligned}\mathfrak{T}(\mathcal{V}, \textbf{while } b \textbf{ do } \{\theta\}\ C_1 \textbf{ od}) = (\,\mathcal{V}'', \mathcal{V}''_e,\ & \textbf{assert}\, \widehat{\mathcal{V}}(\theta)\,;\textbf{assume}\, \widehat{\mathcal{V}'}(\theta)\,;\\ & \textbf{if } \widehat{\mathcal{V}'}(b) \textbf{ then } C'_1\,;\textbf{assert}\, \widehat{\mathcal{V}''}(\theta)\,;\textbf{assume}\,\bot\\ & \textbf{else } \mathsf{mrg}(\mathcal{V}', \mathcal{V}'') \textbf{ fi}\,),\\ \mathcal{V}' = \mathcal{V}[x \mapsto \mathcal{V}(x) + 1 \mid x \in \mathsf{Asgn}(C_1)],\ & \text{and } (\mathcal{V}'', \mathcal{V}''_e, C'_1) = \mathfrak{T}(\mathcal{V}', C_1)\end{aligned}$$

For 1, assume that $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \mathbf{n}(s_f)$. Then it must be the case that $s_i \models \theta$ and $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \langle \mathcal{A}(C), s\rangle$, for some $s \in \Sigma^{\mathrm{SA}}$ such that $s \models \theta \wedge \neg b$. From Lemma 6.6 it holds that $s_0 \oplus \mathcal{V}(s_i) \oplus \mathcal{V}'(s) \models \widehat{\mathcal{V}'}(\theta) \wedge \widehat{\mathcal{V}'}(\neg b)$, for some $s_0 \in \Sigma^{\mathrm{SA}}$, and from Lemmas 6.6, 6.8 and 6.14 that $s_0 \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i) \models \widehat{\mathcal{V}}(\theta) \wedge \widehat{\mathcal{V}'}(\theta) \wedge \widehat{\mathcal{V}'}(\neg b)$. Hence there exists indeed some $s' = s_0 \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i)$ such that $\langle C', s'\rangle \Rightarrow^3 \langle \mathsf{mrg}(\mathcal{V}', \mathcal{V}''), s'\rangle \Rightarrow^* \mathbf{n}(s' \oplus \mathcal{V}''(s))$. The last step is justified by Lemmas 6.8, 6.10 and 6.14, and by the fact that $\mathsf{sup}(\mathcal{V}', \mathcal{V}'') = \mathcal{V}''$. From Lemma 6.5 it holds that $\forall\, x \in \mathbf{Var}.\ (s' \oplus \mathcal{V}''(s))(\widehat{\mathcal{V}''}(x))$.

For 2, assume that $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \mathbf{e}(s_f)$. Then it must be the case that $s_i \models \theta$ and $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \langle \mathcal{A}(C), s\rangle$, for some $s \in \Sigma^{\mathrm{SA}}$ such that $s \models \theta \wedge b$ and $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^* \mathbf{e}(s_f)$.

By IH, $\exists\, s_0, s'_f \in \Sigma^{\mathrm{SA}}.\ \langle C'_1, s_0 \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \mathbf{e}(s'_f)$, and $\forall\, x \in \mathbf{Var}.\ s_f(x) = s'_f(\widehat{\mathcal{V}''_e}(x))$. From Lemmas 6.8, 6.14 and 6.15, $\langle C'_1, s_0 \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \mathbf{e}(s'_f)$. Note also that $s_0 \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i) \models \widehat{\mathcal{V}}(\theta) \wedge \widehat{\mathcal{V}'}(\theta) \wedge \widehat{\mathcal{V}'}(b)$, hence $\langle s_0 \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i), C'\rangle \Rightarrow^* \mathbf{e}(s'_f)$.

For 3, assume that $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \bullet$ and note that in this case, one of three cases must occur:

- $s_i \not\models \theta$. In this case $s' \oplus \mathcal{V}(s_i) \not\models \widehat{\mathcal{V}}(\theta)$ for any $s' \in \Sigma^{\mathrm{SA}}$, and thus $\langle C', s' \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \bullet$.
- $s_i \models \theta \wedge b$, $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \langle \mathcal{A}(C), s\rangle$, for some $s \in \Sigma^{\mathrm{SA}}$ such that $s \models \theta \wedge b$ and $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^* \bullet$. By IH, $\exists\, s' \in \Sigma^{\mathrm{SA}}.\ \langle C'_1, s' \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \bullet$ and from Lemmas 6.8, 6.14 and 6.15, $\exists\, s' \in \Sigma^{\mathrm{SA}}.\ \langle C'_1, s' \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \bullet$. Since $s' \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i) \models \widehat{\mathcal{V}}(\theta) \wedge \widehat{\mathcal{V}'}(\theta) \wedge \widehat{\mathcal{V}'}(b)$, it is possible to conclude that $\langle C', s' \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \bullet$.
- $s_i \models \theta \wedge b$, $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* \langle \mathcal{A}(C), s\rangle$, for some $s \in \Sigma^{\mathrm{SA}}$ such that $s \models \theta \wedge b$, $\langle \mathcal{A}(C_1), s\rangle \Rightarrow^* \mathbf{n}(s_f)$ and $s_f \not\models \theta$. By IH, and using Lemmas 6.8, 6.14 and 6.15 it is possible to obtain $\exists\, s' \in \Sigma^{\mathrm{SA}}.\ \langle C'_1, s' \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \mathbf{n}(s'_f)$ and $\forall\, x \in \mathbf{Var}.\ s_f(x) = s'_f(\widehat{\mathcal{V}''}(x))$. Since $s' \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i) \models \widehat{\mathcal{V}}(\theta) \wedge \widehat{\mathcal{V}'}(\theta) \wedge \widehat{\mathcal{V}'}(b)$, it is the case that $\langle C', s' \oplus \mathcal{V}'(s) \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \langle \mathbf{assert}\, \widehat{\mathcal{V}''}(\theta) \,;\, \mathbf{assume}\, \bot, s'_f\rangle \Rightarrow \bullet$. Note that, it holds that $s'_f \not\models \widehat{\mathcal{V}''}(\theta)$ because $s_f \not\models \theta$ and $s'_f = s'' \oplus \mathcal{V}''(s_f)$ for some $s'' \in \Sigma^{\mathrm{SA}}$.

With the previous cases, we conclude that $\exists\, s' \in \Sigma^{\mathrm{SA}}.\ \langle C', s' \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \bullet$.

Case C is $C_1 \,;\, C_2$. Then we have:

$$\begin{aligned}\mathfrak{T}(\mathcal{V}, C_1 \,;\, C_2) = (\, & \mathcal{V}'', \mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e),\ \mathbf{try}\, C'_1\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}'_e, \mathcal{V}''_e) \,;\, \mathbf{throw\, hc} \,; \\ & \qquad\qquad\qquad\qquad \mathbf{try}\, C'_2\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}''_e, \mathcal{V}'_e) \,;\, \mathbf{throw\, hc}\,), \\ (\mathcal{V}', \mathcal{V}'_e, C'_1) = \mathfrak{T}&(\mathcal{V}, C_1),\ \mathcal{V}' \neq \bot,\ \text{and}\ (\mathcal{V}'', \mathcal{V}''_e, C'_2) = \mathfrak{T}(\mathcal{V}', C_2)\end{aligned}$$

For 1, assume that $\langle \mathcal{A}(C), s_i\rangle \Rightarrow^* s_f$. Then, there exists some $s \in \Sigma$ such that $\langle \mathcal{A}(C_1), s_i\rangle \Rightarrow^* s$ and $\langle \mathcal{A}(C_2), s\rangle \Rightarrow^* s_f$. By IH, and using also Lemma 6.5, we have $\exists\, s_0 \in \Sigma^{\mathrm{SA}}.\ \langle C'_1, s_0 \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \mathbf{n}(s'' \oplus \mathcal{V}'(s))$, for some $s'' \in \Sigma^{\mathrm{SA}}$. Again, by IH we have $\exists\, s_1, s'_f \in \Sigma^{\mathrm{SA}}.\ \langle C'_2, s_1 \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \mathbf{n}(s'_f)$ and $\forall\, x \in \mathbf{Var}.\ s_f(x) = s'_f(\widehat{\mathcal{V}''}(x))$. Now, using Lemma 6.16, $\langle C'_2, s'' \oplus s_1\big|^{>}_{\mathcal{V}'} \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \mathbf{n}(s''_f)$ for some s''_f such that $s''_f(\widehat{\mathcal{V}''}(x)) = s'_f(\widehat{\mathcal{V}''}(x))$. Going back to $\langle C'_1, s_0 \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \mathbf{n}(s'' \oplus \mathcal{V}'(s))$, from Lemma 6.17 we have that $\langle C'_1, s_0 \oplus \mathcal{V}(s_i) \oplus s_1\big|^{>}_{\mathcal{V}'}\rangle \Rightarrow^* \mathbf{n}(s'' \oplus \mathcal{V}'(s) \oplus s_1\big|^{>}_{\mathcal{V}'})$ and from Lemma 2.12 that $\langle C'_1, s_0 \oplus s_1\big|^{>}_{\mathcal{V}'} \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \mathbf{n}(s'' \oplus s_1\big|^{>}_{\mathcal{V}'} \oplus \mathcal{V}'(s))$. So, $\langle C', s_0 \oplus s_1\big|^{>}_{\mathcal{V}'} \oplus \mathcal{V}(s_i)\rangle \Rightarrow^* \langle \mathbf{try}\, C'_2\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}''_e, \mathcal{V}'_e)\, \mathbf{hc}, s'' \oplus s_1\big|^{>}_{\mathcal{V}'} \oplus \mathcal{V}'(s)\rangle \Rightarrow^* \mathbf{n}(s''_f)$, and $s''_f(\widehat{\mathcal{V}''}(x)) = s'_f(\widehat{\mathcal{V}''}(x)) = s_f(x)$ which allows us to conclude 1.

For 2 and 3 the proof is analogous. It starts by using Lemma 6.2, which results in two distinct cases, and then for each case follows by applying IH and using Lemmas 6.16 and 6.17.

Case C is $\mathbf{try}\, C_1\, \mathbf{catch}\, C_2\, \mathbf{hc}$. The proof is analogous to the case in which C is $C_1 \,;\, C_2$.

Case C is $\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}$. Then we have:

$$\begin{aligned}\mathfrak{T}(\mathcal{V}, \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}) = (\, & \mathsf{sup}(\mathcal{V}', \mathcal{V}''), \mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e), \\ & \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then\ try}\ C'_1 \,;\, \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\ \mathbf{catch}\ \mathsf{mrg}(\mathcal{V}'_e, \mathcal{V}''_e) \,;\, \mathbf{throw\ hc} \\ & \mathbf{else\ try}\ C'_2 \,;\, \mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\ \mathbf{catch}\ \mathsf{mrg}(\mathcal{V}''_e, \mathcal{V}'_e) \,;\, \mathbf{throw\ hc\ fi}\,), \\ (\mathcal{V}', \mathcal{V}'_e, C'_1) = \mathfrak{T}(\mathcal{V}, C_1),\ & \text{and}\ (\mathcal{V}'', \mathcal{V}''_e, C'_2) = \mathfrak{T}(\mathcal{V}, C_2)\end{aligned}$$

We show here the proof for 2. The proof for 1 and 3 are analogous. Assume that $\langle \mathcal{A}(C), s_i \rangle \Rightarrow^* \mathbf{e}(s_f)$. Then one of two cases must occur:

- $s_i \models b$. In this case, $\langle \mathcal{A}(C_1), s_i \rangle \Rightarrow^* \mathbf{e}(s_f)$ and thus, by IH $\exists\, s_0 \in \Sigma^{\mathrm{SA}}.\ \langle C_1', s_0 \oplus \mathcal{V}(s_i) \rangle \Rightarrow^* \mathbf{e}(s_f')$ and $\forall\, x \in \Sigma^{\mathrm{SA}}.\ s_f(x) = s_f'(\widehat{\mathcal{V}_e'}(x))$. From Lemma 6.5, $s_f' = s' \oplus \mathcal{V}_e'(s_f)$. Then, $s_0 \oplus \mathcal{V}(s_i) \models \widehat{\mathcal{V}}(b)$, $\langle C', s_0 \oplus \mathcal{V}(s_i) \rangle \Rightarrow^* \langle \mathsf{mrg}(\mathcal{V}_e', \mathcal{V}_e'')\,;\, \mathbf{throw}, s' \oplus \mathcal{V}_e'(s_f) \rangle \Rightarrow^* \mathbf{e}(s' \oplus \mathcal{V}_e'(s_f) \oplus \mathsf{sup}(\mathcal{V}_e', \mathcal{V}_e'')(s_f))$, where the last step is justified by Lemma 6.10, and thus we conclude the proof because $\forall\, x \in \mathbf{Var}.\ s_f(x) = (s' \oplus \mathcal{V}_e'(s_f) \oplus \mathsf{sup}(\mathcal{V}_e', \mathcal{V}_e'')(s_f))(\widehat{\mathsf{sup}(\mathcal{V}_e', \mathcal{V}_e'')}(x))$.

- $s_i \models \neg b$. The proof is analogous.

This allow us to conclude that $\exists\, s', s_f' \in \Sigma^{\mathrm{SA}}.\ \langle C', s' \oplus \mathcal{V}(s_i) \rangle \Rightarrow^* s_f'$ and $\forall\, x \in \Sigma.\ s_f(x) = s_f'(\widehat{\mathsf{sup}(\mathcal{V}_e', \mathcal{V}_e'')}(x))$. □

Resorting to the previous result, we are now able to prove the soundeness of the translation w.r.t. the axiomatic semantics.

Theorem 6.19. *Let $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', \mathcal{V}_e', C')$ and assume that $\models \{\widehat{\mathcal{V}}(\phi)\}\, C'\, \{\widehat{\mathcal{V}'}(\psi), \widehat{\mathcal{V}_e'}(\epsilon)\}$. Then $\models \{\phi\}\, \mathcal{A}(C)\, \{\psi, \epsilon\}$.*

Proof. First of all assume that $\models \{\widehat{\mathcal{V}}(\phi)\}\, C'\, \{\widehat{\mathcal{V}'}(\psi), \widehat{\mathcal{V}_e'}(\epsilon)\}$. Let $s_i \in \Sigma$ be such that $s_i \models \phi$ and assume also that $\langle \mathcal{A}(C), s_i \rangle \Rightarrow^* \sigma$ (note that if the execution of $\mathcal{A}(C)$ gets stuck or does not terminate we are done). Then:

- $\sigma \neq \bullet$, because if $\sigma = \bullet$ then by Proposition 6.18, $\exists\, s' \in \Sigma^{\mathrm{SA}}.\ \langle C', s' \oplus \mathcal{V}(s_i) \rangle \Rightarrow^* \bullet$, which contradicts the hypothesis.

- if $\sigma = \mathbf{n}(s_f')$ for some $s_f \in \Sigma^{\mathrm{SA}}$, then by Proposition 6.18, $\exists\, s', s_f' \in \Sigma^{\mathrm{SA}}.\ \langle C', s' \oplus \mathcal{V}(s_i) \rangle \Rightarrow^* s_f'$ and $\forall\, x \in \mathbf{Var}.\ s_f(x) = s_f'(\mathcal{V}'(x))$. From Lemma 6.5, $s_f' = s_0 \oplus \mathcal{V}'(s_f)$ for some $s_0 \in \Sigma^{\mathrm{SA}}$ and thus from Lemma 6.6 and the hypothesis we have that $s_f' \models \mathcal{V}'(\psi)$.

- if $\sigma = \mathbf{e}(s_f')$ for some $s_f' \in \Sigma^{\mathrm{SA}}$, it also holds that $s_f' \models \mathcal{V}_e'(\epsilon)$. The proof is analogous to the previous case.

From the above we conclude that $\models \{\phi\}\, \mathcal{A}(C)\, \{\psi, \epsilon\}$. □

Corollary 6.20. *If $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', C')$ and $\models \{\widehat{\mathcal{V}}(\phi)\}\, C'\, \{\widehat{\mathcal{V}'}(\psi)\}$, then $\models \{\phi\}\, C\, \{\psi\}$.*

Proof. Follows directly from Theorem 6.19 and Lemma 6.13. □

The next section proves that the translation is also complete w.r.t. system Hg.

6.4 Completeness of SA Translation

It remains to be proved that the translation is complete in the sense that if the original triple is derivable in system Hg then so is the translated triple. This together with the result from the previous section will allow us to conclude that the translation is indeed an SA translation in the sense of Definition 5.8.

Proposition 6.21. *Let $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}$ and $C \in \mathbf{AComm}$. If $\mathfrak{T}(\mathcal{V}, C) = (\mathcal{V}', \mathcal{V}'_e, C')$ and $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$ then $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, C'\, \{\widehat{\mathcal{V}'}(\psi), \widehat{\mathcal{V}'_e}(\epsilon)\}$.*

Proof. By induction on the derivation of $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$.

If the last step is (skip), (assume), (assert), or (throw), then the proof is straightforward.

If the last step is (assign), then the proof is similar to the respective case in the proof of Proposition 3.23.

If the last step is (seq), we have:

$$\begin{aligned}\mathfrak{T}(\mathcal{V}, C_1\,;\, C_2) = (\,\mathcal{V}'', \mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e),\ & \mathbf{try}\, C'_1\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}'_e, \mathcal{V}''_e)\,;\, \mathbf{throw\, hc}\,; \\ & \mathbf{try}\, C'_2\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}''_e, \mathcal{V}'_e)\,;\, \mathbf{throw\, hc}\,), \\ (\mathcal{V}', \mathcal{V}'_e, C'_1) = \mathfrak{T}(\mathcal{V}, C_1),\ \mathcal{V}' \neq \bot,\ & \text{and } (\mathcal{V}'', \mathcal{V}''_e, C'_2) = \mathfrak{T}(\mathcal{V}', C_2)\end{aligned}$$

Let $C = C_1\,;\, C_2$ and $C' = \mathbf{try}\, C'_1\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}'_e, \mathcal{V}''_e)\,;\, \mathbf{throw\, hc}\,;\, \mathbf{try}\, C'_2\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}''_e, \mathcal{V}'_e)\,;\, \mathbf{throw\, hc}$. The goal is to prove that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, C'\, \{\widehat{\mathcal{V}''}(\psi), \widehat{\mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e)}(\epsilon)\}$. Note that from $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi, \epsilon\}$, we have that $\vdash_{\mathsf{Hg}} \{\phi\}\, C_1\, \{\theta, \epsilon\}$ and $\vdash_{\mathsf{Hg}} \{\theta\}\, C_2\, \{\psi, \epsilon\}$ for some $\theta \in \mathbf{Assert}$. From $\mathfrak{T}(\mathcal{V}, C_1\,;\, C_2) = (\mathcal{V}'', \mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e), C')$, we have that $\mathfrak{T}(\mathcal{V}, C_1) = (\mathcal{V}', \mathcal{V}'_e, C'_1)$ and $\mathfrak{T}(\mathcal{V}', C_2) = (\mathcal{V}'', \mathcal{V}''_e, C'_2)$. Therefore, by IH we get that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, C'_1\, \{\widehat{\mathcal{V}'}(\theta), \widehat{\mathcal{V}'_e}(\epsilon)\}$, and with the derivation given for free by Lemma 6.11, applying the (try-catch) rule, we conclude that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathbf{try}\, C'_1\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}'_e, \mathcal{V}''_e)\, \mathbf{hc}\, \{\widehat{\mathcal{V}'}(\theta), \widehat{\mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e)}(\epsilon)\}$. Using the same strategy, it is possible to obtain that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\theta)\}\, \mathbf{try}\, C'_2\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}''_e, \mathcal{V}'_e)\, \mathbf{hc}\, \{\widehat{\mathcal{V}''}(\psi), \widehat{\mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e)}(\epsilon)\}$, and applying the (seq) rule, it follows that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, C'\, \{\widehat{\mathcal{V}''}(\psi), \widehat{\mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e)}(\epsilon)\}$.

If the last step is (try-catch), the proof is analogous to the previous case.

If the last step is (if), we have:

$$\begin{aligned}\mathfrak{T}(\mathcal{V}, \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}) = (\,& \mathsf{sup}(\mathcal{V}', \mathcal{V}''), \mathsf{sup}(\mathcal{V}'_e, \mathcal{V}''_e), \\ & \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then\ try}\, C'_1\,;\, \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}'_e, \mathcal{V}''_e)\,;\, \mathbf{throw\, hc} \\ & \mathbf{else\ try}\, C'_2\,;\, \mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}''_e, \mathcal{V}'_e)\,;\, \mathbf{throw\, hc\ fi}\,), \\ (\mathcal{V}', \mathcal{V}'_e, C'_1) = \mathfrak{T}(\mathcal{V}, C_1),\ & \text{and } (\mathcal{V}'', \mathcal{V}''_e, C'_2) = \mathfrak{T}(\mathcal{V}, C_2)\end{aligned}$$

Let $C = \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}$ and $C' = \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then\ try}\, C'_1\,;\, \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}'_e, \mathcal{V}''_e)\,;\, \mathbf{throw\, hc\ else\ try}\, C'_2\,;\, \mathsf{mrg}(\mathcal{V}'', \mathcal{V}')\, \mathbf{catch}\, \mathsf{mrg}(\mathcal{V}''_e, \mathcal{V}'_e)\,;\, \mathbf{throw\, hc\ fi}$. Since from the hypothesis we have that $\vdash_{\mathsf{Hg}} \{\phi \wedge b\}\, C_1\, \{\psi, \epsilon\}$ and $\mathfrak{T}(\mathcal{V}, C_1) = (\mathcal{V}', \mathcal{V}'_e, C'_1)$, by IH we obtain $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi \wedge b)\}\, C'_1\, \{\widehat{\mathcal{V}'}(\psi), \widehat{\mathcal{V}'_e}(\epsilon)\}$. With the derivation given for free by Lemma 6.11, applying (seq) we obtain $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi \wedge b)\}\, C'_1\,;\, \mathsf{mrg}(\mathcal{V}', \mathcal{V}'')\, \{\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(\psi), \widehat{\mathcal{V}'_e}(\epsilon)\}$, and then by applying (try) with the derivation given for free by Lemma 6.11 we conclude that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi \wedge$

$b)\}$ **try** C_1' ; $\mathsf{mrg}(\mathcal{V}',\mathcal{V}'')$ **catch** $\mathsf{mrg}(\mathcal{V}_e',\mathcal{V}_e'')$; **throw hc** $\{\widehat{\mathsf{sup}(\mathcal{V}',\mathcal{V}'')}(\psi),\widehat{\mathsf{sup}(\mathcal{V}_e',\mathcal{V}_e'')}(\epsilon)\}$. Applying the same strategy it is possible to obtain that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi\wedge\neg b)\}$ **try** C_2' ; $\mathsf{mrg}(\mathcal{V}'',\mathcal{V}')$ **catch** $\mathsf{mrg}(\mathcal{V}_e'',\mathcal{V}_e')$; **throw hc** $\{\widehat{\mathsf{sup}(\mathcal{V}',\mathcal{V}'')}(\psi),\widehat{\mathsf{sup}(\mathcal{V}_e',\mathcal{V}_e'')}(\epsilon)\}$ and then we finish by applying (if).

If the last step is (while), then we have:

$$\begin{aligned}\mathfrak{T}(\mathcal{V},\textbf{while } b \textbf{ do } \{\theta\}\, C_1 \textbf{ od}) = (\,\mathcal{V}'',\mathcal{V}_e'',\ & \textbf{assert}\,\widehat{\mathcal{V}}(\theta)\,;\,\textbf{assume}\,\widehat{\mathcal{V}'}(\theta)\,;\\ & \textbf{if}\ \widehat{\mathcal{V}'}(b)\ \textbf{then}\ C_1'\,;\,\textbf{assert}\,\widehat{\mathcal{V}''}(\theta)\,;\,\textbf{assume}\,\bot\\ & \textbf{else}\ \mathsf{mrg}(\mathcal{V}',\mathcal{V}'')\ \textbf{fi}\,),\\ \mathcal{V}' = \mathcal{V}[x\mapsto\mathcal{V}(x)+1 \mid x\in\mathsf{Asgn}(C_1)],\ \text{and}\ (\mathcal{V}'',\mathcal{V}_e'',C_1') = \mathfrak{T}(\mathcal{V}',C_1)\end{aligned}$$

From the hypothesis it also holds that $\vdash_{\mathsf{Hg}} \{\theta\wedge b\}\,C_1\,\{\theta,\epsilon\}, \models \phi\to\theta$, and $\models \theta\wedge\neg b\to\psi$. By applying (assert) we have $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}$ **assert** $\widehat{\mathcal{V}}(\theta)\,\{\top,\widehat{\mathcal{V}_e''}(\epsilon)\}$ because $\models\phi\to\theta$ and thus $\widehat{\mathcal{V}}(\phi)\to\widehat{\mathcal{V}}(\theta)$, and by applying (assume) we have $\vdash_{\mathsf{Hg}} \{\top\}$ **assume** $\widehat{\mathcal{V}'}(\theta)\,\{\widehat{\mathcal{V}'}(\theta),\widehat{\mathcal{V}_e''}(\epsilon)\}$. Therefore, by applying (seq) we have that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}$ **assert** $\widehat{\mathcal{V}}(\theta)$; **assume** $\widehat{\mathcal{V}'}(\theta)\,\{\widehat{\mathcal{V}'}(\theta),\widehat{\mathcal{V}_e''}(\epsilon)\}$.

By IH, we have that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\theta\wedge b)\}\,C_1'\,\{\widehat{\mathcal{V}''}(\theta),\widehat{\mathcal{V}_e''}(\epsilon)\}$, and since we have $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}''}(\theta)\}$ **assert** $\widehat{\mathcal{V}''}(\theta)$; **assume** $\bot\,\{\widehat{\mathcal{V}''}(\psi),\widehat{\mathcal{V}_e''}(\epsilon)\}$, because $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}''}(\theta)\}$ **assert** $\widehat{\mathcal{V}''}(\theta)\,\{\top,\widehat{\mathcal{V}_e''}(\epsilon)\}$, and $\vdash_{\mathsf{Hg}}$ $\{\top\}$ **assume** $\bot\,\{\widehat{\mathcal{V}''}(\psi),\widehat{\mathcal{V}_e''}(\epsilon)\}$, we can apply (seq) to obtain $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\theta\wedge b)\}\,C_1'$; **assert** $\widehat{\mathcal{V}''}(\theta)$; **assume** $\bot\,\{\widehat{\mathcal{V}''}(\psi),\widehat{\mathcal{V}_e''}(\epsilon)\}$. Note that $\mathsf{sup}(\mathcal{V}',\mathcal{V}'') = \mathcal{V}''$ and $\widehat{\mathcal{V}'}(\theta)\wedge\widehat{\mathcal{V}'}(b)\to\widehat{\mathcal{V}'}(\psi)$ because $\models\theta\wedge\neg b\to\psi$. Therefore, by Lemma 6.11, and Lemma 5.4 we have that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\phi)\wedge\widehat{\mathcal{V}'}(b)\}$ $\mathsf{mrg}(\mathcal{V}',\mathcal{V}'')\,\{\widehat{\mathcal{V}''}(\psi),\widehat{\mathcal{V}_e''}(\epsilon)\}$. Applying the (if) rule we obtain $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\theta)\}$ **if** $\widehat{\mathcal{V}'}(b)$ **then** C_1' ; **assert** $\widehat{\mathcal{V}''}(\theta)$; **assume** $\bot$ **else** $\mathsf{mrg}(\mathcal{V}',\mathcal{V}'')$ **fi** $\{\widehat{\mathcal{V}''}(\psi),\widehat{\mathcal{V}_e''}(\epsilon)\}$. We finalize by applying the (seq) rule. □

We can now conclude that the translation is indeed a valid SA translation according to Definition 5.8.

Corollary 6.22. *The function $\mathfrak{T}$ presented in Figure 6.1 is an SA translation in the sense of Definition 5.8.*

Proof. Follows directly from Proposition 6.21 and Corollary 6.20. □

6.5 Related Work

The translation in this chapter differs from the translation of Chapter 3 mainly because it translates iterating programs into non-iterating SA programs. What is more, the source programming language may contain assume, assert and exception commands as opposed to the While language of Chapter 3. We should refer once more that it is optional for the source language to include assume and assert commands, since they may be present exclusively in the intermediate language just to encode loops. This is sufficient if one just wants to check the validity of Hoare triples annotated with loop invariants.

The idea of encoding loops through assumes and asserts was already used in the context of tools like ESC/Modula-3 [43] and was later inherited by tools like ESC/Java [51], Boogie [10] and Why3 [50]. Similarly, the translation of programs into SA is also not new. It was introduced

by Flanagan and Saxe [52] and used in the context of different tools such as ESC/Java and Boogie. Also in the context of model checking the use of SSA is a common practice, as already mentioned in Section 2.6. Basically, both the removal of iteration and the translation into SA have been seen as two different stages of the generation of verification conditions: in a first stage loops were removed through the use of a command that assigns non-deterministic values to variables, and in a second stage, the non-iterating program was converted into SA form.

Even though Flanagan and Saxe [52] have shown that the second stage preserves the weakest-precondition semantics, as far as we know none of these stages has been shown to be sound or complete with respect to a program semantics. In particular (and again, as far as we know) it has never been shown that the translation preserves the validity of loop invariants. Our translation, as opposed to the works mentioned above, generates directly a non-iterating SA program, and is shown to be sound and complete with respect to a program semantics. In particular this allows us to conclude that the translation preserves the validity of annotated triples and that it does not translate invalid triples into valid triples.

Chapter 7

Evaluation and Experiments

With the aim of evaluating empirically the contributions of the previous chapters, we have put them into practice in different ways. As a first approach, in order to test and validate the single-assignment translations and the VCGens in practice, we created a prototype in Haskell (available from `https://bitbucket.org/belolourenco/while-lang`). The prototype implements a parser[1] for While programs possibly containing annotated loops, assumes, asserts and exceptions, applies a selected single-assignment translations, and then generates a set of VCs using a selected VCGen. Either the SA translation and the VCGen can be chosen by the user. Moreover, it is also possible to remove loops by unwinding them as explained in Section 2.6. The produced VCs can then be converted into Why3 goals[2] and discharged in Why3.

Since our goal is to generate VCs for real world programs, our first evaluation is done over the LLVM intermediate representation (IR) [80]. We have chosen LLVM IR firstly because it is already in SSA form, and secondly because different languages, such as C, Ada or Objective-C, can be compiled into LLVM IR, which means that a single framework can be used for multiple input languages (our work however focuses only on a subset of the C programming language). We have implemented the VCGens discussed in Chapter 4 in a tool that was initially intended for fault localization of LLVM IR, and with the resulting framework we were able to compare the VCGens empirically using a set of existing benchmarks [7, 61].

The evaluation mentioned above was done by unwinding loops, and thus it resembles a bounded model checker of software. With the aim of comparing the VCGens on a major deductive verification tool, we have implemented them in Why3, done several experiments, and compared our implementation with the Why3 native VCGens. In particular we explore the worst-cases (in terms of VC size) of each VCGen in Why3, and present in the appendix sketches of the programs that lead to the worst-case in each VCGen.

The following section presents the empirical evaluation of the VCGens developed in the context of LLVM IR and Section 7.2 presents several experiments with the Why3 native VCGens and our implementations.

[1]Based in `https://github.com/davnils/while-lang-parser`

[2]Encoding done using `https://hackage.haskell.org/package/why3`

7.1 Experimental Evaluation with SNIPER-VCGen

The cube sets the basis for a thorough comparative evaluation of the VCGens with respect to different criteria, using a representative set of benchmark programs. Since no existing tool implements all the algorithms, we developed one on top of SNIPER [79] to analyze the effect of solving VCs generated by different VCGens. This tool, baptized SNIPER-VCGen (available from `http://alfa.di.uminho.pt/~belolourenco/sniper-vcgen.html`), targets the verification of iteration-free LLVM intermediate representation [80]. The use of LLVM as intermediate language is convenient for our purposes, since loop expansion and optimizations involving constant propagation and simplification are readily implemented by the LLVM toolset prior to VC generation. Naturally, an empirical comparison of VCGens requires the generation of VCs of substantial size, which we obtain by expanding loops.

The resulting formulas are encoded in the SMT-LIB v2 language [14], and are then directly sent to different solvers for checking in the QF_AUFLIA logic, which supports quantifier-free formulas, (unbounded) integer arithmetic, and integer arrays. In our experiments we used the Z3 [42] (v. 4.4.1), CVC4 [12] (v. 1.4), and MathSAT [27] (v. 5.3.10) SMT solvers, to evaluate whether our VCGen comparison results hold consistently across a diverse set of solvers, or whether they are solver-dependent. The evaluation was performed on a 1.7 Ghz Intel Core i5 MacBook Air, with 4 GB of RAM and running OS X Yosemite. The given time values represent wall clock time (seconds), and correspond to the solving time only, taken by the solver to run on the SMT-LIB files, excluding generation of the VCs and encoding into SMT-LIB language.

For the first part of the evaluation, we take the program of Example 4.10, translate it to LLVM, unwind loops N times, and generate a set of VCs using one of the VCGens mentioned in Chapter 4. We have measured experimentally the size and the solving time of the corresponding SMT problem for each VCGen. The detailed results for $N = 100, 200, 300$ when feasible (the tool times out after 5 minutes) are shown in Figures 7.1 and 7.2.

The file size data supports the asymptotic analysis that was performed in Chapter 4. With respect to the VCGens based on CNF, the SSA optimization produces a dramatic decrease on the file size when partial contexts are used, in particular if no asserts are included. For VCSP, the file size for both global and partial contexts without asserts are similar to those obtained for SSA, but now the inclusion of assertions as lemmas in the context has only marginal impact on the file size. With the VCLin optimization, VC size becomes linear in all cases; the data confirms that this is by far the most efficient of all the evaluated VCGens regarding file size.

As to the solver execution time, $\mathsf{VCCNF}^{\mathsf{G}}$ performs much better than the remaining VCGens based on CNF (more than 10 times faster than the partial contexts $\mathsf{VCCNF}^{\mathsf{P}}$). The SSA optimization improves solving time only marginally with global contexts, but with partial contexts reduces it to roughly 1/3 (or to 1/2 if asserts are included in contexts). Nonetheless, $\mathsf{VCSSA}^{\mathsf{G}}$ beats $\mathsf{VCSSA}^{\mathsf{P}}$ (resulting in roughly 5 times faster solving). With the SP VCGens again the use of a global context results in several times faster solving than partial contexts: $\mathsf{VCSP}^{\mathsf{G}}$ stands

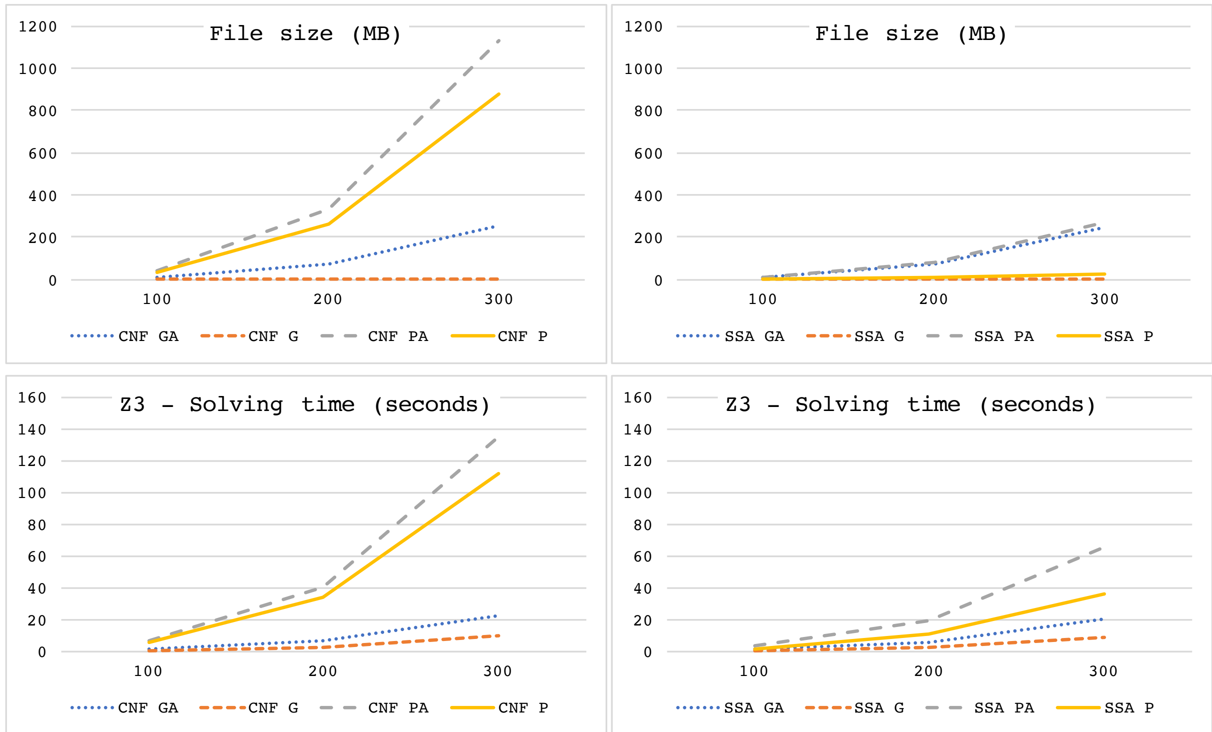


Figure 7.1: Results for the program of Example 4.10: VCCNF (left) and VCSSA (right)

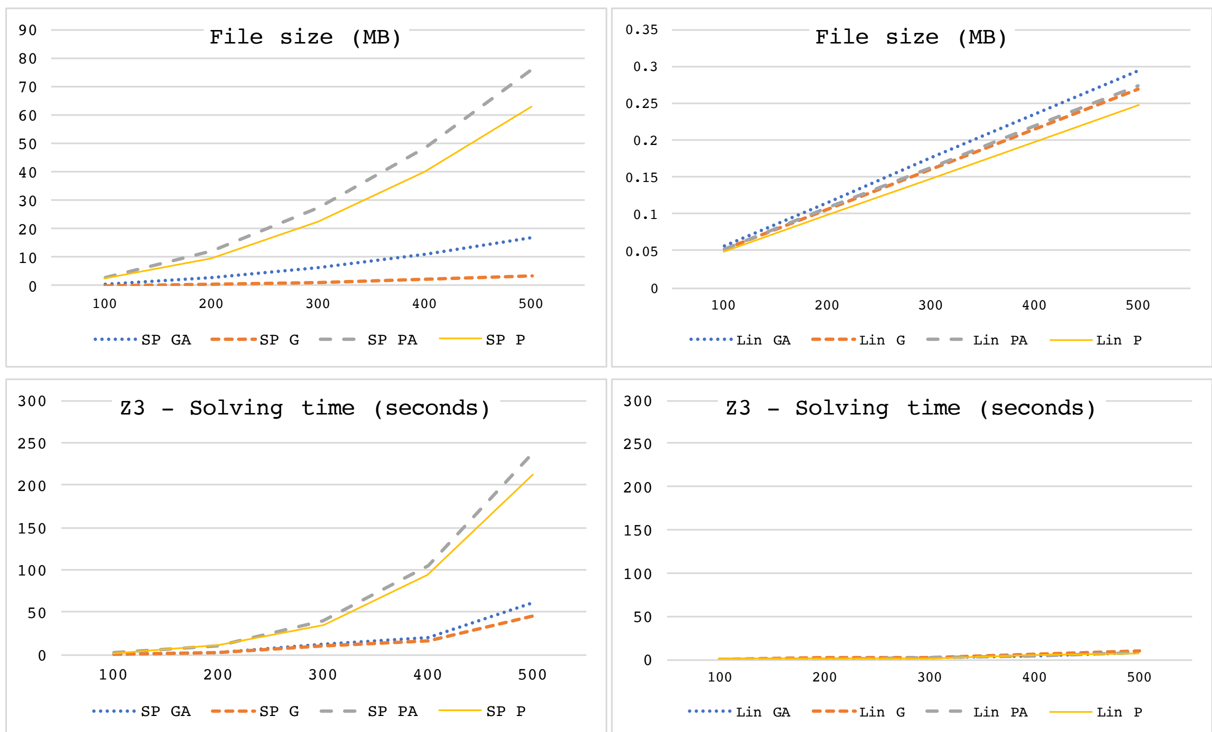


Figure 7.2: Results for program of Example 4.10: VCSP (left) and VCLin (right)

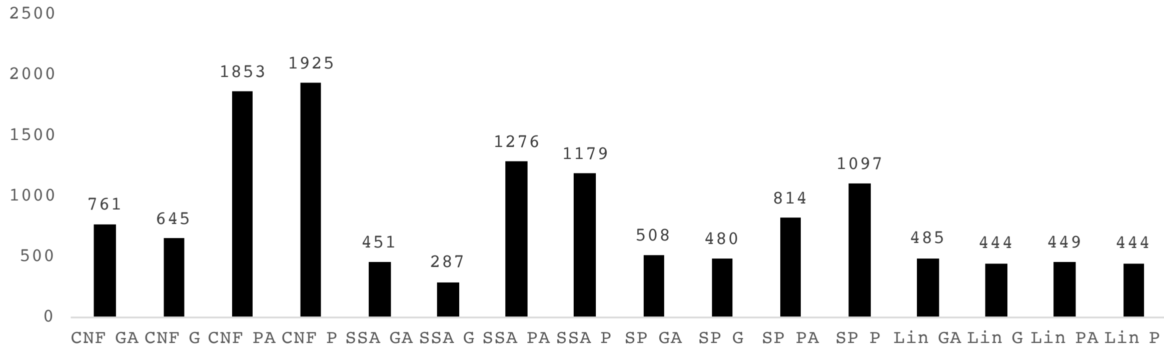


Figure 7.3: Overall Z3 solving time (seconds) for benchmark programs

roughly between $\mathsf{VCCNF}^\mathsf{G}$ and $\mathsf{VCSSA}^\mathsf{G}$, and VCSP^P has similar performance to $\mathsf{VCSSA}^\mathsf{P}$. Finally, and as expected from the analysis of VC size, VCLin leads to significantly more efficient solving than any other VCGen. Although in all other cases it is preferable to use a global context, VCLin performs slightly better with partial contexts.

Although in theory it seems that the example program will expose the benefits of using asserts as lemmas, this kind of reasoning is misleading. In practice, when an automated solver is used, there is no way to influence the proof, and asserts are best left out of contexts, since they result in worse performance (with the exception of VCLin, for which adding asserts does not seem to affect performance).

In addition to the example program of Example 4.10, we evaluated the VCGens using a suite consisting of several case studies from the Eureka and InvGen benchmarks [7, 61], that have been used before to test, validate and evaluate other program verification tools. These programs are algorithmically more complicated, and therefore allow us to compare the VCGens in a more realistic setting. In particular, the Eureka benchmark was created with the aim of assessing the scalability of software model checking tools, with programs of increasing complexity. Nonetheless, since assert statements are scarce in Eureka programs we also use case studies from the InvGen benchmarks, which are rich in asserts and allow us to evaluate the effect of including them in the context (one of the dimensions of the VCGen cube). The Eureka benchmark contains both correct and faulty annotated programs, while the InvGen benchmark contains only correct annotated programs. The properties found in both sets of programs rely on Boolean expressions of the C language, and therefore do not contain any quantifiers.

For the correct *bounded* Eureka programs, loops were expanded the required number of times, and unwinding assertions were introduced to ensure that the expansion was sufficient. For the other programs, loops were expanded a reasonable number of times, with unwinding assumptions introduced to prevent executions with more iterations from being considered. Figure 7.3 shows the overall solving time in Z3 for the VCs generated from the full set of programs, and Table 7.1 shows the results separated by benchmark. The column labeled *Correct* (resp. *Faulty*) refers to the total solving time for all VCs generated from the correct Eureka programs (resp. faulty

	Eureka						InvGen		Total	
VCGen	*Correct (s)*	#	*Faulty (s)*	#	*Total (s)*	#	*Total (s)*	#	*(s)*	#
$\mathsf{VCCNF}^{\mathsf{GA}}$	464	2	146	3	610	5	150	1	760	6
$\mathsf{VCCNF}^{\mathsf{G}}$	417	2	126	2	543	4	101	0	644	4
$\mathsf{VCCNF}^{\mathsf{PA}}$	842	1	447	1	1289	2	564	0	1853	2
$\mathsf{VCCNF}^{\mathsf{P}}$	850	1	520	2	1370	3	553	0	1923	3
$\mathsf{VCSSA}^{\mathsf{GA}}$	183	5	149	4	332	9	117	2	449	11
$\mathsf{VCSSA}^{\mathsf{G}}$	**92**	7	123	**27**	**215**	34	70	2	**285**	36
$\mathsf{VCSSA}^{\mathsf{PA}}$	463	5	376	3	839	8	436	0	1275	8
$\mathsf{VCSSA}^{\mathsf{P}}$	417	4	368	2	785	6	392	0	1177	6
$\mathsf{VCSP}^{\mathsf{GA}}$	325	2	111	8	436	10	70	2	506	12
$\mathsf{VCSP}^{\mathsf{G}}$	301	10	114	10	415	20	63	1	478	21
$\mathsf{VCSP}^{\mathsf{PA}}$	437	1	129	8	566	9	247	1	813	10
$\mathsf{VCSP}^{\mathsf{P}}$	559	1	244	3	803	4	291	0	1094	4
$\mathsf{VCLin}^{\mathsf{GA}}$	328	18	109	5	437	23	46	12	483	35
$\mathsf{VCLin}^{\mathsf{G}}$	291	**35**	109	9	399	**44**	43	**14**	442	**58**
$\mathsf{VCLin}^{\mathsf{PA}}$	331	11	**76**	21	408	32	41	9	449	41
$\mathsf{VCLin}^{\mathsf{P}}$	312	7	92	10	404	17	**39**	12	443	29

Table 7.1: Z3 solving time for benchmark programs (time in seconds)

programs). The columns marked with *Total* refer to the total solving time for each benchmark set of programs; the sum of both is also shown in the final column. Columns marked with # show the number of times that each VCGen performed *better than the others*. The detailed results for each program can be found in the tool's webpage.

A considerable number of Eureka programs use *assumes* and *asserts* simply as *pre-* and *postconditions*, as opposed to InvGen programs which are densely populated with asserts. This is reflected in the table: the solving time difference between VCGens based on partial and global contexts is greater in the InvGen benchmark than it is in the Eureka benchmark. The results also confirm the trends identified previously (global contexts lead to faster solving; placing asserts in contexts increases solving time), but allow for an exception to be identified: in all three data sets (Eureka Correct, Eureka Faulty, and InvGen), $\mathsf{VCSP}^{\mathsf{PA}}$ performs better than $\mathsf{VCSP}^{\mathsf{P}}$ (nonetheless, $\mathsf{VCSP}^{\mathsf{G}}$ behaves much better).

As before, the solving times for $\mathsf{VCSP}^{\mathsf{G}}$ and $\mathsf{VCCNF}^{\mathsf{G}}$ are close, with $\mathsf{VCSP}^{\mathsf{G}}$ performing consistently better in all three benchmark datasets. The optimizations improve performance: $\mathsf{VCSSA}^{\mathsf{G}}$ performs better than $\mathsf{VCCNF}^{\mathsf{G}}$ in all three datasets, and the VCLin variant outperform all the others. Note that this observation is not immediately visible in the Eureka Correct

	MathSAT						CVC4					
VCGen	*Eureka*	#	*Invgen*	#	*Total*	#	*Eureka*	#	*Invgen*	#	*Total*	#
VCCNF^{GA}	6911	6	356	1	3355	1	5169	4	1570	0	5117	1
VCCNF^{G}	6723	**30**	230	4	3123	20	4347	9	1040	0	4085	1
VCCNF^{PA}	8147	7	1376	0	4907	0	8677	2	4800	0	10595	0
VCCNF^{P}	8368	7	1265	0	4725	1	8507	1	4693	0	10210	0
VCSSA^{GA}	5906	7	436	0	2926	5	4073	8	1145	2	4130	5
VCSSA^{G}	**5683**	15	312	11	**2672**	19	**3319**	26	**531**	**29**	**3049**	**36**
VCSSA^{PA}	7911	7	1176	1	4306	6	9391	12	2787	0	8528	4
VCSSA^{P}	7960	7	1053	0	4078	4	9247	9	2120	1	7784	4
VCSP^{GA}	7338	14	307	5	3400	13	4928	9	1036	3	4307	10
VCSP^{G}	7281	15	247	3	3257	10	4835	14	907	1	4155	11
VCSP^{PA}	7704	5	825	0	3722	2	4638	16	3195	1	6454	7
VCSP^{P}	8260	6	892	0	3875	1	4979	7	3650	1	6946	2
VCLin^{GA}	7120	13	163	5	2963	13	4305	23	891	6	3600	19
VCLin^{G}	7131	24	**159**	**12**	2986	**27**	4499	**28**	850	5	3745	27
VCLin^{PA}	7227	18	474	4	3385	14	4534	11	1115	0	4097	8
VCLin^{P}	7142	21	477	7	3329	15	4908	17	1044	4	4232	11

Table 7.2: Overall solving time results for benchmark programs in MathSAT and CVC4 (time in seconds)

data, because the dataset contains an outlier program that biases the data heavily in favor of VCSSA. If this program is removed, the supremacy of VCLin is restored. Finally, it remains to discuss which variant of the VCLin performs better. The aggregate solving time for the benchmarks is inconclusive (with G, PA and P resulting in similar times), but if we take the number of programs in which each VCGen outperforms the others, the clear winner is VCLin^{G} (58 programs against 41 for VCLin^{PA}). This is in accordance with the general trend that using a global context without asserts seems to be the best choice.

The analysis of the data obtained with CVC4 and MathSAT in Table 7.2 confirms the general trends described above, but reveals some points that are solver-specific. In particular it reinforces the fact that VCSP^{PA} performs better than VCSP^{P} *with all solvers*; and with MathSAT several programs in the benchmarks timeout with all VCGens except VCSSA, which causes the latter to have the best aggregate solving time, supplanting VCLin. It is also interesting to note that VCSSA^{G} performs better with CVC4 than all other VCGens in most case studies.

7.2 Experiments with Why3

Our second experimental analysis was performed with the Why3 verification tool set [50], which is entirely based on deductive reasoning with invariants and contracts, and does not support loop unrolling. The results that will be presented in this section were obtained with a working branch of Why3[3] and the modifications we implemented on the tool are available in the following repository: `https://bitbucket.org/belolourenco/why3_claudio`. We will refer to this modified version as Why3-vcgens. With respect to the SMT solvers, our experiments used the Alt-Ergo version 1.30 [21] and Z3 version 4.5.0 [42]. The experiments were performed on a 4.00GHz Intel Core i7 iMAC with 8GB of RAM.

We should start by mentioning that since the tool is normally used to prove the functional correctness of complicated algorithms it has traceability as an important feature. When a VC cannot be proved, having a higher level of traceability helps identifying the part of the specification that cannot be proved. In particular the VCGen that is used in the tool by default is based on weakest-precondition computations, and can generate VCs of exponential size (see Section 2.4), and has advantages from the point of view of traceability: the Why3 graphical interface is able to highlight execution paths corresponding to selected VCs. Note however that the explosion on the size of the VCs in Why3 come exclusively from the duplication of the postcondition (e.g. Example 2.15) and not from the duplication of variables (e.g. Example 2.16): the VCGen is not based on variable substitution, such as the one in Definition 2.15, but instead it uses different version for the variables in a similar way to what happens when a single-assignment form is used. Nevertheless, Why3 also implements and makes available (through a command-line switch) a VCGen based on strongest-postcondition computations, which corresponds to $\mathsf{VCLin}^{\mathsf{PA}}$ in our framework. For simplification purposes, in the rest of this section we will refer to the Why3 VCGen based on weakest-precondition as `w3wp` and to the one based on strongest-postcondition as `w3sp`.

Let us note the fact that the Why3 VCGens start by generating only a single VC and just then allow the user to split it into a set of 'single-goal' VCs. These VCs can then be split even further to reach the desired level of traceability. If the VCGen based on weakest-precondition is used, it is possible to obtain separate VCs for each branch in conditional constructs. When the VC obtained with `w3sp` is split, it originates a set of VCs that are analogous to those generated by $\mathsf{VCSP}^{\mathsf{PA}}$. Note however that Why3 provides several theories that are required for proving the VCs, therefore whenever a VC is split, the axiomatization of such theories must be replicated as well, so they appear in the context of each VC. In what follows if split is applied to a VC obtained with `w3wp` (resp. `w3sp`) we will denote the corresponding 'splitting' VCGen as `w3wp-split` (resp. `w3sp-split`).

The Why3 programming language has two different commands that are similar to the assert command of the previous chapters (in addition to the typical invariants and contracts of deductive verification). The first command is called *assert* and should be used whenever one wants to annotate the code to verify the validity of a property at a given point, and then use it in the

[3] Our copy of Why3 is based on the commit SHA: 4b3854a2c9fdc7fc753a01ed8936a3f3952f88f8

$$\begin{array}{rcl}
\multicolumn{3}{l}{\mathtt{w3wp} : \mathbf{AComm}^{\text{SA}} \times \mathbf{Assert}^{\text{SA}} \rightarrow \mathbf{Assert}^{\text{SA}}} \\
\mathtt{w3wp}(\mathbf{skip}, \psi) & = & \psi \\
\mathtt{w3wp}(x := e, \psi) & = & x = e \rightarrow \psi \\
\mathtt{w3wp}(\mathbf{assume}\,\theta, \psi) & = & \theta \rightarrow \psi \\
\mathtt{w3wp}(\mathbf{assert}\,\theta, \psi) & = & \theta \wedge \psi \\
\mathtt{w3wp}(C_1\,;\,C_2, \psi) & = & \mathtt{w3wp}(C_1, \mathtt{w3wp}(C_2, \psi)) \\
\mathtt{w3wp}(\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, \psi) & = & (b \rightarrow \mathtt{w3wp}(C_1, \psi)) \wedge (\neg b \rightarrow \mathtt{w3wp}(C_2, \psi)) \\
\mathtt{w3wp}(\mathbf{check}\,\theta, \psi) & = & \theta
\end{array}$$

Figure 7.4: Description of Why3 default VCGen based on weakest-precondition

context of the subsequent properties. Basically, it can be seen as the assert command of our language in a VCGen that introduces asserts in the context (the variants denoted by PA and GA). The other command is called *check* and is used exclusively to verify if a property holds at the current point of the program, being not used in the context of subsequent properties. It can be seen as our assert command, in the VCGens that do not use asserts as lemmas (the variants denoted by P and G). So, Why3 leaves it to the user to choose the properties to be used as lemmas, which stands in contrast with our methodology (we just have a single command for checking properties and then depending on the VCGen that is chosen, those properties are inserted in the context or not).

In order to make the discussion of this section easier to follow, let us present the idea behind `w3wp`. First of all, note that Why3 does not rely on the translation of programs into SA form: the tool's workflow relies on different intermediate forms, but only during the generation of verification conditions it produces unique identifiers on the fly for the variables in the program. Nonetheless, the whole idea, and the generated VCs, are basically the same as if single-assignment form is used. The `w3wp` VCGen, adapted to the SA setting, is outlined in Figure 7.4. The function receives a command and a postcondition, and returns the VC based on the weakest-precondition. Since we are assuming that the program is in SA form, the assignment consists simply of an implication, where the left hand-side captures the assignment and the right-hand side is the received postcondition. Note that it does not rely on variable substitution. The check command is a special case, and only makes sense due to a translation that Why3 performs before generating the VCs: basically commands of the form $\mathbf{check}\,\theta\,;\,C$ are translated into $\mathbf{check}\,\theta \,\square\, C$, where $\square$ is the choice command of Section 2.4. Therefore, properties that are found in check commands will not influence subsequent properties as explained above. Finally note that the if command originates a duplication of the postcondition, which originates the explosion in the size of the generated VCs, as explained before.

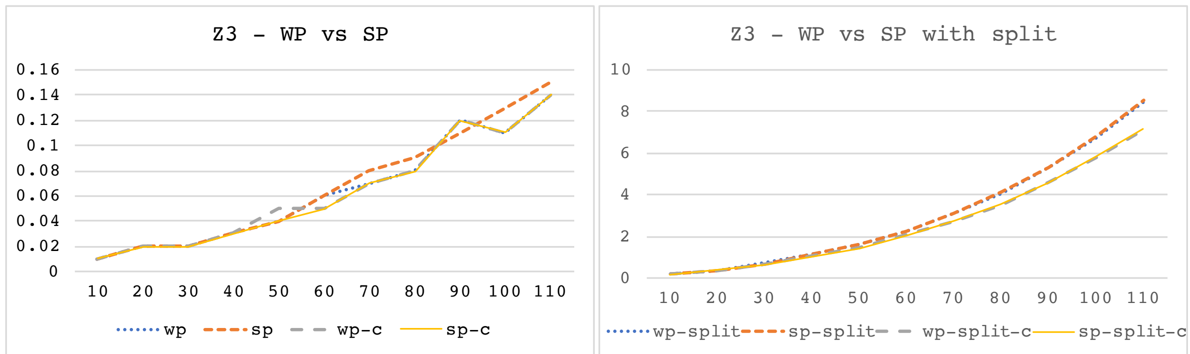


Figure 7.5: Solving time (in seconds) for the VCs obtained by unwinding the program of Example 4.10

Why3 default VCGens and Quantifiers Let us start by comparing the Why3 VCGens with the program of Example 4.10. For this to be possible we implemented loop unrolling in Why3. This is done in one of the intermediate forms before the VCs are generated, and the idea is pretty straightforward: loops are unwound a certain number of times; if invariants are annotated in the program we insert them as asserts at the beginning of the unwound code and after each iteration; and after the last iteration an unwinding assumption or assertion (see Section 2.6) is inserted, depending on the user's choice. Loop unrolling is available in Why3-vcgens through the command line switch `--bmc-assume N`, to unroll loops N times and use an unwinding assumption, or `--bmc-assert N`, to unroll loops N times and use an unwinding assertion.

The program of Example 4.10 was implemented in a Why3 module (see Appendix B) and can be found in the Why3-vcgens repository (as is the case for all the programs presented throughout this section). To evaluate whether or not having proved properties in the context reduces the solving time, we wrote two versions of the program: one using the Why3 assert (in module `Tassert`) and another using the Why3 check command (in module `Tcheck`). Note that in our setting of the previous chapters this corresponds basically to compare the variants P (resp. G) with PA (resp. GA). Our first experiment consists in unwinding the loop a number of times and observing the solving time for the generated VCs, as well as the solving time for the result of applying the split option.

Our first experiment consists in unwinding loops from 10 to 110, by increments of ten. The total solving times with Z3 for the generated VCs are shown in Figure 7.5. In both charts, the x-axis shows the number of times the loop was unwound, and the y-axis the solving time. Since we have two versions of the same program (one with asserts and another with checks), we identify the VCs from the program with checks by appending a '-c' to the VCGen name. For instance, the line labeled '`w3wp-c`' corresponds to the solving time for the VCs generated from the program annotated with checks using the native Why3 VCGen based on weakest-precondition; and similarly the line labeled '`w3wp-split-c`' corresponds to splitting of these same VCs. Note that since there are two asserts in each loop, splitting the VC will originate $2 \times N$ VCs, with N the number of times the loop was unwound.

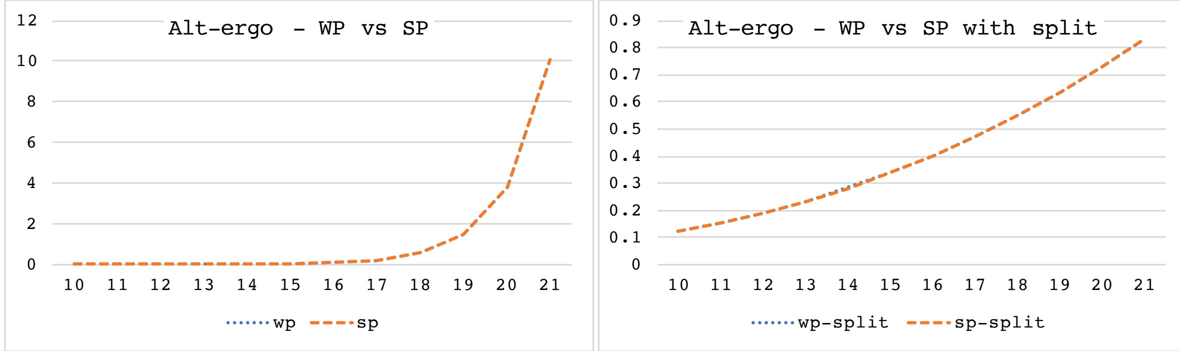


Figure 7.6: Solving time (in seconds) for the VCs obtained by unwinding the program of Example 4.10 (the line referring to `w3wp` is not visible because it coincides with `w3sp`)

The chart on the left in the figure shows that it does not make much difference on whether `w3wp` or `w3sp` is used to generate VCs. Similarly, there is no signifficant difference between using checks or asserts to annotate the properties in the program. The reason for the similarity in the solving time between `w3wp` and `w3sp` is that for this program the VCs generated are themselves very similar. In fact both VCGens generate VCs with the same shape when checks are used, and the difference when asserts are used comes only from the fact that the VCGen based on WP generates a formula of the form $\theta \wedge \psi$ for each assert θ and postcondition ψ, while the VCGen based on SP generates formulas of the form $\theta \wedge (\theta \rightarrow \psi)$, which duplicates θ.

From the chart on the right in Figure 7.5 it is possible to see that checks generate VCs that are more efficient in terms of solving time. This seems to agree with our conclusion in Section 7.1 that having asserts as lemmas in the context of the present program only increases the VC size and thus the solving time.

Let us now consider the Alt-Ergo SMT solver. For this solver, we will unroll loops from 10 to 21 by increments of one. Figure 7.6 shows the solving time for the VCs generated using `w3wp` and `w3sp` (on the left), and then using `w3wp-split` and `w3sp-split` (on the right). Once again observe that there is not much difference between using a VCGen based on weakest-precondition or strongest postcondition. We should also mention that these results are for the program annotated with asserts, but the same results are produced for the one annotated with checks. Observe that in the current scenario the split version of the VCGens is much more efficient. In fact the solving time when the VCs are not split grows exponentially, which makes the split versions of the VCGen ideal for this setting.

After some analysis we concluded that the explosion was caused by the method used by Why3 to quantify variables in the generated VC. It does so, by placing the quantifier as close as possible to the quantified variables, therefore, quantifiers (both existential and universal) appear in several places in the generated VCs. In our modified version of Why3, we changed the VCGens in such a way that variables are not quantified during VC generation; instead they are globally quantified in the end, so quantifiers are placed externally. These modified versions

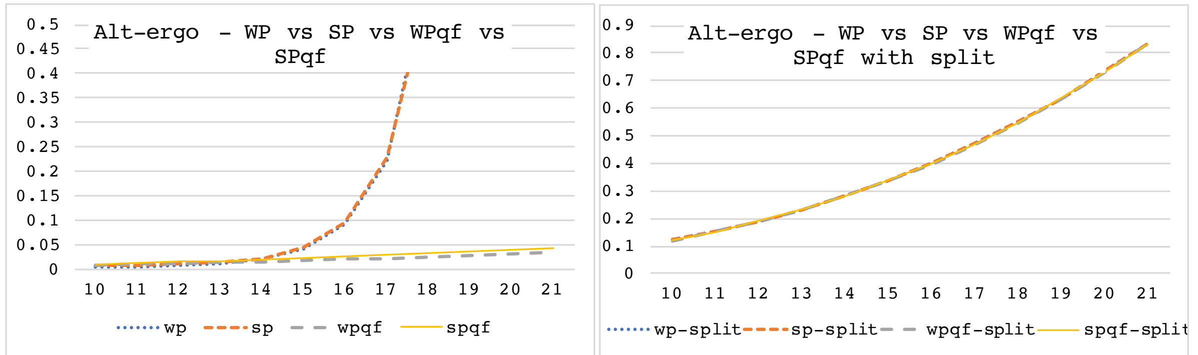


Figure 7.7: The result of moving the quantifiers to the outermost position of the VCs

of the VCGens will be denoted in what follows respectively by wpqf, spqf, wpqf-split, and spqf-split. Figure 7.7 shows that moving quantifiers to the outermost position in the VC is clearly beneficial. If VCs are split then the solving time does not change, but if they are not split then the solving time is drastically reduced, as shown on the left hand-side of the figure.

In fact this observation is not only useful in the context of Alt-Ergo but also when the Z3 solver is used. Even though in Z3 the solving time does not grow exponentially, as shown in Figure 7.8, it grows considerably when comparing to our version of the VCGens. Another difference w.r.t. Alt-Ergo is that in Z3 the solving time is also reduced when VCs are split. Lastly, it should be noted that the observations made for the results obtained with Z3 also hold consistently across CVC3 [15] and CVC4 [13] SMT solvers.

VCGen based on WP vs SP. Let us now focus on the differences between the VCGens based on WP and SP. The results that will be discussed next, hold consistently across the native Why3 VCGens and our versions, in which quantifiers are moved outside the VC, and thus, as a matter of simplification we will only discuss results obtained with the native Why3 VCGens. Moreover, since the results also hold across different solvers, we will only show results obtained with the Z3 SMT solver. To make this discussion more interesting we will add the

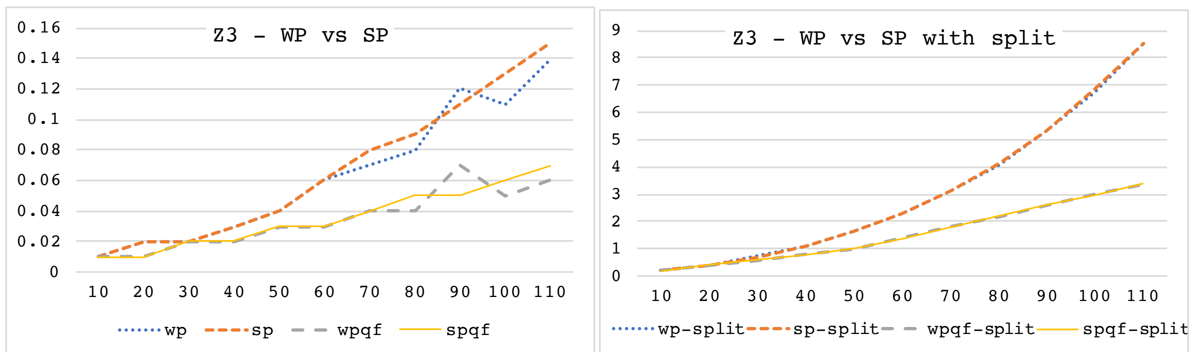


Figure 7.8: The result of moving the quantifiers to the outermost position of the VCs

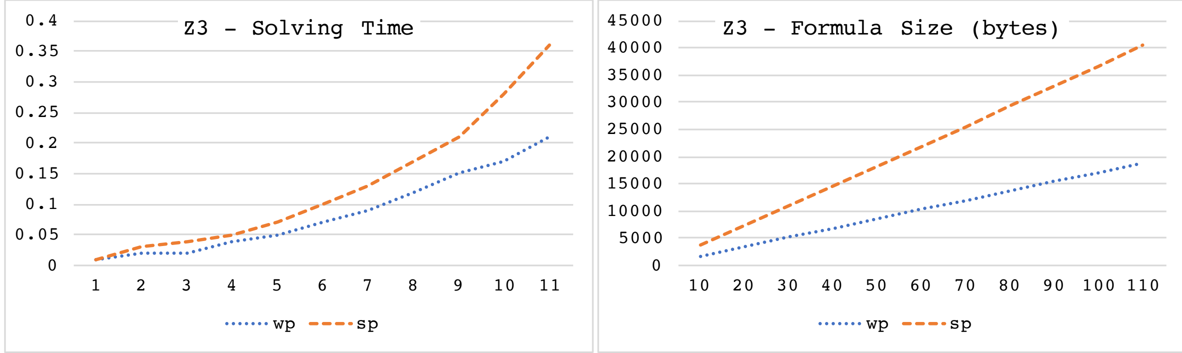


Figure 7.9: Solving time (left) and formula size (right) for the program of e Example 4.10 the postcondition $x \leq 101$

trivial postcondition $x \leq 101$ to the previous program. The resulting Why3 input program is given in Appendix B, under the module name `TassertPostcondition` for the version with asserts and `TcheckPostcondition` for the version with checks. Once again, since the results hold consistently for both versions, we only discuss the version with asserts. With respect to our version of the program of Example 4.10, we can imagine that the postcondition is added as an assert at the end of the program (this is in fact what Why3 does in the intermediate language that is used before the VCs are generated).

The charts in Figure 7.9 show the solving time using Z3 and the VC formula's size originated for this solver. The main observation is that `w3sp` generates bigger formulas, and thus less efficient VCs (in terms of solving time), than `w3wp`, which reinforces the idea that even though `w3wp` generates formulas of exponential size in the worst case, it is in many cases more efficient than `w3sp`. Let us then analyze the reasons for `w3wp` to behave better than `w3sp` for this concrete case.

In order to make the explanation easier to follow, let us consider a new program that abstracts the relevant part of the program under analysis as a set of nested if constructs followed by an assert statement with the post condition. The parts of the program that influences the difference on the solving time are the if statements containing asserts and followed by other asserts. So, let us consider the program **if** b **then** $x_1 := e_1$; **assert** θ_1 **else** $x_1 := e_2$ **fi** ; **assert** θ_2 that has exactly these ingredients. For this program, the VCGen based on SP generates the VC $(b \rightarrow x_1 = e_1 \rightarrow \theta_1) \wedge ((b \wedge x_1 = e_1 \wedge \theta_1) \vee (\neg b \wedge x_1 = e_2) \rightarrow \theta_2)$, which duplicates the encoding $x_1 = e_1$ of the assignment $x_1 := e_1$. This might not seem a big deal in this abstract case, but if we go back to our concrete running example, the encoding of the whole set of nested ifs will be duplicated. We should note however that this duplication only occurs once, and does not generate any kind of explosion.

As a matter of fact, the VCGen based on WP also duplicates formulas, more precisely it duplicate the postcondition. For the abstract program referred above, `w3wp` will generate the VC $(b \rightarrow x_1 = e_1 \rightarrow \theta_1 \wedge \theta_2) \vee (\neg b \rightarrow x_1 = e_2 \rightarrow \theta_2)$, in which the condition θ_2 occurs twice. Note that even though the condition is duplicated, the encoding of the structure of the program is preserved, and thus the replication of the postcondition is done in a controlled way. In our

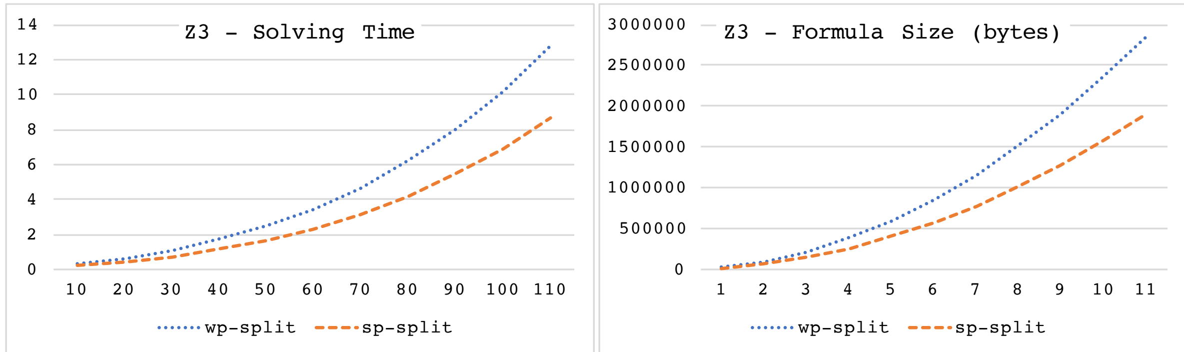


Figure 7.10: Solving time (left) and formula size (right) for the program of Example 4.10 with the postcondition $x \leq 101$

concrete running example, the postcondition will be replicated through all the execution paths that can possibly reach it, that is, it will be propagated to the innermost then branch, and then through all the else branches. The idea is that the postcondition can be reached from the innermost then branch, if all the branching conditions hold, or else through an else branch, if one of the branching conditions fails to hold.

The situation, both in terms of VC size and solving time, becomes different when VCs are split. The charts in Figure 7.10 shows the results for this scenario. It is possible to observe that in this case `w3sp` generates smaller, and thus more efficient VCs. The reason for this is not because the duplication referred above disappeared, but because of the number of VCs that result from splitting the VC. When using `w3sp`, the split will originate one VC for each assert in the program, so if we unwind the program N times, this will originate $2 \times N + 1$ VCs, that is, one for each assert and another one for the postcondition. Nonetheless, as we mentioned previously, the postcondition in `w3wp` is propagated backwards through all the else branches and also to the innermost then branch. Therefore, splitting the VC will originate $2 \times N + N + 1$ VCs, that is, one for each assert, N VCs of the else branches and another one from the innermost then branch.

VCGens from the Cube. As we have explained at the beginning of this section, Why3 does not use an implicit translation of programs into SA form. Instead it produces unique symbols for the variables in the program on the fly during the generation of VCs. To fill the gap between our work and the Why3 tool, we have implemented an explicit translation of programs into SA form, which captures what the default VCGen functions do, returning a new program in SA form instead of a VC. The VCs are then generated from this SA program.

Using this translation, we have implemented some of the VCGens discussed in Chapter 4. Since the translation produces DSA programs we were unable to implement the VCSSA family of VCGens, and due to the existence of the check and assert command in Why3, we do not distinguish between the P and PA (resp. G and GA) variants. Therefore, it is left to the user to choose which properties are (not) to be used as lemmas through the use of the assert (resp.

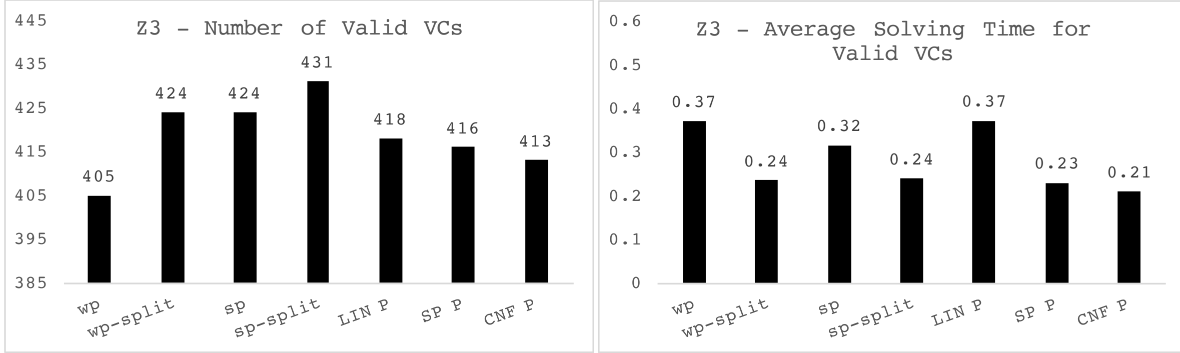


Figure 7.11: Number of valid VCs in a total of 647 (left) and average solving time in seconds for the Valid VCs (right)

check) command. Moreover, in what follows the results for the global variants will be omitted because the number of valid VCs and their solving time are always poor in terms of efficiency and traceability when compared to the partial context variants.

We have compared our VCGens with the native Why3 VCGens using the gallery of Why3 programs, and the results obtained with the Z3 SMT solver are shown in Figure 7.11. The chart on the left shows the number of functions that were shown to be valid w.r.t. their contract, and the one on the right shows the average solving time. When using `w3sp` or $\mathsf{VCLin}^\mathsf{P}$ a timeout of twenty seconds was considered for the single-generated VC, and when using `w3sp-split`, $\mathsf{VCLin}^\mathsf{P}$, or VCSP^P the timeout value was of five seconds for each VC.

It is possible to observe that `w3sp` with split is the one that generates the greatest amount of VCs that can be discharged. In total, Z3 can prove that 431 out of 647 functions are correct with respect to their contracts. The second best VCGens in terms of proved functions are `w3sp` (with no split) and `w3wp-split`, which can prove 424 functions. The Why3 default VCGen with no splitting can only prove 405 functions. Regarding our VCGens based on the translation into SA, $\mathsf{VCLin}^\mathsf{P}$ can only prove the correctness of 418 functions. For VCSP^P and $\mathsf{VCCNF}^\mathsf{P}$, this number is 416 and 413 respectively. In terms of average solving time, we note that even though $\mathsf{VCCNF}^\mathsf{P}$ is the VCGen that generates the greatest amount of VCs that are not proved correct for the given timeout, the average time is the lowest, immediately followed by VCSP^P and the native `w3wp-split` and `w3sp-split`.

Even though we are not certain about the reason for the poor performance of $\mathsf{VCLin}^\mathsf{P}$ when compared to `w3sp`, we discuss here some possibilities. One of the reasons might be because we were unable to implement some optimizations that are present in the Why3 native VCGens, such as for instance the elimination of some variables that are introduced by the Why3 intermediate representations. Another reason might be that we are quantifying variables in the outermost position of the VC, as opposed to `w3sp`: while this seems to improve the solving time in some cases, it may have the opposite effect in other cases.

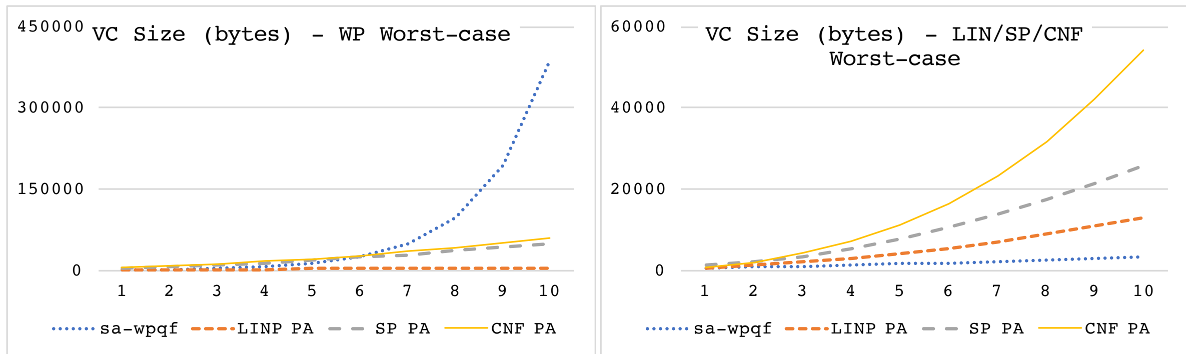


Figure 7.12: VC size growth w.r.t. the size of the respective program that causes the worst-case in WP (left) and in VCLin, $\mathsf{VCSP}^{\mathsf{PA}}$, and VCCNF (right)

Exploring the Worst Cases. Let us now explore the worst-case scenario for each VCGen in terms of the size of the generated VCs w.r.t. the input program. The VCGens used in this part of the experiment are based on the translation of programs into SA, and therefore, quantifiers always appear in the outermost position of the VC. Moreover, the sole purpose of the programs considered in this part of the thesis is to explore the growth of the VCs, and thus they do nothing in particular (in fact most of the generated VCs are not even valid). We first consider the worst-case for programs with no exceptions and then for programs containing exceptions.

Our first program together with a VCGen based on weakest-precondition originates an explosion in the size of the VCs. The program is shown in Appendix B under the module name WPExplosion and, in order to explore the growth of the VC, we considered different versions of the same program replicating the initial structure. To make the comparison fair, the results that will be presented here do not correspond to the native Why3 VCGen, but to our own implementation over the SA intermediate form (as a matter of fact, the overall observations hold across the native Why3 VCGens).

The chart in Figure 7.12 (left) shows the growth of the VCs generated by our version of sa-wpqf, $\mathsf{VCLin}^{\mathsf{PA}}$, $\mathsf{VCSP}^{\mathsf{PA}}$ and $\mathsf{VCCNF}^{\mathsf{PA}}$. The chart seems to indicate that the size of the VCs generated by sa-wpqf grow exponentially w.r.t. the size of the program, while $\mathsf{VCSP}^{\mathsf{PA}}$, $\mathsf{VCCNF}^{\mathsf{PA}}$ have quadratic growth, and $\mathsf{VCLin}^{\mathsf{PA}}$ linear growth.

Recall the program of Example 4.13. It was shown that such a program with the sequences associated to the left originates VCs that have quadratic size even when VCLin is used. In fact this quadratic growth does not occur in Why3 because one of the intermediate steps of the Why3 associates all sequences to the right. Nonetheless, in Section 4.3 we also mentioned that the problem is not only the association of sequences. Having 'blocks' of code containing asserts that are also followed by other asserts also causes a quadratic growth. A way of generating a program with such a pattern is by creating a chain of nested if commands, and inserting an assert command after each one. A sketch of such a program in Why3 is shown in Appendix B under the module name SPCNFWorstCase.

The chart on the right in Figure 7.12 shows the size of the VCs as the number of nested if

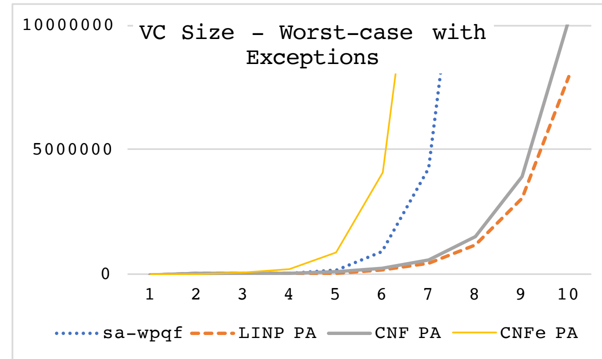


Figure 7.13: VC size growth in the worst-case w.r.t. the size of the program when exceptions are considered

commands is incremented. In fact the current example is not only the worst-case for the $\mathsf{VCLin}^{\mathsf{PA}}$ VCGen, but also for $\mathsf{VCSP}^{\mathsf{PA}}$ and $\mathsf{VCCNF}^{\mathsf{PA}}$. As explained in sections 4.2 and 4.3 $\mathsf{VCSP}^{\mathsf{PA}}$ and $\mathsf{VCLin}^{\mathsf{PA}}$ generates quadratic size VCs, and $\mathsf{VCCNF}^{\mathsf{PA}}$ cubic size VCs, which is confirmed in the chart. Finally, it can also be seen that the VCGen based on WP is now the one that generates smaller VCs, being for this case of linear size with respect to the size of the program.

Section 4.4 has already given an overview of the worst-case when exceptions are taken into account. In short, all the VCGens may produce VCs of exponential size w.r.t. the size of the input program. Here we just confirm the discussion of Section 4.4 with the data obtained with the VCGens that were implemented in Why3. Once again, the same overall results hold also for the original Why3 VCGens.

The chart in Figure 7.13 shows the growth of the VCs for a program similar to the one in Example 4.18. A sketch of the Why3 program that was used in practice is shown in Appendix B under the module name ExceptionsWorstCase (the exponential growth is obtained when the code of the current try-catch command is replicated in a nested way).

The first conclusion one may draw from the chart is that all VCGens indeed generate VCs that grow exponentially. In our experiments we also considered the $\mathsf{VCCNF}_{\mathsf{e}}$ (by using Reduced Ordered Binary Decision Diagrams for keeping path conditions simplified) alternative of the VCCNF but, as can be observed in the chart, the size of the generated VCs is even worse than those generated by VCCNF.

7.3 Related Work

The experiments presented in this chapter resemble the ones that are typically carried out to evaluate two distinct families of tools. In the first family stand tools like software model checkers that unwind or abstract loops, and check for (simple) Boolean properties annotated in the code through assert statements. In this kind of setting the logic is normally decidable, and related to how fast a tool is to prove that the program is correct, or what the required resources for that are. Therefore, tools belonging to this family of tools resort normally to a large set of

benchmark programs to test their efficiency [72, 37, 70, 17]. Verification competitions targeting this type of tools, such as for instance SV-COMP [18], also employ a large set of benchmark programs to compare and evaluate the participating tools (e.g. [78, 24]) in different categories.

On the other hand, in the context of deductive verification tools such as Why3, programs are algorithmically more complicated; properties are expressed through a more expressive logic; and loops are annotated with invariants. In this setting the challenge is not only on how fast the tool is to check a property, but also whether the properties can be proved (sometimes the proof is only possible resorting to an interactive theorem prover). Therefore, deductive verification tools are normally used to establish the correctness of complicated algorithms with respect to a specification, such as in [25, 26, 34]. Verification competitions targeting this type of tools, such as VerifyThis [68] focus on complicated problems (e.g. [47, 33]), and the challenge is not only the proof but also the implementation of the algorithm in a way that allows it to be proved with respect to the specification.

SNIPER-VCGen together with the evaluation of Section 7.1 belong to the first family, whereas the experiments of Section 7.2 belong to the second.

Chapter 8

Conclusion

Motivated by the fact that single-assignment intermediate forms are used extensively in software verification, both in model checking and in deductive verification, we have proposed two proved verification frameworks. Both families of tools eliminate iterating constructs before programs are converted to SA form. In our first approach we deviate from this notion and introduce the notion of iterating single-assignment program. Based on a Hoare-style logic for these programs, we have formalized a program verification technique that consists in translating annotated programs and specifications into an intermediate SA language, and generating compact VCs from it. An adaptation-complete variant of the logic is obtained by adding a dedicated consequence rule with a simple condition. The framework's workflow is proved to be sound and complete, including the translation of annotated programs to SA form. We remark that the translation of loop invariants is a crucial component of the workflow, that does not trivially lead to completeness. To the best of our knowledge, this is the first time that completeness is established for a verification technique based on the use of an intermediate SA form for programs annotated with invariants.

Our second approach was prompted by an inspection of different methods for generating verification conditions for (non-iterating) SA programs. Based on two well-known fundamental VCGen algorithms, we have identified three orthogonal design dimensions, and proposed a conceptual framework (the VCGen cube) that allowed us to define in a uniform way 6 hybrid VCGens. The VCGen cube was then extended to incorporate two optimizations implemented in popular tools. We remark that the two fundamental VCGens from which we departed have their origins in two different traditions and families of tools and thus we expect that our work here will help bridging a gap between the deductive verification and software model checking communities, contributing towards a uniform framework for program verification.

Tools based on predicate transformers and bounded model checking incorporate many advanced features that our framework does not cover. For instance, Boogie includes automatic inference of loop invariants based on abstract interpretation, and CBMC, which natively uses a SAT (rather than SMT) solver, incorporates constant propagation and simplification functionality that is essential for making bounded verification work in practice. Still, our work here proposes a common theoretical foundation for program verification based on intermediate single-assignment form, unifying ideas from predicate transformers, bounded model checking of

software, and program logic.

From the theoretical point of view, we have introduced a program logic for (non-iterating) single-assignment language with assume, assert, and exceptions. Based on this logic we were able to prove the soundness and completeness of the VCGens in the cube, including the two foremost VCGens used in deductive verification and bounded model checking of software. As stated before, these two families of VCGens were developed in a completely independent way and thus have not before been proved to be equivalent or compared in any way. With respect to the conditional normal form transformation, this is the first time (as far as we know) that VC generation is formulated for programs with exceptions, and that soundness and completeness results are studied. For predicate transformers, not all the results are new, but the use of a theoretical framework based on program logic is (the results available in the literature are based on the predicate transformer semantics, which collapses the operational and axiomatic semantics into a single definition).

A verification framework based on the translation of programs into (non-iterating) single-assignment and on the subsequent generation of verification conditions with one of the VCGens of the cube is formalized and proved sound and complete w.r.t. a programming semantics. Although commonly used by verification tools, this is the first time that such a technique, including the translation of programs containing annotated loops into non-iterating single-assignment form, is proved to be sound and complete.

Finally we have presented results obtained in two distinct empirical experiments comparing different VCGens. The first was carried out in the context of LLVM intermediate representation, and was based on unwinding loops before generating VCs. The main goal of this experiment was to observe the behavior of the VCGens in the context of a tool that resembles a bounded model checker of software: loops do not contain annotated invariants; they are unwound a given number of times; and properties to be verified are inserted through assumes and asserts. Even though the benchmarks considered are perhaps not entirely representative of real world programs, their diversity is sufficient to provide evidence of the importance of studying different methods for verification condition generation. In particular by observing the results we believe that studying different VCGens can lead to improvements in state-of-the-art verification tools.

The second experiment was performed in the Why3 deductive verification tool. We started by studying the native Why3 VCGens, which were later compared with our VCGen algorithms that rely on a translation into SA form and subsequent generation of VCs. Additionally we also used the Why3 native VCGens and our own versions to study the worst-case of each VCGen.

The results obtained with the two experiments should be compared with caution. First of all note that the LLVM programs were subject to loop unwinding, while the Why3 programs were annotated with invariants. Moreover, the programs used in the context of LLVM depend only on the QF_AUFLIA logic (quantifier-free, linear integer arithmetic with uninterpreted functions, and arrays), and normally the verification procedure is a simple matter of time and resources. On the other hand, Why3 relies on different background theories, some of which are not decidable. Finally, Why3 is a well established verification framework that has been

used in different contexts and extensively validated, while SNIPER-VCGen has its origins in a fault localization tool and does not implement optimizations targeting verification condition generation.

As future work we identify the following topics:

- Even tough we are confident in the formalization and proofs that were exposed throughout this thesis, it would be interesting to express those with the help of a proof assistant, such as for instance Coq. In particular this can serve as a basis for creating a 'correct by construction' verification tool.
- In terms of defining a proved framework for bounded verification we see two possible pathways: (i) to use the iterating single-assignment language from Chapter 3 and study a bounded logic (resp. bounded VCGen) that constructs bounded derivations (resp. bounded VCs); or (ii) to create a translation into (non-iterating) SA that unwinds loops during the translation. In both cases the main goal would be to establish soundness and/or completeness results, depending on whether unwinding assumptions or assertions were used. We remark that our work in this thesis addresses VC generation as implemented by BMC tools after loop expansion, but not loop expansion itself.
- As far as the target programming language is concerned, initially we considered a fully structured language and then moved into a 'less structured' language by considering exceptions. The next step would be to consider a non structured language with *goto* statements, starting with a restricted form allowing only backward jumps.
- Additionally, it will be interesting to investigate how other important features of verification tools, such as ghost code [48], or the ability to handle aliasing and pointer-based dynamic data structures, would be affected by (and could take advantage of) the use of a single-assignment setting.
- In our experiments in Why3 we observed that moving quantifiers to the outermost position of the VC has benefits for some particular cases in terms of solving time. Even though for the examples we considered this does not seem to introduce any inconsistency, it deserves a deeper analysis for it to be used as default.
- We also believe that it would be interesting to implement and test the VCGens in other verification tools, for instance in Boogie which is another major deductive verification tool, and in the flagship bounded model checker tool CBMC which uses by default a SAT instead of SMT solver.

Appendix A

Derivations

In this chapter we present the derivation in system H and Hg from Chapter 5, of the program from Example 4.1. For simplicity reasons, and since this program never throws exceptions, we simplify the exceptional postcondition to false. This way, when using system H we avoid to use the consequence rule with the trivial side condition of the form $\bot \rightarrow \delta$, for some $\delta \in \textbf{Assert}$. We will sometimes use $C_1, \ldots, C_6$ to refer to the statements of the program as follows:

$$\begin{array}{ll} C_1: & \textbf{if } x_0 > 0 \textbf{ then } y_1 := 1 \textbf{ else } y_1 := 0 \textbf{ fi} \\ C_2: & \textbf{assert } y_1 = 0 \vee y_1 = 1 \\ C_3: & \textbf{if } x_0 > 0 \textbf{ then } y_2 := 1 \textbf{ else } y_2 := 0 \textbf{ fi} \\ C_4: & \textbf{assert } y_2 = y_1 \\ C_5: & \textbf{if } x_0 > 0 \textbf{ then } y_3 := 1 \textbf{ else } y_3 := 0 \textbf{ fi} \\ C_6: & \textbf{assert } y_3 = y_1 \end{array}$$

In the derivation using system H we omit part of the derivation because it is analogous to the other part.

Derivation in H.

$\{\top\}\, C_1; C_2; C_3; C_4; C_5; C_6\, \{\top, \bot\}$
(seq)

1. $\{\top\}\, C_1; C_2; C_3\, \{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)), \bot\}$
   (seq)
   1. $\{\top\}\, \textbf{if } x_0 > 0 \textbf{ then } y_1 := 1 \textbf{ else } y_1 := 0 \textbf{ fi}\, \{(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0), \bot\}$
      (if)
      1. $\{\top \wedge x_0 > 0\}\, y_1 := 1\, \{(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0), \bot\}$
         (conseq)
         1. (assign) $\{x_0 > 0 \wedge 1 = 1\}\, y_1 := 1\, \{x_0 > 0 \wedge y_1 = 1, \bot\}$
      2. $\{\top \wedge \neg x_0 > 0\}\, y_0 := 0\, \{(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0), \bot\}$
         (conseq)
         1. (assign) $\{\neg x_0 > 0 \wedge 0 = 0\}\, y_1 := 0\, \{\neg x_0 > 0 \wedge y_1 = 0, \bot\}$

2. $\{(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)\}\, C_2; C_3\, \{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)), \bot\}$
   (seq)
   1. $\{(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)\}\, \textbf{assert}\, y_1 = 0 \vee y_1 = 1\, \{(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0), \bot\}$
      (conseq)
      1. (assert) $\{(y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0))\}\, \textbf{assert}\, y_1 = 0 \vee y_1 = 1\, \{(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0), \bot\}$
   2. $\{(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)\}\, \textbf{if}\, x_0 > 0\, \textbf{then}\, y_2 := 1\, \textbf{else}\, y_2 := 0\, \textbf{fi}\, \{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)), \bot\}$
      (if)
      1. $\{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge x_0 > 0\}\, y_2 := 1\, \{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)), \bot\}$
         (conseq)
         1. (assign) $\{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (x_0 > 0 \wedge 1 = 1)\}\, y_2 := 1\, \{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (x_0 > 0 \wedge y_2 = 1), \bot\}$
      2. $\{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge \neg(x_0 > 0)\}\, y_2 := 0\, \{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0)), \bot\}$
         (conseq)
         1. (assign) $\{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (\neg x_0 > 0 \wedge 0 = 0)\}\, y_2 := 0\, \{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (\neg x_0 > 0 \wedge y_2 = 0), \bot\}$

2. $\{((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0))\}\, C_4; C_5; C_6\, \{\top, \bot\}$
   (seq)
   ...

   Side conditions for application of the (conseq) rules:

   - $\top \wedge x_0 > 0 \to x_0 > 0 \wedge 1 = 1$ **and** $x_0 > 0 \wedge y_1 = 1 \to (x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)$
   - $\top \wedge \neg x_0 > 0 \to \neg x_0 > 0 \wedge 0 = 0$ **and** $\neg x_0 > 0 \wedge y_1 = 0 \to (x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)$
   - $(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0) \to (y_1 = 0 \vee y_1 = 1) \wedge ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0))$ **and** $(x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0) \to (x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)$
   - $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge x_0 > 0 \to ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (x_0 > 0 \wedge 1 = 1)$ **and** $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (x_0 > 0 \wedge y_2 = 1) \to ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0))$
   - $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge \neg(x_0 > 0) \to ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (\neg x_0 > 0 \wedge 0 = 0)$ **and** $((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge (\neg x_0 > 0 \wedge y_2 = 0) \to ((x_0 > 0 \wedge y_1 = 1) \vee (\neg x_0 > 0 \wedge y_1 = 0)) \wedge ((x_0 > 0 \wedge y_2 = 1) \vee (\neg x_0 > 0 \wedge y_2 = 0))$
   - ...

Derivation in Hg

$\{\top\}\, C_1; C_2; C_3; C_4; C_5; C_6\, \{\top, \bot\}$
(seq)

1. $\{\top\}\, C_1; C_2; C_3\, \{((y_1 = 1 \wedge x_0 > 0) \vee (y_1 = 0 \wedge \neg x_0 > 0)) \wedge y_2 = y_1, \bot\}$
   (seq)

1. $\{\top\}$ **if** $x_0 > 0$ **then** $y_1 := 1$ **else** $y_1 := 0$ **fi** $\{(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0) \land (y_1 = 0 \lor y_1 = 1), \bot\}$
   (if)
   1. (assign) $\{\top \land x_0 > 0\}\, y_1 := 1\, \{(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0) \land (y_1 = 0 \lor y_1 = 1), \bot\}$
   2. (assign) $\{\top \land \neg x_0 > 0\}\, y_1 := 0\, \{(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0) \land (y_1 = 0 \lor y_1 = 1), \bot\}$
2. $\{(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0) \land (y_1 = 0 \lor y_1 = 1)\}\, C_2; C_3\, \{((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land y_2 = y_1, \bot\}$
   (seq)
   1. (assert) $\{(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0) \land (y_1 = 0 \lor y_1 = 1)\}$ **assert** $y_1 = 0 \lor y_1 = 1\, \{(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0), \bot\}$
   2. $\{(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)\}$ **if** $x_0 > 0$ **then** $y_2 := 1$ **else** $y_2 := 0$ **fi** $\{((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land y_2 = y_1, \bot\}$
      (if)
      1. (assign) $\{((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land x_0 > 0\}\, y_2 := 1\, \{((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land y_2 = y_1, \bot\}$
      2. (assign) $\{((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land \neg x_0 > 0\}\, y_2 := 0\, \{((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land y_2 = y_1, \bot\}$

2. $\{((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land y_2 = y_1\}\, C_4; C_5; C_6\, \{y_3 = y_1, \bot\}$
   (seq)
   1. (assert) $\{((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land y_2 = y_1\}$ **assert** $y_2 = y_1\, \{(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0), \bot\}$

   2. $\{(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)\}\, C_5; C_6\, \{y_3 = y_1, \bot\}$ (seq)
      1. $\{(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)\}$ **if** $x_0 > 0$ **then** $y_3 := 1$ **else** $y_3 := 0$ **fi** $\{y_3 = y_1, \bot\}$
         (if)
         1. (assign) $\{((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land x_0 > 0\}\, y_3 := 1\, \{y_3 = y_1, \bot\}$
         2. (assign) $\{((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land \neg x_0 > 0\}\, y_3 := 0\, \{y_3 = y_1, \bot\}$
      2. (assert) $\{y_3 = y_1\}$ **assert** $y_3 = y_1\, \{\top, \bot\}$

   Side conditions for application of the (assign) rules:

   - $\top \land x_0 > 0 \to ((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0) \land (y_1 = 0 \lor y_1 = 1))[1/y_1]$
   - $\top \land \neg x_0 > 0 \to ((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0) \land (y_1 = 0 \lor y_1 = 1))[0/y_1]$
   - $((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land x_0 > 0 \to (((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land y_2 = y_1)[1/y_2]$
   - $((y_1 = 1 \land \neg x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land x_0 > 0 \to (((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land y_2 = y_1)[0/y_2]$
   - $((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land x_0 > 0 \to (y_3 = y_1)[y_1/y_3]$
   - $((y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)) \land \neg x_0 > 0 \to (y_3 = y_1)[y_1/y_3]$

   Side conditions for application of the (assert) rules:

   - $(y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0) \land (y_1 = 0 \lor y_1 = 1) \to (y_1 = 0 \lor y_1 = 1) \land (y_1 = 1 \land x_0 > 0) \lor (y_1 = 0 \land \neg x_0 > 0)$

- $((y_1 = 1 \wedge x_0 > 0) \vee (y_1 = 0 \wedge \neg x_0 > 0)) \wedge y_2 = y_1 \rightarrow y_2 = y_1 \wedge ((y_1 = 1 \wedge x_0 > 0) \vee (y_1 = 0 \wedge \neg x_0 > 0))$
- $y_3 = y_1 \rightarrow y_3 = y_1$

Appendix B

Why3 Experimental Programs

```
module Tassert
  use import int.Int
  use import ref.Ref

  let h (x:int) (y:int)
    requires { x >= 0 /\ x <= 50}
    requires {y < x}
  = let x,y = ref x, ref y in
    while !x < 100 do
      assert {!y < 100}; x := !x + 1; y := !y + 1; assert {!y <= 100}
    done;
    !x
end
```

```
module Tcheck
  use import int.Int
  use import ref.Ref

  let h (x:int) (y:int)
    requires { x >= 0 /\ x <= 50}
    requires {y < x}
  = let x,y = ref x, ref y in
    while !x < 100 do
      check {!y < 100}; x := !x + 1; y := !y + 1; check {!y <= 100}
    done;
    !x
end
```

```
module TassertPostcondition
  use import int.Int
  use import ref.Ref

  let h (x:int) (y:int)
    requires { x >= 0 /\ x <= 50}
    requires {y < x}
    ensures {result <= 101}
  = let x,y = ref x, ref y in
    while !x < 100 do
      assert {!y < 100}; x := !x + 1; y := !y + 1; assert {!y <= 100}
    done;
    !x
end


module TcheckPostcondition
  use import int.Int
  use import ref.Ref

  let h (x:int) (y:int)
    requires { x >= 0 /\ x <= 50}
    requires {y < x}
  ensures {result <= 101}
  = let x,y = ref x, ref y in
    while !x < 100 do
      check {!y < 100}; x := !x + 1; y := !y + 1; check {!y <= 100}
    done;
    !x
end


module WPExplosion
  use import int.Int
  use import ref.Ref

  let f (x:ref int) (y:ref int)
    requires {!x > 0}
  = if !x = 0 then (x := !x + 1; assert {!x = !y + 1})
              else (y := !y + 10; assert {!y = !x + 1});
```

```
    if !x = 0 then (x := !x + 1; assert {!x = !y + 2})
              else (y := !y + 10; assert {!y = !x + 2});
    (* ... *)
    assert {!x + 999 = !y - 999}
end


module SPCNFWorstCase
  use import int.Int
  use import ref.Ref

  let h (x:ref int) (y:ref int) (z: ref int)
    requires { !x >= 0}
  = x := !x + !y + 1;
    if !x = 0 then
      (x := !x + !y + 2;
      if !x = 0 then
        (* ... *)
        x := !x + !y + 11
      else x := !x + !z + 2;
      assert {!x + 1 = !y + 2})
    else x := !x + !z + 1;
    assert {!x + 1 = !y + 1}
end


module ExceptionsWorstCase
  use import int.Int
  use import ref.Ref

  exception Jump

  let h (z:int) (y:int)
  requires {z > 0 /\ y <= z}
  ensures {result <= y}
  raises {Jump -> true}
  = let x = ref z in
    try
      if !x = 10 then x := !x + 1 else raise Jump;
      if !x = 20 then x := !x + 2 else raise Jump
    with Jump ->
```

```
      if !x = 10 then x := !x + 1 else raise Jump;
      if !x = 20 then x := !x + 2 else raise Jump
    end;
    (* ... *)
    !x + 10
end
```

Bibliography

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In *VSTTE*, volume 8471 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2014.

[2] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. *Rigorous Software Development - An Introduction to Program Verification*. Undergraduate Topics in Computer Science. Springer, 2011.

[3] Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. Efficiently intertwining widening and narrowing. *Sci. Comput. Program.*, 120:1–24, 2016.

[4] Pierre America and Frank S. de Boer. Proving total correctness of recursive procedures. *Inf. Comput.*, 84(2):129–162, 1990.

[5] Krzysztof R. Apt. Ten years of Hoare's logic: A survey - part 1. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[6] Krzysztof R. Apt. Ten years of Hoare's logic: A survey part II: nondeterminism. *Theor. Comput. Sci.*, 28:83–109, 1984.

[7] Alessandro Armando, Massimo Benerecetti, and Jacopo Mantovani. Counterexample-guided abstraction refinement for linear programs with arrays. *Autom. Softw. Eng.*, 21(2):225–285, 2014.

[8] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *STTT*, 11(1):69–83, 2009.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[10] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[11] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87. ACM, 2005.

[12] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[13] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[14] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[15] Clark Barrett and Cesare Tinelli. CVC3. In *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007.

[16] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[17] Mohammed Bekkouche. Combining techniques of bounded model checking and constraint programming to aid for error localization. *Constraints*, 22(1):93–94, 2017.

[18] Dirk Beyer. Software verification with validation of results - (report on SV-COMP 2017). In *TACAS (2)*, volume 10206 of *Lecture Notes in Computer Science*, pages 331–349, 2017.

[19] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *DAC*, pages 317–320. ACM Press, 1999.

[20] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[21] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.

[22] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[23] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *LICS*, pages 428–439. IEEE Computer Society, 1990.

[24] Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamaric, and Michael Emmi. SMACK software verification toolchain. In *ICSE (Companion Volume)*, pages 589–592. ACM, 2016.

[25] Razvan Certezeanu, Sophia Drossopoulou, Benjamin Egelund-Müller, K. Rustan M. Leino, Sinduran Sivarajan, and Mark J. Wheelhouse. Quicksort revisited - verifying alternative versions of quicksort. In *Theory and Practice of Formal Methods*, volume 9660 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 2016.

[26] Ran Chen, Martin Clochard, and Claude Marché. A formally proved, complete algorithm for path resolution with symbolic links. *J. Formalized Reasoning*, 10(1):51–66, 2017.

[27] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[28] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[29] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *POPL*, pages 342–354. ACM Press, 1992.

[30] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[31] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[32] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.

[33] Martin Clochard, Léon Gondelman, and Mário Pereira. The matrix reproved (verification pearl). In *VSTTE*, volume 9971 of *Lecture Notes in Computer Science*, pages 107–118, 2016.

[34] Martin Clochard, Léon Gondelman, and Mário Pereira. The matrix reproved (verification pearl). *J. Autom. Reasoning*, 60(3):365–383, 2018.

[35] David R. Cok. Openjml: JML for java 7 by extending openjdk. In *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer, 2011.

[36] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.

[37] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.

[38] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[39] Flaviu Cristian. Correct and robust programs. *IEEE Trans. Software Eng.*, 10(2):163–174, 1984.

[40] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[41] Daniela Carneiro da Cruz, Maria João Frade, and Jorge Sousa Pinto. Verification conditions for single-assignment programs. In *SAC*, pages 1264–1270. ACM, 2012.

[42] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[43] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Available from `www.research.digital.com/SRC/publications/src-rr.html`, 1998.

[44] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[45] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[46] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer, 1990.

[47] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. KIV: overview and VerifyThis competition. *STTT*, 17(6):677–694, 2015.

[48] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2014.

[49] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[50] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[51] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.

[52] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205. ACM, 2001.

[53] Robert W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19(19-32):1, 1967.

[54] Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. *Computer Science Review*, 5(3):252–277, 2011.

[55] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, New York, NY, USA, 1989.

[56] Patrice Godefroid and Shuvendu K. Lahiri. From program to logic: An introduction. In *LASER Summer School*, volume 7682 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2011.

[57] Herman H. Goldstine and John Von Neumann. Planning and coding of problems for an electronic computing instrument. Technical report, Institute of Advanced Studies, Princeton, 1947.

[58] Mike Gordon and Hélène Collavizza. Forward with Hoare. In *Reflections on the Work of C. A. R. Hoare*, pages 101–121. Springer, 2010.

[59] G. A. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical report, University of Toronto, Department of Computer Science, 1975.

[60] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *FTfJP@ECOOP*, pages 6:1–6:7. ACM, 2009.

[61] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 634–640. Springer, 2009.

[62] Matthew Hennessy. *Semantics of programming languages - an elementary introduction using structural operational semantics.* John Wiley and Sons, 1990.

[63] Paolo Herms, Claude Marché, and Benjamin Monate. A certified multi-prover verification condition generator. In *VSTTE*, volume 7152 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2012.

[64] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[65] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, 1971.

[66] Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *Comput. J.*, 38(2):131–141, 1995.

[67] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *FASE*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer, 2000.

[68] Marieke Huisman, Vladimir Klebanov, Rosemary Monahan, and Michael Tautschnig. VerifyThis 2015 - a program verification competition. *STTT*, 19(6):763–771, 2017.

[69] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer, 2010.

[70] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A symbolic execution tool for verification. In *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 758–766. Springer, 2012.

[71] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, 2009.

[72] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.

[73] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[74] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[75] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Asp. Comput.*, 11(5):541–566, 1999.

[76] Donald E. Knuth. Backus Normal Form vs. Backus Naur Form. *Commun. ACM*, 7(12):735–736, 1964.

[77] Daniel Kroening. Software verification. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 505–532. IOS Press, 2009.

[78] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, 2014.

[79] Si-Mohamed Lamraoui and Shin Nakajima. A formula-based approach for automatic fault localization of imperative programs. In *ICFEM*, volume 8829 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2014.

[80] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[81] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[82] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[83] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[84] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *ECOOP Workshops*, volume 1743 of *Lecture Notes in Computer Science*, pages 110–111. Springer, 1999.

[85] Cláudio Belo Lourenço, Maria João Frade, Shin Nakajima, and Jorge Sousa Pinto. A generalized approach to program verification. In *COMPSAC*, page (to appear). IEEE Computer Society, 2018.

[86] Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto. A bounded model checker for SPARK programs. In *ATVA*, volume 8837 of *Lecture Notes in Computer Science*, pages 24–30. Springer, 2014.

[87] Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto. Formalizing single-assignment program verification: An adaptation-complete approach. In *ESOP*, volume 9632 of *Lecture Notes in Computer Science*, pages 41–67. Springer, 2016.

[88] Cláudio Belo Lourenço, Si-Mohamed Lamraoui, Shin Nakajima, and Jorge Sousa Pinto. Studying verification conditions for imperative programs. *ECEASST*, 72, 2015.

[89] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[90] Kenneth Mcmillan. *Symbolic Model Checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[91] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *VSTTE*, volume 7152 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2012.

[92] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[93] Felipe R. Monteiro, Erickson H. da S. Alves, Isabela da Silva, Hussama Ismail, Lucas C. Cordeiro, and Eddie B. de L. Filho. ESBMC-GPU A context-bounded model checking tool to verify CUDA programs. *Sci. Comput. Program.*, 152:63–69, 2018.

[94] Peter Müller and Martin Nordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS*, pages 39–46. ACM, 2007.

[95] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer.* Undergraduate Topics in Computer Science. Springer, 2007.

[96] John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, New York, USA, 1st edition, 1998.

[97] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[98] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2004.

[99] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 703–719. Springer, 2011.

[100] Stefan Sokolowski. Total correctness for procedures. In *MFCS*, volume 53 of *Lecture Notes in Computer Science*, pages 475–483. Springer, 1977.

[101] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.

[102] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, and Francky Catthoor. A practical dynamic single assignment transformation. *ACM Trans. Design Autom. Electr. Syst.*, 12(4):40, 2007.

[103] Glynn Winskel. *The formal semantics of programming languages - an introduction.* Foundation of computing series. MIT Press, 1993.