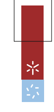
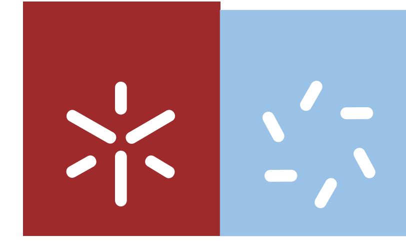


José João Peixoto Pereira Bounded Model Checking de Programas Imperativos

UMinho|2013

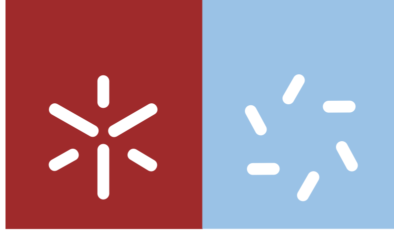


Universidade do Minho
Escola de Ciências

José João Peixoto Pereira

Bounded Model Checking de Programas Imperativos

outubro de 2013



Universidade do Minho
Escola de Ciências

José João Peixoto Pereira

Bounded Model Checking de Programas Imperativos

Dissertação de Mestrado
Mestrado em Matemática e Computação

Trabalho realizado sob orientação do
Professor Doutor Luís Filipe Ribeiro Pinto
e do
Professor Doutor Jorge Sousa Pinto

outubro de 2013

DECLARAÇÃO

Nome _José João Peixoto Pereira

Endereço electrónico: jj.peixotopereira@gmail.com

Número do Bilhete de Identidade: 130 53 826

Título dissertação: Bounded Model Checking de Programas Imperativos

Orientador(es): Professor Doutor Luís Filipe Ribeiro Pinto

Professor Doutor Jorge Sousa Pinto

Ano de conclusão: _Outubro 2013

Designação do Mestrado: Mestrado em Matemática e Computação

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

Universidade do Minho, ____/___/_______

Assinatura: ____

Agradecimentos

Quero deixar os meus especiais agradecimentos ao Doutor Luís Filipe Ribeiro Pinto pela solidariedade, paciência e dedicação demonstrada, por ter estado disponível, por todas as discussões e apoio que prestou e levaram a concretização deste trabalho.

Ao Doutor Jorge Sousa Pinto pela sua disponibilidade e generosos conhecimentos, pela sua incansável paciência e dedicação ao longo deste trabalho.

O meu sincero obrigado.

Agradece-se o apoio do projecto:
"CROSS - Uma Infraestrutura para a certificação e re-engenharia de Software Livre"
FCOMP-01-0124-FEDER-010049 (Refª FCT: PTDC/EIA-CCO/108995/2008), financiado por:

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

QREN
QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL

Resumo

A detecção de erros em sistemas computacionais, em particular em sistemas críticos, é uma tarefa fundamental para garantir o correcto funcionamento dos sistemas e, por este motivo, a verificação formal é um elemento primordial no desenvolvimento de sistemas computacionais. Muitas das técnicas de verificação formal de sistemas computacionais são baseadas em *Model Checking* (MC). O MC é uma técnica de verificação automática que assenta na modelação de sistemas através de máquinas de estados finitas e na especificação das propriedades pretendidas para os sistemas através de fórmulas lógicas (tipicamente em alguma lógica temporal). O *Bounded Model Checking* (BMC) é uma técnica de MC que restringe a exploração do espaço de estados até um determinado comprimento (*bound*) dos traços de execução, como forma de evitar o problema da explosão de estados do MC tradicional. Por isso, as propriedades garantidas por este método aplicam-se apenas à execução até um certo *bound*.

Nesta tese focamos no *Bounded Model Checking* aplicado a *software* (BMCsw), no contexto de linguagens imperativas, cujas ideias essenciais passam por usar *asserts* ao longo dos programas, como forma de declarar propriedades que devem ser satisfeitas por qualquer execução desses programas, e por expandir ciclos, escrevendo-os à custa de construções *if then else*. O BMCsw pode ser visto como uma sequência de transformações ao nível do programa e por uma transformação final que produz uma fórmula lógica, cuja validade garante a correcção do programa.

Especificamente, nesta tese de mestrado estudamos a validade do método BMC no contexto de programas $\mathcal{W}hile$ com *asserts*. Por um lado, estabelecemos uma semântica operacional para uma linguagem $\mathcal{W}hile$ com *asserts*, que designamos por $\mathcal{W}hile_{\mathbf{assert}}$, onde o estado de erro desempenha um papel fundamental. Por outro lado, estudamos propriedades desta linguagem e, com base nestas propriedades, provamos a validade (correcção e completude) da transformação final envolvida no método BMCsw.

O BMC apresenta problemas de escalabilidade que são amenizados pelo actual poder de decisão dos *Satisfiability problem* (SAT) *solvers*. Durante esta tese foi desenvolvida uma ferramenta de *software* que implementa o método BMC para a linguagem minimalista $\mathcal{W}hile_{\mathbf{assert}}$. Dado um programa $\mathcal{W}hile_{\mathbf{assert}}$, esta ferramenta produz uma fórmula lógica, codificando as propriedades do programa, cuja validade garante a correcção do programa. A validade da fórmula é então avaliada com recurso ao SAT *solver* Z3.

Abstract

Error detection in computer systems, particularly in critical systems, is a key task to ensure the correct functioning of systems and, therefore, formal verification is a key element in the development of computer systems. Many of the techniques of formal verification of computer systems are based on *Model Checking* (MC). MC is an automatic verification technique which is based on using finite state machines to model systems, and on the specification of the desired properties for the systems by logical formulas (typically in some *temporal logic*). *Bounded Model Checking* (BMC) is a technique of MC that restricts the exploration of the state space up to a certain execution trace length (*bound*), in order to avoid the problem of state explosion in traditional MC. Therefore, the properties guaranteed by this method apply only to a certain *bound* enforcement.

In this thesis we focus on *Bounded Model Checking* applied to *software* (BMCsw), in the context of imperative languages, whose key ideas are to use *assertions* throughout programs as a way to declare properties that must be satisfied by any execution of these programs, and to expand cycles, rewriting them in terms of the construct *if then else*. The BMCsw can be seen as a sequence of transformations at the level of the program and a final transformation which produces a logical formula whose validity ensures the correctness of the program.

Specifically, in this master thesis we study the validity of the method BMC in the context of $\mathcal{W}hile$ programs with *asserts*. On the one hand, we establish an operational semantics for a $\mathcal{W}hile$ language with *assertions*, which we call $\mathcal{W}hile_{\mathbf{assert}}$, where the error state plays a key role. On the other hand, we study properties of this language and, based on these properties, we prove the validity (correctness and completeness) involved in the final transformation of the BMCsw method.

BMC presents scalability issues that are mitigated by the decision-making power of the current *Satisfiability problem* (SAT) *solvers*. During this thesis we have developed a *software* tool that implements the BMC method for the minimal language $\mathcal{W}hile_{\mathbf{assert}}$. Given a program $\mathcal{W}hile_{\mathbf{assert}}$, this tool produces a logical formula, encoding properties of the program, whose validity ensures the correctness of the program. The validity of the formula is then evaluated using the Z3 SAT *solver*.

Conteúdo

Agradecimentos **i**

Resumo **iii**

Abstract **v**

1 Introdução **1**

2 $\mathcal{W}hile_{\textbf{assert}}$: A linguagem imperativa base **5**

2.1 Aspectos sintáticos 5

2.1.1 Expressões inteiras 6

2.1.2 Expressões booleanas 6

2.1.3 Comandos 7

2.2 A *semântica operacional* 9

2.2.1 Estados 9

2.2.2 Avaliação de expressões inteiras 10

2.2.3 Avaliação de expressões booleanas 11

2.2.4 Estados com erro 12

2.2.5 Relação de avaliação $\rightsquigarrow$ 13

2.2.6 Propriedades elementares da relação de avaliação 16

2.2.7 Equivalência de programas 19

2.3 Correcção de programas 21

2.4 Classes especiais de programas 23

2.4.1 Comandos $\mathcal{SA}$ 23

2.4.2 Comandos $\mathcal{C}nf$ 25

3 *Bounded Model Checking* **29**

3.1 *Model Checking* 29

3.2 *Bounded Model Checking* 31

3.3 Transformação BMC de programas $\mathcal{W}hile_{\textbf{assert}}$ 35

3.3.1 Transformação $\mathcal{T}_1$ 35

3.3.2 Transformação $\mathcal{T}_2$ 37

3.3.3 Transformação $\mathcal{T}_3$ 39

3.3.4 Transformação $\mathcal{T}_4$ 40

4 Correcção e Completude da transformação de programas $\mathcal{C}nf$ em fórmulas **43**

4.1 Resultados preliminares 43

4.2 Correcção de $\mathcal{T}_4$ 55

4.3 Completude de $\mathcal{T}_4$ 57

5 Implementação de um *Bounded Model Checker* **59**
5.1 Conceitos 60
5.2 Front-end 61
5.2.1 Análise léxica - Reconhecimento dos tokens da linguagem 61
5.2.2 Análise hierárquica 63
5.3 Middle-end 69
5.3.1 Árvore de síntaxe abstracta dos comandos 69
5.3.2 Transformações 71
5.4 Back-end: Validação das fórmulas com recurso a um SMT 73

6 Conclusões e trabalho futuro **75**

A Apêndice **81**
A.1 Aspectos técnicos sobre a ferramenta 81
A.2 Ficheiro SMT-LIB v2.0 resultante do exemplo ilustrativo 83
A.3 Modelo obtido pela ferramenta Z3 84

Lista de Figuras

2.1 Classe dos Comandos SA . . . 24
2.2 Exemplo ilustrativo . . . 24
2.3 Classe dos comandos CNF . . . 25

3.1 Transformações BMC . . . 35
3.2 Instância de um programa sem ciclos . . . 35
3.3 Exemplo base . . . 36
3.4 Exemplo base sem ciclos . . . 37
3.5 Static Single Assignment . . . 37
3.6 Dynamic Single Assignment . . . 38
3.7 Forma Dynamic Single Assignment do programa na Figura 3.4 . . . 38
3.8 Conditional Normal Form do programa na Figura 3.7 . . . 40
3.9 Conjunto $\mathcal{C}$ resultante . . . 41
3.10 Conjunto $\mathcal{P}$ resultante . . . 41

5.1 Tokens: Comandos . . . 62
5.2 Tokens: Operações aritméticas . . . 62
5.3 Tokens: Expressões Booleanas . . . 63
5.4 Classes simbólicas: White Spaces, letras e digitos . . . 63
5.5 Tokens: Números inteiros e identificadores de variáveis . . . 63
5.6 Representação simbólica da estrutura em C . . . 64
5.7 Expressões de Inteiros . . . 65
5.8 Argumentos dos identificadores segundo a filosofia aplicacional (λ-*calculus*) . . . 65
5.9 Definição recursiva à direita de uma lista não vazia de expressões inteiras . . . 66
5.10 Expressões Booleanas . . . 67
5.11 Comandos . . . 68
5.12 Procedimentos . . . 69
5.13 Raíz da gramática . . . 69
5.14 Representação Haskell da Classe de Expressões Inteiras . . . 70
5.15 Representação Haskell da Classe de Expressões Booleanas . . . 70
5.16 Representação Haskell da Classe dos Comandos . . . 71
5.17 Pré-processamento do comando call . . . 71
5.18 Transformação $\mathcal{T}_1$ (implementação) . . . 72
5.19 Transformação $\mathcal{T}_2$ (implementação) . . . 72
5.20 Transformação $\mathcal{T}_3$ (implementação) . . . 73
5.21 Transformação $\mathcal{T}_4$ (implementação) . . . 73

A.1 Argumentos reconhecíveis . . . 81
A.2 Ilustração de versão . . . 81
A.3 Ilustração de *bounds* . . . 82
A.4 Ilustração de input com ficheiros . . . 82
A.5 Ilustração de erros gerados . . . 82

Lista de Tabelas

2.1 Classe das **expressões inteiras** 6
2.2 Classe das **expressões booleanas** 7
2.3 Classe de comandos 8
2.4 Regras da semântica operacional 14
2.5 Simplificação da semântica operacional para expressões CNF 27

Lista de Acrónimos

ABV *Assertion-Based Verification*

API *Interface de Programação de Aplicativos*

AST *Abstract Syntax Tree*

ASIC *Circuitos Integrados de Aplicação Específica*

BDD *Binary Decision Diagram*

BMC *Bounded Model Checking*

BMCsw *Bounded Model Checking* aplicado a *software*

BNF *Backus Naur Form*

CMU *Carnegie Mellon University*

CNF *Conditional Normal Form*

DSA *Dynamic Single Assignment*

DSL *Domain-Specific Language*

FOL *Lógica de Primeira Ordem*

LTL *Linear Temporal Logic*

MC *Model Checking*

PSP *Propositional Satisfiability Problem*

RoBDD *Reduced Ordered Binary Decision Diagrams*

SA *Single Assignment*

SAT *Satisfiability problem*

SMC *Symbolic Model Checking*

SMT *Satisfiability Modulo Theories*

SoC *Sistemas Integrados em Chips*

SSA *Static Single Assignment*

STS *Sistema de Transição de Estados*

WCET *high-level Worst-Case Execution Time*

Capítulo 1

Introdução

Enquadramento

A detecção de erros em sistemas críticos é uma tarefa fundamental e por este motivo a verificação formal é um elemento primordial no desenvolvimento deste tipo de sistemas. Intuitivamente, um erro num sistema computacional é algo indesejável. Indesejável pois pode acarretar custos temporais, económicos, biológicos, e em sistemas críticos a ocorrência de erros poderá ter consequências agravadas, possivelmente com custos humanos. Neste tipo de sistemas, embora se utilizem por vezes técnicas formais de verificação, tradicionalmente são implementados mecanismos *fail-safe*, que recorrem a práticas de detecção empírica de erros para prevenir a sua falha, através de testes e seus derivados.

Também a crescente complexidade evidenciada nos mais recentes sistemas tem salientado a importância de aplicar técnicas de verificação formal, em particular na indústria de *hardware* [42, 26]. De acordo com Harry Foster [24], em média, a procura de '*bugs*' em sistemas digitais chega a consumir mais de 60 por cento do esforço de desenvolvimento nos *Circuitos Integrados de Aplicação Específica* (ASIC) e *Sistemas Integrados em Chips* (SoC). Em [1] afirma-se que as empresas que adotaram uma metodologia baseada em afirmações de verificação, *i.e.*, *Assertion-Based Verification* (ABV), verificaram uma significativa redução no tempo de simulação e de depuração, uma melhoria de até cerca de 50 por cento. Claramente, a verificação formal de sistemas é um tema que a indústria, quer de *hardware*, quer de *software*, deve enfrentar, e algumas organizações já o têm feito: grande parte dos maiores fabricantes de sistemas digitais têm procurado aplicar sistemas de verificação formal.

Enquanto na indústria do *hardware* o desenvolvimento de técnicas automáticas de verficação têm levado à sua adopção generalizada, o mesmo não sucede ainda na área do *software*. A *verificação formal de sistemas de software* é o acto de provar a correcção dos algoritmos com respeito a *especificações formais*, expressas num conjunto de *propriedades*, usando métodos formais matemáticos. Os processos de verificação consistem em validar aspectos estruturais (estáticos) e comportamentais (dinâmicos). Os aspectos estáticos envolvem análises detalhadas ao código, enquanto que os aspectos dinâmicos estão intimamente ligados a baterias de testes que são aplicados à execução do sistema.

As técnicas usadas na verificação formal de *software* são tipicamente baseadas em *Model Checking* (MC) [9, 14] ou em provas dedutivas [27]. O MC assenta na modelação de sistemas através de máquinas de estados finitas, na especificação das propriedades pretendidas para os sistemas através de fórmulas lógicas (tipicamente em alguma lógica temporal), sendo então efectuada uma procura exaustiva no espaço de estados, no sentido de testar a validade das fórmulas que representam as propriedades pretendidas nos sistemas. Na abordagem através de provas dedutivas, como por exemplo em lógicas de programas baseadas em *Lógica de Hoare* [27], a correcção dos programas é garantida através da construção de provas para uma determinada fórmula lógica ou conjunto de fórmulas lógicas (as chamadas *condições de verificação* em *Lógica de Hoare*), tipicamente com recurso a ferramentas computacionais de demonstração (semi-)automática.

O *Bounded Model Checking* (BMC) é uma técnica de MC, que, como o nome sugere, restringe a exploração do espaço de estados, impondo um certo limite (*bound*), aos comprimento dos traços de execução das máquinas de estados que modelam os sistemas. Assim sendo, este método procurará apenas contra-exemplos para a validade das propriedades esperadas para o sistema dentro do *bound*, pelo que se não encontrar um contra-exemplo dentro do *bound* nada permite concluir acerca da validade do sistema. Contudo, este método tem-se revelado como uma técnica eficaz para ultrapassar o problema da explosão de estados no MC tradicional e é uma das técnicas de verificação mais utilizadas na indústria dos semicondutores, para verificar circuitos lógicos [6].

Bounded Model Checking de *software*

Atualmente as ideias do BMC são também usadas na verificação de software, o chamado *Bounded Model Checking* aplicado a *software* (BMCsw), designadamente no contexto de linguagens de programação imperativas, como por exemplo a linguagem C [31]. Neste contexto, a ideia essencial é "expandir" ciclos, escrevendo-os à custa de construções if then else e impondo limites/*bounds* à expansão. Uma outra ideia usada neste contexto é a introdução de *asserts* ao longo do programa, que correspondem a asserções/propriedades que devem ser satisfeitas durante a execução do programa.

O mecanismo de especificação da *Lógica de Hoare* apenas permite raciocinar sobre afirmações feitas como pré-condições e pós-condições do programa, o que não permite de forma fácil, detectar a validade de uma certa propriedade num determinado ponto de execução do programa. A possibilidade de introduzir *asserts* ao longo do programa permite contornar esta dificuldade. Para tal a noção de validade de um programa terá que garantir que em qualquer execução do programa, todos os *asserts* coleccionados são efectivamente válidos.

Uma outra mais valia do BMCsw reside no facto de ser um método automático em contraste com os métodos dedutivos que, normalmente, para lidarem com a verificação de ciclos requerem a introdução de invariantes de ciclo por parte do programador.

O BMCsw pode ser visto como uma sequência de transformações que conduzem a uma fórmula lógica, cuja validade garante a correcção do programa. As ferramentas mais aplicadas para decidir sobre a validade das fórmulas geradas pelo BMCsw são os chamados *Satisfiability problem* (SAT) *solvers* [25] e *Satisfiability Modulo Theories* (SMT) *solvers* [22].

O método BMCsw, só por si, tem problemas de escalabilidade, essencialmente associados à capacidade de decisão das enormes fórmulas resultantes da aplicação do método. No entanto, o poder crescente dos SAT *solvers* e SMT *solvers*, tem potenciado a aplicação de métodos como o BMCsw. O CBMC [31] é uma ferramenta de BMC para programas escritos na linguagem ANSI-C e C++. Foi desenvolvido por *Daniel Kröning* na *Carnegie Mellon University* (CMU). A ferramenta permite verificar limites em `arrays`, `buffer overflows`, `pointer safety`, verificação de excepções e afirmações especificadas pelo programador, recorrendo a SAT *solvers* para validar as fórmulas que gera. Diversos artigos salientam o sucesso desta ferramenta [10, 32, 11].

Motivação e objectivos da tese

Nesta tese de mestrado considera-se uma linguagem imperativa simples, designadamente a linguagem $\mathcal{While}$ enriquecida com o comando **assert** (b), e estuda-se a aplicação do método BMC a esta linguagem.

A motivação para o tema desta tese surgiu do resultado e do contributo da bolsa de investigação com referência BI4-2011_PTDC/EIA-CCO/108995/2008_UMINHO. Esta bolsa de investigação permitiu a familiarização com o método BMC e a construção de uma ferramenta exploratória de aplicação do BMC a programas $\mathcal{While}$ com *asserts*. Nos trabalhos relativos a esta bolsa, não tivemos preocupações em justificar a correcção da aplicação do método. No entanto, na altura, também não encontramos trabalhos dedicados à justificação do método BMC no contexto de programas $\mathcal{While}$ com *asserts*, embora se encontrem diversos artigos sobre os sucessos da técnica BMC e diversas aplicações práticas.

Nesta tese de mestrado procuramos dar os primeiros passos no sentido de justificar a validade do método BMC no contexto de programas $\mathcal{W}hile$ com *asserts*. Por um lado, estabelece-se uma *semântica operacional* para uma linguagem $\mathcal{W}hile$ com *asserts*, que designamos por $\mathcal{W}hile_{\mathbf{assert}}$, onde o estado de erro desempenha um papel fundamental. Por outro lado, estudam-se propriedades desta linguagem que permitem garantir a validade de parte das transformaçõs envolvidas no método BMC. Durante a realização desta tese de mestrado, foi também desenvolvida uma ferramenta que implementa a técnica BMC para a linguagem $\mathcal{W}hile_{\mathbf{assert}}$.

Estrutura da tese

Esta tese de mestrado está organizada da seguinte forma: O Capítulo 2 está dividido em 4 (quatro) secções. Na Secção 2.1 são abordados os aspectos sintáticos da linguagem básica $\mathcal{W}hile_{\mathbf{assert}}$, na Secção 2.2 é abordada a *semântica operacional* da referida linguagem, na Secção 2.3 é introduzida a noção de correcção de programas no contexto desta tese de mestrado. Finalmente, na Secção 2.4 são formalizadas 2 (duas) classes especiais de programas necessárias a todo o processo de verificação BMC.

O Capítulo 3 está dividido em 3 (três) secções. Na Secção 3.1 é introduzida uma visão histórica da área de verificação até ao ponto de criação do método BMC. Na Secção 3.2 é apresentada a génese do método BMC para *hardware* e *software*, a partir de duas indústrias distintas. Finalmente, na Secção 3.3 são apresentadas em detalhe as 4 (quatro) transformações ($\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ e $\mathcal{T}_4$) que constituem os vários passos do método BMC no contexto de uma linguagem $\mathcal{W}hile$ com *asserts*.

O Capítulo 4 está dividido em 3 (três) secções. Na Secção 4.1 são apresentadas as observações/resultados necessários às conclusões que pretendemos retirar nas secções seguintes. Na Secção 4.2 é apresentado o teorema da correcção associado à transformação $\mathcal{T}_4$, e, finalmente, na Secção 4.3 é apresentado o teorema da completude associado à transformação $\mathcal{T}_4$.

O Capítulo 5 está dividido em 4 (quatro) secções. Na Secção 5.1 são apresentados alguns conceitos relativos a compiladores, à compilação e à construção da ferramenta de verificação. Na Secção 5.2 aborda-se o front-end e apresentam-se as decisões tomadas para o reconhecimento da linguagem. Na Secção 5.3 aborda-se o middle-end e descrevem-se as decisões tomadas na implementação das transformações $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ e $\mathcal{T}_4$ e construção da fórmula lógica resultante. Finalmente, na Secção 5.4 aborda-se o back-end, a apresentação da documentação gerada pela ferramenta e faz-se uma pequena referência a um dos SMT *solvers* existentes no mercado, em particular aquele que foi aplicado durante o desenvolvimento desta tese de mestrado.

No Capítulo 6 apresentam-se os resultados extraídos desta tese de mestrado, trabalho futuro a realizar no sentido da continuação deste projecto e algumas considerações finais.

Capítulo 2

$\mathcal{W}hile_{\textbf{assert}}$: A linguagem imperativa base

Este capítulo define uma linguagem de programação que será utilizada ao longo da tese de mestrado. Esta linguagem é uma linguagem imperativa básica que apenas procura representar mecanismos essenciais de progamação. A esta linguagem daremos o nome $\mathcal{W}hile_{\textbf{assert}}$ e, embora seja simples, será suficientemente poderosa para podermos raciocinar sobre as transformações que ocorrem no processo BMC.

Neste capítulo são definidos conceitos que nos permitirão definir as transformações em que assenta o método BMC e raciocinar sobre a validade do BMC. Abordar-se-ão questões de *sintaxe* da linguagem, bem como as questões da *semântica* da linguagem. Por um lado, a *sintaxe* estabelece a forma das frases de uma linguagem. Por outro lado, a *semântica* é a área que estuda a forma de expressar o significado dessas frases numa linguagem, sendo ambas necessárias para definir e raciocinar sobre uma linguagem.

A linguagem $\mathcal{W}hile_{\textbf{assert}}$ permite exprimir **ciclos** com o comando **while** (b) **do** c_w, **condicionais** com o comando **if** (b) **then** c_t **else** c_f, **atribuições** com o comando $x := e$, as **asserções** com o comando **assert** (b) , **nop**es (*"no operation"*) com o comando **skip**, e, finalmente, a construção típica das linguagens imperativas, **composição** de comandos com o comando c_i **;** c_j.

Este capítulo está dividido em 4 (quatro) secções. Na primeira secção abordam-se aspectos sintáticos da linguagem, como se representam as expressões inteiras, as expressões booleanas e os comandos da linguagem. Na segunda secção definem-se estados de computação e aborda-se o significado das expressões sobre a manipulação de estados. Na terceira secção definimos a noção de correcção de um programa na nossa linguagem e finalmente, na última secção, definem-se as classes intermédias de comandos usadas pelas transformações presentes no método BMC. Nas secções 2.2 e 2.4 provam-se algumas propriedades básicas que serão necessárias no Capítulo 4.

2.1 Aspectos sintáticos

A *sintaxe* de uma linguagem de programação especifica a estrutura e a forma de organizar os constituintes da linguagem (as palavras, os chamados `tokens` da linguagem). Geralmente expressos em texto, `strings` ou frases da linguagem.

A *sintaxe* procura descrever um conjunto de regras, conhecidas como *produções da linguagem*, que definem possíveis construções de uma frase da linguagem. Estas regras descrevem a organização possível dos elementos da linguagem (`tokens`), e organizam a estrutura em classes sintáticas. A *sintaxe* de uma linguagem diz-se ambígua quando permite mais do que uma forma de construção da mesma frase na linguagem.

Como dito anteriormente, a *sintaxe* é definida por uma gramática, normalmente numa gramática normalizada em BNF ou uma variante desta. No nosso caso, iremos exprimir a nossa *sintaxe* numa

variante desta meta-notação. Uma vez que não nos iremos preocupar com questões como *"o que é um número"* e *"como deve ser um número representado"*, iremos apresentar a nossa *sintaxe* em termos de uma *sintaxe abstracta*. Uma *sintaxe abstracta* é uma notação que permite definir tipos de dados simples, tipos de dados complexos e especificar quais os valores que estes tipos de dados podem assumir. As sequências de símbolos permitidos numa frase sintaticamente correta do programa é definido nas seguintes subsecções. A linguagem que estudamos é simples, pelo que iremos apenas manipular objectos conhecidos e pouco complexos.

2.1.1 Expressões inteiras

Precisamos de definir as variáveis do programa que irão representar números inteiros. As variáveis serão objectos que a linguagem manipula. Assumiremos que o conjunto de variáveis é numerável.

Convenção 2.1. *Assumimos a existência de variáveis* $x_0, x_1, \ldots, x_n, \ldots$
Esta colecção de variáveis é numerável e notada por $\mathcal{V}_{Integer}$.
Usamos as letras $x, x', y, y', \ldots$ *como meta-variáveis sobre* $\mathcal{V}_{Integer}$.

As variáveis de $\mathcal{V}_{Integer}$, no contexto da avaliação de expressões, assumirão valores em $\mathbb{Z}$. Habitualmente usamos $z, v, z', v', z_0, v_0, \ldots$ como meta-variáveis sobre $\mathbb{Z}$

Por forma a fornecer algum poder à linguagem, são permitidas algumas **expressões inteiras**, para representar operações sobre os números inteiros. Introduzimos a noção de expressão na Definição 2.2 e porteriormente, na Definição 2.12, definimos a interpretação das expressões.

Definição 2.2. *A notação* $\mathcal{E}xpression_{Integer}$ *representará a classe das* **expressões inteiras**. *A classe sintática* $\mathcal{E}xpression_{Integer}$ *é definida indutivamente do seguinte modo:*

$$
\begin{array}{rcll}
e & ::= & z & se\ z \in \mathbb{Z} \\
 & | & x & se\ x \in \mathcal{V}_{Integer} \\
 & | & -e & \\
 & | & (\ e_i \,\square\, e_j\ ) & ,\ \square \in \{\ +,\ -,\ \times,\ \div,\quad \text{mod}\ \}
\end{array}
$$

Tabela 2.1: Classe das **expressões inteiras**

Habitualmente usamos $e, e', e_i, e_j, e_1, \ldots$ *como meta-variáveis sobre* $\mathcal{E}xpression_{Integer}$.

Recorremos à função *FV* (de *free variables*) para obter os elementos de $\mathcal{V}_{Integer}$ presentes numa expressão inteira.

Definição 2.3. *A função* $FV : \mathcal{E}xpression_{Integer} \to \mathcal{P}(\ \mathcal{V}_{Integer}\ )$ *é definida recursivamente por:*

$$
\begin{array}{rcll}
FV\ (\ z\ ) & = & \emptyset & se\ z \in \mathbb{Z} \\
FV\ (\ x\ ) & = & \{\ x\ \} & se\ x \in \mathcal{V}_{Integer} \\
FV\ (\ -e\ ) & = & FV\ (\ e\ ) & \\
FV\ (\ e_i \,\square\, e_j\ ) & = & FV\ (\ e_i\ ) \cup FV\ (\ e_j\ ) & se\ \square \in \{\ +,\ -,\ \times,\ \div,\quad \text{mod}\ \}
\end{array}
$$

Exemplo 1. $FV\ (\ (\ 3 + x_0\ ) \times (\ x_3 - (\ 4 \div 96\ )\ )\ ) = \{\ x_0,\ x_3\ \}$

2.1.2 Expressões booleanas

Esta subsecção formaliza a forma de expressar condições boleanas na nossa linguagem. Estas expressões são necessárias para exprimir ciclos, condicionais e mesmo afirmações e asserts dentro dos programas. Além disto, o método de BMC, por si, transforma um programa numa fórmula lógica que procura modelar todo o comportamento expresso pelo programa.

Convenção 2.4.
Usamos o símbolo $\top$ *para denotar o* **valor lógico verdadeiro**.
Usamos o símbolo $\bot$ *para denotar o* **valor lógico falso**.
Denotaremos o conjunto dos valores lógicos, por $\mathbb{B}$. *Por vezes, chamaremos* **valores booleanos** *aos valores lógicos.*

Além dos valores de verdade, nas nossas expressões booleanas também permitimos algumas operações lógicas como a negação, a conjunção ($\wedge$), a disjunção ($\vee$) e a implicação ($\rightarrow$). A implicação lógica não é comum nas linguagens de programação usuais, uma vez que se pode interpretar a implicação lógica $p \rightarrow q$ como $(\neg p) \vee q$. Contudo, optamos por introduzir a implicação, evitando a formalização de novas classes e interpretações associadas quando for necessário realizar o estudo das expressões lógicas resultantes do processo BMC. Introduzimos também relações de comparação, que são o mecanismo fundamental para a construção de expressões booleanas a partir dos valores das variáveis, *i.e.*, dos programas e que portanto permite fazer afirmações acerca dos valores das variáveis.

De seguida descrevemos a classe das nossas expressões booleanas. A avaliação de expressões desta classe sintática é definida posteriormente, na Definição 2.14.

Definição 2.5. *A notação* $\mathcal{E}xpression_{Boolean}$ *representará a classe das* **expressões booleanas**. *A classe sintática* $\mathcal{E}xpression_{Boolean}$ *é definida indutivamente do seguinte modo:*

$$
\begin{array}{rcll}
b & ::= & \top & \\
 & | & \bot & \\
 & | & (\, e_i \,\Box\, e_j \,) & ,\Box \in \{\, \leq, <, >, \geq, =, \neq \,\} \\
 & | & \neg\, b & \\
 & | & (\, b_i \,\Box\, b_j \,) & ,\Box \in \{\, \wedge, \vee, \rightarrow \,\}
\end{array}
$$

Tabela 2.2: Classe das **expressões booleanas**

Habitualmente usamos $b, b', b_i, b_j, b_1, \ldots$ *como meta-variáveis sobre* $\mathcal{E}xpression_{Boolean}$.

Recorremos à função FV para obter os elementos de $\mathcal{V}_{Integer}$ presentes numa expressão booleana.

Definição 2.6. *A função* $FV : \mathcal{E}xpression_{Boolean} \rightarrow \mathcal{P}(\mathcal{V}_{Integer})$ *é definida recursivamente por:*

$$
\begin{array}{rcll}
FV\,(\,\top\,) & = & \emptyset & \\
FV\,(\,\bot\,) & = & \emptyset & \\
FV\,(\,e_i \,\Box\, e_j\,) & = & FV\,(\,e_i\,) \cup FV\,(\,e_j\,) & ,\,\Box \in \{\, \leq, <, >, \geq, =, \neq \,\} \\
FV\,(\,\neg\, b\,) & = & FV\,(\,b\,) & \\
FV\,(\,b_i \,\Box\, b_j\,) & = & FV\,(\,b_i\,) \cup FV\,(\,b_j\,) & ,\,\Box \in \{\, \wedge, \vee, \rightarrow \,\}
\end{array}
$$

Exemplo 2. $FV\,(\,(\,\top \rightarrow \bot\,) \vee (\,x_2 - 4\,) \geq (\,x_1 \div 96\,)\,) = \{\, x_2,\ x_1 \,\}$

2.1.3 Comandos

A classe sintática expressa pela Tabela 2.3, corresponde aos comandos presentes na linguagem $\mathcal{W}hile_{\mathbf{assert}}$. Conforme indicado na introdução, esta classe indica a ordem e sequenciação de símbolos que exprimem os ciclos, os condicionais, os mecanismos de modificação de variávies, de afirmação de propriedades do programa, etc.

O comando nulo: **skip** representa uma "não operação". Pode ser considerado o elemento neutro, pois pretende-se que não efectue qualquer intervenção no estado da computação. A sua existência, embora não afecte o estado da computação, permite programar condicionais em que apenas um dos ramos tem conteúdo computacional. O comando de afirmação/*assert* é notado

por **assert** (b) e permite realizar afirmações/asserções sobre o programa em pontos específicos do programa, sendo que um programa para ser correcto terá que tornar a asserção verdadeira no estado com que chegar a esse ponto de execução. O comando de atribuição $x \coloneqq e$ permite manipular variáveis do programa; a variável x passará a assumir o valor da avaliação da expressão e no estado em que esta atribuição é realizada. Este mecanismo de substituição é o único capaz de alterar o estado. O comando de composição c_i **;** c_j permite sequênciar um conjunto de passos de computação, simulando a execução sequêncial das linguagens imperativas. O comando condicional **if** (b) **then** c_t **else** c_f permite computar em termos de casos, onde a condição b rege a decisão de computar uma das duas possibilidades, ou c_t, ou c_f. Finalmente, o comando **while** (b) **do** c_w expressa ciclos de programação, onde o condifional b rege a decisão de computar o comando c_w.

Definição 2.7. *A notação $\mathcal{C}ommand$ representará a classe de comandos. A classe sintática $\mathcal{C}ommand$ é definida indutivamente do seguinte modo:*

$$
\begin{array}{rcc}
c & ::= & \textbf{skip} \\
 & | & \textbf{assert } (b) \\
 & | & x \coloneqq e \\
 & | & c_i \textbf{ ; } c_j \\
 & | & \textbf{if } (b) \textbf{ then } c_t \textbf{ else } c_f \\
 & | & \textbf{while } (b) \textbf{ do } c_w
\end{array}
$$

Tabela 2.3: Classe de comandos

A uma expressão da classe $\mathcal{C}ommand$ chamaremos comando ou programa. Habitualmente usamos $c, c', c_i, c_j, c_1, \ldots$ como meta-variáveis sobre $\mathcal{C}ommand$.

Recorremos à função FV para obter os elementos de $\mathcal{V}_{Integer}$ presentes num comando.

Definição 2.8. *A função $FV : \mathcal{C}ommand \rightarrow \mathcal{P}(\ \mathcal{V}_{Integer}\ )$ é definida recursivamente por:*

$$
\begin{array}{rcl}
FV(\ \textbf{\textit{skip}}\ ) & = & \emptyset \\
FV(\ \textbf{\textit{assert}}\ (\ b\ )\ ) & = & FV(\ b\ ) \\
FV(\ x \coloneqq e\ ) & = & \{\ x\ \} \cup FV(\ e\ ) \\
FV(\ c_i \textbf{\textit{ ; }} c_j\ ) & = & FV(\ c_i\ ) \cup FV(\ c_j\ ) \\
FV(\ \textbf{\textit{if}}\ (\ b\ )\ \textbf{\textit{then}}\ c_t\ \textbf{\textit{else}}\ c_f\ ) & = & FV(\ b\ ) \cup FV(\ c_t\ ) \cup FV(\ c_f\ ) \\
FV(\ \textbf{\textit{while}}\ (\ b\ )\ \textbf{\textit{do}}\ c_w\ ) & = & FV(\ b\ ) \cup FV(\ c_w\ )
\end{array}
$$

Exemplo 3. *Seja p o comando da classe $\mathcal{C}ommand$,*
*tal que $p\ =\ x_7 \coloneqq 1 + x_0$ **;** $x_7 \coloneqq 1 \times x_2$ **;** $x_7 \coloneqq 1 \mod x_3$ **;** $x_7 \coloneqq 1 \div x_4$ **;** $x_7 \coloneqq x_3 \times\ x_7$.*
Então $FV(\ p\ ) = \{\ x_0,\ x_2,\ x_3,\ x_4,\ x_7\ \}$

Adiante usaremos com frequência a função **assign**, que permite obter os elementos de $\mathcal{V}_{Integer}$ que são alvo de uma atribuíção num comando.

Definição 2.9. *A função **assign** $: \mathcal{C}ommand \rightarrow \mathcal{P}(\ \mathcal{V}_{Integer}\ )$ é definida recursivamente por:*

$$
\begin{array}{rcl}
\textbf{\textit{assign}}(\ \textbf{\textit{skip}}\ ) & = & \emptyset \\
\textbf{\textit{assign}}(\ \textbf{\textit{assert}}\ (\ b\ )\ ) & = & \emptyset \\
\textbf{\textit{assign}}(\ x \coloneqq e\ ) & = & \{\ x\ \} \\
\textbf{\textit{assign}}(\ c_i \textbf{\textit{ ; }} c_j\ ) & = & \textbf{\textit{assign}}(\ c_i\ ) \cup \textbf{\textit{assign}}(\ c_j\ ) \\
\textbf{\textit{assign}}(\ \textbf{\textit{if}}\ (\ b\ )\ \textbf{\textit{then}}\ c_t\ \textbf{\textit{else}}\ c_f\ ) & = & \textbf{\textit{assign}}(\ c_t\ ) \cup \textbf{\textit{assign}}(\ c_f\ ) \\
\textbf{\textit{assign}}(\ \textbf{\textit{while}}\ (\ b\ )\ \textbf{\textit{do}}\ c_w\ ) & = & \textbf{\textit{assign}}(\ c_w\ )
\end{array}
$$

Exemplo 4. *Seja p o programa em $\mathcal{C}ommand$, tal que $p = x_1 \coloneqq 1$; $x_1 \coloneqq 1$ **;** **if** ($x_1 \leq\ x_2$) **then** $x_2 \coloneqq$ 1 **else** $x_2 \coloneqq x_3$.*
*Temos **assign**$(\ p\ ) = \{\ x_1, x_2\ \}$*

2.2 A *semântica operacional*

Duas formas normais de descrever a *semântica operacional* de uma linguagem de programação são a *semântica operacional estrutural* e a *semântica natural*. Por um lado, a *semântica operacional estrutural*, também conhecida como **small-step semantics**, define o significado individual de cada passo da computação. Por outro, a *semântica natural*, também conhecida como **big-step semantics**, define o significado geral/global das computações que terminam, *i.e.*, o que deve acontecer no final da computação [35].

A *semântica operacional* é uma forma rigorosa de formalizar o significado de uma computação. Descreve a forma como uma frase válida da linguagem deve ser interpretada, em sequências de passos computacionais. Estas sequências reflectem o significado da frase, o significado do programa. Cada expressão de uma linguagem, como **if** (b) **then** c_t **else** c_f, possui um intuito, possui um significado, um resultado desejado, *i.e.*, procura exprimir algo. A *semântica operacional* é uma ferramenta essencial para a construção, verificação e validação de uma linguagem, por providenciar uma **descrição formal** do seu comportamento.

Uma alternativa à *semântica operacional* é a *semântica denotacional* que pode ser considerada como a interpretação matemática de programas, uma vez que os programas são traduzidos para uma certa estrutura matemática e permite **raciocinar** sobre o programa nessa estrutura. Ambas as semânticas (*semântica operacional* e *semântica denotacional*) são convenientes para a descrição completa de uma linguagem [44].

2.2.1 Estados

Necessitaremos de definir a noção de estado de um programa. Como foi dito, as semânticas procuram representar o significado da execução de um programa. Os programas são executados em máquinas finitas que recorrem à manipulação de memória e recursos disponíveis ao sistema de computação. Para representar o significado da computação iremos representar o estado dos objectos que são manipulados pelo programa. Consideraremos que um estado s de execução do programa, contém toda a informação gerida pelo programa, *i.e.*, o valor de todas as variáveis que o programa manipula. Por convenção, às variáveis que não são atribuídas, mas que ocorrem nas expressões, consideramos que possuem o valor 0 por defeito.

Definição 2.10. *Chamaremos* **estado** *a uma função do tipo* $\mathcal{V}_{Integer} \to \mathbb{Z}$.
O conjunto de todos os estados possíveis será denotado por Σ. *Usamos as letras* $s, s', s_i, s_1, \ldots$ *como meta-variáveis sobre* Σ.

Uma vez que um estado é uma função, a aplicação do estado a uma variável denota o valor da variável nesse estado.

Muitas vezes precisamos de modificar o valor das variáveis, e para tal será útil usar a função, conhecida por vezes por "*override*" de um estado.

Definição 2.11. *Para* $s \in \Sigma$, $x \in \mathcal{V}_{Integer}$ *e* $v \in \mathbb{Z}$

$$s\begin{pmatrix} x \\ v \end{pmatrix}(\, y \,) = \begin{cases} v & se\ x = y \\ s(\, y \,) & se\ x \neq\ y \end{cases} \quad , \textit{para todo } y \in \mathcal{V}_{Integer}.$$

O resultado desta *"substituição"* é por si um estado, onde x, a variável substítuida, toma o valor do número inteiro v.

Exemplo 5.
No estado $\left(s\begin{pmatrix} x_0 \\ 2 \end{pmatrix}\right)\begin{pmatrix} x_0 \\ 3 \end{pmatrix}$, *o valor da variável* x_0 *é* 3.

2.2.2 Avaliação de expressões inteiras

Nesta subsecção abordamos a avaliação de expressões inteiras. Raciocinamos sobre o valor da expressão de acordo com um determinado estado. A avaliação de uma mesma expressão em estados diferentes poderá produzir resultados diferentes. Por definição, um estado avalia um número inteiro nele próprio (inteiros são por si valores). O símbolo 1 significa o valor 1, o símbolo 2 significa o valor 2 e assim consecutivamente. O estado avalia uma variável para o valor que é armazenado no estado para essa variável. Isto resulta do valor obtido pela aplicação do estado à variável. As restantes expressões são avaliadas de um modo recursivo, dependendo da respectiva construção semântica. A Definição 2.12 formaliza este conceito.

Neste trabalho não nos preocupamos com a geração de erros por parte da avaliação de expressões inteiras (caso da divisão por zero). Contudo, na avaliação do comando de atribuição, é necessário avaliar a expressão que pode gerar uma fonte de outro tipo de erro/excepção que não a considerada neste projecto. Neste projecto não se torna relevante, mas poderia ter sido considerado. Referimos esta questão como possível trabalho futuro no Capítulo 6.

Definição 2.12. *A avaliação de expressões inteiras,*
$[\![\ .\ ]\!] : \mathcal{E}xpression_{Integer} \rightarrow \Sigma \rightarrow \mathbb{Z}$ *é definida recursivamente por:*

$$\begin{array}{rcll}
[\![\ z\ ]\!]_s & = & z & ,\ se\ z \in \mathbb{Z} \\
[\![\ x\ ]\!]_s & = & s\ (\ x\ ) & ,\ se\ x \in \mathcal{V}_{Integer} \\
[\![\ -\ e\ ]\!]_s & = & -\ (\ [\![\ e\ ]\!]_s\ ) & \\
[\![\ e_1 \,\Box\, e_2\ ]\!]_s & = & [\![\ e_1\ ]\!]_s \,\Box\, [\![\ e_2\ ]\!]_s & ,\ para\ \Box \in \{\ +,\ -,\ \times\ \} \\
[\![\ e_1 \,\Box\, e_2\ ]\!]_s & = & \begin{cases} [\![\ e_1\ ]\!]_s \,\Box\, [\![\ e_2\ ]\!]_s & se\ [\![\ e_2\ ]\!]_s \neq 0 \\ 0 & de\ outro\ modo \end{cases} & ,\ para\ \Box \in \{\ div,\ mod\ \}
\end{array}$$

De seguida (Lema 2.13), estabelecemos uma propriedade fundamental da avaliação de expressões. Intuitivamente, esta propriedade afirma que se todas as variáveis presentes numa expressão de $\mathcal{E}xpression_{Integer}$ têm atribuído o mesmo valor em dois estados, então o resultado de avaliar a expressão nesses dois estados é igual.

Lema 2.13. *Para quaisquer* $e \in \mathcal{E}xpression_{Integer}$

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV\ (\ e\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )) \Rightarrow [\![\ e\ ]\!]_s\ =\ [\![\ e\ ]\!]_{s'}$$

Demonstração. Por indução estrutural em $e \in \mathcal{E}xpression_{Integer}$.

i) Para qualquer $z \in \mathbb{Z}$,
Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV\ (\ z\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )) \Rightarrow [\![\ z\ ]\!]_s\ =\ [\![\ z\ ]\!]_{s'}$$

Por definição, $[\![\ z\ ]\!]_s = z = [\![\ z\ ]\!]_{s'}$.

ii) Para qualquer $x \in \mathcal{V}_{Integer}$,
Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV\ (\ x\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )) \Rightarrow [\![\ x\ ]\!]_s\ =\ [\![\ x\ ]\!]_{s'}$$

Tendo em conta que $FV\ (\ x\ ) = \{\ x\ \}$, $s\ (\ x\ ) = s'\ (\ x\ )$.
Assim, temos $[\![\ x\ ]\!]_s = s\ (\ x\ ) = s'\ (\ x\ ) = [\![\ x\ ]\!]_{s'}$.

iii) Para qualquer $e \in \mathcal{E}xpression_{Integer}$, assumimos que por hipótese de indução temos,

$$HI:\ \ \forall s,\ s' \in \Sigma,\ (\forall y \in FV\ (\ e\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )) \Rightarrow [\![\ e\ ]\!]_s\ =\ [\![\ e\ ]\!]_{s'}$$

Vamos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV\,(\,-e\,)\,,\ s\,(\,y\,)\ =\ s'\,(\,y\,)) \Rightarrow [\![\,-e\,]\!]_s\ =\ [\![\,-e\,]\!]_{s'}$$

Por definição temos $[\![\,-e\,]\!]_s = -\,(\,[\![\,e\,]\!]_s\,)$.
Como $FV\,(\,-e\,) = FV\,(\,e\,)$, da hipótese $\forall\ y \in FV\,(\,-e\,)\,,\ s\,(\,y\,) = s'\,(\,y\,)$, segue que $\forall\ y \in FV\,(\,e\,)\,,\ s\,(\,y\,) = s'\,(\,y\,)$.
Logo, pela hipótese de indução, temos $[\![\,e\,]\!]_s = [\![\,e\,]\!]_{s'}$ e consequentemente temos também $[\![\,-e\,]\!]_s = [\![\,-e\,]\!]_{s'}$, pois, por definição, $-[\![\,e\,]\!]_s = [\![\,-e\,]\!]_s$ e $-[\![\,e\,]\!]_{s'} = [\![\,-e\,]\!]_{s'}$.

iv) Os casos em que $e = e_i \,\Box\, e_j$, com $e_i,\ e_j \in \mathcal{E}xpression_{Integer}$ e $\Box \in \{\ +,\ -,\ \times,\ \div,\ \text{mod}\ \}$, provam-se a partir das hipóteses de indução, de forma análoga ao caso iii. $\Box$

2.2.3 Avaliação de expressões booleanas

Nesta subsecção, abordamos a avaliação de expressões booleanas. Estados diferentes proderão produzir resultados diferentes, tal como acontece na avaliação de expressões inteiras. Por definição, qualquer estado avalia os valores de verdade $\top$ e $\bot$ neles próprios. As operações lógicas são avaliadas na sua forma usual, bem como as relações de comparação.

Definição 2.14. *A relação de avaliação de expressões booleanas,* $[\![\,.\,]\!] : \mathcal{E}xpression_{Boolean} \to \Sigma \to \mathbb{B}$ *é definida recursivamente por:*

$$\begin{aligned}
[\![\,\top\,]\!]_s &= \top \\
[\![\,\bot\,]\!]_s &= \bot \\
[\![\,e_1 \,\Box\, e_2\,]\!]_s &= \begin{cases} \top & \textit{se } [\![\,e_1\,]\!]_s \,\Box\, [\![\,e_2\,]\!]_s{}^{1} \\ \bot & \textit{caso contrário} \end{cases} \\
[\![\,\neg\, b\,]\!]_s &= \begin{cases} \top & \textit{se } [\![\,b\,]\!]_s\ =\ \bot \\ \bot & \textit{caso contrário} \end{cases} \\
[\![\,b_1\ \wedge\ b_2\,]\!]_s &= \begin{cases} \bot & \textit{se } [\![\,b_1\,]\!]_s\ =\ \bot \\ [\![\,b_2\,]\!]_s & \textit{caso contrário} \end{cases} \\
[\![\,b_1\ \vee\ b_2\,]\!]_s &= \begin{cases} \top & \textit{se } [\![\,b_1\,]\!]_s\ =\ \top \\ [\![\,b_2\,]\!]_s & \textit{caso contrário} \end{cases} \\
[\![\,b_1\ \to\ b_2\,]\!]_s &= \begin{cases} \top & \textit{se } [\![\,b_1\,]\!]_s = \bot \\ [\![\,b_2\,]\!]_s & \textit{caso contrário} \end{cases}
\end{aligned}$$

Notação 1. *Dado um estado s e dada uma expressão booleana b:*
usaremos a notação $s \models b$ para expressar que $[\![\,b\,]\!]_s = \top$;
usaremos a notação $s \not\models b$ para expressar que $[\![\,b\,]\!]_s = \bot$.

De imediato, surge a propriedade expressa pelo Lema 2.15, análoga à propriedade expressa pelo Lema 2.13 para expressões inteiras. Intuitivamente, afirma que para quaisquer dois estados, se todas as variáveis presentes numa expressão booleana **têm atribuido o mesmo valor**, então a expressão tem o mesmo valor de verdade para os dois estados.

Lema 2.15. *Para quaisquer $b \in \mathcal{E}xpression_{Boolean}$*

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV\,(\,b\,)\,,\ s\,(\,y\,)\ =\ s'\,(\,y\,)) \Rightarrow [\![\,b\,]\!]_s = [\![\,b\,]\!]_{s'}$$

[1] Nesta expressão o símbolo $\Box$ significa a respectiva relação binária em $\mathbb{Z}$ com a interpretação usual.

Demonstração. Por indução estrutural em $b \in \mathcal{E}xpression_{Boolean}$.

i) Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV\ (\ \top\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )) \Rightarrow [\![\ \top\ ]\!]_s = [\![\ \top\ ]\!]_{s'}$$

Por definição, $[\![\ \top\ ]\!]_s = \top = [\![\ \top\ ]\!]_{s'}$.

ii) Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV\ (\ \bot\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )) \Rightarrow [\![\ \bot\ ]\!]_s = [\![\ \bot\ ]\!]_{s'}$$

Por definição, $[\![\ \bot\ ]\!]_s = \bot = [\![\ \bot\ ]\!]_{s'}$.

iii) Para quaisquer $e_i,\ e_j \in \mathcal{E}xpression_{Integer}$ e $\Box \in \{\ \leq, <, >, \geq, =, \neq\ \}$,
Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV\left(\ e_i \Box e_j\ \right),\ s\ (\ y\ )\ =\ s'\ (\ y\ )) \Rightarrow [\![\ e_i \Box e_j\ ]\!]_s = [\![\ e_i \Box e_j\ ]\!]_{s'}$$

Como $FV\ (\ e_i \Box e_j\ ) = FV\ (\ e_i\ ) \cup FV\ (\ e_j\ )$, da hipótese $\forall\ y \in FV\ (\ e_i \Box e_j\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )$, sabemos que:

1) $\forall\ y \in FV\ (\ e_i\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )$;
2) $\forall\ y \in FV\ (\ e_j\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )$.

Daqui, pelo Lema 2.13, segue que $[\![\ e_i\ ]\!]_s = [\![\ e_i\ ]\!]_{s'}$ e $[\![\ e_j\ ]\!]_s = [\![\ e_j\ ]\!]_{s'}$.
Logo, temos $[\![\ e_i \Box e_j\ ]\!]_s = [\![\ e_i\ ]\!]_s \Box [\![\ e_j\ ]\!]_s = [\![\ e_i\ ]\!]_{s'} \Box [\![\ e_j\ ]\!]_{s'} = [\![\ e_i \Box e_j\ ]\!]_{s'}$ como pretendido.

iv) Para qualquer $b \in \mathcal{E}xpression_{Boolean}$,
Por hipótese de indução temos,

$$HI:\quad \forall s,\ s' \in \Sigma,\ (\forall y \in FV\ (\ b\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )) \Rightarrow [\![\ b\ ]\!]_s = [\![\ b\ ]\!]_{s'}$$

Queremos mostrar que

$$\forall s,\ s' \in \Sigma,\ (\forall y \in FV\ (\ \neg\, b\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )) \Rightarrow [\![\ \neg\, b\ ]\!]_s = [\![\ \neg\, b\ ]\!]_{s'}$$

Como $FV\ (\ \neg\, b\ ) = FV\ (\ b\ )$,
da hipótese $\forall\ y \in FV\ (\ b\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )$, segue $\forall\ y \in FV\ (\ b\ ),\ s\ (\ y\ )\ =\ s'\ (\ y\ )$.

Logo, de HI, vem $[\![\ b\ ]\!]_s = [\![\ b\ ]\!]_{s'}$ e consequentemente $[\![\ \neg\, b\ ]\!]_s = [\![\ \neg\, b\ ]\!]_{s'}$.

v) Os casos $b = b_i \Box b_j$, com $b_i,\ b_j \in \mathcal{E}xpression_{Boolean}$ e $\Box \in \{\ \wedge, \vee, \rightarrow\ \}$, seguem através das hipóteses de indução, de modo semelhante ao caso anterior.

$\Box$

2.2.4 Estados com erro

A noção de *"programa correcto"* que adotamos nesta tese de mestrado é a de programas que não produzem *"erros"* durante a sua execução. **Um erro não é um estado** e, para o propósito do nosso estudo, não é possível recuperar de um erro. Pelo que um erro é algo diferente dos estadoos que consideramos até aqui. No entanto, apresentaremos o *erro* como um estado especial. **O estado de erro não é um estado** no sentido de possuir informação sobre as variáveis manipuladas pelo programa. O estado de erro servirá apenas para simbolizar que algo correu mal na computação, uma *"excepção abstracta"*. Um erro resulta de uma afirmação que não é verdadeira, indicando que o programa falha, em pelo menos um ponto, a especificação dada para o programa.

Definição 2.16. *Assumimos a existência de um único* **estado de erro***, que denotamos por erro. Denotamos por* Σ_{erro} *o conjunto* $\Sigma \cup \{\ erro\ \}$ *que contém todos os estados de* Σ*, e ainda o estado de erro.*

Note-se que para o estado $s = erro$, não faz sentido falar de valor de uma variável ou de uma expressão, pelo que notações como $s\,(\,x\,)$, $s\begin{pmatrix} x \\ [\![\, z \,]\!]_s \end{pmatrix}$ e $s \models b$ não estão definidas neste caso.

2.2.5 Relação de avaliação $\rightsquigarrow$

Definida a noção de estado de um programa, podemos agora definir a relação de avaliação que irá modelar a noção de computação na nossa linguagem de programação. A relação de avaliação é ternária, denotada por $\rightsquigarrow$, e expressa como computar com as várias construções presentes na Definição 2.7. A relação $\rightsquigarrow$ relaciona um comando $c \in \mathcal{C}ommand$ e um estado $s \in \Sigma$ arbitrários, com *"o resultado da computação de c em s"*, em semântica *big-step*. A relação $\rightsquigarrow$ representa a alteração de um estado s arbitrário por um comando c que produz um estado s', exprimindo s' o resultado de uma computação bem sucedida (um elemento de Σ) ou um *erro*.

Definição 2.17. *A relação de avaliação $\rightsquigarrow$ é uma relação ternária entre $\mathcal{C}ommand$, estados e estados com erro, ou seja, $\rightsquigarrow \subseteq \mathcal{C}ommand \times \Sigma \times \Sigma_{\text{erro}}$. Usualmente, em vez de escrevermos $(\,c,\ s,\ s'\,) \in \rightsquigarrow$, escrevemos $(\,c,\ s\,) \rightsquigarrow s'$, e em vez de escrevermos $(\,c,\ s,\ s'\,) \notin \rightsquigarrow$, escrevemos $(\,c,\ s\,) \not\rightsquigarrow s'$. A relação $\rightsquigarrow$ é definida indutivamente pelas regras presentes na Tabela 2.4.*

É de salientar que em $(\,c,\ s\,) \rightsquigarrow s'$, s é necessáriamente um estado que não é *erro*, mas s' pode ser o estado de erro. De seguida analisaremos as alterações de estados produzidos por cada uma das regras.

Regra SKIP . A execução do comando **skip** não produz qualquer alteração ao estado, sendo que no final da computação o estado final é exactamente o estado inicial.

$$\frac{}{(\,\mathbf{skip},\ s\,) \rightsquigarrow s}\ \textbf{SKIP}$$

Regra ASSERT . A execução do comando **assert** $(\,b\,)$ permite, num determinado *"ponto"* de execução, afirmar propriedades que se espera que o estado satisfaça nesse *"ponto"*. O comando **assert** $(\,b\,)$ não altera o estado corrente de execução caso a afirmação seja avaliada como verdadeira no presente estado. Neste caso o estado não é alterado.

$$\frac{s \models b}{(\,\mathbf{assert}\ (\,b\,),\ s\,) \rightsquigarrow s}\ \textbf{ASSERT}$$

Regra ASSERT - Error . Por outro lado, o comando **assert** $(\,b\,)$, deve indicar que o programa entra no incumprimento da afirmação/asserção b, avaliando para o estado de erro, caso b não seja satisfeito no estado em causa.

$$\frac{s \not\models b}{(\,\mathbf{assert}\ (\,b\,),\ s\,) \rightsquigarrow erro}\ \textbf{ASSERT - Error}$$

Regra ASSIGN . A execução do comando $x\ :=\ e$ permite, num determinado *"ponto"* de execução, atribuir um novo valor inteiro a uma variável. Efectua-se uma substituição no estado, onde o valor de x passa a tomar o valor resultante da avaliação da expressão e.

$$\frac{}{(\,x\ :=\ e,\ s\,) \rightsquigarrow s\begin{pmatrix} x \\ [\![\, e \,]\!]_s \end{pmatrix}}\ \textbf{ASSIGN}$$

Regra COMPOSITION . O comando c_i **;** c_j permite a composição de comandos, começando

$$\frac{}{(\ \textbf{skip},\ s\ ) \rightsquigarrow s}\ \textbf{SKIP}$$

$$\frac{s \models b}{(\ \textbf{assert}\ (\ b\ )\ ,\ s\ ) \rightsquigarrow s}\ \textbf{ASSERT}$$

$$\frac{s \not\models b}{(\ \textbf{assert}\ (\ b\ )\ ,\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{ASSERT - Error}$$

$$\frac{}{(\ x\ :=\ e,\ s\ ) \rightsquigarrow s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}}\ \textbf{ASSIGN}$$

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow \mathit{erro}}{(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{COMPOSITION - Break}$$

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow s'' \quad s'' \neq \mathit{erro} \quad (\ c_j,\ s''\ ) \rightsquigarrow s'}{(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow s'}\ \textbf{COMPOSITION}$$

$$\frac{s \models b \quad (\ c_t,\ s\ ) \rightsquigarrow s'}{(\ \textbf{if}\ (\ b\ )\ \textbf{then}\ c_t\ \textbf{else}\ c_f,\ s\ ) \rightsquigarrow s'}\ \textbf{IF - True}$$

$$\frac{s \not\models b \quad (\ c_f,\ s\ ) \rightsquigarrow s'}{(\ \textbf{if}\ (\ b\ )\ \textbf{then}\ c_t\ \textbf{else}\ c_f,\ s\ ) \rightsquigarrow s'}\ \textbf{IF - False}$$

$$\frac{s \not\models b}{(\ \textbf{while}\ (\ b\ )\ \textbf{do}\ c_w,\ s\ ) \rightsquigarrow s}\ \textbf{WHILE - False}$$

$$\frac{s \models b \quad (\ c_w,\ s\ ) \rightsquigarrow \mathit{erro}}{(\ \textbf{while}\ (\ b\ )\ \textbf{do}\ c_w,\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{WHILE - Break}$$

$$\frac{s \models b \quad (\ c_w,\ s\ ) \rightsquigarrow s'' \quad s'' \neq \mathit{erro} \quad (\ \textbf{while}\ (\ b\ )\ \textbf{do}\ c_w,\ s''\ ) \rightsquigarrow s'}{(\ \textbf{while}\ (\ b\ )\ \textbf{do}\ c_w,\ s\ ) \rightsquigarrow s'}\ \textbf{WHILE - True}$$

Tabela 2.4: Regras da semântica operacional

por executar o comando c_i, e executado posteriormente o comando c_j, a partir do estado resultante da execução de c_i.

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow s'' \quad s'' \neq \mathit{erro} \quad (\ c_j,\ s''\ ) \rightsquigarrow s'}{(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow s'}\ \textbf{COMPOSITION}$$

Regra COMPOSITION - Break . Se num programa c_i **;** c_j ocorre um erro na computação do comando c_i, todo o programa passa ao estado de erro, sendo desnecessária qualquer avaliação posterior.

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow \mathit{erro}}{(\ c_i\ ;\ c_j,\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{COMPOSITION - Break}$$

Neste estudo, não é possível recuperar de um erro, e logo que é encontrada uma afirmação/asserção no programa que não é satisfeita, a computação termina.

Regra IF - True . A execução do comando **if** (b) **then** c_t **else** c_f permite, num determinado *"ponto"* de execução, expressar uma computação condicional. Se o estado validar a expressão booleana b, no *"ponto"* em que o condicional é executado, a computação prossegue com c_t e o mesmo estado.

$$\frac{s \models b \quad (\ c_t,\ s\ ) \rightsquigarrow s'}{(\ \textbf{if}\ (\ b\ )\ \textbf{then}\ c_t\ \textbf{else}\ c_f,\ s\ ) \rightsquigarrow s'}\ \textbf{IF - True}$$

Regra IF - False . No caso em que o estado não validar a expressão booleana b, o programa avalia de acordo com o comando c_f.

$$\frac{s \not\models b \quad (\ c_f,\ s\ ) \rightsquigarrow s'}{(\ \textbf{if}\ (\ b\ )\ \textbf{then}\ c_t\ \textbf{else}\ c_f,\ s\ ) \rightsquigarrow s'}\ \textbf{IF - False}$$

O comando **while** (b) **do** c_w permitirá ciclos num determinado *"ponto"* da computação. O corpo c_w do ciclo é executado um número determinado de vezes, sendo esse número de execuções controlado pela condição booleana b.

Regra WHILE - False . Se a condição booleana de ciclo for falsa, *i.e.*, o estado atual da execução não valida a condição, nada mais é executado e é retornado o estado prévio, com o estado das computações executadas até então.

$$\frac{s \not\models b}{(\ \textbf{while}\ (\ b\ )\ \textbf{do}\ c_w,\ s\ ) \rightsquigarrow s}\ \textbf{WHILE - False}$$

Regra WHILE - Break . Caso a condição booleana de ciclo seja validada no estado atual de execução, queremos executar o corpo do ciclo, e o resultado dessa computação é transmitido às próximas computações. Assim se $s \models b$ e (c_w, s) $\rightsquigarrow s''$, caso $s'' = \mathit{erro}$, devemos indicar que, independentemente de posteriores computações, o programa termina no estado de erro.

$$\frac{s \models b \quad (\ c_w,\ s\ ) \rightsquigarrow \mathit{erro}}{(\ \textbf{while}\ (\ b\ )\ \textbf{do}\ c_w,\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{WHILE - Break}$$

Regra WHILE - True . Caso contrário, o resultado da computação do corpo do ciclo s'' deve ser transmitido ao próximo passo de computação, que se espera ser outra iteração do ciclo, se b ainda se verificar em s''.

$$\frac{s \models b \quad (\ c_w,\ s\ ) \rightsquigarrow s'' \quad s'' \neq \mathit{erro} \quad (\ \textbf{while}\ (\ b\ )\ \textbf{do}\ c_w,\ s''\ ) \rightsquigarrow s'}{(\ \textbf{while}\ (\ b\ )\ \textbf{do}\ c_w,\ s\ ) \rightsquigarrow s'}\ \textbf{WHILE - True}$$

Note-se que se $(c,\ s) \rightsquigarrow s'$ (*i.e.* se o triplo $(c,\ s,\ s')$ pertence à relação $\rightsquigarrow$), então existe uma "árvore de derivação" construída com as regras da *semântica operacional* cuja conclusão/raíz é $(c,\ s) \rightsquigarrow s'$.

2.2.6 Propriedades elementares da relação de avaliação

O nosso modelo de computação é determinista, *i.e.*, dado um comando e um estado poderemos computar um único estado. Esta propriedade fundamental de $\rightsquigarrow$ é provada adiante, no Teorema 2.19. Além desta propridade, provamos também na próxima proposição que as variáveis que não ocorrem num comando não influenciam, nem são alteradas pela execução do comando.

Proposição 2.18. $\forall\ c \in \mathcal{C}ommand,\ s \in \Sigma,\ s' \in \Sigma_{\text{erro}},\ \ y \in \mathcal{V}_{Integer},$
$(c,\ s) \rightsquigarrow s' \wedge\ y \notin FV\ (c) \Rightarrow \forall\ z \in \mathbb{Z}$

$$\begin{cases} 1) & \left(c,\ s \begin{pmatrix} y \\ z \end{pmatrix} \right) \rightsquigarrow s' \begin{pmatrix} y \\ z \end{pmatrix} & , se\ s' \neq \text{erro} \\ 2) & \left(c,\ s \begin{pmatrix} y \\ z \end{pmatrix} \right) \rightsquigarrow \text{erro} & , se\ s' = \text{erro} \end{cases}$$

Demonstração. Por indução nos comandos $c \in \mathcal{C}ommand$

i) Caso $c = \mathbf{skip}$.

Atendendo à semântica operacional, existe uma única regra aplicável ao comando **skip**, e teremos que ter $s' = s$ (e portanto $s' \neq erro$). Usando a regra **SKIP** temos, como pretendido, $\left(\mathbf{skip},\ s \begin{pmatrix} y \\ z \end{pmatrix} \right) \rightsquigarrow s \begin{pmatrix} y \\ z \end{pmatrix}$.

ii) Caso $c = \mathbf{assert}\ (b)$.

Caso $s \models b$, temos que ter $(\mathbf{assert}\ (b),\ s) \rightsquigarrow s$, ou seja $s' = s$. Análogo ao caso anterior.

Caso $s \not\models b$, temos $(\mathbf{assert}\ (b),\ s) \rightsquigarrow erro$ e portanto $s' = erro$.
Dado que $s \not\models b$, $\forall\ y \notin FV\ (\mathbf{assert}\ (b))$, e $\forall\ z \in \mathbb{Z}$,
também $s \begin{pmatrix} y \\ z \end{pmatrix} \not\models b$, atendendo ao lema 2.15.

Logo, pela regra **ASSERT - Error**, tem-se $\left(\mathbf{assert}\ (b),\ s \begin{pmatrix} y \\ z \end{pmatrix} \right) \rightsquigarrow erro$,
como pretendido.

iii) Caso $c = x\ \coloneqq\ e$.

Atendendo à semântica operacional, temos que ter $s' = s \begin{pmatrix} x \\ [\![e]\!]_s \end{pmatrix}$, e portanto $s' \neq erro$.

Usando a regra **ASSIGN**, $\left(x\ \coloneqq\ e,\ s \begin{pmatrix} y \\ z \end{pmatrix} \right) \rightsquigarrow s \begin{pmatrix} y \\ z \end{pmatrix} \begin{pmatrix} x \\ [\![e]\!] \end{pmatrix}$.

Mas, com $y \notin FV\ (c)$, $y \neq\ x$, pelo que

$$s \begin{pmatrix} x \\ [\![e]\!] \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} = s \begin{pmatrix} y \\ z \end{pmatrix} \begin{pmatrix} x \\ [\![e]\!] \end{pmatrix}$$

e tem-se o pretendido.

iv) Caso $c = c_i\ \mathbf{;}\ c_j$.
Por hipótese de indução temos,

HI_i: $\forall\ s \in \Sigma,\ s' \in \Sigma_{erro},\ y \in \mathcal{V}_{Integer},\ \ (\ c_i,\ s\ ) \rightsquigarrow s' \wedge\ y \notin FV\ (\ c_i\ ) \Rightarrow \forall\ z \in \mathbb{Z}$

$$\begin{array}{lll} 1) & \left(c_i,\ s\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow s'\begin{pmatrix} y \\ z \end{pmatrix} & ,se\ s' \neq erro \\ 2) & \left(c_i,\ s\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow erro & ,se\ s' = erro \end{array}$$

HI_j: $\forall\ s \in \Sigma,\ s' \in \Sigma_{erro},\ y \in \mathcal{V}_{Integer},\ (\ c_j,\ s\ ) \rightsquigarrow s' \wedge\ y \notin FV\ (\ c_j\ ) \Rightarrow \forall\ z \in \mathbb{Z}$

$$\begin{array}{lll} 1) & \left(c_j,\ s\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow s'\begin{pmatrix} y \\ z \end{pmatrix} & ,se\ s' \neq erro \\ 2) & \left(c_j,\ s\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow erro & ,se\ s' = erro \end{array}$$

Queremos mostrar que $\forall\ s \in \Sigma,\ s' \in \Sigma_{erro},\ y \in \mathcal{V}_{Integer},\ (\ c_i\ \mathbf{;}\ c_j,\ s\ ) \rightsquigarrow s' \wedge\ y \notin FV\ (\ c_i\ \mathbf{;}\ c_j\ ) \Rightarrow$ $\forall\ z \in \mathbb{Z}$

$$\begin{array}{lll} 1) & \left(c_i\ \mathbf{;}\ c_j,\ s\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow s'\begin{pmatrix} y \\ z \end{pmatrix} & ,se\ s' \neq erro \\ 2) & \left(c_i\ \mathbf{;}\ c_j,\ s\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow erro & ,se\ s' = erro \end{array}$$

Caso **COMPOSITION - Break**.

Suponhamos que $(\ c_i\ \mathbf{;}\ c_j,\ s\ ) \rightsquigarrow s'$ segue pela regra **COMPOSITION - Break**.

Então, $s' = erro$ e temos $(\ c_i,\ s\ ) \rightsquigarrow erro$. Dado $y \notin FV\ (\ c_i\ \mathbf{;}\ c_j\ )$, $y \notin FV\ (\ c_i\ )$ e por HI_i temos $\left(c_i,\ s\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow erro$, para qualquer $z \in \mathbb{Z}$.

Aplicando a regra **COMPOSITION - Break** retiramos $\left(c_i\ \mathbf{;}\ c_j,\ s\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow erro$.

Caso **COMPOSITION**.

Suponhamos que $(\ c_i\ \mathbf{;}\ c_j,\ s\ ) \rightsquigarrow s'$ segue pela regra **COMPOSITION**.

Então, existe $s'' \neq erro$ tal que $(\ c_i,\ s\ ) \rightsquigarrow s''$ e $(\ c_j,\ s''\ ) \rightsquigarrow s'$.

Tomemos $z \in \mathbb{Z}$ arbitrário. Assumindo que $y \notin FV\ (\ c_i\ \mathbf{;}\ c_j\ )$, $y \notin FV\ (\ c_i\ )$ e $y \notin FV\ (\ c_j\ )$.

Daqui segue que:

1) por HI_i, $\left(c_i,\ s\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow s''\begin{pmatrix} y \\ z \end{pmatrix}$,

2) por HI_j, $\left(c_j,\ s''\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow s'\begin{pmatrix} y \\ z \end{pmatrix}$, se $s' \neq erro$

e também $\left(c_j,\ s''\begin{pmatrix} y \\ z \end{pmatrix}\right) \rightsquigarrow erro$, se $s' = erro$.

Assim sendo, basta aplicar a regra **COMPOSITION** para obter o resultado pretendido.

v) Caso $c =$ **if** $(\ b\ )$ **then** c_t **else** c_f.

Há dois casos, consoante $(\ c,\ s\ ) \rightsquigarrow s'$ é obtido por **IF - True** ou **IF - False**.

Em cada um dos casos a prova segue com o auxílio da hipótese de indução

vi) Caso $c =$ **while** $(\ b\ )$ **do** c_w.

Suponhamos que $(\ c,\ s\ ) \rightsquigarrow s'$ segue pela regra **WHILE - False**.

Então, $s' = s$ (com $s' \neq erro$) e $s \not\models b$.

Assim, tomando $y \notin FV\ (\ c\ )$, tem-se $y \notin FV\ (\ b\ )$ e pelo Lema 2.15 segue que

$$s\begin{pmatrix} y \\ z \end{pmatrix} \not\models b, \text{ para todo } z \in \mathbb{Z}.$$

Consequentemente, usando a regra **WHILE - False** vem que

$$\left(\textbf{while} \ (\ b \ ) \ \textbf{do} \ c_w, \ s \begin{pmatrix} y \\ z \end{pmatrix} \right) \rightsquigarrow s \begin{pmatrix} y \\ z \end{pmatrix}$$

como pretendido (note-se que com $s = s'$, $s \begin{pmatrix} y \\ z \end{pmatrix} = s' \begin{pmatrix} y \\ z \end{pmatrix}$).

Os casos **WHILE - Break** e **WHILE - True** seguem com auxílio da hipótese de indução e são análogos aos casos **COMPOSITION - Break** e **COMPOSITION** respectivamente.

□

Uma propriedade elementar da relação de avaliação que surge naturalmente é o determinismo da relação de avaliação. Intuitivamente, esta propriedade afirma que o resultado de avaliar um comando é sempre o mesmo, *i.e.*, caso existam duas computações de um qualquer comando, num qualquer estado, o resultado de ambas as computações é o mesmo.

Teorema 2.19. *(Determinismo da relação de avaliação)*

$$\forall s \in \Sigma, \ \forall s', s'' \in \Sigma_{\text{erro}}, \ \forall c \in \mathcal{C}ommand, \ (\ c, \ s \ ) \rightsquigarrow s' \wedge (\ c, \ s \ ) \rightsquigarrow s'' \Rightarrow s' = s''$$

Demonstração. Por indução na relação de avaliação.

Caso **SKIP**.
Supondo $(\ \textbf{skip}, \ s \ ) \rightsquigarrow s'$ e $(\ \textbf{skip}, \ s \ ) \rightsquigarrow s''$, necessariamente $s = s'$ e $s = s''$.
Por transitividade da relação de igualdade, $s' = s''$.

Caso **ASSERT**.
Supondo que $c = \textbf{assert} \ (\ b \ )$ e $(\ \textbf{assert} \ (\ b \ ), \ s \ ) \rightsquigarrow s'$ é obtido pela regra **ASSERT**, tem-se $s' = s$ e $s \models b$.
Assim, de $(\ c, \ s \ ) \rightsquigarrow s''$ conclui-se $s'' = s$, pois a regra **ASSERT - Error** não é aplicável, e consequentemente $s'' = s'$.

Caso **ASSERT - Error**.
Supondo que $c = \textbf{assert} \ (\ b \ )$ e $(\ \textbf{assert} \ (\ b \ ), \ s \ ) \rightsquigarrow s'$ é obtido pela regra **ASSERT**, neste caso $s' = \textit{erro}$ e $s \not\models b$.
Logo $s' = s''$, por um raciocínio análogo ao do caso anterior.

Caso **ASSIGN**.
Supondo $(\ x \ := \ e, \ s \ ) \rightsquigarrow s'$ e $(\ x \ := \ e, \ s \ ) \rightsquigarrow s''$,
necessariamente $s' = s \begin{pmatrix} x \\ [\![\ e \ ]\!]_s \end{pmatrix}$ e $s'' = s \begin{pmatrix} x \\ [\![\ e \ ]\!]_s \end{pmatrix}$.
Logo $s' = s''$.

Caso **COMPOSITION - Break**.
Supondo que $c = c_i \ \textbf{;} \ c_j$ e $(\ c_i \ \textbf{;} \ c_j, \ s \ ) \rightsquigarrow s'$ é obtido pela regra **COMPOSITION - Break**, $s' = \textit{erro}$ e $(\ c_i, \ s \ ) \rightsquigarrow \textit{erro}$.
Assim, por hipótese de indução,

$$\forall \ s'' \in \Sigma_{erro}, \ (\ c_i, \ s \ ) \rightsquigarrow s''' \Rightarrow s''' = \textit{erro} \tag{2.1}$$

Logo, $s'' = \textit{erro}$, pois de outro modo teríamos que ter usado a regra **COMPOSITION**

e existiria $s''' \neq erro$ tal que $(c_i, s) \rightsquigarrow s'''$, contrariando (2.1).

Caso **COMPOSITION**.

Supondo que $c = c_i \,\mathbf{;}\, c_j$ e $(c_i \,\mathbf{;}\, c_j, s) \rightsquigarrow s'$ é obtido pela regra **COMPOSITION**, existe $s''' \neq erro$ tal que $(c_i, s) \rightsquigarrow s'''$ e $(c_j, s''') \rightsquigarrow s'$.
Atendendo à hipótese de indução relativa a $(c_i, s) \rightsquigarrow s'''$,
não podemos ter $(c_i, s) \rightsquigarrow erro$ e portanto $(c_i, s) \rightsquigarrow s'$ tem também que ser obtido pela regra **COMPOSITION**, pelo que existe s'''' tal que $(c_i, s) \rightsquigarrow s''''$ e $(c_j, s'''') \rightsquigarrow s''$.
Daqui, atendendo à hipótese de indução relativa a $(c_i, s) \rightsquigarrow s'''$
tira-se $s''' = s''''$ e atendendo à hipótese de indução relativa a $(c_j, s''') \rightsquigarrow s'$
tira-se $s' = s''$, como pretendido.

Caso **IF - True**.

Supondo que $c = \textbf{if} (b) \textbf{ then } c_t \textbf{ else } c_f$ e $(c, s) \rightsquigarrow s'$ é obtido pela regra **IF - True**, tem-se $s \models b$ e $(c_t, s) \rightsquigarrow s'$.
Assim, $(c, s) \rightsquigarrow s''$ também tem que ser obtido pela regra **IF - True** e tem-se $(c_t, s) \rightsquigarrow s''$.
Pela hipótese de indução podemos então concluir que $s'' = s'$.

Caso **IF - False**.

Análogo ao caso anterior.

Caso **WHILE - False**.

Supondo que $c = \textbf{while} (b) \textbf{ do } c_w$ e que $(c, s) \rightsquigarrow s'$ é obtido pela regra **WHILE - False**, tem-se $s' = s$ e $s \not\models b$.
Uma vez que $s \not\models b$, $(c, s) \rightsquigarrow s''$ também só pode ser obtido pela regra **WHILE - False**, donde $s'' = s = s'$.

Caso **WHILE - Break**.

Análogo ao caso da regra **COMPOSITION - Break**.

Caso **WHILE - True**.

Análogo ao caso da regra **COMPOSITION**.

□

2.2.7 Equivalência de programas

Uma noção fundamental em semânticas de liguagens de programação é a noção de *equivalência de programas*. Necessitaremos de raciocinar sobre a equivalência entre dois comandos. Essencialmente, dois programas são equivalentes se a partir de um qualquer estado produzem estados iguais. Uma propriedade desejável acerca de transformações de programas noutros programas é a de que as transformações produzem programas equivalentes aos programas dados. Embora nesta tese de mestrado não se faça um estudo das propriedades das 3 (três) primeiras transformações envolvidas no método de BMC (definidas na Secção 3.3), espera-se que as transformações $\mathcal{T}_2$ e $\mathcal{T}_3$ (2ª e 3ª transformações respectivamente) produzam programas equivalentes. Já a transformação $\mathcal{T}_1$, uma vez que faz apenas uma expansão limitada dos *"whiles"* pode não produzir programas equivalentes.

Definição 2.20. *A relação binária $\approx$ sobre $\mathcal{C}ommand$ é definida por:*

$$c_1 \approx c_2 \textit{ sse } (c_1, s) \rightsquigarrow s' \Leftrightarrow (c_2, s) \rightsquigarrow s',$$

para qualquer $s \in \Sigma$, $s' \in \Sigma_{\text{erro}}$ e $c_1, c_2 \in \mathcal{C}ommand$.

Como seria de esperar, a relação de equivalência de programas é uma **relação de equivalência**, *i.e.*, é reflexiva, é simétrica e é transitiva.

Lema 2.21. *(Reflexividade da relação* $\approx$ *)*
Para qualquer $c \in \mathcal{C}ommand$*,*

$$c \approx c.$$

Demonstração.
Para qualquer $c \in \mathcal{C}ommand$, é obvio que para quaisquer $s \in \Sigma,\ s' \in \Sigma_{erro}$, $(\ c,\ s\ ) \rightsquigarrow s'$ sse $(\ c,\ s\ ) \rightsquigarrow s'$.
Logo, temos $c \approx c$.

□

Lema 2.22. *(Simetria da relação* $\approx$ *)*
Para quaisquer $c_1,\ c_2 \in \mathcal{C}ommand$*,*

$$c_1 \approx c_2 \Rightarrow c_2 \approx c_1.$$

Demonstração. Suponhamos que

$$c_1 \approx c_2,\ \textit{i.e., para quaisquer } s \in \Sigma,\ s' \in \Sigma_{erro},\ (\ c_1,\ s\ ) \rightsquigarrow s' \textit{ sse } (\ c_2,\ s\ ) \rightsquigarrow s'.$$

Queremos mostrar que

$$c_2 \approx c_1,\ \textit{i.e., para quaisquer } s \in \Sigma,\ s' \in \Sigma_{erro},\ (\ c_2,\ s\ ) \rightsquigarrow s' \textit{ sse } (\ c_1,\ s\ ) \rightsquigarrow s'$$

Pela hipótese, $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_2,\ s\ ) \rightsquigarrow s'$.
Logo, $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_1,\ s\ ) \rightsquigarrow s'$.

□

Lema 2.23. *(Transitividade da relação* $\approx$ *)*
Para quaisquer $c_1,\ c_2,\ c_3 \in \mathcal{C}ommand$*,*

$$(\ c_1 \approx c_2\ ) \wedge (\ c_2 \approx c_3\ ) \Rightarrow c_1 \approx c_3.$$

Demonstração. Suponhamos que
H_a: $c_1 \approx c_2$, *i.e.*, para quaisquer $s_a \in \Sigma,\ s'_a \in \Sigma_{erro}$, $(\ c_1,\ s_a\ ) \rightsquigarrow s'_a$ sse $(\ c_2,\ s_a\ ) \rightsquigarrow s'_a$
H_b: $c_2 \approx c_3$, *i.e.*, para quaisquer $s_b \in \Sigma,\ s'_b \in \Sigma_{erro}$, $(\ c_2,\ s_b\ ) \rightsquigarrow s'_b$ sse $(\ c_3,\ s_b\ ) \rightsquigarrow s'_b$
Queremos mostrar que
$c_1 \approx c_3$, *i.e.*, para quaisquer $s \in \Sigma,\ s' \in \Sigma_{erro}$, $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_3,\ s\ ) \rightsquigarrow s'$

Da hipótese H_a, em particular, tomando $s_a = s$ e $s'_a = s'$, temos $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_2,\ s\ ) \rightsquigarrow s'$
Da hipótese H_b, em particular, tomando $s_b = s$ e $s'_b = s'$, temos $(\ c_2,\ s\ ) \rightsquigarrow s'$ sse $(\ c_3,\ s\ ) \rightsquigarrow s'$
Logo, $(\ c_1,\ s\ ) \rightsquigarrow s'$ sse $(\ c_3,\ s\ ) \rightsquigarrow s'$

□

Proposição 2.24. *A relação* $\approx$ *é uma relação de equivalência.*

Demonstração. Consequência dos lemas 2.21, 2.22 e 2.23. □

2.3 Correcção de programas

Durante a execução de um programa, queremos ser capazes de descrever o que não deve acontecer, ou de igual forma, o que deve sempre acontecer. Necessitamos de uma forma de representar propriedades de "safety" nos comandos.

Para qualquer $c \in \mathcal{C}ommand$, usaremos a notação *safe*(c), para indicar uma propriedade sobre comandos, com o intuito de captar a ideia de que o programa c executa sem erros. A noção de erro, neste contexto, é a execução dum programa que não satisfaz todas as afirmações presentes na sua específicação, *i.e.*, *safe*(c) indica que o programa só avalia por estados que verificam as afirmações/asserções expressadas pela específicação. Desta forma, conseguiremos garantir que um *"comando safe"* satisfaz todas as suas especificações/afirmações. Em particular, queremos raciocinar sobre as propriedades de *safety* de um comando, em relação a um estado inicial em particular.

Assim, a noção expressa em *safe*(c) indica que um programa satisfaz esta propriedade, se não existir nenhum triplo que resultou de um comando que foi avaliado num *erro* de execução, *i.e.*, não existe nenhum triplo da forma ($c,\ s,\ erro$) na relação de avaliação $\rightsquigarrow$. Deste modo raciocinar sobre a propriedade *safe*(c) é raciocinar exactamente sobre a definição da relação $\rightsquigarrow$.

Definição 2.25. *Dado* $c \in \mathcal{C}ommand$ *e* $s \in \Sigma$*,*

$$\text{safe}(\ c,\ s\ ) \Leftrightarrow_{def} (\ c,\ s\ ) \not\rightsquigarrow \text{erro}$$

Recorde-se que a notação ($c,\ s$) $\not\rightsquigarrow$ *erro* significa ($c,\ s,\ erro$) $\notin \rightsquigarrow$, *i.e.*, significa que não existe na relação $\rightsquigarrow$ o triplo ($c,\ s,\ erro$).

De igual modo, um comando é considerado *safe* se e só se é *safe* para todos os estados.

Definição 2.26. *Dado* $c \in \mathcal{C}ommand$*,*

$$\text{safe}(\ c\ ) \Leftrightarrow_{def} \forall\ s \in \Sigma :\ \text{safe}(\ c,\ s\ )$$

De seguida estudaremos duas propriedades acerca do predicado *safe* (Lema 2.28 e Lema 2.27), que serão necessárias no Capítulo 4.

Uma propriedade imediata é a de que um programa equivalente a um programa *safe* é também *safe*.

Proposição 2.27.

$$\forall\ c_1,\ c_2 \in \mathcal{C}ommand,\ c_1\ \approx\ c_2 \Rightarrow \text{safe}(\ c_1\ )\ sse\ \text{safe}(\ c_2\ )$$

Demonstração.
Suponhamos que $c_1\ \approx\ c_2$, *i.e.*, $\forall\ s \in \Sigma,\ s' \in \Sigma_{erro},\ (\ c_1,\ s\ ) \rightsquigarrow s' \Leftrightarrow (\ c_2,\ s\ ) \rightsquigarrow s'$.
Queremos mostrar que *safe*(c_1) *sse* *safe*(c_2).

Supondo *safe*(c_1), *i.e.*, $\forall\ s \neq erro,\ (\ c_1,\ s\ ) \not\rightsquigarrow erro$. Então, da suposição $c_1\ \approx\ c_2$, segue $\forall\ s \neq erro,\ (\ c_2,\ s\ ) \not\rightsquigarrow erro$, e portanto temos *safe*(c_2).

O recíproco tem demonstração análoga. □

A propriedade expressa pelo Lema 2.28, intuitivamente, afirma que, para quaisquer dois estados $s_a,\ s_b \in \Sigma$, se todas as variáveis presentes numa expressão $b' \in \mathcal{E}xpression_{Boolean}$ **não são atribuídas** pelo comando $c \in \mathcal{C}ommand$, *i.e.*, se o comando não manipula as variáveis livres da expressão, mas s_b é resultado da avaliação em s_a, então avaliar b' no estado s_a terá o mesmo resultado de avaliar b' em s_b, e vice-versa.

Lema 2.28. $\forall\ c \in \mathcal{C}ommand,\ \forall s_a, s_b \in \Sigma,\ \forall\ b \in \mathcal{E}xpression_{Boolean}$:

$$\begin{gathered}(\ FV\ (\ b'\ ) \cap \textbf{\textit{assign}}\,(\ c\ ) = \emptyset\ ) \wedge (\ c,\ s_a\ ) \leadsto s_b \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{gathered}$$

Demonstração. Por indução estrutural em $c \in \mathcal{C}ommand$.

i) Queremos mostrar que para quaisquer $b' \in \mathcal{E}xpression_{Boolean}$ e $s_a,\ s_b \in \Sigma$

$$\begin{gathered}(\ FV\ (\ b'\ ) \cap \textbf{assign}\,(\ \textbf{skip}\ ) = \emptyset\ ) \wedge (\ \textbf{skip},\ s_a\ ) \leadsto s_b \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{gathered}$$

De $(\ \textbf{skip},\ s_a\ ) \leadsto s_b$, atendendo à definição da relação $\leadsto$, conluímos que $s_a = s_b$ e a equivalência é imediata.

ii) Queremos mostrar que para quaisquer $b' \in \mathcal{E}xpression_{Boolean}$ e $s_a,\ s_b \in \Sigma$

$$\begin{gathered}(\ FV\ (\ b'\ ) \cap \textbf{assign}\,(\ \textbf{assert}\ (\ b\ )\ ) = \emptyset\ ) \wedge (\ \textbf{assert}\ (\ b\ )\ ,\ s_a\ ) \leadsto s_b \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{gathered}$$

Uma vez que $s_b \neq erro$ ($s_b \in \Sigma$), atendendo à definição da relação $\leadsto$, sabemos que $s_a = s_b$ logo, $s_a \models b'$ sse $s_b \models b'$.

iii) Queremos mostrar que para quaisquer $b' \in \mathcal{E}xpression_{Boolean}$ e $s_a,\ s_b \in \Sigma$

$$\begin{gathered}(\ FV\ (\ b'\ ) \cap \textbf{assign}\,(\ x\ \coloneqq\ e\ ) = \emptyset\ ) \wedge (\ x\ \coloneqq\ e,\ s_a\ ) \leadsto s_b \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{gathered}$$

Atendendo à definição da relação $\leadsto$, sabemos que $s_b = s_a \begin{pmatrix} x \\ [\![\ e\ ]\!]_{s_a} \end{pmatrix}$.

Temos que mostrar que $s_a \models b'$ sse $s_a \begin{pmatrix} x \\ [\![\ e\ ]\!]_{s_a} \end{pmatrix} \models b'$.

Pela suposição, $FV\ (\ b'\ ) \cap \textbf{assign}\,(\ x\ \coloneqq\ e\ ) = \emptyset$.
Logo, $x \notin FV\ (\ b'\ )$. Assim, pelo Lema 2.15, $s_a \models b'$ sse $s_b \models b'$.

iv) Vamos mostrar que

$$\begin{gathered}(\ FV\ (\ b'\ ) \cap \textbf{assign}\,(\ \textbf{if}\ (\ b\ )\ \textbf{then}\ c_t\ \textbf{else}\ c_f\ ) = \emptyset\ ) \wedge (\ \textbf{if}\ (\ b\ )\ \textbf{then}\ c_t\ \textbf{else}\ c_f,\ s_a\ ) \leadsto s_b \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{gathered}$$

Como hipóteses de indução temos

$$HI_t : \quad \begin{gathered}(\ FV\ (\ b'\ ) \cap \textbf{assign}\,(\ c_t\ ) = \emptyset\ ) \wedge (\ c_t,\ s_a\ ) \leadsto s_b \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{gathered}$$

$$HI_f : \quad \begin{gathered}(\ FV\ (\ b'\ ) \cap \textbf{assign}\,(\ c_f\ ) = \emptyset\ ) \wedge (\ c_f,\ s_a\ ) \leadsto s_b \\ \Downarrow \\ (\ s_a \models b'\ ) \Leftrightarrow (\ s_b \models b'\ )\end{gathered}$$

Da hipótese $FV\ (\ b'\ ) \cap \textbf{assign}\,(\ \textbf{if}\ (\ b\ )\ \textbf{then}\ c_t\ \textbf{else}\ c_f\ ) = \emptyset$, segue em particular $FV\ (\ b'\ ) \cap \textbf{assign}\,(\ c_t\ ) = \emptyset$ e $FV\ (\ b'\ ) \cap \textbf{assign}\,(\ c_f\ ) = \emptyset$.

Caso $s_a \models b$: Temos que ter $(\ c_t,\ s_a\ ) \rightsquigarrow s_b$. Daqui e de $FV(\ b'\ ) \cap \textbf{assign}(\ c_t\ ) = \emptyset$, segue pela hipótese de indução HI_t $s_a \models b'$ sse $s_b \models b'$.

Caso $s_a \not\models b$: Temos que ter $(\ c_f,\ s_a\ ) \rightsquigarrow s_b$. Daqui e de $FV(\ b'\ ) \cap \textbf{assign}(\ c_f\ ) = \emptyset$, segue pela hipótese de indução HI_f $s_a \models b'$ sse $s_b \models b'$.

vi) Os casos relativos aos comandos de composição (c_i **;** c_j) e while (**while** (b) **do** c_w) seguem por razões semelhantes às dos casos anteriores.

□

O lema seguinte apresenta uma caracterização da noção de *safe* para a composição de comandos, em termos dos subcomandos.

Lema 2.29. $\forall c_i, c_j \in \mathcal{C}ommand,\ \forall\ s \in \Sigma,$

$$\text{safe}(\ c_i \,\boldsymbol{;}\, c_j,\ s\ )$$
$$sse$$
$$\text{safe}(\ c_i,\ s\ ) \wedge (\ \forall\ s'' \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow s'' \Rightarrow \text{safe}(\ c_j,\ s''\ )\ )$$

Demonstração.
"⇒": Por definição, de $safe(\ c_i \,\boldsymbol{;}\, c_j,\ s\ )$ temos $\forall\ s \in \Sigma :\ (\ c_i \,\boldsymbol{;}\, c_j,\ s\ ) \not\rightsquigarrow erro$. Daqui, pela definição da relação $\rightsquigarrow$:

a) Não podemos ter $(\ c_i,\ s\ ) \rightsquigarrow erro$ (senão, teríamos também $(\ c_i \,\boldsymbol{;}\, c_j,\ s\ ) \rightsquigarrow erro$ pela regra **COMPOSITION - Break** e consequentemente não teriamos $safe(\ c_i \,\boldsymbol{;}\, c_j,\ s\ )$) e, portanto $safe(\ c_i,\ s\ )$;

b) Também não podemos ter $(\ c_i,\ s\ ) \rightsquigarrow s''$ e $(\ c_j,\ s''\ ) \rightsquigarrow erro$ para algum $s'' \neq erro$ (senão teríamos $(\ c_i \,\boldsymbol{;}\, c_j,\ s\ ) \rightsquigarrow erro$ pela regra **COMPOSITION** e consequentemente não teriamos $safe(\ c_i \,\boldsymbol{;}\, c_j,\ s\ )$) e,

por conseguinte, $\forall\ s'' \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow s'' \Rightarrow safe(\ c_j,\ s''\ )$.

"⇐": Supondo que $(\ c_i \,\boldsymbol{;}\, c_j,\ s\ ) \rightsquigarrow erro$. Uma vez que por suposição temos $safe(\ c_i,\ s\ )$, tem-se $(\ c_i,\ s\ ) \not\rightsquigarrow erro$ e portanto $(\ c_i \,\boldsymbol{;}\, c_j,\ s\ ) \rightsquigarrow erro$ não pode ser obtido, pela regra **COMPOSITION - Break**. Assim sendo, resta a possibilidade de aplicar a regra **COMPOSITION**. Mas isto também não é possível, pois teria que existir $s'' \in \Sigma$ tal que $(\ c_i,\ s\ ) \rightsquigarrow s''$ e $(\ c_i,\ s\ ) \rightsquigarrow erro$, o que contraria a suposição $\forall\ s'' \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow s'' \Rightarrow safe(\ c_j,\ s''\ )$.

□

2.4 Classes especiais de programas

2.4.1 Comandos $\mathcal{SA}$

O *Single Assignment* (SA) é uma condição que afirma que todas as variáveis de um programa, no nosso caso de um comando, são atribuídas uma e uma só vez. Além disso, num programa *single assignment*, não pode ser realizada uma atribuição de variáveis que já ocorreram previamente no programa (por exemplo, uma *expressão booleana*). Neste sentido, as variáveis são imutáveis. Os comandos da classe $\mathcal{SA}$ exprimem esta condição em comandos. O método de conversão de um comando para a forma SA é efectuada pela transformação $\mathcal{T}_2$ expressa na Subsecção 3.3.2.

Definimos um comando SA da seguinte forma:

Definição 2.30. *A notação $\mathcal{SA}$ representará a classe das expressões de comandos single assignment. A classe sintática $\mathcal{SA}$ é definida indutivamente do seguinte modo:*

$$
\begin{array}{lclll}
c & ::= & \textbf{skip} & & \\
 & | & \textbf{assert}\ (\ b\ ) & & \\
 & | & x\ :=\ e & & \text{, se } x \in \mathcal{V}_{Integer} \text{ e } x \notin FV\ (\ e\ ) \\
 & | & c_i\ \textbf{;}\ c_j & & \text{, se } \textbf{assign}\ (\ c_i\ ) \cap \textbf{assign}\ (\ c_j\ ) = \emptyset \\
 & & & & \text{e } FV\ (\ c_i\ ) \cap \textbf{assign}\ (\ c_j\ ) = \emptyset \\
 & | & \textbf{if}\ (\ b\ )\ \textbf{then}\ c_t\ \textbf{else}\ c_f & & \text{, se } \textbf{assign}\ (\ c_t\ ) \cap \textbf{assign}\ (\ c_f\ ) = \emptyset \\
 & & & & \text{e } FV\ (\ b\ ) \cap (\ \textbf{assign}\ (\ c_t\ ) \cup \textbf{assign}\ (\ c_f\ )\ ) = \emptyset
\end{array}
$$

Figura 2.1: Classe dos Comandos SA

A classe $\mathcal{SA}$ reflecte comandos da classe $\mathcal{C}ommand$ cujas variáveis, em particular, não possuem mais do que uma única atribuição. Os comandos **skip**, **assert** $(\ b\ )$ não têm atribuições e são por si comandos SA, e portanto pertencem à classe $\mathcal{SA}$. O comando $x\ :=\ e$ tem uma única atribuição. No entanto, para este comando ser SA, a variável atribuída x não pode ocorrer na expressão e. A composição de comandos c_i **;** c_j expressa no entanto duas condições:

1) as variáveis que são atribuidas em c_i e em c_j, para satisfazer esta propriedade, nunca podem ser iguais ($\textbf{assign}\ (\ c_i\ ) \cap \textbf{assign}\ (\ c_f\ ) = \emptyset$);

2) as variáveis atribuídas em c_j não podem ter sido usadas em c_i ($FV\ (\ c_i\ ) \cap \textbf{assign}\ (\ c_j\ ) = \emptyset$).

O comando condicional **if** $(\ b\ )$ **then** c_t **else** c_f expressa duas condições:

1) as variáveis que são atribuidas no ramo c_t e no ramo c_f nunca são iguais, *i.e.*, não pode haver atribuições às mesmas variáveis ($\textbf{assign}\ (\ c_t\ ) \cap \textbf{assign}\ (\ c_f\ ) = \emptyset$);

2) ($FV\ (\ b\ ) \cap (\ \textbf{assign}\ (\ c_t\ ) \cup \textbf{assign}\ (\ c_f\ )\ ) = \emptyset$) que indica que as atribuições às variáveis do ramo c_t e do ramo c_f nunca podem ser efectuadas a variáveis que já ocorreram na condição b (ou, dito de outro modo, as variáveis de b não podem ser afectadas pelos comandos dos ramos do condicional *if*).

Para tornarmos a exposição mais concisa, estabelecem-se algumas convenções para os exemplos. Embora a meta notação apenas permita a existência de variáveis $x_0, x_1, \ldots, x_n, \ldots$, conforme a Convenção 2.1, a título ilustrativo, iremos permitir considerar a notação a, b, c, d, $\ldots$ para as variávies e variáveis com índices $a_0, a_1, \ldots, a_n, \ldots$ referirindo as versões geradas pelo "processo SA".

Consideremos os exemplos da figura seguinte (Figuras 2.2a e 2.2b). O programa 2.2a não está na classe $\mathcal{SA}$ (por exemplo, b é atribuído duas vezes). O programa 2.2b é já um programa *single assignment*. De facto, este programa é o que resulta da transformação $\mathcal{T}_2$ aplicada ao programa 2.2a.

$$
\begin{array}{l}
a = 0; \\
b = b; \\
b = a + b; \\
c = c + 8; \\
d = a + b - f
\end{array}
$$

(a) Sequência de comandos

$$
\begin{array}{l}
a_1 = 0; \\
b_1 = b_0; \\
b_2 = a_1 + b_1; \\
c_1 = c_0 + 8; \\
d_1 = a_1 + b_3 - f_0
\end{array}
$$

(b) Resultado da transformação $\mathcal{T}_2\ (\ c\ )$

Figura 2.2: Exemplo ilustrativo

Note-se que o comando **while** $(\ b\ )$ **do** c_w não faz parte da classe $\mathcal{SA}$, pois, neste estudo, os argumentos para a transformação $\mathcal{T}_2$ serão os programas que resultam da transformação $\mathcal{T}_1$ que irá eliminar os comandos *while*, fazendo a expansão limitada do corpo do ciclo, característica do BMC.

2.4.2 Comandos $\mathcal{C}nf$

Introduzimos agora uma subclasse dos comandos *single assignment*, com uma estrutura muito própria, a que chamaremos a classe de comandos em *Conditional Normal Form* (CNF). Esta classe de comandos é a que resultará da transformação $\mathcal{T}_3$ (definida na Subsecção 3.3.3), responsável pelo 3º passo na sequência de transformações do BMC. A $\mathcal{C}nf$ é uma forma de exprimir um $c \in \mathcal{SA}$ num comando *"normalizado"* como uma sequência de *"ifes"*. Por exemplo, **if** ($\top$) **then** c_1 **else skip**; **if** ($\top$) **then** c_2 **else skip**; **if** ($\top$) **then** c_3 **else skip** será a representação do comando $c_1; c_2; c_3$.

Por comodidade, e visto que os comandos $\mathcal{C}nf$ corresponderão a uma sequência de `ifes` cujo termo falso é o **skip**, iremos simplificar a notação conforme apresentado de seguida.

Notação 2. *Para quaisquer* $b \in \mathcal{E}xpression_{Boolean}$ *e* $c \in \mathcal{C}ommand$,
if *(* b *)* ***then*** c *denotará o comando* ***if*** *(* b *)* ***then*** c ***else skip***

O método BMC faz uso de uma representação intermédia, onde todos o comandos básicos do programa são normalizados na forma $b' \rightarrow c$, representada por **if** (b') **then** c, onde c são comandos básicos ($c \in \{$ **skip**, **assert** (b) , $x := e$ $\}$). A expressão booleana b' irá representar um caminho de decisão reflectindo os condicionais presentes no programa: se o traço de execução seguiu os condicionais que geraram o caminho b', então os comandos básicos associados a estes condicionais foram executados. Tomem-se os seguintes exemplos.

```
assert ( g == h );
if(x == 0)then{
    g := h + x;
    assert ( (g − h) == x )
}else{
    if(g+h == x)then{
        assert ( (x − g − h) == 0 )
    }else{
        assert ( ⊥ )
    }
};
```

(a) programa na classe $\mathcal{SA}$

```
if (⊤) then assert ( g == h );
if (⊤ ∧ ( x == 0 )) then g := h + x;
if (⊤ ∧ ( x == 0 )) then assert ( (g − h) == x );
if (⊤ ∧ ¬( x == 0 ) ∧ ( g + h == x )) then
        assert ( (x − g − h) == 0 );
if (⊤ ∧ ¬( x == 0 ) ∧ ¬( g + h == x )) then
        assert ( ⊥ );
```

(b) programa representado na classe $\mathcal{C}nf$

O programa (2.3b) está em CNF e irá corresponder a uma representação do programa (2.3a) que não está na classe das expressões CNF. Por exemplo, a 1ª linha de (2.3b) reflecte o facto de que o comando base **assert** ($g == 4$) deverá ser sempre executado (a guarda é $\top$). Também na última linha de (2.3b) indica que a afirmação **assert** ($\perp$) só será considerada se a negação dos condição dos condicionais ($x == 0$) e ($g + h == x$) for satisfeita.

A classe de programas $\mathcal{C}nf$ é formalmente descrita na definição seguinte.

Definição 2.31. *A classe sintática* $\mathcal{C}nf$ *é definida indutivamente do seguinte modo:*

c	::=	**if** (b') **then skip**	*se* $b' \in \mathcal{E}xpression_{Boolean}$
	|	**if** (b') **then assert** (b)	*se* $b,\ b' \in \mathcal{E}xpression_{Boolean}$
	|	**if** (b') **then** $x := e$	*se* $b' \in \mathcal{E}xpression_{Boolean},\ x \in \mathcal{V}_{Integer}$ *e se* $e \in \mathcal{E}xpression_{Integer}$
	|	c_i **;** c_j	

Figura 2.3: Classe dos comandos CNF

Uma vez que a definição da relação de avaliação $\rightsquigarrow$ é feita no estilo *"big-step"*, quando é gerado um erro pelo programa, o resultado da avaliação, não apresenta qualquer tipo de informação

sobre as variáveis. Iremos definir a função **Eval** que avaliará o estado das variáveis que resultam da execução de um programa, independentemente das afirmações/asserts presentes no programa serem verdadeiras ou falsas. Este mecanismo irá permitir racicíonar sobre o estado das variáveis mesmo quando o programa não é válido, *i.e.*, quando avalia para *erro*. Assim, o resultado da função **Eval** será o estado com os valores das variáveis no final da computação de todo o programa. A função **Eval** será uma ferramenta importante no Capítulo 4.

Definição 2.32. *A função* $\textbf{\textit{Eval}} : \mathcal{C}nf \to \Sigma \to \Sigma$ *é definida recursivamente em comandos* $\mathcal{C}nf$ *por:*

$$
\begin{array}{rcl}
\textbf{\textit{Eval}}(\ \textbf{\textit{if}}\ (\ b'\ )\ \textbf{\textit{then skip}},\ s\ ) & = & s \\
\textbf{\textit{Eval}}(\ \textbf{\textit{if}}\ (\ b'\ )\ \textbf{\textit{then assert}}\ (\ b\ ),\ s\ ) & = & s \\
\textbf{\textit{Eval}}(\ \textbf{\textit{if}}\ (\ b'\ )\ \textbf{\textit{then}}\ x\ \coloneqq\ e,\ s\ ) & = & \begin{cases} s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix} & se,\ s \models b' \\ s & se,\ s \not\models b' \end{cases} \\
\textbf{\textit{Eval}}(\ c_i\ \textbf{\textit{;}}\ c_j,\ s\ ) & = & \textbf{\textit{Eval}}(\ c_j,\ \textbf{\textit{Eval}}(\ c_i,\ s\ )\ )
\end{array}
$$

Tendo em atenção que **if** (b) **then** c não é mais do que uma abreviatura para **if** (b) **then** c **else skip**, as regras na Tabela 2.5 são deriváveis (*i.e.*, podem ser obtidas conjugando regras da Tabela 2.4). No contexto dos comandos $\mathcal{C}nf$, habitualmente ignoraremos esta distinção e usaremos as regras da Tabela 2.5 como regras primitivas, como forma de tornar mais leve a exposição de ideias.

$$\frac{}{(\ \textbf{if}\ (\ b'\ )\ \textbf{then skip},\ s\ ) \rightsquigarrow s}\ \textbf{CNF - SKIP}$$

$$\frac{s \models b' \qquad s \models b}{(\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ ),\ s\ ) \rightsquigarrow s}\ \textbf{CNF - ASSERT}$$

$$\frac{s \models b' \qquad s \not\models b}{(\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ ),\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{CNF - ASSERT - Error}$$

$$\frac{s \not\models b'}{(\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ ),\ s\ ) \rightsquigarrow s}\ \textbf{CNF - ASSERT - False}$$

$$\frac{s \models b'}{(\ \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ \coloneqq\ e,\ s\ ) \rightsquigarrow s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}}\ \textbf{CNF - ASSIGN}$$

$$\frac{s \not\models b'}{(\ \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ \coloneqq\ e,\ s\ ) \rightsquigarrow s}\ \textbf{CNF - ASSIGN - False}$$

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow s'' \quad s'' \neq \mathit{erro} \quad (\ c_j,\ s''\ ) \rightsquigarrow s'}{(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow s'}\ \textbf{CNF - COMPOSITION}$$

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow \mathit{erro}}{(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow \mathit{erro}}\ \textbf{CNF - COMPOSITION - Break}$$

Tabela 2.5: Simplificação da semântica operacional para expressões CNF

A proposição que termina este capítulo estabelece que qualquer computação a partir de $(\ c,\ s\ )$ conduz a um *erro* ou conduz ao estado $\textbf{Eval}(\ c,\ s\ )$. Note-se que este "ou" é exclusivo dado o determinismo da relação $\rightsquigarrow$ (Proposição 2.19).

Proposição 2.33.

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ (\ c,\ s\ ) \rightsquigarrow erro \vee (\ c,\ s\ ) \rightsquigarrow \textbf{\textit{Eval}}(\ c,\ s\ )$$

Demonstração. Por indução em $c \in \mathcal{C}nf$.

i) $c = \textbf{if}\ (\ b'\ )\ \textbf{then skip}$.
$\textbf{Eval}(\ \textbf{if}\ (\ b'\ )\ \textbf{then skip},\ s\ ) = s$ e atendendo à semântica operacional,
teremos pela regra **CNF - SKIP**, $(\ \textbf{if}\ (\ b'\ )\ \textbf{then skip},\ s\ ) \rightsquigarrow s$.

ii) $c = \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )$.
$\textbf{Eval}(\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ ),\ s\ ) = s$.
Caso $s \models b'$:
Caso $s \models b$: pela regra **CNF - ASSERT**, $(\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ ),\ s\ ) \rightsquigarrow s$.
Caso $s \not\models b$: pela regra **CNF - ASSERT - Error**, $(\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ ),\ s\ ) \rightsquigarrow$ *erro*.
Caso $s \not\models b'$: pela regra **CNF - ASSERT - False**, $(\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ ),\ s\ ) \rightsquigarrow s$.

iii) $c = \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ \coloneqq\ e$.
Caso $s \models b'$:
$\textbf{Eval}(\ \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ \coloneqq\ e,\ s\ ) = s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}$.
Pela regra **CNF - ASSIGN**, $(\ \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ \coloneqq\ e,\ s\ ) \rightsquigarrow s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}$.
Caso $s \not\models b'$:
$\textbf{Eval}(\ \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ \coloneqq\ e,\ s\ ) = s$.
Pela regra **CNF - ASSIGN - False**, $(\ \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ \coloneqq\ e,\ s\ ) \rightsquigarrow s$.

iv) $c = c_i\ \textbf{;}\ c_j$.
Suponhamos que
HI_i: $\forall\ s \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow erro \vee (\ c_i,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c_i,\ s\ )$.
HI_j: $\forall\ s \in \Sigma,\ (\ c_j,\ s\ ) \rightsquigarrow erro \vee (\ c_j,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c_j,\ s\ )$.

Queremos mostrar que $\forall\ s \in \Sigma,\ (\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow erro \vee (\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c_i\ \textbf{;}\ c_j,\ s\ )$.

Tome-se $s \in \Sigma$.
Pela HI_i, tem-se $(\ c_i,\ s\ ) \rightsquigarrow$ *erro* ou $(\ c_i,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c_i,\ s\ )$.
1) Se $(\ c_i,\ s\ ) \rightsquigarrow$ *erro*, pela regra **CNF - COMPOSITION - Break**, $(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow$ *erro*.
2) Se $(\ c_i,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c_i,\ s\ )$, como $\textbf{Eval}(\ c_i,\ s\ ) \neq$ *erro*, sabemos pela HI_j que:
a) $(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \rightsquigarrow$ *erro*, ou
b) $(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \rightsquigarrow \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$.
Temos também que $\textbf{Eval}(\ c_i\ \textbf{;}\ c_j,\ s\ ) = \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$ por definição.
Assim, aplicando a regra **CNF - COMPOSITION**, $(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow$ *erro* ou $(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c_i\ \textbf{;}\ c_j,\ s\ )$.

□

Como nota final deste capítulo, observa-se que os resultados que se encontram previamente neste documento acerca dos comandos em geral se mantém válidos na classe $\mathcal{C}nf$. Em geral, estes resultados serão usados no Capítulo 4.

Capítulo 3

Bounded Model Checking

Este capítulo apresenta uma introdução sobre o método *Bounded Model Checking* (BMC) e o *"workflow"* típico desta técnica no domínio da sua aplicação à verificação de software, incluindo quatro transformações, que vão desde a remoção de ciclos em comandos até à construção da fórmula em lógica de primeira ordem que codifica o modelo do programa. Discutimos a sua origem, de onde surgiu, quais foram as necessidades que levaram a este tipo de investigação, em que ponto se encontra actualmente, as abordagens concorrentes, e abordamos algumas ferramentas que implementam esta técnica para verificar sistemas de *software*.

O BMC é uma técnica de verificação formal que começou por ser aplicada no domínio do *hardware*, mas que posteriormente passou também para o domínio do *software*, sendo apelidada de *Bounded Model Checking* aplicado a *software* (BMCsw). A verificação formal de *software* baseada em BMC surgiu de uma outra área de investigação. Parece-nos que este facto evidencia a importância do trabalho neste tipo de técnicas, visto que surgiu como uma necessidade proveniente de uma área que não está normalmente ligada à área de verificação formal. Essencialmente o BMCsw é um processo de dois passos. Em primeiro lugar um modelo de *software* é extraído do código fonte, e é codificado numa fórmula lógica o comportamento sequencial do sistema de transições deste modelo, sobre um conjunto finito de estados (esta limitação está na base da designação BMC). Seguidamente, no segundo passo, a fórmula lógica produzida no primeiro passo é avaliada por um processo de decisão, usualmente um SAT *solver*, por forma a obter uma interpretação lógica que refute a correcção do programa (providenciando um contra-exemplo) ou obter uma conclusão sobre a validade da fórmula, indicando a correcção do programa até ao limite (*bound*) aplicado sobre o conjunto dos caminhos de execução. Na sua forma mais simples, o BMC carece de correcção (*soundness*), uma vez que alguma das asserções do programa poderá ser violada além do limite estabelecido. No entanto, a versão do BMC implementada durante esta tese, introduz *unwinding assertions*, conforme apresentado na Subsecção 3.3.1. Deste modo pode-se argumentar que o BMC é *sound*, mas isto é apenas um "truque" conceptual, obtendo-se uma forma de *soundness* às custas de perda de completude.

Este capítulo está dividido em três secções. Na primeira aborda-se o *Model Checking* (MC) e como evoluiu para *Symbolic Model Checking* (SMC), com a utilização de *Binary Decision Diagrams* (BDDs) para verificar modelos, em particular de *hardware*. Em seguida, na segunda secção, aborda-se a técnica BMC que evoluiu separadamente para colmatar algumas técnicas de MC, concluindo com a constatação que a técnica BMC aplicada à verificação de *hardware* é uma ideia que também é aplicável à verificação do *software*. Na terceira secção apresentam-se transformações que caracterizam o método BMC para programas $\mathcal{W}hile_{\mathbf{assert}}$.

3.1 *Model Checking*

Na área da verificação formal de sistemas, um *Sistema de Transição de Estados* (STS) modela de forma efectiva as especificações ou o *design* de diversos projectos de *hardware*, em especial

especificações de harware digital [15]. As técnicas automáticas de verificação formal de STS finitos desenvolveram-se ao ponto em que as empresas de *design* de *chips* e *hardware* as têm integrado nos processos de garantia de qualidade do *hardware*. As técnicas de MC têm sido as mais adoptadas [16, 14].

As técnicas de MC permitem verificar de forma algorítmica os complexos STS [13, 9, 14, 38]. A designação MC refere-se a um conjunto de algoritmos e técnicas de verificação de propriedades em STS recorrendo à exploração do espaço de estados. Para determinar se o *design* obedece à especificação do comportamento pretendido, efectuam-se buscas nos grafos de transição associados aos modelos [13, 9, 14, 38].

Estes algoritmos conseguem executar uma verificação exaustiva do espaço de estados e conseguem-no fazer de forma automática. Por este motivo, atraíram um elevado interesse por parte da indústria de *hardware*. No entanto, o MC é afectado pelo problema da explosão do número dos estados que necessitam de ser verificados. Este problema resulta do facto do número de estados, num sistema, crescer exponencialmente com o número de componentes presentes nesse sistema. Muita investigação tem sido dedicada a atenuar este problema [15].

As propriedades provadas pelo MC são geralmente classificadas como propriedades de *safety*, declarando o que não deve nunca acontecer (ou equivalentemente, o que deve acontecer sempre) ou classificadas como propriedades de *liveness*, propriedades que declaram o que temos a certeza que deverá acontecer no futuro [5]. Os projectos de *hardware* são usualmente modelados por máquinas de estados finitas, e as suas especificações são formalizadas por propriedades em lógica temporal, que é um formalismo para se raciocinar sobre a ordenação temporal de eventos no tempo, sem introduzir explicitamente o elemento tempo. Os estados alcançáveis no sistema de transição de estados são percorridos, a fim de verificar as propriedades modeladas na especificação. Quando uma propriedade não é verificada, é gerado de forma automática um contra-exemplo, na forma de uma sequência de estados, que representa o *"raciocínio"* que o método constroi para encontrar a falha do modelo, relativamente à específicação [5].

O termo MC foi adotado por Clarke e Emerson [16] nos anos oitenta. Nos primórdios, os algoritmos de MC enumeravam de forma explícita os estados alcançáveis pelo sistema, a fim de verificar a correcção de uma especificação. A capacidade dos programas que utilizavam técnicas de MC para verificar projectos ficava assim restringida a uns poucos milhões de estados, visto que o número de estados cresce exponencialmente com o número de variáveis. As implementações iniciais não eram capazes de lidar com projetos de complexidade à escala industrial [5]. Mesmo os mais avançados algoritmos de MC explícito são incapazes de verificar todas as propriedades desejadas no sistema em tempo útil, devido ao problema da explosão dos estados. Contudo, não é possível saber se a especificação de um sistema é correcta ou mesmo completa, não é possível saber se o que se especifica traduz efectivamente o significado que se pretende dar ao sistema. Conclui-se assim que não existe o chamado "sistema correcto", sendo apenas possível verificar se o sistema satisfaz a especificação dada.

O MC é frequentemente utilizado para encontrar uma falsificação na especificação do sistema, procurando erros lógicos, em vez de provar que eles não existem. As ferramentas MC são muitas vezes capazes de encontrar erros que dificilmente seriam encontrados pela simulação. Os simuladores apenas consideram um número restrito de estados enquanto as ferramentas de MC consideram de forma exaustiva todos os possíveis comportamentos do sistema. Por outro lado, o recurso à lógica temporal também clarifica algumas ambiguidades que podem ter sido criadas inadvertidamente pela especificação.

As ferramentas de MC não são consideradas como alternativas, mas antes como um complemento essencial aos métodos tradicionais de teste e simulação de sistemas. O facto de se tratar de um método automático, não dependendo de interações complexas com o utilizador para a construção de provas; a geração dos contra exemplos gerados pelo caminho de estados percorrido pelo método; o facto de se focar em sistemas finitos devido ao recurso a modelos de STS; e o recurso a variantes da lógica temporal para a especificação das propriedades da especificação, caracterizam o conceito de MC e explicam a razão da sua vasta utilização. Em resumo, o MC é uma técnica algorítmica para a verificação de propriedades temporais em sistemas de estados finitos.

SMC com recurso a BDDs

Conforme dito anteriormente, as primeiras implementações das técnicas MC no início da década de 80 recorriam à representação explícita de grafos de transição dos STS que procuravam explorar de forma exaustiva o universo de estados, com recurso a técnicas de travessia de grafos, tão eficientes quanto possível. Por este motivo, estas técnicas tornam-se inviáveis e inadequadas para a maioria das aplicações industriais, pois limitavam-se projetos de *hardware* a circuitos com cerca de 20 componentes. O universo de estados tornava-se demasiado vasto para ser explorado, em tempo útil.

Técnicas que utilizam a exploração de representações simbólicas do universo de estados de um sistema de transição, surgiram por volta de 1990 [8, 20, 37]. O primeiro avanço no sentido da utilização mais ampla destas técnicas de verificação por parte da indústria foi realizado aquando da introdução do *Symbolic Model Checking* (SMC) [8, 19]. A técnica SMC, representa e manipula conjuntos integrais de estados implicitamente, através de funções booleanas.

Os BDDs são mais eficientes a lidar com funções booleanas, que representam conjuntos de estados, em vez de manter e manipular uma lista extensiva e explícita de estados, como é realizado no MC explícito. Em particular [7], a representação em grafo de funções booleanas permite uma manipulação eficiente de fórmulas booleanas através dos conhecidos *Reduced Ordered Binary Decision Diagrams* (RoBDD). O SMC realiza através de BDDs, a travessia do grafo de transição de estados utilizando um algoritmo de pesquisa em largura (`breadth first search`) [7]. Os BDDs mantêm as propriedades das funções características dos conjuntos de estados, e permitem a computação das transições entre conjuntos de vários estados, em vez de computar cada estado individualmente.

Desde a sua introdução na década de 90, os BDDs têm sido adotados pela indústria nos processos de garantia de várias empresas produtoras de *hardware*. A principal desvantagem do método BDDs reside, tal como o método explícito, no crescimento exponencial dos estados a serem verificados, com o número de componentes presente no sistema. Infelizmente, é um método cuja quantidade de memória disponível restringe o tamanho dos circuitos que podem ser verificados de forma eficiente. Apesar disto, sistemas finitos de estados, tais como os circuitos em série e alguns protocolos de comunicação, têm sido verificados formalmente com sucesso na última década através de técnicas de SMC com recurso a BDDs.

De acordo com E. M. Clarke [8], a combinação de SMC com BDDs [34, 19] permite verificar eficientemente sistemas com mais de 10^{20} estados. Pela primeira vez, um número significativo de sistemas de *hardware* no mercado, puderam ser verificados eficientemente por estes métodos, o que levou à sua crescente adopção. Empresas como a Intel® e a IBM® desenvolveram as suas próprias ferramentas de MC, inicialmente com projectos experimentais, e posteriormente integrando-as nos seus processos de garantia de qualidade de *hardware*.

3.2 *Bounded Model Checking*

O BMC é uma das técnicas de verificação mais utilizadas na indústria dos semicondutores, para verificar circuitos lógicos. Em vez de recorrer a técnicas de manipulação de BDDs, como é o caso do SMC, o BMC recorre a ferramentas de decisão como SAT *solvers* e SMT *solvers*.

A combinação de técnicas de MC, com ferramentas de decisão (SAT-solvers) permite aos métodos conhecidos por BMC, efectuar uma exploração do espaço de estados de forma muito rápida [15, 5]. Para algumas classes de problemas, verifica-se uma performance superior relativamente a técnicas anteriores ao BMC.

O método foi introduzido pela primeira vez em 1999 por Biere et al. [4], como uma técnica complementar à aplicação de SMC com recurso a BDDs. O método deve o seu nome (**Bounded**) ao facto de verificar o modelo apenas até uma determinada profundidade. Tal como no MC explícito, o projecto de *hardware* é modelado por um sistema de transição de estados. O sistema de transição de estados é depois *"desenrolado"* em k iterações e conjugado com uma propriedade de forma a construir uma fórmula proposicional para procurar um contra-exemplo à especificação, dentro das

computações do sistema até um limite máximo de $k \in \mathbb{N}$ iterações. O principal conceito associado ao BMC é a consideração de um traço de execução finito que pode ser uma testemunha/contra-exemplo na verificação de uma propriedade de "*safety*". O método limita o tamanho do traço de execução, procurando aumentar progressivamente o k até ser encontrado um contra-exemplo à especificação, até a verificação exceder em tempo útil a sua decisão, ou, no caso de *software*, ser atingido o *Completeness Threshold* do sistema. Em *software*, chamamos *Completeness Threshold* à profundidade de exploração do modelo, além da qual as propriedades permanencem válidas (se forem validadas até essa profundidade) [33]. Contudo, aproximações ao *Completeness Threshold* são, na prática, suficientes uma vez que encontrar este ponto concretamente é tão difícil quanto realizar o próprio MC. Uma forma de obter esta aproximação, em termos de verficação de *software*, resume-se a determinar o *high-level Worst-Case Execution Time* (WCET) [23, 6].

O WCET é o limite ao número máximo de iterações de ciclo. Existem ferramentas que calculam o WCET através de análise sintática às estruturas cíclicas. Caso o resultado destas ferramentas não seja adequado, é possível aplicar um processo iterativo onde o limite é constantemente aumentado [12, 11]. O método de encontrar WCET apenas é aplicável a programas que possuem um limite de *run-time*, que se verifica em aplicações embebidas, onde há uma maior necessidade de verificação do *software*.

A fórmula gerada é posteriormente avaliada por meio de um **procedimento de decisão** (usualmente SAT). Os SAT *solvers* são aplicados neste método porque é possível reduzir, de forma eficiente, a um *Propositional Satisfiability Problem* (PSP) o problema da explosão de estados do método BMC, e, deste modo, solucionar com ferramentas de decisão como SAT *solvers* e mesmo SMT *solvers*, pois os procedimentos PSP não sofrem do problema da explosão de estados evidenciado pelos métodos baseados em BDDs. Os SAT *solvers* modernos, lidam fácilmente com PSP de elevada complexidade. Se a fórmula for satisfeita, então existe uma sequência de estados que refuta a propriedade. Se não existir valoração que satisfaça a fórmula gerada, a técnica é inconclusiva, uma vez que poderão existir sequências (de estados) com mais de k estados que refutem a propriedade, sabendo-se apenas que até k iterações não existe erro na avaliação do programa.

Apesar da incompletude inerente ao método BMC, este método tem-se demonstrado útil, uma vez que diversas inconsistências têm sido identificadas, que de outro modo passariam despercebidas.Por permitir encontrar muitos erros lógicos em sistemas com alguma complexidade, que não podem ser tratados por meio de técnicas concorrentes, o BMC tem sido acolhido pela indústria, como uma técnica complementar às técnicas de MC e SMC baseadas em BDDs. Diversas publicações que efectuam esta comparação têm apoiado este ponto de vista.

A motivação original que levou ao desenvolvimento da técnica de BMC, prendeu-se com uma tentativa de aproveitar o recente sucesso da satisfazibilidade SAT de resolver fórmulas booleanas para realizar MC. Durante os últimos anos tem-se observado um enorme aumento no poder de raciocínio e decisão de SAT *solvers*. Dependendo da extensão da fórmula lógica a ser validada, os SAT *solvers* modernos conseguem lidar com centenas de milhar de variáveis, e milhões de cláusulas. O SMC com BDDs, por outro lado, pode apenas verificar sistemas com não mais do que algumas centenas de ciclos.

O BMC é muito atraente para o uso industrial, devido à sua robustez, a sua superior capacidade de verificação e o facto de ser um método automático de verificação. Ferramentas de decisão como GRASP [40], SATO [47], e o algoritmo Stålmarck [41], que raramente necessitam de espaço exponencial, têm contribuido para este sucesso. As desvantagens, no entanto, devem-se ao facto do método carecer de "integridade" e os tipos de propriedades que podem ser verificados actualmente são muito limitadas. Alguns destes inconvenientes têm sido abordados em trabalhos mais recentes, como visto em [39, 43, 2], onde foram alcançados resultados encorajadores.

Com esta proposta, o problema de complexidade do MC não é solucionado pelo BMC, por este também se basear num procedimento exponencial, o que limita a sua capacidade. Contudo, a aplicação empírica do método tem mostrado que o BMC resolve muitos casos que não podem ser resolvidos por técnicas baseadas em BDDs, e o inverso também se verifica. Há problemas que são solucionados de forma mais eficiente por técnicas baseadas em BDDs. Outra desvantagem reflete-se no facto do BMC ser incapaz de provar a ausência de erros em certas circunstâncias.

Por este motivo as técnicas BMC não substituem nenhuma ferramenta de verificação automática, juntando-se ao arsenal de ferramentas utilizadas pela indústria.

O BMC visa resolver os mesmos problemas que as técnicas de SMC tradicional baseado em BDDs. O BMC possui duas características: 1) Se o limite não for suficientemente alto, o método pode não funcionar satisfatoriamente uma vez que é incompleto. 2) utiliza técnicas PSP em vez de BDDs. Empiricamente tem-se verificado que se o limite for suficientemente pequeno (dependendo do modelo e do SAT *solver*), o método BMC supera as técnicas baseadas em BDDs. Resultados empíricos também mostraram que há pouca correlação entre os problemas que são difíceis (`hard`) para PSP e os problemas que são difíceis (`hard`) para as técnicas baseadas em BDDs. Verifica-se que problemas da classe `hard` para BDDs, muitas vezes são decidíveis por SAT.

Acreditamos que caso se configure os SAT *solvers* para tirar partido da estrutura original das fórmulas resultantes do process BMC, o método melhora significativamente [39].

Uma pesquisa publicada pela Intel® [18] mostra que o BMC tem vantagens tanto na capacidade de verificação como na produtividade dos resultados relativamente a ferramentas SMC baseadas em BDDs, quando aplicada à modelação de sistemas dos conhecidos Pentium™. As vantagens de produtividade resultam de que, normalmente, as técnicas baseadas em BDDs, para optimizar o seu desempenho, requerem algum nível de orientação manual.

De acordo com [30], uma abordagem semelhante foi tomada para resolver o clássico *planning problem* na área da Inteligência Artificial [30]. Neste problema procura-se encontrar uma sequêcia de passos a fim de realizar uma tarefa ("por exemplo, posicionar cubos por ordem decrescente de tamanho"). Tal como no BMC, encontrar um plano é equivalente a encontrar caminhos até um determinado limite máximo de passos.

BMC aplicado na verificação de *software*

Os sistemas de computação consistem em complexos aglomerados de *hardware* e *software*. Assegurar a correcção do *software* tem-se revelado um desafio mais complexo do que garantir a correcção do *hardware* que é aplicado no sistema de computação [23]. Existem numerosas ferramentas para procurar falhas funcionais nas arquitecturas dos *waffers* de silicone, no entanto poucas são as ferramentas que asseguram qualidade no *software*. O custo de garantir a ausência de erros no *software* é tão elevada que as ferramentas desenvolvidas raramente garantem completamente a correcção do *software*.

Um modelo de *software* consiste em estados de execução e transições entre estados, e a sua verificação procura determinar se o modelo do sistema satisfaz a especificação.

Os algoritmos de MC para *software* examinam de forma exaustiva todos os estados alcançáveis pelo modelo de *software*. Com a evolução do crescente poder de computação, o tamanho e a complexidade dos programas de *software* têm crescido ao ponto de se tornar difícil assegurar a correcção e a qualidade dos produtos de *software*. Especialmente nas áreas da comunicação e transportes, a correcção do *software* toma uma importância agravada. É comum encontrar-se erros de *software* que resultam em enormes perdas financeiras e até mesmo perdas de vidas.

A ideia subjacente ao BMC, técnica aplicada à verificação de *hardware*, pode também ser aplicada a sistemas de *software*. Segundo Biere [6] a forma mais linear de realizar esta transição resume-se em considerar todo o programa como uma relação de transição.

O sistema de *software* pode ser visto como um STS, consistindo de uma relação de transição, um conjunto de estados iniciais e um conjunto de estados que codificam a localização de comandos e variáveis do programa bem como os seus valores ao longo da sua execução. No entanto, a abordagem BMCsw dispensa a consideração explícita de um STS, efectuando, em vez disso, transformações no proprio programa. Destas transformações é obtida uma fórmula lógica, cuja validade é então estudada com recurso a alguma ferramenta de decisão.

Ferramentas de Verificação de Software com BMC

Existem algumas ferramentas de verificação de *software* baseadas em BMC. A primeira implementação data de 2000 onde Currie et al. [21] implementaram uma procura simbólica em

profundidade e limitada.

Outra das implementações iniciais do método BMC é apresentado pela ferramenta CBMC desenvolvida na CMU. Procura simular uma vasta gama de arquitecturas para testar os programas de *software* a serem verificados. Providencia suporte para memórias *little* e *big-Endian* bem como suporte a header files para Linux, Windows® e OS X®. A ferramenta além de implementar técnicas de desenrolar ciclos, também implementa `bit-flattening`, ou `bit-blasting` para decidir fórmulas de `bit-vector`. As fórmulas geradas por esta ferramenta são traduzíveis para muitos dos standards conhecidos, sendo a única ferramenta que suporta SystemC, SpecV e C++. A IBM® desenvolveu uma versão do CBMC para verificar programas concorrentes.

O F-Soft [29], desenvolvido pela NEC Research, é a única ferramenta que desenrola por completo todo o STS subjacente ao programa. Esta ferramenta fornece um SAT *solver* próprio que foi optimizado aos problemas de decisão gerados pelo método BMC.

A ferramenta Saturn também é digna de se notar pela sua escalabilidade [45]. Foi aplicada ao *kernel* do Linux para detectar *NULL-pointer dereferences* e convenções de *locks* na *Interface de Programação de Aplicativos* (API), demonstrando assim que a técnica aplicada na ferramenta Saturn é escalável o suficiente para analisar todo o *kernel* do sistema operativo Linux.

A ferramenta EXE [46] aplica simulação simbólica para detectar erros no código de sistemas de ficheiros, usando um modelo de memória de baixo nível.

Têm surgido várias versões alternativas destas ferramentas, como a conhecida versão do CBMC de Armando et al. [31] que gera um problema de decisão para SMT em aritmética de inteiros linear. A aplicação de constraint solvers nos problemas de decisão gerados por BMC também é estudado por [17].

Em resumo, o BMC é uma técnica excepcional para detectar *erros de software*. Outras técnicas serão mais apropriadas para *provar propriedades* sobre *software*.

3.3 Transformação BMC de programas $\mathcal{W}hile_{\textbf{assert}}$

O BMC de programas $\mathcal{W}hile_{\textbf{assert}}$ assenta em 4 (quatro) transformações que vão transformando um comando da linguagem $\mathcal{W}hile_{\textbf{assert}}$ sucessivamente, até chegar a uma fórmula lógica que codifica o programa:

$\mathcal{T}_1$ Fase de desenrolamento de ciclos até um limite k (*bound*);

$\mathcal{T}_2$ Fase de conversão a *Single Assignment* (SA);

$\mathcal{T}_3$ Fase de normalização para *Conditional Normal Form* (CNF);

$\mathcal{T}_4$ Fase de construção da fórmula lógica a ser validada por um processo de decisão.

No nosso estudo o processo BMCsw é a composição sequencial das transformações $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ e $\mathcal{T}_4$, como esquematicamente apresentado pela Figura 3.1.

$$\mathcal{C}ommand \xrightarrow{\mathcal{T}_1} \mathcal{C}ommand_{\text{sem ciclos}} \xrightarrow{\mathcal{T}_2} \mathcal{SA} \xrightarrow{\mathcal{T}_3} \mathcal{C}nf \xrightarrow{\mathcal{T}_4} \mathcal{E}xpression_{Boolean}$$

Figura 3.1: Transformações BMC

O capítulo 5 descreve o *bounded model checker* que foi construído durante esta tese, que designamos por `gbmc` e que implementa a sequência de transformações acima. Os exemplos apresentados nesta secção para ilustrar as várias transformações são retirados directamente dos relatórios da ferramenta `gbmc` .

3.3.1 Transformação $\mathcal{T}_1$

A transformação $\mathcal{T}_1$ transforma um qualquer comando da linguagem $\mathcal{W}hile_{\textbf{assert}}$, que pode conter ciclos, num comando, possivelmente não equivalente, onde os ciclos são desenrolados até um dado limite k. Esta transformação reduz, portanto, o espaço de comandos a ser analisado, substituindo todas as ocorrências do comando **while** (b) **do** c_w por combinações de outras construções presentes na classe $\mathcal{C}ommand$. Grosso modo, esta transformação substitui recursivamente o comando **while** (b) **do** c_w, pelo comando **if** (b) **then** c_w **; while** (b) **do** c_w **else skip**, k vezes. A última ocorrência do comando é simplesmente eliminada, razão pela qual esta transformação produz um programa que não é necessariamente equivalente, mas antes uma aproximação do original. Um programa sem ciclos não sofre qualquer tipo de alteração, como é o caso do programa expresso na Figura 3.2.

```
if(x > 3)then{
    assert(y > 0)
}else{
    x = 4
}
```

Figura 3.2: Instância de um programa sem ciclos

Considerando um programa com ciclos *while* aninhados, esta é a transformação que melhor evidencia o problema da explosão dos estados para este tipo de técnicas. Apresenta-se seguidamente a definição da transformação $\mathcal{T}_1$.

Definição 3.1. *Com $k \in \mathbb{N}$ e $c \in \mathcal{C}ommand$, $\mathcal{T}_1(c, k) := \mathcal{T}_1(k, c, k)$ onde a transformação $\mathcal{T}_1(i, c, k)$ (com $i, k \in \mathbb{N}_0$, $c \in \mathcal{C}ommand$) é definida recursivamente do seguinte modo:*

$$
\begin{array}{rcl}
\mathcal{T}_1(i, \textbf{\textit{skip}}, k) & := & \textbf{\textit{skip}} \\
\mathcal{T}_1(i, x := e, k) & := & x := e \\
\mathcal{T}_1(i, \textbf{\textit{assert}}(b), k) & := & \textbf{\textit{assert}}(b) \\
\mathcal{T}_1(i, c_i \,\textbf{;}\, c_j, k) & := & \mathcal{T}_1(i, c_i, k) \,\textbf{;}\, \mathcal{T}_1(i, c_j, k) \\
\mathcal{T}_1(i, \textbf{\textit{if}}(b)\, \textbf{\textit{then}}\, c_t \,\textbf{\textit{else}}\, c_f, k) & := & \textbf{\textit{if}}(b)\, \textbf{\textit{then}}\, \mathcal{T}_1(i, c_t, k) \,\textbf{\textit{else}}\, \mathcal{T}_1(i, c_f, k) \\
\mathcal{T}_1(0, \textbf{\textit{while}}(b)\, \textbf{\textit{do}}\, c_w, k) & := & \textbf{\textit{assert}}(\neg b) \\
\mathcal{T}_1(i, \textbf{\textit{while}}(b)\, \textbf{\textit{do}}\, c_w, k) & := & \textbf{\textit{if}}(b)\, \textbf{\textit{then}} \\
 & & \quad \mathcal{T}_1(k, c_w, k) \,\textbf{;} \\
 & & \quad \mathcal{T}_1(i-1, \textbf{\textit{while}}(b)\, \textbf{\textit{do}}\, c_w, k) \\
 & & \textbf{\textit{else}} \\
 & & \quad \textbf{\textit{skip}} \qquad\qquad (i > 0)
\end{array}
$$

Esta definição expande todos os ciclos (internos e externos) $k+1$ vezes/iterações. Note-se também o recurso a acumuladores, a "recursão externa" sobre os comandos e a "recursão interna" sobre os inteiros, presentes nesta definição. Note-se o caso de paragem, onde se introduz o comando **assert** $(\neg b)$ (a chamada *unwinding assertion*), para indicar a ocorrência de um erro. Neste trabalho, não se destingue entre a geração de um erro por parte das afirmações/asserts escritos pelo programador e a geração de um erro devido ao limite de verificação, relativo à *unwinding assertion*.

A relação entre a correcção de um programa e a correcção do programa que resulta da sua transformação por $\mathcal{T}_1$ pode ser resumida do seguinte modo:

i) Se $\mathcal{T}_1(c, k)$ é correcto, então c é correcto;

ii) Se $\mathcal{T}_1(c, k)$ é incorrecto e não é violada uma "*unwinding assertion*", então a asserção que dita a incorrecção é também violada por c;

iii) Se $\mathcal{T}_1(c, k)$ é incorrecto, sendo violada uma "*unwinding assertion*", então não ocorre nenhuma violação de asserção nas primeiras $k-1$ iterações dos ciclos de c, mas nada pode ser dito sobre a correcção de c em geral;

iv) Se c é correcto, então $\mathcal{T}_1(c, k)$ não é necessariamente correcto, podendo ser violadas "*unwinding assertion*" e apenas estas.

Consideremos o seguinte exemplo para ilustrar esta transformação (o mesmo exemplo será utilizado para as restantes transformações).

$x_0 := 2$;
while$(y_0 < x_0)$ do{
    $y_0 := y_0 + 1$
} ;
assert$(y_0 == x_0 \vee y_0 > 2)$

Figura 3.3: Exemplo base

Numa execução do programa da Figura 3.3, sabemos que a variável y_0 irá ser incrementada enquanto satisfazer a condição booleana $y_0 < x_0$. Uma qualquer execução deste programa irá satisfazer a asserção final. O resultado de aplicar a transfomação $\mathcal{T}_1$ ao programa da Figura 3.3 com limite $k = 2$ é expresso na Figura 3.4.

```
x₀ := 2 ;
if(y₀ < x₀){
        y₀ := y₀ + 1 ;
        if(y₀ < x₀){
                y₀ := y₀ + 1 ;
                assert(¬(y₀ < x₀))
        }else{
                skip
        }
}else{
        skip
} ;
assert(y₀ == x₀ ∨ y₀ > 2)
```

Figura 3.4: Exemplo base sem ciclos

Note-se que no programa da Figura 3.4, a asserção final e a *unwinding assertion* serão satisfeitas apenas quando $y_0 \geq 0$.

3.3.2 Transformação $\mathcal{T}_2$

A transformação $\mathcal{T}_2$ transforma um qualquer comando sem ciclos num comando SA, ou seja um comando onde cada variável só pode ser atribuída uma vez e apenas no caso de não ter sido ainda "usada" (ver Subsecção 2.4.1). A ideia essencial da transformação $\mathcal{T}_2$ é a de ao encontrar uma atribuição a uma variável x num programa, gerar uma "nova versão da variável" x (que corresponderá efectivamente a uma nova variável) e actualizar as ocorrências da variável x nos comandos que se seguem a esta atribuição com o novo nome. No caso de comandos **if** (b) **then** c_t **else** c_f, a aplicação desta ideia exige adicionalmente algum mecanismo de sincronização das versões das variáveis geradas em cada um dos ramos. Esta transfomação cria novas versões das mesmas variáveis, pelo que, normaliza o comando com atribuições que são únicas.

Neste projecto, para tratar as expressões condicionais if then else, optamos por aplicar *Dynamic Single Assignment* (DSA), em vez de *Static Single Assignment* (SSA). A diferença entre estas duas opções, reside na forma como se unificam as variáveis após a ocorrência de um condicional, uma vez que em DSA se admite que a mesma variável ocorra mais do que uma vez do lado esquerdo de atribuições, desde que seja em caminhos de execução diferentes. O SSA unifica as variáveis exactamente a seguir à ocorrência do condicional. Assim é adicionada uma sequência de versões dos comandos adicional entre o if e o comando seguinte, como simplificadamente esquematizada pela Figura 3.5. Esta sequência de sincronização renomeia todas as variáveis alteradas para os valores calculados em ambos os ramos, dependendo da condição inicial.

$$\mathcal{T}_2\,(\,(\,\textbf{if}\,(\,b\,)\,\textbf{then}\,c_t\,\textbf{else}\,c_f\,)\,;\,\textit{comando}\,)$$
$$\downarrow$$
$$(\,\textbf{if}\,(\,b\,)\,\textbf{then}\,\mathcal{T}_2\,(\,c_t\,)\,\textbf{else}\,\mathcal{T}_2\,(\,c_f\,)\,)\,;\,\textit{sequência de sincronização};\,\mathcal{T}_2\,(\,\textit{comando}\,)$$

Figura 3.5: Static Single Assignment

O DSA unifica as variáveis dentro de cada ramo, gerando uma nova versão da variável que é partilhada por comandos dos ramos do condicional. Esquematicamente:

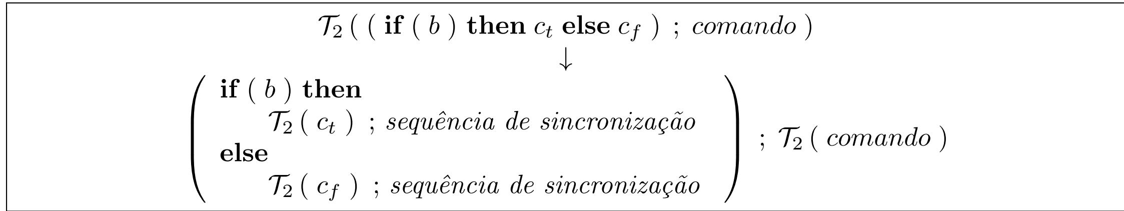


Figura 3.6: Dynamic Single Assignment

Note-se que, na opção DSA uma variável atribuída pode ter uma outra atribuição no programa, mas esta atribuição terá que ser no contexto de uma condicional if then else e terá que ser no outro ramo desse condicional. Como numa execução apenas um dos ramos de cada conficional if then else pode ser executado, em *"runtime"* a mesma variável não é atribuída mais do que uma vez.

O resultado de aplicar a transfomação $\mathcal{T}_2$ (descrita adiante) ao programa da Figura 3.4 é expresso na Figura 3.7.

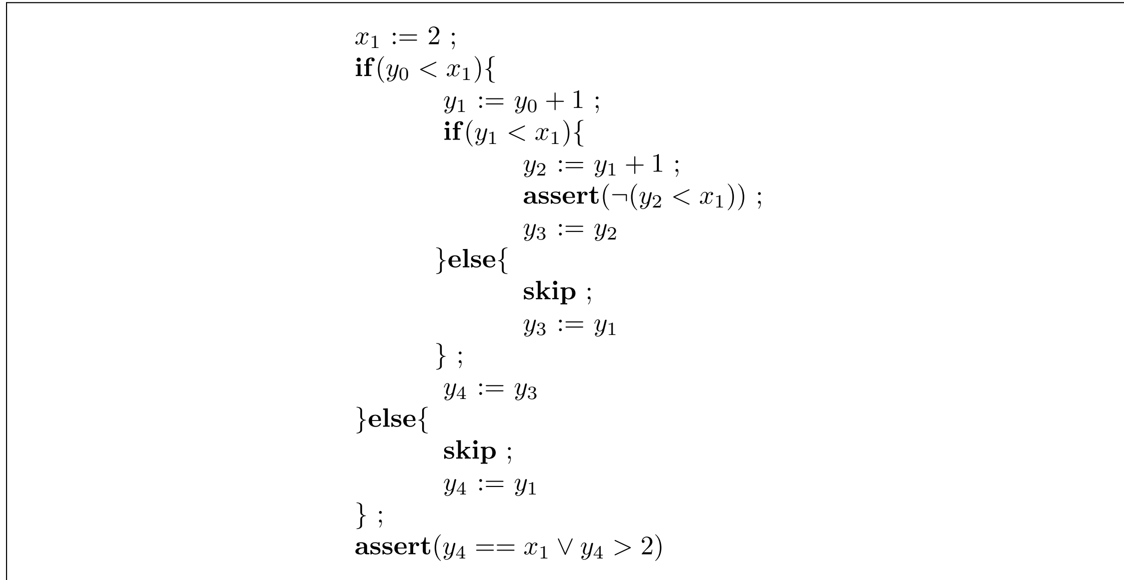


Figura 3.7: Forma Dynamic Single Assignment do programa na Figura 3.4

Observe-se o modo como é feita a sincronização neste caso: x_1 é a única nova versão de x_0 necessária (há apenas uma única atribuição a x_0 no programa inicial); y_1, y_2, y_3 e y_4 são novas versões da variável y_0; a variável y_3 assegura a sincronização no *if* interior e a variável y_4 assegura a sincronização no *if* exterior, sendo y_4 a versão mais actual de y_0. Por definição, a variável x_0 deveria também ser actualizada em cada *if*, contudo, visto que a variável x_0 não é atribuida nos ramos *if*, optamos por optimizar a apresentação de resultados.

Definição 3.2. *Para qualquer comando c, que já não tenha estruturas* ***while*** (b) ***do*** c_w*, a transformação* $\mathcal{T}_2$ *é definida pela aplicação da função* $\mathcal{T}_2'$ *(definida abaixo na Definição 3.3) a todas as variáveis do programa. Mais em concreto, e assumido que l é uma lista que contém todas as variáveis do programa c,* $\mathcal{T}_2$ *tem a seguinte definição:*

$$\mathcal{T}_2 (c) \coloneqq \mathcal{T}_2 (c,\ l)$$

$$\mathcal{T}_2 (c,\ []) \coloneqq c$$

$$\mathcal{T}_2 (c,\ v :: l') \coloneqq \mathcal{T}_2 (c',\ l') \ \textit{onde} \ (_,\ c') \coloneqq \mathcal{T}_2' (v,\ c)$$

A função $\mathcal{T}_2'(v, c)$ produz um par onde a primeira componente é a última versão criada da variável v que se pretende normalizar no comando c. A segunda componente do par é o comando c em SA. Quando ocorrem atribuições à variável v, são criadadas novas versões da variável v. Quando são criadas novas versões, as ocorrências posteriores da variável v são actualizadas e finalmente os ramos dos condicionais são sincronizados, recorrendo ao DSA. Apresenta-se seguidamente a definição deste conceito.

Abaixo, usaremos a notação $x \in [v]$ para indicar que x é versão de v e $x \notin [v]$ para indicar que x não é versão de v.

Definição 3.3. *Para qualquer comando c, que já não tenha estruturas* ***while*** (b) ***do*** c_w*, a função* $\mathcal{T}_2'$ *é definida recursivamente do seguinte modo:*

$$\mathcal{T}_2'(v,\ \textbf{\textit{skip}}) := (v,\ \textbf{\textit{skip}})$$

$$\mathcal{T}_2'(v,\ \textbf{\textit{assert}}\ (b)) := (v,\ \textbf{\textit{assert}}\ (\textbf{\textit{subs}}(b,\ v)))$$

$$\mathcal{T}_2'(v,\ x := e) := (v,\ x := \textbf{\textit{subs}}(e,\ v)) \quad \text{, se } x \notin [v]$$
$$\mathcal{T}_2'(v,\ x := e) := (v',\ v' := \textbf{\textit{subs}}(e,\ v)) \quad \text{, se } x \in [v]\ (v' \text{ é nova versão de } v)$$

$$\mathcal{T}_2'(v,\ c_i\ ;\ c_j) := (v_j,\ c_i'\ ;\ c_j') \quad \text{, onde } (v_i,\ c_i') := \mathcal{T}_2'(v,\ c_i) \text{ onde } (v_j,\ c_j') := \mathcal{T}_2'(v_i,\ c_j)$$

$$\mathcal{T}_2'(v,\ \textbf{\textit{if}}\ (b)\ \textbf{\textit{then}}\ c_t\ \textbf{\textit{else}}\ c_f) := \left(v',\ \begin{array}{l} \textbf{\textit{if}}\ (\textbf{\textit{subs}}(b,\ v))\ \textbf{\textit{then}} \\ \quad c_t'\ ;\ v' := v_t \\ \textbf{\textit{else}} \\ \quad c_f'\ ;\ v' := v_f \end{array} \right) \quad \begin{array}{l} \text{, onde } (v_t,\ c_t') := \mathcal{T}_2'(v,\ c_t) \\ \text{onde } \left(v_f,\ c_f'\right) := \mathcal{T}_2'(v,\ c_f) \\ (v' \text{ é nova versão de } v) \end{array}$$

A função recursiva auxiliar **subs** (e, v) substitui em e todas as ocorrências das versões de v por v.

Definição 3.4. *Para qualquer expressão e, e* $v \in \mathcal{V}_{Integer}$*,* $\textbf{\textit{subs}}(e,\ v)$ *é definido recursivamente do seguinte modo:*

$$\begin{array}{rcll} \textbf{\textit{subs}}(e_1 \Box e_2,\ v) & := & \textbf{\textit{subs}}(e_1,\ v) \Box \textbf{\textit{subs}}(e_2,\ v) & \\ \textbf{\textit{subs}}(n,\ v) & := & n & \\ \textbf{\textit{subs}}(x,\ v) & := & x & \text{, onde } x \notin [v] \\ \textbf{\textit{subs}}(x,\ v) & := & v & \text{, onde } x \in [v] \end{array}$$

3.3.3 Transformação $\mathcal{T}_3$

A transformação $\mathcal{T}_3$ transforma um comando SA, num comando em CNF (ver Subsecção 2.4.2). Em particular, esta transfomação elimina os condicionais aninhados, assim como os condicionais com dois ramos, e usa apenas os condicionais especiais da forma **if** (b) **then** c, onde o ramo `else` corresponde ao comando **skip**. O programa é assim transformado numa sequência de condicionais *"single-branch"* não contendo outros condicionais. A transformação $\mathcal{T}_3$ tem como parâmetro uma fórmula lógica que representa o "caminho" necessário à execução de um dado "comando atómico" (**skip**, $x := e$ ou **assert** (b)). Este parâmetro booleano é "actualizado" de cada vez que se encontra um comando **if** (b) **then** c_t **else** c_f. Na composição das 4 (quatro) transformações que constituem o *Bounded Model Checking* aplicado a *software* (BMCsw), esta transformação é invocada com o parâmetro "$\top$", como forma de representar o "início do caminho".

Definição 3.5. *Com* $c \in \mathcal{SA}$*,* $\mathcal{T}_3(c) := \mathcal{T}_3(c,\ \top)$*, sendo a transformação* $\mathcal{T}_3(c,\ b)$ *(para*

$c \in \mathcal{SA}$ *e* $b \in \mathcal{E}xpression_{Boolean}$*) definida recursivamente do seguinte modo:*

$$
\begin{array}{rcl}
\mathcal{T}_3\,(\,\textbf{\textit{skip}},\ b'\,) & := & \textbf{\textit{if}}\,(\,b'\,)\ \textbf{\textit{then skip}} \\
\mathcal{T}_3\,(\,\textbf{\textit{assert}}\,(\,b\,)\,,\ b'\,) & := & \textbf{\textit{if}}\,(\,b'\,)\ \textbf{\textit{then assert}}\,(\,b\,) \\
\mathcal{T}_3\,(\,x\ :=\ e,\ b'\,) & := & \textbf{\textit{if}}\,(\,b'\,)\ \textbf{\textit{then}}\ x\ :=\ e \\
\mathcal{T}_3\,(\,c_i\ \textbf{;}\ c_j,\ b'\,) & := & \mathcal{T}_3\,(\,c_i,\ b'\,)\ \textbf{;}\ \mathcal{T}_3\,(\,c_j,\ b'\,) \\
\mathcal{T}_3\,(\,\textbf{\textit{if}}\,(\,b\,)\ \textbf{\textit{then}}\ c_t\ \textbf{\textit{else}}\ c_f,\ b'\,) & := & \mathcal{T}_3\,(\,c_t,\ b' \wedge\ b\,)\ \textbf{;}\ \mathcal{T}_3\,(\,c_f,\ b' \wedge \neg\ b\,)
\end{array}
$$

O resultado de aplicar a transfomação $\mathcal{T}_3$ ao programa da Figura 3.7 encontra-se na Figura 3.8.

$$
\begin{array}{l}
\textbf{if}(\top)\{\ x_1 := 2\ \}\ ; \\
\textbf{if}(\top \wedge y_0 < x_1)\{\ y_1 := y_0 + 1\ \}\ ; \\
\textbf{if}(\top \wedge y_0 < x_1 \wedge y_1 < x_1)\{\ y_2 := y_1 + 1\ \}\ ; \\
\textbf{if}(\top \wedge y_0 < x_1 \wedge y_1 < x_1)\{\ \textbf{assert}(\neg(y_2 < x_1))\ \}\ ; \\
\textbf{if}(\top \wedge y_0 < x_1 \wedge y_1 < x_1)\{\ y_3 := y_2\ \}\ ; \\
\textbf{if}(\top \wedge y_0 < x_1 \wedge \neg(y_1 < x_1))\{\ \textbf{skip}\ \}\ ; \\
\textbf{if}(\top \wedge y_0 < x_1 \wedge \neg(y_1 < x_1))\{\ y_3 := y_1\ \}\ ; \\
\textbf{if}(\top \wedge y_0 < x_1)\{\ y_4 := y_3\ \}\ ; \\
\textbf{if}(\top \wedge \neg(y_0 < x_1))\{\ \textbf{skip}\ \}\ ; \\
\textbf{if}(\top \wedge \neg(y_0 < x_1))\{\ y_4 := y_1\ \}\ ; \\
\textbf{if}(\top)\{\ \textbf{assert}(y_4 == x_1 \vee y_4 > 2)\ \}
\end{array}
$$

Figura 3.8: Conditional Normal Form do programa na Figura 3.7

Observe-se que um programa na forma CNF está já muito próximo de uma leitura lógica uma vez que cada condicional (comando *if*) pode ser visto como uma fórmula implicativa.

3.3.4 Transformação $\mathcal{T}_4$

A transformação $\mathcal{T}_4$ transforma um comando na forma CNF, numa sua representação através de uma fórmula lógica. Mais concretamente, a transformação $\mathcal{T}_4$ aplicada a um comando irá produzir dois conjuntos de fórmulas $\mathcal{C}_c$ e $\mathcal{P}_c$, que devem ser pensados como a fórmula $\bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$. Por um lado, em $\mathcal{C}_c$ será coleccionada a informação relativa aos *assignments* ($x\ :=\ e$) do comando c. Em $\mathcal{P}_c$ será coleccionada a informação sobre os *asserts* (**assert** $(\ b\ )$) do comando c. O programa c será válido se for possível garantir que todas as asserções em $\mathcal{P}_c$ seguem dos *assignments* representados em $\mathcal{C}_c$.

Definição 3.6. *Para qualquer* $c \in \mathcal{C}nf$*, a transformação* $\mathcal{T}_3\,(\,c\,)$ *produz um par de conjuntos de fórmulas lógicas, definido recursivamente do seguinte modo:*

$$
\begin{array}{rcll}
\mathcal{T}_4\,(\,\textbf{\textit{if}}\,(\,b'\,)\ \textbf{\textit{then skip}}\,) & := & (\,\emptyset,\ \emptyset\,) & \\
\mathcal{T}_4\,(\,\textbf{\textit{if}}\,(\,b'\,)\ \textbf{\textit{then assert}}\,(\,b\,)\,) & := & (\,\emptyset,\ \{\,b' \rightarrow\ b\,\}\,) & \\
\mathcal{T}_4\,(\,\textbf{\textit{if}}\,(\,b'\,)\ \textbf{\textit{then}}\ x\ :=\ e\,) & := & (\,\{\,b' \rightarrow (\,x = e\,)\,\},\ \emptyset\,) & \\
\mathcal{T}_4\,(\,c_i\ \textbf{;}\ c_j\,) & := & \big(\,\mathcal{C}_{c_i} \cup \mathcal{C}_{c_j},\ \mathcal{P}_{c_i} \cup \mathcal{P}_{c_j}\,\big) & \textit{onde}\quad (\,\mathcal{C}_{c_i},\ \mathcal{P}_{c_i}\,) = \mathcal{T}_4\,(\,c_i\,) \\
 & & & \qquad\quad\ (\,\mathcal{C}_{c_j},\ \mathcal{P}_{c_j}\,) = \mathcal{T}_4\,(\,c_j\,)
\end{array}
$$

Para tornarmos a exposição mais concisa, estabelecem-se algumas convenções de notação. A Convenção 3.7, simplifica a representação dos conjuntos que resultam da aplicação da transformação $\mathcal{T}_4$: quando nos referimos a $\mathcal{C}_c$, referimo-nos à primeira componente do par $(\,\mathcal{C},\ \mathcal{P}\,) := \mathcal{T}_4\,(\,c\,)$, e analogamente para $\mathcal{P}_c$.

Convenção 3.7. *Dado um comando* $c \in \mathcal{C}nf$*,* $\mathcal{C}_c$ *e* $\mathcal{P}_c$ *denotam respectivamente a 1ª e a 2ª componentes de* $\mathcal{T}_4\,(\,c\,)$ *(o resultado de aplicar a transformação* $\mathcal{T}_4\,(\,c\,)$ *ao comando c).*

Neste capítulo adotaremos a convenção que se segue, que fixa que a conjunção lógica de um conjunto de fórmulas lógicas, quando este é vazio (denotado por $\emptyset$), não é mais do que uma representação da fórmula $\top$.

Convenção 3.8. *Seja $\mathcal{S}$ um conjunto de expressões em $\mathcal{E}xpression_{Boolean}$. Convencionamos que $\bigwedge \mathcal{S}$ denota a conjunção lógica de todos os elementos do conjunto $\mathcal{S}$, assumindo, em particular, que quando $\mathcal{S} = \emptyset$, $\bigwedge \mathcal{S} =_{def} \top$.*

O resultado de aplicar a transfomação $\mathcal{T}_4$ ao programa da Figura 3.8 é expresso nas Figuras 3.9 e 3.10, que indicam os conjuntos $\mathcal{C}$ e $\mathcal{P}$ respectivamente.

$$\left\{ \begin{array}{rl} \top \rightarrow & x_1 == 2, \\ \top \wedge y_0 < x_1 \rightarrow & y_1 == y_0 + 1, \\ \top \wedge y_0 < x_1 \wedge y_1 < x_1 \rightarrow & y_2 == y_1 + 1, \\ \top \wedge y_0 < x_1 \wedge y_1 < x_1 \rightarrow & y_3 == y_2, \\ \top \wedge y_0 < x_1 \wedge \neg(y_1 < x_1) \rightarrow & y_3 == y_1, \\ \top \wedge y_0 < x_1 \rightarrow & y_4 == y_3, \\ \top \wedge \neg(y_0 < x_1) \rightarrow & y_4 == y_1 \end{array} \right\}$$

Figura 3.9: Conjunto $\mathcal{C}$ resultante

$$\left\{ \begin{array}{rl} \top \wedge y_0 < x_1 \wedge y_1 < x_1 \rightarrow & \neg(y_2 < x_1), \\ \top \rightarrow & y_4 == x_1 \vee y_4 > 2 \end{array} \right\}$$

Figura 3.10: Conjunto $\mathcal{P}$ resultante

Note-se que a fórmula $\bigwedge \mathcal{C} \rightarrow \bigwedge \mathcal{P}$ resultante não é válida, uma vez que é possível satisfazer a fórmula $\bigwedge \mathcal{C} \wedge \neg \bigwedge \mathcal{P}$. De facto, quando passamos esta fórmula (Anexo A.2) ao SAT *solver* Z3 usado na implementação, este indica a violação da fórmula quando y_0 é -1, y_1 é 0, y_2 é 1, y_3 é 1, y_4 é 1 e x_1 é 2 (Anexo A.3). Neste caso, existem outros contra-modelos para a fórmula, mas a ferramenta Z3 apenas se preocupa em fornecer um contra-modelo.

Capítulo 4

Correcção e Completude da transformação de programas $\mathcal{C}nf$ em fórmulas

Neste capítulo, dão-se alguns passos no sentido de formalizar provas relacionadas com a correcção e a completude do método BMCsw. Em particular, focaremos na correcção e completude da transformação $\mathcal{T}_4$, onde se realiza a codificação de um programa $\mathcal{W}hile_{\mathbf{assert}}$, após normalização pelas transformações $\mathcal{T}_1$, $\mathcal{T}_2$ e $\mathcal{T}_3$, numa fórmula lógica.

Neste capítulo estabelece-se a correcção (Teorema 4.12) e a completude (Teorema 4.14) da transformação $\mathcal{T}_4$, mostrando que é possível codificar um programa imperativo numa fórmula lógica de tal modo que o programa é correcto (todos os seus asserts são validados) se e só se a fórmula for válida em lógica de primeira ordem.

Este capítulo está dividido em três secções. Na primeira secção apresentam-se alguns resultados preliminares sobre os comandos $\mathcal{C}nf$ e a função **Eval**, que auxiliam os resultados que pretendemos demonstrar. Na segunda secção provamos a correcção da transformação $\mathcal{T}_4$. Na terceira secção provamos a completude da transformação $\mathcal{T}_4$.

4.1 Resultados preliminares

Uma primeira observação simples acerca das variáveis das fórmulas geradas por $\mathcal{T}_4$ é a seguinte.

Lema 4.1.

$$\forall\ c \in \mathcal{C}nf,\ FV\left(\bigwedge \mathcal{C}_c \right) \subseteq FV\ (\ c\ ) \tag{4.1}$$

Demonstração. Por indução em $c \in \mathcal{C}nf$.

Caso $c =$ **if** $(\ b'\ )$ **then skip**:

Então $\mathcal{C}_c = \emptyset$ e $FV\ (\ \top\ ) = \emptyset \subseteq FV\ (\ b'\ ) = FV\ (\ c\ )$.

Caso $c =$ **if** $(\ b'\ )$ **then assert** $(\ b\ )$:

Análogo ao caso anterior.

Caso $c =$ **if** $(\ b'\ )$ **then** $x\ :=\ e$:

Então $\mathcal{C}_c = \{\ b' \to\ x = e\ \}$ e $FV\ (\ \bigwedge \mathcal{C}_c\ ) = FV\ (\ b'\ ) \cup \{\ x\ \} \cup FV\ (\ e\ ) = FV\ (\ c\ )$.

Caso $c = c_i \,;\, c_j$:

Então $\mathcal{C}_c = \mathcal{C}_{c_i} \cup \mathcal{C}_{c_j}$ e $FV\left(\bigwedge \mathcal{C}_{c_i} \right) \subseteq FV\left(c_i \right)$ e $FV\left(\bigwedge \mathcal{C}_{c_j} \right) \subseteq FV\left(c_j \right)$ por hipótese de indução.

Logo, $FV\left(\bigwedge \left(\mathcal{C}_{c_i} \cup \mathcal{C}_{c_j} \right) \right) = FV\left(\mathcal{C}_{c_i} \right) \cup FV\left(\mathcal{C}_{c_j} \right) \subseteq FV\left(c_i \right) \cup FV\left(c_j \right) = FV\left(c_i \,;\, c_j \right)$.

□

O Lema 4.2 surge da constatação que se num dado estado s, um comando $c \in \mathcal{C}nf$ não é avaliado no estado de *erro*, então o resultado de avaliar o comando c no estado s através da relação $\rightsquigarrow$ é o estado resultante da aplicação da função **Eval**.

Lema 4.2.

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ \text{safe}\left(c,\ s \right) \Rightarrow \left(c,\ s \right) \rightsquigarrow \textbf{\textit{Eval}}\left(c,\ s \right)$$

Demonstração. Por indução estrutural em $c \in \mathcal{C}nf$

i) Caso $c = \textbf{if}\ (b')\ \textbf{then skip}$.

Por um lado, pela definição da função **Eval**, $\textbf{Eval}\left(\textbf{if}\ (b')\ \textbf{then skip},\ s \right) = s$.

Por outro, pela semântica operacional para comandos $\mathcal{C}nf$ (Tabela 2.5),

$\left(\textbf{if}\ (b')\ \textbf{then skip},\ s \right) \rightsquigarrow s$.

ii) Caso $c = \textbf{if}\ (b')\ \textbf{then assert}\ (b)$.

Por um lado, pela definição de **Eval**, $\textbf{Eval}\left(\textbf{if}\ (b')\ \textbf{then assert}\ (b),\ s \right) = s$.

Por outro, pela semântica operacional para comandos $\mathcal{C}nf$ (Tabela 2.5),

Caso $s \models b'$:

Da hipótese $\textit{safe}\left(\textbf{assert}\ (b),\ s \right)$, temos, $\left(\textbf{assert}\ (b),\ s \right) \not\rightsquigarrow \textit{erro}$.

Portanto, teremos que ter $s \models b$, pelo que pela regra **CNF - ASSERT**

temos $\left(\textbf{if}\ (b')\ \textbf{then assert}\ (b),\ s \right) \rightsquigarrow s$.

Caso $s \not\models b'$:

Pela regra **CNF - ASSERT - False**,

temos de imediato $\left(\textbf{if}\ (b')\ \textbf{then assert}\ (b),\ s \right) \rightsquigarrow s$.

iii) Caso $c = \textbf{if}\ (b')\ \textbf{then}\ x \ := \ e$.

Caso $s \models b'$:

Por um lado, pela semântica operacional,

temos $\left(\textbf{if}\ (b')\ \textbf{then}\ x \ := \ e,\ s \right) \rightsquigarrow s\begin{pmatrix} x \\ [\![e]\!]_s \end{pmatrix}$ pela regra **CNF - ASSIGN**.

Pelo outro, a definição de **Eval** revela que

$$\textbf{Eval}\left(\textbf{if}\ (b')\ \textbf{then}\ x \ := \ e,\ s \right) = s\begin{pmatrix} x \\ [\![e]\!]_s \end{pmatrix}$$

Caso $s \not\models b'$:

Por um lado, temos $\left(\textbf{if}\ (b')\ \textbf{then}\ x \ := \ e,\ s \right) \rightsquigarrow s$ pela regra **CNF - ASSIGN - False**.

Por outro lado, a devinição de **Eval** revela que $\textbf{Eval}\left(\textbf{if}\ (b')\ \textbf{then}\ x \ := \ e,\ s \right) = s$.

iv) Caso $c = c_i \,;\, c_j$.

Por hipótese de indução temos,

$$HI_i : \forall\ s \in \Sigma,\ \textit{safe}\left(c_i,\ s \right) \Rightarrow \left(c_i,\ s \right) \rightsquigarrow \textbf{Eval}\left(c_i,\ s \right)$$

$$HI_j : \forall\ s \in \Sigma,\ \textit{safe}\left(c_j,\ s \right) \Rightarrow \left(c_j,\ s \right) \rightsquigarrow \textbf{Eval}\left(c_j,\ s \right)$$

Queremos mostrar que

$$\forall\ s \in \Sigma,\ \textit{safe}\left(c_i \,;\, c_j,\ s \right) \Rightarrow \left(c_i \,;\, c_j,\ s \right) \rightsquigarrow \textbf{Eval}\left(c_i \,;\, c_j,\ s \right)$$

Pela definição da função **Eval**, $\textbf{Eval}(\ c_i\ \textbf{;}\ c_j,\ s\ ) = \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$.

Usando a hipótese e o Lema 2.29 temos:
a) $\mathit{safe}(\ c_i,\ s\ )$
b) $\forall\ s'' \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow s'' \Rightarrow \mathit{safe}(\ c_j,\ s''\ )$

Por a) e HI_i temos

$$(\ c_i,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c_i,\ s\ ) \tag{4.2}$$

De (4.2), por b), segue $\mathit{safe}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$.

Recorrendo agora a HI_j temos

$$(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \rightsquigarrow \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \tag{4.3}$$

Finalmente, usando (4.2) e (4.3) temos $(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$, aplicando a regra **CNF - COMPOSITION** da semântica operacional.

□

O recíproco do lema anterior é também verdadeiro e temos a seguinte caracterização alternativa de *safe*:

Proposição 4.3.

$$\forall\ c \in \mathcal{C}nf,\ \forall\ s \in \Sigma,\ (\ c,\ s\ ) \rightsquigarrow \boldsymbol{Eval}(\ c,\ s\ ) \Leftrightarrow \mathsf{safe}(\ c,\ s\ )$$

Demonstração. Em 4.2 mostramos já que:

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ \mathit{safe}(\ c,\ s\ ) \Rightarrow (\ c,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c,\ s\ )$$

A implicação recíproca é uma consequência do determinismo da relação $\rightsquigarrow$ (ver Proposição 2.33): Se não tivessemos $\mathit{safe}(\ c,\ s\ )$, teriamos $(\ c,\ s\ ) \rightsquigarrow \mathit{erro}$ e como por hipótese temos $(\ c,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c,\ s\ )$, pelo determinismo de $\rightsquigarrow$ teriamos que ter $\mathit{erro} = \textbf{Eval}(\ c,\ s\ )$, o que é falso. □

Dado um comando $c \in \mathcal{C}nf$ e um estado arbitrário s, o Lema 4.4 permite garantir que a avaliação do comando c num estado s é o próprio s, se a relação $\rightsquigarrow$ não avaliar $(\ c,\ s\ )$ em *erro* e o estado s validar $\bigwedge \mathcal{C}_c$. Intuitivamente: s já atribui às variáveis os valores que resultam da execução de c.

Lema 4.4.

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ (\ s \models \wedge \mathcal{C}_c\ ) \wedge \text{safe}(\ c,\ s\ ) \Rightarrow s = \textbf{\textit{Eval}}(\ c,\ s\ )$$

Demonstração. Por indução estrutural em $c \in \mathcal{C}nf$

i) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then skip}$ ou $c = \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )$.
Por definição de **Eval** tem-se de imediato $s = \textbf{Eval}(\ c,\ s\ )$.

iii) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ :=\ e$.
Se $s \models b'$:
Como $\textbf{Eval}(\ c,\ s\ ) = s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}$, basta mostrar que $s(\ x\ ) = [\![\ e\ ]\!]_s$.
A hipótese diz-nos que $s \models b' \rightarrow\ x = e$.
Assim, com $s \models b'$, $s \models\ x = e$.
Daqui segue que $[\![\ x\ ]\!]_s = s(\ x\ ) = [\![\ e\ ]\!]_s$.

Se $s \not\models b'$:
De imediato, por definição de **Eval**, tem-se $s = \textbf{Eval}(\ c,\ s\ )$.

iv) Caso $c = c_i\ \textbf{;}\ c_j$.
Por hipótese de indução temos,

$$HI_i : \forall\ s \in \Sigma,\ (\ s \models \wedge \mathcal{C}_{c_i}\ ) \wedge \textit{safe}(\ c_i,\ s\ ) \Rightarrow s = \textbf{Eval}(\ c_i,\ s\ )$$

$$HI_j : \forall\ s \in \Sigma,\ \left(\ s \models \wedge \mathcal{C}_{c_j}\ \right) \wedge \textit{safe}(\ c_j,\ s\ ) \Rightarrow s = \textbf{Eval}(\ c_j,\ s\ )$$

Queremos mostrar que

$$\forall\ s \in \Sigma,\ \left(\ s \models \wedge \mathcal{C}_{c_i\ ;\ c_j}\ \right) \wedge \textit{safe}(\ c_i\ \textbf{;}\ c_j,\ s\ ) \Rightarrow s = \textbf{Eval}(\ c_i\ \textbf{;}\ c_j,\ s\ )$$

1) Por um lado, $\textbf{Eval}(\ c_i\ \textbf{;}\ c_j,\ s\ ) = \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$.
2) Da hipótese sabemos que:
a) $s \models \bigwedge \mathcal{C}_i$;
b) $s \models \bigwedge \mathcal{C}_j$;
Por a) e HI_i, tiramos que

$$s = \textbf{Eval}(\ c_i,\ s\ ) \tag{4.4}$$

Portanto b) é equivalente a termos $\textbf{Eval}(\ c_i,\ s\ ) \models \bigwedge \mathcal{C}_j$ e daqui, por HI_j, vem

$$\textbf{Eval}(\ c_i,\ s\ ) = \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )\,.$$

Logo, atendendo a (4.4) e a 1), conclui-se $s = \textbf{Eval}(\ c_i\ \textbf{;}\ c_j,\ s\ )$.

□

O próximo lema constata que as variáveis que não são manipuladas (assigned) pelos comandos, mantêm o mesmo valor, quando aplicada a função **Eval**. A parte 1) do lema surge como contra-recíproco da parte 2).

Lema 4.5.

$$1)\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ \forall y \in \mathcal{V}_{Integer},\ \textbf{\textit{Eval}}(\ c,\ s\ )(\ y\ ) \neq s(\ y\ ) \Rightarrow y \in \textbf{\textit{assign}}(\ c\ )$$
$$2)\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ \forall y \in \mathcal{V}_{Integer},\ y \notin \textbf{\textit{assign}}(\ c\ ) \Rightarrow \textbf{\textit{Eval}}(\ c,\ s\ )(\ y\ ) = s(\ y\ )$$

Demonstração. Comecemos por notar que 1) é o contra-recíproco de 2). A demonstração de 2) segue por indução estrutural nos comandos $c \in \mathcal{C}nf$.

i) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then skip}$.
Segue da definição que $\textbf{Eval}(\ c,\ s\ ) = s$.
Logo $\forall\ y \in \mathcal{V}_{Integer},\ \textbf{Eval}(\ c,\ s\ )(\ y\ ) = s(\ y\ )$.

ii) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )$.
Análogo ao caso anterior.

iii) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ :=\ e$.
Consideremos $y \notin \textbf{assign}(\ c\ )$, *i.e.*, $y \notin \{\ x\ \}$.
Caso $s \not\models b'$:
Análogo aos casos anteriores.

Caso $s \models b'$:
Por definição, $\textbf{Eval}(\ c,\ s\ ) = s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}$,
mas como $x \neq\ y$, temos $s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}(\ y\ ) = s(\ y\ )$.

iv) Caso $c = c_i\ ;\ c_j$.
Temos as seguintes hipóteses de indução:
HI_i: $\forall\ s \in \Sigma,\ \forall\ y \in \mathcal{V}_{Integer},\ y \notin \textbf{assign}(\ c_i\ ) \Rightarrow \textbf{Eval}(\ c_i,\ s\ )(\ y\ ) = s(\ y\ )$
HI_j: $\forall\ s \in \Sigma,\ \forall\ y \in \mathcal{V}_{Integer},\ y \notin \textbf{assign}(\ c_j\ ) \Rightarrow \textbf{Eval}(\ c_j,\ s\ )(\ y\ ) = s(\ y\ )$

Queremos mostrar que $\forall\ s \in \Sigma,\ \forall\ y \in \mathcal{V}_{Integer}$,

$$y \notin \textbf{assign}(\ c_i\ ) \cup \textbf{assign}(\ c_j\ ) \Rightarrow \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )(\ y\ ) = s(\ y\ )$$

Pela hipótese HI_j, tomando $\textbf{Eval}(\ c_i,\ s\ )$, para s tem-se

$$\textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )(\ y\ ) = \textbf{Eval}(\ c_i,\ s\ )(\ y\ )$$

Pela hipótese HI_i, $\textbf{Eval}(\ c_i,\ s\ )(\ y\ ) = s(\ y\ )$.

Retira-se que $\textbf{Eval}(\ c_i\ ;\ c_j,\ s\ )(\ y\ ) = s(\ y\ )$.

□

Os Lemas 4.6 e 4.7 devem ler-se em conjunto. O Lema 4.6 surge da constatação que dados dois estados arbitrários s_1 e s_2, e um comando $c \in \mathcal{C}nf$ se ambos os estados atribuem os mesmos valores às variáveis presentes em c, então o resultado de avaliar $\textbf{Eval}(\ c,\ s_1\ )$ e $\textbf{Eval}(\ c,\ s_2\ )$ será o mesmo.

Lema 4.6.

$$\forall c \in \mathcal{C}nf,\ \forall s_1, s_2 \in \Sigma,$$
$$(\ \forall\ x \in FV(\ c\ ):\ s_1(\ x\ ) = s_2(\ x\ )\ ) \Rightarrow (\ \forall\ x \in FV(\ c\ ):\ \textbf{\textit{Eval}}(\ c,\ s_1\ )(\ x\ ) = \textbf{\textit{Eval}}(\ c,\ s_2\ )(\ x\ )\ )$$

Demonstração. Por indução em $c \in \mathcal{C}nf$.

Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then skip}$ ou $c = \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )$. Por definição, tem-se para qualquer $s \in \Sigma$, $\textbf{Eval}\,(\ c,\ s\ ) = s$. Assim o antecedente e o consequente da implicação coincidem.

Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ := \ e$.

Caso $s_1 \not\models b'$.

Da hipótese e do Lema 2.15, $s_2 \not\models b'$ pois $FV\ (\ b'\ ) \subseteq FV\ (\ c\ )$.
Por definição, $\textbf{Eval}\,(\ c,\ s_1\ ) = s_1$ e $\textbf{Eval}\,(\ c,\ s_2\ ) = s_2$
e novamente temos a mesma condição no antecedente e consequente da implicação.

Caso $s_1 \models b'$.

Da hipótese e do Lema 2.15, $s_2 \models b'$.
Por definição, $\textbf{Eval}\,(\ c,\ s_1\ ) = s_1 \begin{pmatrix} x \\ [\![\ e\ ]\!]_{s_1} \end{pmatrix}$ e $\textbf{Eval}\,(\ c,\ s_2\ ) = s_2 \begin{pmatrix} x \\ [\![\ e\ ]\!]_{s_2} \end{pmatrix}$.
Tendo em atenção a hipótese, basta mostrar que $[\![\ e\ ]\!]_{s_1} = [\![\ e\ ]\!]_{s_2}$.

Da hipótese, segue em particular que $\forall\ x \in FV\ (\ e\ ) :\ s_1\ (\ x\ ) = s_2\ (\ x\ )$.
Logo, pelo Lema 2.13, $[\![\ e\ ]\!]_{s_1} = [\![\ e\ ]\!]_{s_2}$

Caso $c = c_i\ \textbf{;}\ c_j$,

Suponhamos que $\forall\ x \in FV\ (\ c_i\ \textbf{;}\ c_j\ ) :\ s_1\ (\ x\ ) = s_2\ (\ x\ )$.
Em particular $\forall\ x \in FV\ (\ c_i\ ) :\ s_1\ (\ x\ ) = s_2\ (\ x\ )$.
Logo por HI,

$$\forall\ x \in FV\ (\ c_i\ ) :\ \textbf{Eval}\,(\ c_i,\ s_1\ )\,(\ x\ ) = \textbf{Eval}\,(\ c_i,\ s_2\ )\,(\ x\ ) \tag{4.5}$$

Temos também $\forall\ x \in FV\ (\ c_j\ ) :\ \textbf{Eval}\,(\ c_i,\ s_1\ )\,(\ x\ ) = \textbf{Eval}\,(\ c_i,\ s_2\ )\,(\ x\ )$ pois:

a) caso $x \in FV\ (\ c_i\ )$: tal segue de (4.5)

b) caso $x \notin FV\ (\ c_i\ )$: então $x \notin \textbf{assign}\,(\ c_i\ )$ e assim,

$$\textbf{Eval}\,(\ c_i,\ s_1\ )\,(\ x\ ) =_{i)} s_1\ (\ x\ ) =_{ii)} s_2\ (\ x\ ) =_{i)} \textbf{Eval}\,(\ c_i,\ s_2\ )\,(\ x\ )$$

i) $x \notin \textbf{assign}\,(\ c_i\ )$ e Lema 4.5 (parte 2).
ii) $\forall\ x \in FV\ (\ c_j\ )\ ,\ s_1\ (\ x\ ) = s_2\ (\ x\ )$.

Logo, por hipótese de indução,

$$\begin{array}{rl} 1)\forall\ x \in FV\ (\ c_j\ ) : & \textbf{Eval}\,(\ c_j,\ \textbf{Eval}\,(\ c_i,\ s_1\ )\ )\,(\ x\ ) = \textbf{Eval}\,(\ c_j,\ \textbf{Eval}\,(\ c_i,\ s_2\ )\ )\,(\ x\ ) \\ \Leftrightarrow & \textbf{Eval}\,(\ c_i\ \textbf{;}\ c_j,\ s_1\ )\,(\ x\ ) = \textbf{Eval}\,(\ c_i\ \textbf{;}\ c_j,\ s_2\ )\,(\ x\ ) \end{array}$$

Adicionalmente,

$$\begin{array}{rl} 2)\forall\ x \in FV\ (\ c_i\ ) : & \textbf{Eval}\,(\ c_j,\ \textbf{Eval}\,(\ c_i,\ s_1\ )\ )\,(\ x\ ) = \textbf{Eval}\,(\ c_j,\ \textbf{Eval}\,(\ c_i,\ s_2\ )\ )\,(\ x\ ) \\ \Leftrightarrow & \textbf{Eval}\,(\ c_i\ \textbf{;}\ c_j,\ s_1\ )\,(\ x\ ) = \textbf{Eval}\,(\ c_i\ \textbf{;}\ c_j,\ s_2\ )\,(\ x\ ) \end{array}$$

por $\textbf{Eval}\,(\ c_i,\ s_1\ )\,(\ x\ ) = \textbf{Eval}\,(\ c_i,\ s_2\ )\,(\ x\ )$ de (4.5),
por $x \notin \textbf{assign}\,(\ c_j\ )$ (uma vez que $c_i\ \textbf{;}\ c_j \in \mathcal{SA}$ e $x \in FV\ (\ c_i\ )$)
e pelo Lema 4.5.

□

Do modo semelhante, o Lema 4.7, refere que dados dois estados arbitrários s_1 e s_2, e um comando $c \in \mathcal{C}nf$ se ambos os estados atribuem os mesmos valores às variáveis presentes em c e se a relação $\rightsquigarrow$ avalia o par $(\ c,\ s_1\ )$ em *erro*, então, a relação $\rightsquigarrow$ também avaliará o par $(\ c,\ s_2\ )$ em *erro*.

Lema 4.7.

$$\forall c \in \mathcal{C}nf,\ \forall s_1, s_2 \in \Sigma,\quad (\ c,\ s_1\ ) \rightsquigarrow \text{erro} \wedge (\ \forall\ x \in FV\ (\ c\ ):\ s_1\ (\ x\ ) = s_2\ (\ x\ )\ ) \Rightarrow (\ c,\ s_2\ ) \rightsquigarrow \text{erro}$$

Demonstração. Por indução em $c \in \mathcal{C}nf$. Consideramos abaixo os casos que não são vacuosos (*i.e.* onde efectivamente é usada uma regra de avaliação que produz *erro*).

Caso **CNF - ASSERT - Error**.
Sabemos então que $c = \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )$ e que

$$s_1 \models b' \quad e \quad s_1 \not\models\ b \tag{4.6}$$

Sabemos também por suposição

$$\forall\ x \in FV\ (\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )\ ):\ s_1\ (\ x\ ) = s_2\ (\ x\ ) \tag{4.7}$$

Queremos mostrar que $(\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )\ ,\ s_2\ ) \rightsquigarrow \textit{erro}$.
Ora de (4.7) e (4.6), pelo Lemma 2.15, temos $s_2 \models b'$ e $s_2 \not\models\ b$ e consequentemente, pela regra **CNF - ASSERT - Error** segue $(\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )\ ,\ s_2\ ) \rightsquigarrow \textit{erro}$.

Caso **CNF - COMPOSITION - Break**.
Sabemos então que $c = c_i\ \textbf{;}\ c_j$ e que
1) $(\ c_i,\ s_1\ ) \rightsquigarrow \textit{erro}$
2) $\forall\ x \in FV\ (\ c_i\ ) \cup FV\ (\ c_j\ ):\ s_1\ (\ x\ ) = s_2\ (\ x\ )$
A hipótese de indução relativa a c_i diz-nos que:

$$HI_i:\ \forall\ s_1, s_2 \in \Sigma,\ (\ c_i,\ s_1\ ) \rightsquigarrow \textit{erro} \wedge \forall\ x \in FV\ (\ c_i\ )\ ,\ s_1\ (\ x\ ) = s_2\ (\ x\ ) \Rightarrow (\ c_i,\ s_2\ ) \rightsquigarrow \textit{erro}$$

De 2), temos em particular $\forall\ x \in FV\ (\ c_i\ ) : s_1\ (\ x\ ) = s_2\ (\ x\ )$ e daqui e de 1), por HI_i, segue $(\ c_i,\ s_2\ ) \rightsquigarrow \textit{erro}$.
Daqui, pela regra **CNF - COMPOSITION - Break**, conclui-se, como pretendido, $(\ c_i\ \textbf{;}\ c_j,\ s_2\ ) \rightsquigarrow \textit{erro}$.

Caso **CNF - COMPOSITION**.
Sabemos então que $c = c_i\ \textbf{;}\ c_j$ e que
1) para algum $s'' \neq \textit{erro}$, se tem $(\ c_i,\ s_1\ ) \rightsquigarrow s''$ e $(\ c_j,\ s''\ ) \rightsquigarrow \textit{erro}$
2) $\forall\ x \in FV\ (\ c_i\ ) \cup FV\ (\ c_j\ ):\ s_1\ (\ x\ ) = s_2\ (\ x\ )$ e, em particular,
  i) $\forall\ x \in FV\ (\ c_i\ ):\ s_1\ (\ x\ ) = s_2\ (\ x\ )$
  ii) $\forall\ x \in FV\ (\ c_j\ ):\ s_1\ (\ x\ ) = s_2\ (\ x\ )$

Uma vez que $s'' \neq \textit{erro}$, $s'' = \textbf{Eval}\ (\ c_i,\ s_1\ )$ pelo Lema 4.2
Pretendemos concluir que

$$(\ c_i,\ s_2\ ) \rightsquigarrow \textbf{Eval}\ (\ c_i,\ s_2\ ) \tag{4.8}$$

Sabemos

$$(\ c_i,\ s_1\ ) \rightsquigarrow \textbf{Eval}\ (\ c_i,\ s_1\ ) \tag{4.9}$$

Se tivessemos $(\ c_i,\ s_2\ ) \rightsquigarrow \textit{erro}$.
Dado que $\forall\ x \in FV\ (\ c_i\ )\ ,\ s_1\ (\ x\ ) = s_2\ (\ x\ )$, por HI_i, $(\ c_i,\ s_1\ ) \rightsquigarrow \textit{erro}$ o que contradiz (4.9).

Logo pela Proposição 2.33, temos que ter $(c_i, s_2) \rightsquigarrow \textbf{Eval}(c_i, s_2)$.

Pretendemos também concluir que

$$\forall x \in FV(c_j), \textbf{Eval}(c_i, s_1)(x) = \textbf{Eval}(c_i, s_2)(x) \tag{4.10}$$

Seja $x \in FV(c_j)$. Queremos mostrar $\textbf{Eval}(c_i, s_1)(x) = \textbf{Eval}(c_i, s_2)(x)$.
Caso $x \notin FV(c_i)$. Temos:

$$\textbf{Eval}(c_i, s_1)(x) =_{i)} s_1(x) =_{ii)} s_2(x) =_{i)} \textbf{Eval}(c_i, s_2)(x)$$

i) $x \notin \textbf{assign}(c_i)$ e Proposição 4.5 (parte 2)
ii) $\forall x \in FV(c_j), s_1(x) = s_2(x)$ (das hipóteses)

Caso $x \in FV(c_i)$.
Como $\forall x \in FV(c_i), s_1(x) = s_2(x)$ (da hipótese), pelo Lema 4.6,

$$\textbf{Eval}(c_i, s_1)(x) = \textbf{Eval}(c_i, s_2)(x)$$

Por (4.10) e por $(c_j, \textbf{Eval}(c_i, s_1)) \rightsquigarrow \textit{erro}$ (ver 1)), vem pela hipótese de indução relativa a c_j que

$$(c_j, \textbf{Eval}(c_i, s_2)) \rightsquigarrow \textit{erro} \tag{4.11}$$

Finalmente, de (4.8) e (4.11), aplicando a regra **CNF - COMPOSITION**
segue $(c_i \,\mathbf{;}\, c_j, s_2) \rightsquigarrow \textit{erro}$.

□

Na próxima proposição observaremos que o estado que resulta de calcular **Eval** de um comando c a partir de um qualquer estado, valida $\bigwedge \mathcal{C}_c$.

Proposição 4.8.

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ \textbf{\textit{Eval}}(\ c,\ s\ ) \models \bigwedge \mathcal{C}_c$$

Demonstração. Por indução estrutural em $c \in \mathcal{C}nf$.

i) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then skip}$.
Por definição de $\mathcal{T}_4$, $\mathcal{C}_c = \emptyset$, e deste modo, $\bigwedge \mathcal{C}_c = \top$, pelo que, trivialmente, temos $\models \bigwedge \mathcal{C}_c$.

ii) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )$.
Pela definição de $\mathcal{T}_4$, $\mathcal{C}_c = \emptyset$, e tal como no caso anterior,
$\bigwedge \mathcal{C}_c = \top$, e consequentemente, $\models \bigwedge \mathcal{C}_c$.

iii) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ := \ e$.
Pela definição de $\mathcal{T}_4$, $\mathcal{C}_c = \{\ b' \rightarrow (\ x = e\ )\ \}$.
Caso $s \models b'$: $\textbf{Eval}(\ c,\ s\ ) = s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}$, e como tal

$$s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix} \models (\ x = e\ )\ \ sse\ [\![\ e\ ]\!]_s = [\![\ e\ ]\!]_{s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}} \tag{4.12}$$

Como $x \notin FV\ (\ e\ )$ por $c \in \mathcal{SA}$, $[\![\ e\ ]\!]_{s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}} = [\![\ e\ ]\!]_s$, pelo Lema 2.13

Caso $s \not\models b'$:
Como neste caso $\textbf{Eval}(\ c,\ s\ ) = s$, também $\textbf{Eval}(\ c,\ s\ ) \not\models b'$.
Portanto, $\textbf{Eval}(\ c,\ s\ ) \models b' \rightarrow (\ x = e\ )$ (o antecedente da implicação é falso).

iv) Caso $c = c_i\ \textbf{;}\ c_j$.
Por hipótese de indução temos,

$$\begin{array}{ll} HI_i: & \forall\ s \in \Sigma,\ \textbf{Eval}(\ c_i,\ s\ ) \models \bigwedge \mathcal{C}_{c_i} \\ HI_j: & \forall\ s \in \Sigma,\ \textbf{Eval}(\ c_j,\ s\ ) \models \bigwedge \mathcal{C}_{c_j} \end{array}$$

Por definição, $\textbf{Eval}(\ c_i\ \textbf{;}\ c_j,\ s\ ) = \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$.
Queremos então mostrar que $\textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_i} \bigwedge \mathcal{C}_{c_j}$.

a) Por HI_j, tomando $\textbf{Eval}(\ c_j,\ s\ )$ para s, temos $\textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_j}$.

b)Falta provar que $\textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_i}$
Por $FV\ (\ \bigwedge \mathcal{C}_{c_i}\ ) \subseteq FV\ (\ c_i\ )$ (pelo Lema 4.1) e $FV\ (\ c_i\ ) \cap \textbf{assign}\ (\ c_j\ ) = \emptyset$ (por c ser $\mathcal{SA}$)
segue que $\textbf{assign}\ (\ c_j\ ) \cap FV\ (\ \bigwedge \mathcal{C}_{c_i}\ ) = \emptyset$. Então, pela Proposição 4.5,

$$\forall\ x \in FV\left(\bigwedge \mathcal{C}_{c_i}\right),\ \textbf{Eval}(\ c_i,\ s\ )(\ x\ ) = \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )(\ x\ )$$

Logo, pelo Lema 2.15, $\textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_i}$ sse $\textbf{Eval}(\ c_i,\ s\ ) \models \bigwedge \mathcal{C}_{c_i}$.

Logo, como $\textbf{Eval}(\ c_i,\ s\ ) \models \bigwedge \mathcal{C}_{c_i}$, por HI_i, também $\textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_i}$.

□

As proposições 4.9 e 4.10 têm um espírito semelhante. Estas proposições garantem, respectivamente, que dados um comando $c \in \mathcal{C}nf$ e um estado arbitrário s, se a relação $\rightsquigarrow$ avalia um par $(\ c,\ s\ )$ em $\textbf{Eval}(\ c,\ s\ )$ (respectivamente *erro*) então a relação $\rightsquigarrow$ também avaliará o par $(\ c,\ \textbf{Eval}(\ c,\ s\ )\ )$ no mesmo resultado, $\textbf{Eval}(\ c,\ s\ )$ (respectivamente *erro*).

A Proposição 4.9 incorpora uma ideia de *"ponto fixo"* ao afirmar que uma computação de c a partir do estado $\textbf{Eval}(\ c,\ s\ )$ produzirá o mesmo estado $\textbf{Eval}(\ c,\ s\ )$.

Proposição 4.9.

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\quad (\ c,\ s\ ) \rightsquigarrow \textbf{\textit{Eval}}(\ c,\ s\ ) \Rightarrow (\ c,\ \textbf{\textit{Eval}}(\ c,\ s\ )\ ) \rightsquigarrow \textbf{\textit{Eval}}(\ c,\ s\ )$$

Demonstração. Por indução estrutural em $c \in \mathcal{C}nf$.

i) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then skip}$.

Por definição, $\textbf{Eval}(\ c,\ s\ ) = s$ e portanto o antecedente e o consequente da implicação coincidem.

ii) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )$.

Análogo ao caso anterior.

iii) Caso $c = \textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ :=\ e$.

Caso $s \not\models b'$.

Análogo aos casos anteriores.

Caso $s \models b'$.

Por definição, $\textbf{Eval}(\ c,\ s\ ) = s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}$.

Por c ser um comando $\mathcal{SA}$, $x \notin FV\ (\ b'\ )$ e portanto, pelo Lema 2.15, $s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix} \models b'$.

Usando a regra **CNF - ASSIGN**, tem-se

$$\left(\textbf{if}\ (\ b'\ )\ \textbf{then}\ x\ :=\ e,\ s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}\right) \rightsquigarrow s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}\begin{pmatrix} x \\ [\![\ e\ ]\!]_{s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}} \end{pmatrix}$$

, e isto é o que se pretende mostrar, pois

$$s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}\begin{pmatrix} x \\ [\![\ e\ ]\!]_{s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}} \end{pmatrix} = s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix}$$

, por $x \notin\ c$ $(c \in \mathcal{SA})$, e $s\begin{pmatrix} x \\ [\![\ e\ ]\!]_s \end{pmatrix} = \textbf{Eval}(\ c,\ s\ )$.

iv) Caso $c = c_i\ \textbf{;}\ c_j$.

Supondo que $(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$,

queremos mostrar que $(\ c_i\ \textbf{;}\ c_j,\ \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$.

Atendendo à suposição e ao Lema 2.33 teremos que ter

1) $(\ c_i,\ s\ ) \rightsquigarrow \textbf{Eval}(\ c_i,\ s\ )$

2) $(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \rightsquigarrow \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$

De 1) por HI tem-se $(\ c_i,\ \mathbf{Eval}(\ c_i,\ s\ )\ ) \rightsquigarrow \mathbf{Eval}(\ c_i,\ s\ )$ e daqui, pelo Lemma 2.18, tem-se

3) $(\ c_i,\ \mathbf{Eval}(\ c_j,\ \mathbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \mathbf{Eval}(\ c_j,\ \mathbf{Eval}(\ c_i,\ s\ )\ )$,

pois para todo $y \in \mathcal{V}_{Integer}$, se $\mathbf{Eval}(\ c_j,\ \mathbf{Eval}(\ c_i,\ s\ )\ )(\ y\ ) \neq \mathbf{Eval}(\ c_i,\ s\ )(\ y\ )$, então, pelo Lema 4.5, $y \in \mathbf{assign}(\ c_j\ )$ e, como $c_i\ \mathbf{;}\ c_j \in \mathcal{SA}$, $y \notin FV(\ c_i\ )$. De 2) por HI tem-se

4) $(\ c_j,\ \mathbf{Eval}(\ c_j,\ \mathbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \mathbf{Eval}(\ c_j,\ \mathbf{Eval}(\ c_i,\ s\ )\ )$.

Finalmente, de 3) e 4), pela regra **CNF - COMPOSITION**,
tem-se $(\ c_i\ \mathbf{;}\ c_j,\ \mathbf{Eval}(\ c_j,\ \mathbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \mathbf{Eval}(\ c_j,\ \mathbf{Eval}(\ c_i,\ s\ )\ )$.

□

Proposição 4.10.

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ (\ c,\ s\ ) \rightsquigarrow erro \Rightarrow (\ c,\ \textbf{\textit{Eval}}(\ c,\ s\ )\ ) \rightsquigarrow erro$$

Demonstração. Por indução em $c \in \mathcal{C}nf$.

Os casos relativos às regras **CNF - SKIP**, **CNF - SKIP - False**, **CNF - ASSERT**, **CNF - ASSERT - False**, **CNF - ASSIGN** e **CNF - ASSIGN - False** são imediatos, por a hipótese $(\ c,\ s\ ) \rightsquigarrow erro$ não ser satisfeita.

Caso **CNF - ASSERT - Error**.
Neste caso tem-se $c =$ **if** $(\ b'\ )$ **then assert** $(\ b\ )$, $s \models b'$ e $s \not\models b$.
Como, por definição, $\textbf{Eval}(\ c,\ s\ ) = s$ segue que $\textbf{Eval}(\ c,\ s\ ) \models b'$ e $\textbf{Eval}(\ c,\ s\ ) \not\models\ b$, pelo que, usando a regra **CNF - ASSERT - Error**, se tem $(\ c,\ \textbf{Eval}(\ c,\ s\ )\ ) \rightsquigarrow erro$.

Caso **CNF - COMPOSITION - Break**.
Então, tem-se $c = c_i\ \textbf{;}\ c_j$ e $(\ c_i,\ s\ ) \rightsquigarrow erro$.
1) Por HI, $(\ c_i,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \rightsquigarrow erro$.
2) Como $c_i\ \textbf{;}\ c_j \in \mathcal{SA}$, $FV\ (\ c_i\ ) \cap \textbf{assign}\,(\ c_j\ ) = \emptyset$.
Logo, pelo Lema 4.5,

$$\forall\ x \in FV\ (\ c_i\ ):\ \textbf{Eval}(\ c_i,\ s\ )(\ x\ ) = \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )(\ x\ )$$

3) Daqui e de 1), pelo Lema 4.7, $(\ c_j,\ \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow erro$
4) Logo, pela regra **CNF - COMPOSITION - Break**, $(\ c_i\ \textbf{;}\ c_j,\ \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow erro$,
e $\textbf{Eval}(\ c_i\ \textbf{;}\ c_j,\ s\ ) = \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$.

Caso **CNF - COMPOSITION**.
Então, tem-se $c = c_i\ \textbf{;}\ c_j$,
e i) $(\ c_i,\ s\ ) \rightsquigarrow s''$, para algum $s'' \neq erro$,
e ii) $(\ c_j,\ s''\ ) \rightsquigarrow erro$.

1) Assim, usando a Proposição 2.33 podemos concluir que $s'' = \textbf{Eval}(\ c_i,\ s\ )$.
2) Por HI, $(\ c_j,\ \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow erro$.

3) Mostremos que $(\ c_i,\ \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$:
a) Se tivessemos $(\ c_i,\ \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow erro$,
Como $\forall\ x \in FV\ (\ c_i\ )$, $\textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )(\ x\ ) = \textbf{Eval}(\ c_i,\ s\ )(\ x\ )$,
pelo Lema 4.7, $(\ c_i,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \rightsquigarrow erro$, contrariando i).

b) Mostraremos que $\textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_i}$, pelo que, tendo em atenção à alinea a), ao Lema 4.4 e à Proposição 2.33,
$(\ c_i,\ \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$.
$\textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ ) \models \bigwedge \mathcal{C}_{c_i}$ dado que:
b.1) $\textbf{Eval}(\ c_i,\ s\ ) \models \bigwedge \mathcal{C}_{c_i}$ (Prop. 4.8) e
b.2) $\forall\ x \in FV\ (\ \bigwedge \mathcal{C}_{c_i}\ )$, $\textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )(\ x\ ) = \textbf{Eval}(\ c_j,\ s\ )(\ x\ )$;
esta observação segue pelo Lema 4.5, e pelo facto de $FV\ (\ \bigwedge \mathcal{C}_{c_i}\ ) \subseteq FV\ (\ c_i\ )$ (pelo Lema 4.1) e $FV\ (\ c_i\ ) \cap \textbf{assign}\,(\ c_j\ ) = \emptyset$ ($c_i\ \textbf{;}\ c_j \in \mathcal{SA}$).
b.3)Atendendo a b.1) e a b.2), basta agora usar o Lema 2.15.

4) Finalmente, de 3) e de 2), usando a regra **CNF - COMPOSITION** tem-se

$$(\ c_i\ \textbf{;}\ c_j,\ \textbf{Eval}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )\ ) \rightsquigarrow erro$$

□

4.2 Correcção de $\mathcal{T}_4$

Nesta secção provaremos a correcção da transformação $\mathcal{T}_4$. Mais concretamente, provaremos que um comando $c \in \mathcal{C}nf$ é *safe* se e só se a fórmula $\bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$ que lhe é associada pela transformação $\mathcal{T}_4$ é uma fórmula válida.

Além de resultados que já estabelecemos na Secção 4.1, precisamos ainda do resultado que se segue. Este resultado afirma que se uma execução de c atingiu o estado de *erro* a partir de um estado s que satisfazia as "condições do programa", esse estado s já não pode satisfazer "os *assert's* do programa".

Proposição 4.11.

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ (\ c,\ s\ ) \rightsquigarrow erro \wedge s \models \bigwedge \mathcal{C}_c \Rightarrow s \not\models \bigwedge \mathcal{P}_c$$

Demonstração. Por indução em $(\ c,\ s\ ) \rightsquigarrow erro$.

Pela semântica operacional (Tabela 2.5), as únicas regras que permitem a avaliação para o estado *erro* são as regras **CNF - ASSERT - Error**, **CNF - COMPOSITION - Break** e **CNF - COMPOSITION** (e , portanto, os restantes casos são demonstrados por vacuosidade).

Caso **CNF - ASSERT - Error**.
Suponhamos que $(\ \textbf{if}\ (\ b'\ )\ \textbf{then assert}\ (\ b\ )\ ,\ s\ ) \rightsquigarrow erro$ e $s \models \bigwedge \mathcal{C}_{\textbf{if}\,(\ b'\ )\,\textbf{then assert}\,(\ b\ )}$.
Queremos mostrar que $s \not\models \bigwedge \mathcal{P}_{\textbf{if}\,(\ b'\ )\,\textbf{then assert}\,(\ b\ )}$.

Sabemos então que $\bigwedge \mathcal{C}_{\textbf{if}\,(\ b'\ )\,\textbf{then assert}\,(\ b\ )} = \top$ e $\bigwedge \mathcal{P}_{\textbf{if}\,(\ b'\ )\,\textbf{then assert}\,(\ b\ )} = b' \rightarrow\ b$.
Para se aplicar a regra **CNF - ASSERT - Error** é necessário ter $s \models b'$ e $s \not\models b$. Ora, nestas condições, $s \not\models b' \rightarrow\ b$.

Caso **CNF - COMPOSITION - Break**.

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow erro}{(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow erro}\ \textbf{CNF - COMPOSITION - Break}$$

Sabemos então que
1) $(\ c_i,\ s\ ) \rightsquigarrow erro$,
2) $s \models \bigwedge \mathcal{C}_{c_i} \bigwedge \mathcal{C}_{c_j}$, *i.e.*, $s \models \bigwedge \mathcal{C}_{c_i}$ e $s \models \bigwedge \mathcal{C}_{c_j}$.
Sabemos ainda, por HI, que
3) $(\ c_i,\ s\ ) \rightsquigarrow erro \wedge s \models \bigwedge \mathcal{C}_{c_i} \Rightarrow s \not\models \bigwedge \mathcal{P}_{c_i}$.

Queremos mostrar que $s \not\models \mathcal{P}_{c_i\,;\,c_j}$, *i.e.*, $s \not\models \bigwedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$,

Ora, de 1), 2) e 3) temos $s \not\models \bigwedge \mathcal{P}_{c_i}$ e disto, também $s \not\models \bigwedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$.

Caso **CNF - COMPOSITION**.

$$\frac{(\ c_i,\ s\ ) \rightsquigarrow s'' \quad s'' \neq erro \quad (\ c_j,\ s''\ ) \rightsquigarrow erro}{(\ c_i\ \textbf{;}\ c_j,\ s\ ) \rightsquigarrow erro}\ \textbf{CNF - COMPOSITION}$$

Por suposição, sabemos então que,
$s \models \bigwedge \mathcal{C}_{c_i} \bigwedge \mathcal{C}_{c_j}$, *i.e.*, $s \models \bigwedge \mathcal{C}_{c_i}$ e $s \models \bigwedge \mathcal{C}_{c_j}$.

Queremos mostrar que $s \not\models \mathcal{P}_{c_i\,;\,c_j}$, *i.e.*, $s \not\models \bigwedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$,

Como $s'' \neq \mathit{erro}$, temos $\mathit{safe}(\,c_i,\ s\,)$ e, atendendo à Proposição 2.33, temos também $s'' = \mathbf{Eval}(\,c_i,\ s\,)$. Como também temos $s \models \bigwedge \mathcal{C}_{c_i}$, pelo Lema 4.4 segue que $s'' = \mathbf{Eval}(\,c_i,\ s\,) = s$. Portanto, como $s \models \bigwedge \mathcal{C}_{c_j}$, também $s'' \models \bigwedge \mathcal{C}_{c_j}$. Daqui, de $(\,c_j,\ s''\,) \rightsquigarrow \mathit{erro}$ e da hipótese de indução associada à última premissa da inferência **CNF - COMPOSITION**, temos $s'' \not\models \bigwedge \mathcal{P}_{c_j}$ e disto, segue $s \not\models \bigwedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$ (recorde-se que $s'' = s$).

□

Teorema 4.12. *(Correcção de $\mathcal{T}_4$)*

$$\forall c \in \mathcal{C}nf,\ \models \bigwedge \mathcal{C}_c \to \bigwedge \mathcal{P}_c \Rightarrow \mathrm{safe}(\,c\,)$$

Demonstração. Mostraremos o contra-recíproco da implicação.
Suponhamos que não temos $\mathit{safe}(\,c\,)$, *i.e.*, suponhamos que $\exists\ s \in \Sigma,\ (\,c,\ s\,) \rightsquigarrow \mathit{erro}$.

Logo, pela Proposição 4.10, temos

$$(\,c,\ \mathbf{Eval}(\,c,\ s\,)\,) \rightsquigarrow \mathit{erro} \tag{4.13}$$

Pela Proposição 4.8, temos

$$\mathbf{Eval}(\,c,\ s\,) \models \bigwedge \mathcal{C}_c \tag{4.14}$$

Logo de (4.13) e (4.14), pela Proposição 4.11,

$$\mathbf{Eval}(\,c,\ s\,) \not\models \wedge \mathcal{P}_c \tag{4.15}$$

De (4.14) e (4.15) segue que $\not\models \bigwedge \mathcal{C}_c \to \bigwedge \mathcal{P}_c$, como pretendido.

□

4.3 Completude de $\mathcal{T}_4$

Finalmente, estamos em condições de provar o Teorema da Completude (Corolário 4.14) e assim garantir que para concluir que a validade da fórmula gerada pela transformação $\mathcal{T}_4$ para *"representar"* o comando c basta provar que c é *"safe"* (não conduz a *erro* durante a sua execução).

Começamos por demonstrar a ideia essencial da completude da transformação $\mathcal{T}_4$.

Teorema 4.13.

$$\forall c \in \mathcal{C}nf,\ \forall s \in \Sigma,\ \text{safe}(\ c,\ s\ ) \Rightarrow s \models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$$

Demonstração. Por indução estrutural em $c \in \mathcal{C}nf$

i) Caso $c =$ **if** (b') **then skip**.
Por definição, $\mathcal{C}_c = \emptyset$ e $\mathcal{P}_c = \emptyset$.
Como $\bigwedge \emptyset = \top$, temos que mostrar $s \models \top \rightarrow \top$, o que é imediato pois $\top \rightarrow \top$ é tautologia.

ii) Caso $c =$ **if** (b') **then assert** (b).
Por definição, $\mathcal{C}_c = \emptyset$ e $\mathcal{P}_c = \{\ b' \rightarrow\ b\ \}$.
Assim, temos que mostrar $s \models b' \rightarrow\ b$.

Caso $s \models b'$:
Como, por hipótese, *safe*(**if** (b') **then assert** (b) , s) temos que ter $s \models\ b$.
De outro modo teríamos (**if** (b') **then assert** (b) , s) $\rightsquigarrow$ *erro*,
pois (**assert** (b) , s) $\rightsquigarrow$ *erro* se $s \not\models b$,
contrariando a suposição *safe*(**if** (b') **then assert** (b) , s).

Caso $s \not\models b'$:
De imediato temos $s \models b' \rightarrow\ b$.

iii) Caso $c =$ **if** (b') **then** $x\ :=\ e$.
Por definição, $\mathcal{C}_c = \{\ b' \rightarrow (\ x = e\ )\ \}$ e $\mathcal{P}_c = \emptyset$ e portanto,
temos que mostrar $s \models (\ b' \rightarrow (\ x = e\ )\ ) \rightarrow \top$.
Ora, isto é verdade porque o consequente da implicação é uma tautologia.

iv) Caso $c = c_i\ \mathbf{;}\ c_j$.
Por definição, $\bigwedge \mathcal{C}_c = \bigwedge \mathcal{C}_{c_i} \bigwedge \mathcal{C}_{c_j}$ e $\bigwedge \mathcal{P}_c = \bigwedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$.
Como hipóteses de indução temos,

$$HI_i : \forall\ s \in \Sigma,\ \mathit{safe}(\ c_i,\ s\ ) \Rightarrow s \models \bigwedge \mathcal{C}_{c_i} \rightarrow \bigwedge \mathcal{P}_{c_i}$$

$$HI_j : \forall\ s \in \Sigma,\ \mathit{safe}(\ c_j,\ s\ ) \Rightarrow s \models \bigwedge \mathcal{C}_{c_j} \rightarrow \bigwedge \mathcal{P}_{c_j}$$

Queremos mostrar que

$$\forall\ s \in \Sigma,\ \mathit{safe}(\ c_i\ \mathbf{;}\ c_j,\ s\ ) \Rightarrow s \models \bigwedge \mathcal{C}_{c_i} \bigwedge \mathcal{C}_{c_j} \rightarrow \bigwedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$$

Seja $s \in \Sigma$ tal que *safe*($c_i\ \mathbf{;}\ c_j,\ s$) e suponhamos que $s \models \bigwedge \mathcal{C}_{c_i} \bigwedge \mathcal{C}_{c_j}$,
temos que mostar que $s \models \wedge \mathcal{P}_{c_i} \bigwedge \mathcal{P}_{c_j}$.
Como *safe*($c_i\ \mathbf{;}\ c_j$), segue pelo Lema 2.29:
1) *safe*($c_i,\ s$)
2) $\forall\ s' \in \Sigma,\ (\ c_i,\ s\ ) \rightsquigarrow s' \Rightarrow \mathit{safe}(\ c_j,\ s'\ )$

a) Ora, por *safe*($c_i,\ s$) e por HI_i, sabemos que $s \models \bigwedge \mathcal{C}_{c_i} \rightarrow \bigwedge \mathcal{P}_{c_i}$
e como tal, dado que $s \models \wedge \mathcal{C}_{c_i}$ (por $s \models \bigwedge \mathcal{C}_{c_i} \bigwedge \mathcal{C}_{c_j}$), também $s \models \wedge \mathcal{P}_{c_i}$.

b) Falta mostrar que $s \models \wedge \mathcal{P}_{c_j}$.

Uma vez que $\textit{safe}(\ c_i,\ s\ )$, pelo Lema 4.2, $(\ c_i,\ s\ ) \leadsto \textbf{Eval}(\ c_i,\ s\ )$
e, como tal, por 2), $\textit{safe}(\ c_j,\ \textbf{Eval}(\ c_i,\ s\ )\ )$.
Daqui, por HI_j,

$$\textbf{Eval}(\ c_i,\ s\ ) \models \bigwedge \mathcal{C}_{c_j} \rightarrow \bigwedge \mathcal{P}_{c_j} \tag{4.16}$$

Uma vez que $s \models \wedge \mathcal{C}_{c_i}$ e por 1) sabemos que $\textit{safe}(\ c_i,\ s\ )$,
usando o Lema 4.4, temos que $s = \textbf{Eval}(\ c_i,\ s\ )$.
Logo, como $s \models \wedge \mathcal{C}_{c_j}$, $\textbf{Eval}(\ c_i,\ s\ ) \models \wedge \mathcal{C}_{c_j}$ e portanto,
por (4.16), $\textbf{Eval}(\ c_i,\ s\ ) \models \wedge \mathcal{P}_{c_j}$. Sendo $\textbf{Eval}(\ c_i,\ s\ ) = s$, $s \models \wedge \mathcal{P}_{c_j}$, como pretendido.

□

Corolário 4.14. *(Completude de $\mathcal{T}_4$)*

$$\forall c \in \mathcal{C}nf,\ \text{safe}(\ c\ ) \Rightarrow \models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$$

Demonstração. Suponhamos que $\textit{safe}(\ c\ )$, *i.e.*, suponhamos que $\forall\ s \in \Sigma :\ \textit{safe}(\ c,\ s\ )$.
Queremos mostrar que $\models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$, *i.e.*, $\forall\ s \in \Sigma :\ s \models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$.

Seja $s \in \Sigma$. Pela suposição, sabemos $\textit{safe}(\ c,\ s\ )$.
Logo, pelo Teorema 4.13, segue $s \models \bigwedge \mathcal{C}_c \rightarrow \bigwedge \mathcal{P}_c$, como pretendido. □

Capítulo 5

Implementação de um *Bounded Model Checker*

Este capítulo aborda os aspectos técnicos da implementação da ferramenta de BMC de programas $\mathcal{While}_{\textbf{assert}}$ estendidos com procedimentos, realizada durante este projecto de mestrado.

O desenvolvimento da ferramenta de BMCsw envolveu tecnologias de diversas áreas. Foi necessário construir um sistema para reconhecer sequências de símbolos da linguagem válidas (**palavras** e **palavras reservadas**); construir um sistema para reconhecer a estrutura da linguagem; codificar esta estrutura (que reflete de perto a gramática da linguagem $\mathcal{While}_{\textbf{assert}}$) numa *Domain-Specific Language* (DSL); implementar na linguagem de programação Haskell os algoritmos que caracterizam o método BMCsw, recebendo a codificação na DSL como argumento; e construir um sistema de documentação apropriado para a ferramenta.

Embora a transição entre estes sistemas seja simples, envolvendo transformações não muito complicadas até à geração das fórmulas lógicas, a implementação da ferramenta no seu todo trata-se, efectivamente, de um projecto complexo. Em torno do nosso objectivo está um sistema similar aos sistemas de compilação de programas.

A implementação de uma ferramenta para a linguagem de programação desenvolvida envolve 3 (três) fases de desenvolvimento. Designemos estas fases por front-end, middle-end e back-end.

No front-end está o cerne de toda a computação referente à transformação de uma frase válida da linguagem imperativa base $\mathcal{While}_{\textbf{assert}}$, numa representação da estrutura sintática, útil para raciocinar em termos de geração de condições de verificação. Assim, é necessário reconhecer os construtores da linguagem, sintetizar a informação recolhida numa *Abstract Syntax Tree* (AST) e codificar a AST construída numa string de acordo com as regras da DSL que foi fixada neste projecto.

O cerne do middle-end foca-se na travessia da string produzida pelo front-end, que contém uma representação normalizada do programa $\mathcal{While}_{\textbf{assert}}$. O middle-end é construído na linguagem de programação Haskell. Por um lado, são desenhados tipos de dados apropriados para representar as AST's dos programas $\mathcal{While}_{\textbf{assert}}$. Por outro lado, são implementados os algoritmos das transformações sucessivas que compõem o método BMCsw e conduzem à geração de uma fórmula lógica que representa a validade de um programa $\mathcal{While}_{\textbf{assert}}$.

O back-end do sistema envia para o SMT *solver* escolhido (formato SMT-LIB v2.0) a fórmula lógica produzida pelo middle-end e efectua todo o esforço computacional de representação da informação gerada, e da sua formatação tipográfica em documentação no formato LaTeX e SMT-LIB v2.0.

De seguida, iremos passar pelas tecnologias envolvidas no parsing da linguagem de programação definida no Capítulo 2 e na geração de condições de verificação, pela representação da informação, formatação e processamento dos resultados. As tecnologias usadas no desenvolvimento da ferramenta de BMCsw provêm de diferentes domínios, a saber:

Processamento de Linguagens, envolvendo Lex e Yacc para formalizar computacionalmente a linguagem de programação simples $\mathcal{W}hile_{\textbf{assert}}$, e reconhecer frases válidas desta mesma linguagem.

Tipografia, recorrendo ao sistema tipográfico LaTeX 2_ε para formatar e apresentar os resultados obtidos numa forma gráfica.

Satisfiability Modulo Theories, envolvendo a ferramenta Z3 da Microsoft® Corporation para decidir a satisfazibilidade das condições de verificação geradas.

Sabendo que apenas um conjunto restrito de técnicas de construção de compiladores, produzem tradutores eficientes para uma variedade de linguagens e máquinas, decidimos construir uma ferramenta BMC para verificação de *Software*, com base nos nossos conhecimentos de construção de compiladores, já adquiridos, em contraste com aprender novas tecnologias como é o caso do *AnTLR* [36] e outras ferramentas de geração de compiladores.

Este capítulo está dividido em quatro secções. Na Secção 5.1 são apresentados alguns conceitos de compiladores, compilação e construção da ferramenta de verificação.s:frontEnd Na Secção 5.2 aborda-se o front-end e em particular, na Subsecção 5.2.1 apresentam-se as decisões tomadas para o reconhecimento dos tokens da linguagem e na Subsecção 5.2.2 quais foram as decisões tomadas para a construção das frases da linguagem. Na Secção 5.3 aborda-se o middle-end e a implementação das transformações $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ e $\mathcal{T}_4$. Finalmente, na Secção 5.4 faz-se uma pequena referência a um dos SMT *solvers* existentes no mercado, em particular aquele que foi aplicado durante o desenvolvimento desta tese de mestrado.

Apresenta-se a documentação criada para a ferramenta e explica-se os aspectos da utilização e funcionamento da ferramenta. Aborda-se as ferramentas utilizadas, a instalação da ferramenta em sistemas UNIX, e a apresentação das decisões tomadas ao longo do projecto. Também se refere alguns aspectos de compilação e reconhecimento de linguagens.

5.1 Conceitos

Um compilador interpreta um programa escrito numa linguagem, a *linguagem fonte*, no nosso caso a linguagem dos comandos, e traduz este programa, noutro programa, escrito numa outra linguagem, a *linguagem objectivo*.

Como resultado da aplicação de compiladores a uma linguagem, a codificação presente na representação do reconhecimento efectuada por parte de um compilador, indica se um programa está **bem escrito** na *linguagem fonte*, *i.e.*, se um método automático consegue extrair significado do programa. Por ser um processo automático, garante-se que o resultado está bem escrito, de acordo com as regras da linguagem presentes no compilador.

Existe uma vasta gama de compiladores, tipos de compiladores, tipos de linguagens (fonte e objectivo), e tipos de regras. Regra geral, dependendo da forma como são construidos, ou do intuíto para o qual foram desenvolvidos, os compiladores são classificados como `single-pass`, `multi-pass`, `load-and-go`, `debugging` ou `optimizers` [3]. A compilação requer duas fases distintas, a fase de análise do programa na linguagem original, e a fase de síntese do programa na *linguagem objectivo*.

A parte referente à análise da linguagem, divide o programa em elementos e estrutura-os de forma a tomarem significado numa representação intermédia. A parte referente à síntese, constrói um programa na *linguagem objectivo* de acordo com a representação intermédia que codifica o significado do programa analisado na *linguagem fonte*, interpretado pelo compilador.

Destas duas partes, a síntese é a parte do compilador que, actualmente, requer as técnicas mais especializadas. Muito desenvolvimento foi realizado na organização e desenvolvimento na construção de compiladores, desde o seu aparecimento no início da década de 1950.

Durante a fase de análise da linguagem, as operações expressas pelo programa são determinadas, interpretadas e registadas numa estrutura hierárquica denominada por `árvore de sintaxe`.

Alguns tipos de árvores, e nós usaremos algumas, representam as operações nas raízes e os operandos nas folhas da referida `árvore de sintaxe`. A análise de um programa subdivide-se em 3 (três) partes:

A **análise linear** (Subsecção 5.2.1), onde o programa é analisado como uma sequência de símbolos, que são agrupados de acordo com o significado que geram ao se associarem em grupo. A análise linear, é também chamada de `análise léxica` ou `scanning`;

A **análise de hierarquia** (Subsecção 5.2.2), onde estes agrupamentos são organizados hierarquicamente, de acordo com o significado que possuem associado à sua posição relativa. A análise hierárquica, também chamada de `análise sintática` ou `parsing`, é usualmente expressa por definições recursivas;

E a **análise semântica**, onde um grupo de testes são efectuados para certificar que a ordem pela qual estão posicionados, ou e sequência de evocações, contém algum significado semântico, i.e, se a frase/programa possui algum sentido (Secção 5.3).

Após a análise do código fonte, alguns compiladores, geram uma representação intermédia explícita que deve possuir duas propriedades: ser de fácil construção; e de fácil tradução para a *linguagem objectivo*. Existem diversas formas de representação intermédia.

5.2 Front-end

O front-end do sistema que compõe a ferramenta considera e codifica a interpretação de um programa dotado do significado pretendido pelo programador. O front-end é realizado em duas fases que passamos a descrever.

5.2.1 Análise léxica - Reconhecimento dos tokens da linguagem

Esta secção foca-se na análise léxica da lingugem $\mathcal{W}hile_{\mathbf{assert}}$ através da ferramenta Lex. Esta ferramenta foi utilizada para reconhecer e representar os tokens da linguagem de programação, bem como transformar a estrutura sintática da linguagem numa estrutura normalizada para efeitos de geração da frase na *linguagem objectivo* por parte da ferramenta Yacc. Introduziremos a definição dos *lexemas* da linguagem definida na Secção 2.1.

Lex é uma ferramenta de geração automática de **analisadores léxicos**. A ferramenta foi originalmente descrita formalmente por *Mike Lesk* e *Eric Schmidt*. Escolheu-se esta ferramenta por ser uma ferramenta *freeware* disponível em diversas distribuições dos sistemas operativos `UNIX` e devido à sua facilidade na geração do analisador léxico necessário à ferramenta implementada. Na verdade, Lex é conhecido como o gerador de analisadores léxicos padrão.

Dada uma string, o analisador léxico gera uma sequência de objectos (tokens) representativos dos *lexemas* presentes nas palavras identificadas no ficheiro, de acordo com as especificações fornecidas à ferramenta Lex. A ferramenta Lex baseia-se no reconhecimento de tokens a partir de expressões regulares, pelo que é conveniente explicitar os mesmos.

As *expressões regulares* constituem uma notação para definir linguagens regulares e os lexemas, por serem linguagens regulares, podem ser reconhecidos por *expressões regulares*. As *expressões regulares* estão intimamente ligadas aos *automatos finitos não determinísticos* e podem ser considerados como uma notação alternativa à pesada notação dos *automatos finitos não determinísticos*. As *expressões regulares* são estremamente úteis para reconhecer palavras das liguagens regulares. As expressões regulares tornam-se assim uma importante ferramenta na identificação de tokens.

Seguidamente (Figura 5.1) apresentam-se os tokens essenciais à linguagem $\mathcal{W}hile_{\mathbf{assert}}$, bem como a ligeiras extensões à linguagem que implementamos. A ferramenta `gbmc` implementa a linguagem $\mathcal{W}hile_{\mathbf{assert}}$, com ligeiras extensões. As extenções adoptadas permitem facilitar a programação de comandos (escrever **if** (b) **then** c em vez de **if** (b) **then** c_t **else skip**), a organização do documento devido à implementação de procedimentos e invocação de procedimentos

(introdução do comando **call** (p)), entre outras pequenas extensões criadas para futura investigação.

A linguagem de programação é rica em palavras reservadas, o que torna relativamente simples a desta em termos de tokens. Algumas *expressões regulares* necessárias são sequências bem definidas de caracteres como é o caso das palavras `proc` para indicar a definição de procedimentos, `call` para indicar a invocação de procedimentos, `skip` representando o comando **skip**, `assert` representando a invocação do comando **assert** (b), ":=" representando o construtor do comando $x := e$, o símbolo ";" representando o construtor do comando c_i **;** c_j, `if` e `else` nessessários à construção dos comandos **if** (b) **then** c_t **else** c_f e **if** (b) **then** c, bem como as palavras reservadas `while` e `do` para representar o comando **while** (b) **do** c_w.

```
1  /* command expression tokens */
2  SKIP         "skip"
3  ASSERT       "assert"
4  ASSIGN       ":="
5  COMPOSITION  ";"
6  IF           "if"
7  ELSE         "else"
8  WHILE        "while"
9  DO           "do"
10 CALL         "call"
11 PROC         "proc"
```

Figura 5.1: Tokens: Comandos

As operações aritméticas sobre inteiros da linguagem são expressas e identificadas pelos símbolos presentes na Figura 5.2.

```
1 /* integer expression tokens */
2 I_Plus       "+"
3 I_Minus      "-"
4 I_Times      "*"
5 I_Div        "div"
6 I_Mod        "mod"
```

Figura 5.2: Tokens: Operações aritméticas

Na Tabela 5.3 ilustram-se os tokens referentes a valores e operadores da classe de expressões booleanas. As sequências de caracteres "true" e "false" representam os valores lógicos $\top$ e $\bot$. O caracter "!" representa a negação lógica ($\neg\, b$), "&&" representa a conjunção lógica ($b_1 \wedge\ b_2$), "||" representa a disjunção lógica ($b_1 \vee\ b_2$), e "−>" representa a implicação lógica ($b_1 \rightarrow\ b_2$). Os tokens identificados pelos caracteres "==", "! =", "<", "<=", ">"e ">=" representam respectivamente os comparadores $=$, $\neq$, $<$, $\leq$, $>$ e $\geq$.

```
1  /* boolean expression tokens */
2  B_Top        "true"
3  B_Bottom     "false"
4  B_Not        "!"
5  B_And        "&"
6  B_Or         "|"
7  B_Then       "->"
8  B_Eq         "=="
9  B_Neq        "!="
10 B_Lt         "<"
11 B_Le         "<="
12 B_Gt         ">"
13 B_Ge         ">="
```

Figura 5.3: Tokens: Expressões Booleanas

A Figura 5.4 apresenta classes de tokens auxiliares à idêntifição dos tokens que representarão os números inteiros e as variáveis. O `token` LETTER representará a classe das letras reconhecidas pelo analisador léxico, abrangendo as letras do alfabeto latino, sem acentuação. Uma letra é um de (2×26) elementos. O `token` DIGIT representará a classe dos números reconhecidas pelo analisador léxico, abrangendo os caracteres 0,1,2,3,4,5,6,7,8 ou 9. Um dígito é um de 10 elementos. A detecção de caracteres em branco, representado pelo token WSPACES é uma classe de elementos, *i.e.*, a classe dos espaços, tabulações, carriage returns, line feeds e newlines.

```
1 /* non reservaded words (identifiers, numbers, etc.)*/
2 LETTER       [a-zA-Z]
3 DIGIT        [0-9]
4 /* white spaces */
5 WSPACES      [ \t\r\f\n]
```

Figura 5.4: Classes simbólicas: White Spaces, letras e digitos

Definidas as expressões referentes às palavras reservadas, definem-se agora as expressões regulares que correspondem aos números e às variáveis. Um número é uma sequência de dígitos com uma formatação apropriada e uma variável é um identificador próprio. Assim, um número inteiro é representado pelo token NUMBER, que por sua vez representa uma sequência finita de 1 ou mais dígitos. A definição usual de uma variável é representada pelo token VARIABLE, que por sua vez representa um caracter que pode ser uma letra (LETTER) ou um `underscore`, seguido de uma sequência finita de zero ou mais letras, `underscores`, digitos ou pontos.

```
1 NUMBER       {DIGIT}+
2 VARIABLE     [a-zA-Z_][a-zA-Z_0-9.]*
```

Figura 5.5: Tokens: Números inteiros e identificadores de variáveis

5.2.2 Análise hierárquica

Esta secção foca-se na análise hierárquica da lingugem $\mathcal{While}_{\mathbf{assert}}$ através da ferramenta Yacc. Esta ferramenta foi utilizada para exprimir a ordem pela qual os tokens devem ser organizados e estruturados por forma a tomarem significado na linguagem que definimos. O Yacc vai produzir uma AST, representada numa DSL. A ferramenta Yacc constrói uma representação textual do programa numa AST. A estrutura desta AST é uma implementada através de uma pequena DSL inspirada nas estruturas de dados nativas do Haskell, e permite a comunicação dos dois

sistemas. Uma DSL é uma linguagem de programação dedicada à representação de um problema em particular. O uso do Lex e Yacc, embora dispendioso, permite funcionalidades que seriam complexas de implementar em Haskell com os mesmo recursos.

O Yacc é uma ferramenta genérica de formalização da estrutura de programas, vistos como uma frase de uma linguagem. Especificada a estrutura da linguagem, é construído um compilador para a linguagem (no nosso caso um tradutor). A ferramenta Yacc codifica uma especificação em rotinas que processam frases de uma linguagem. Especificada a estrutura da linguagem de comandos, recorremos neste projecto ao Yacc para construir uma **Gramática Tradutora**. Esta tradução permite abstrair os problemas inerentes ao reconhecimento da linguagem, e focar nos problemas centrais deste projecto (geração da fórmula da *lógica de primeira ordem*).

Uma AST é uma representação em árvore da **estrutura sintática abstracta** de um **programa** escrito numa **linguagem de programação**. O objectivo desta fase centrou-se em mecanizar a representação de qualquer programa na linguagem de comandos numa *árvores de sintaxe abstracta* na linguagem Haskell.

A riqueza a nível de tokens permite que a linguagem não seja rígida no sentido estrutural, permitindo fazer definições em qualquer parte do documento, contrastando com linguagens como o C. A título de exemplo, na Figura 5.6 ilustra-se a estrutura lógica que os programas escritos na linguagem C têm que seguir. A ordem das declarações é relevante, sendo boa prática a explicitação dos includes, das definições iniciais, das declarações de variáveis e das funções por ordem sequencial de invocação.

```
1 <includes>
2 <defines>
3 <variable declarations>
4 <function declarations>
```

Figura 5.6: Representação simbólica da estrutura em C

Como veremos mais à frente, na ferramenta gbmc , determinamos que cada ficheiro deve ser constituído por uma sequência "não-ordenada" de procedimentos, cabendo ao middle-end a interpretação do mesmo. Sendo cada procedimento uma peça de *software*, a ferramenta validará um dado procedimento, sendo as dependências calculadas pelo próprio método através de um pré-processamento dos programas. Esta opção foi tomada por se tratar de um caso de estudo e permitir um maior grau de liberdade na programação, facilitando a construção pretendida por parte do programador e permitindo generalidade suficiente para possibilitar as modificações que um projecto académico necessita. Cabe ao programador a implementação das boas práticas de programação nos seus documentos.

Por ser uma linguagem de programação simples, o reconhecimento dos tokens da linguagem torna-se relativamente simples. A linguagem não possui construtores complexos, como é o caso do aninhamento de blocos de código e a detecção da associatividade entre operadores. É maioritariamente definida à custa de palavras reservadas. Recorrendo à notação *Backus Naur Form* (BNF), apresentamos na Figura 5.7 a gramática independente de contexto que define a representação das expressões inteiras.

```
/* integer expression grammar */
exp_int
        : NUMBER
            {sprintf(buffer,"(Number %s)",$1);          $$=strdup(buffer);}

        | VARIABLE
            {sprintf(buffer,"(Variable %s)",$1);        $$=strdup(buffer);}/*(d)*/

        | VARIABLE '[' exp_int ']'
            {sprintf(buffer,"(Array %s %s)",$1,$3);     $$=strdup(buffer);}/*(a)*/

        | VARIABLE arguments
            {sprintf(buffer,"(Function %s %s)",$1,$2);  $$=strdup(buffer);}

        | exp_int I_Plus  exp_int
            {sprintf(buffer,"(I_Plus%s%s)",$1,$3);      $$=strdup(buffer);}

        | exp_int I_Minus exp_int
            {sprintf(buffer,"(I_Minus%s%s)",$1,$3);     $$=strdup(buffer);}

        | exp_int I_Times exp_int
            {sprintf(buffer,"(I_Times%s%s)",$1,$3);     $$=strdup(buffer);}

        | exp_int I_Div   exp_int
            {sprintf(buffer,"(I_Div%s%s)",$1,$3);       $$=strdup(buffer);}

        | exp_int I_Mod   exp_int
            {sprintf(buffer,"(I_Mod%s%s)",$1,$3);       $$=strdup(buffer);}
        ;
```

Figura 5.7: Expressões de Inteiros

As produções gramaticais na Figura 5.7 indicam que uma expressão inteira pode ser um número, uma variável, ou uma operação aritmética sobre inteiros em notação `infixe`. Note-se que permitimos que uma variável tome 3 (três) comportamentos distintos. Embora não estudado, a linguagem permite a um identificador comportar-se como uma *variável*, `array` ou *função*. Uma variável seguida do caracter '[', de uma expressão inteira e do caracter ']', indica um acesso a uma posição de uma variável. Uma variável seguida de uma lista de argumentos separados por ',' e delimitados pelos símbolos '(' e ')' representa a aplicação de funções aos seus argumentos. Os argumentos de uma função são sempre explicitados, mesmo quando não há argumentos; os símbolos '(' e ')' expressam a aplicação de uma função. Note-se que poderíamos agrupar as produções referentes às variáveis, numa só produção, tornando os argumentos de funções e os acessos à memória das variáveis, opcionais. Contudo isso traria uma complexidade superior ao tratamento e representação das variáveis na AST transportada para o Haskell. Optámos por esta aproximação à Definição 2.2. A gramática implementada permite a aplicação de funções e `arrays` mas como não pretendíamos estudar funções nem acessos a memória, a linguagem utilizada é mais restrita.

```
arguments
        : '(' exp_int_list ')'
            {sprintf(buffer,"[%s]",$2);                 $$=strdup(buffer);}/*(b)*/
        | '(' ')'
            {sprintf(buffer,"[]");                      $$=strdup(buffer);}/*(c)*/
        ;
```

Figura 5.8: Argumentos dos identificadores segundo a filosofia aplicacional (λ-*calculus*)

(a) A 3ª produção da Figura 5.7, permite identificar uma referência a uma posição de um array, dada pela expressão.

(b) e (c) As produções da Figura 5.8, permite identificar a invocação de uma função com uma lista de expressões.

(d) A 3ª produção da Figura 5.7, permite identificar uma variável.

```
1 exp_int_list
2         : exp_int ',' exp_int_list
3             {sprintf(buffer,"%s,%s",$1,$3);             $$=strdup(buffer);}
4         | exp_int
5             {sprintf(buffer,"%s",$1);                   $$=strdup(buffer);}
6         ;
```

Figura 5.9: Definição recursiva à direita de uma lista não vazia de expressões inteiras

A Figura 5.9 representa uma lista não vazia de expressões de inteiros, separados pelo token ";". A lista de expressões inteiras aplica uma definição recursiva à direita.

As produções da gramática, presentes na Figura 5.10, reconhecem os tokens B_Top e B_Bottom, a negação por B_Not seguido de uma expressão booleana, os operadores lógicos B_And, B_Or e B_Then em notação `infix`, e as operações de comparação B_Eq, B_Neq, B_Lt, B_Le, B_Gt e B_Ge também em notação `infix`.

```
 1 /* boolean expression grammar */
 2 exp_bool
 3         : B_Top
 4             {sprintf(buffer,"(B_Top)");                  $$=strdup(buffer);}
 5
 6         | B_Bottom
 7             {sprintf(buffer,"(B_Bottom)");               $$=strdup(buffer);}
 8
 9         | B_Not exp_bool
10             {sprintf(buffer,"(B_Not%s)"    , $2);        $$=strdup(buffer);}
11
12         | exp_bool B_And  exp_bool
13             {sprintf(buffer,"(B_And%s%s)"  , $1,$3);     $$=strdup(buffer);}
14
15         | exp_bool B_Or   exp_bool
16             {sprintf(buffer,"(B_Or%s%s)"   , $1,$3);     $$=strdup(buffer);}
17
18         | exp_bool B_Then exp_bool
19             {sprintf(buffer,"(B_Then%s%s)", $1,$3);      $$=strdup(buffer);}
20
21         | exp_int  B_Eq   exp_int
22             {sprintf(buffer,"(B_Eq%s%s)"   , $1,$3);     $$=strdup(buffer);}
23
24         | exp_int  B_Neq  exp_int
25             {sprintf(buffer,"(B_Neq%s%s)"  , $1,$3);     $$=strdup(buffer);}
26
27         | exp_int  B_Lt   exp_int
28             {sprintf(buffer,"(B_Lt%s%s)"   , $1,$3);     $$=strdup(buffer);}
29
30         | exp_int  B_Le   exp_int
31             {sprintf(buffer,"(B_Le%s%s)"   , $1,$3);     $$=strdup(buffer);}
32
33         | exp_int  B_Gt   exp_int
34             {sprintf(buffer,"(B_Gt%s%s)"   , $1,$3);     $$=strdup(buffer);}
35
36         | exp_int  B_Ge   exp_int
37             {sprintf(buffer,"(B_Ge%s%s)"   , $1,$3);     $$=strdup(buffer);}
38         ;
```

Figura 5.10: Expressões Booleanas

As produções da gramática relativas aos comandos são semelhantes às definidas na Subsecção 2.1.3. O comando **skip** é reconhecido pelo token SKIP, o comando **assert** (b) é reconhecido pelo token ASSERT, seguido de uma expressão booleana delimitada pelos caracteres '(' e ')'.

O comando $x := e$ é reconhecido pelo posicionamento do token ASSIGN, após uma variável (reconhecida pelo token VARIABLE) e antes de uma expressão inteira.

O comando c_i **;** c_j é reconhecido pelo token COMPOSITION, relacionando dois comandos, em notação `infix`. Note-se que não se define a associatividade deste comando.

O comando **if** (b) **then** c_t **else** c_f é reconhecido pelo token IF seguido de uma expressão booleana delimitada pelos caracteres '(' e ')', um comando delimitado pelos caracteres '{' e '}', a que pretendemos dar significado, e a possível construção idêntificada pelo token ELSE. Note-se que caso não seja detectada a explicitação do ramo ELSE, este é implicitamnente construído na AST com o comando **skip**, conforme a Convenção 2.

O comando **while** (b) **do** c_w é reconhecido pelo token WHILE, seguido de uma expressão booleana delimitada pelos caracteres '(' e ')', o token DO e um comando delimitado pelos caracteres '{' e '}'. As frases da linguagem, expressa por esta gramática, exigem a explicitação de blocos gramaticais, a fim de tornar o programa perceptível ao programador, tentando tornar a *sintaxe* concreta da linguagem $\mathcal{W}hile_{\textbf{assert}}$ numa *sintaxe* semelhante à utilizada pelas linguagens imperativas comuns.

```
1  /* commands expression grammar */
2  command
3          : SKIP
4             {sprintf(buffer,"(Skip)");                  $$=strdup(buffer);}
5
6          | ASSERT '(' exp_bool ')'
7             {sprintf(buffer,"(Assert %s)", $3);         $$=strdup(buffer);}
8
9          | VARIABLE ASSIGN exp_int
10            {sprintf(buffer,"(Assign(%s)%s)", $1, $3);  $$=strdup(buffer);}
11
12         | command COMPOSITION command
13            {sprintf(buffer,"(Comp%s%s)", $1, $3);      $$=strdup(buffer);}
14
15         | IF   '(' exp_bool ')'   '{' command '}' if_else
16            {sprintf(buffer,"(If%s%s%s)", $3, $6, $8 ); $$=strdup(buffer);}
17
18         | WHILE '(' exp_bool ')' DO '{' command '}'
19            {sprintf(buffer,"(While%s%s)", $3, $7);     $$=strdup(buffer);}
20
21         | CALL VARIABLE
22            {sprintf(buffer,"(Call %s)", $2);           $$=strdup(buffer);}
23         ;
24
25 if_else
26         : ELSE '{' command '}'                        {$$=$3;}
27         |                                             {$$="(Skip)";}
28         ;
```

Figura 5.11: Comandos

As produções presentes na Figura 5.11 identificam a estrutura que reconhece um comando da linguagem. Note-se que a gramática permite ao programador exprimir construções if-then sem else. Contudo a linguagem de programação $\mathcal{W}hile_{\textbf{assert}}$, expressa pela classe $\mathcal{C}ommand$, não permite tal construção. A solução relativa a este problema é trivial visto que a construção if-then pode ser reduzida à construção if-then-else considerando o comando skip na construção else.

```
1 /* procedures */
2 procedure
3     : PROC '(' VARIABLE ')' '{' command '}'
4             {sprintf(buffer,"(Proc %s %s)", $3, $6);$$=strdup(buffer);}
5     ;
```

Figura 5.12: Procedimentos

As produções presentes na Figura 5.12 ilustram a definição de um procedimento como uma sequência de tokens, nomeadamente a palavra reservada `proc` seguida de uma variável que identifica o procedimento, o token "=" (açúcar sintático na programação) e o corpo do procedimento, resumido a um comando da linguagem.

Um **procedimento** é identificado pelo token PROC seguido por uma variável que identifica o procedimento/comando e um comando na linguagem definida. Temos assim uma sequência de comandos num ficheiro, identificados como procedimentos.

```
1 /* file root */
2 root
3         : root procedure        {printf("%s\n",$2);}
4         | procedure             {printf("%s\n",$1);}
5         |                       {;}
6         ;
```

Figura 5.13: Raíz da gramática

Finalmente, e para concluir os pontos importantes da estrutura dos programas, as produções presentes na Figura 5.13 ilustram a meta-estrutura dos programas que permite que os procedimentos possam ser declarados sem ordem específica. Assim, a raiz da AST é uma construção de declarações, *i.e.*, o documento é seccionado e visto como uma lista de declarações que podem apenas ser procedimentos. Após a construção da AST cabe ao middle-end a interpretação da mesma.

Neste projecto, um ficheiro com procedimentos na linguagem definida, é representado por uma AST, cuja raíz pode representar uma sequência vazia, um procedimento, ou uma sequência de procedimentos.

5.3 Middle-end

Haskell é uma linguagem de programação, em particular, é uma linguagem puramente funcional, polimórfica de tipos estáticos. A linguagem foi dedicada a *Haskell Brooks Curry*, cujo trabalho realizado no campo da *Lógica Matemática* serve de fundamento às linguagens funcionais. Haskell baseia-se no sistema formal do λ-*calculus*.

O middle-end foi implementado na linguagem de programação Haskell. O middle-end recebe uma representação intermédia gerada pelo front-end, a AST que representa programas na linguagem dos comandos e normaliza os ficheiros, obtendo a estrutura dos programas avaliados. Cabe ao middle-end, onde estão codificadas as 4 (quatro) transformações que constituem o método BMCsw, extrair um modelo do programa/software e codificá-lo de forma automática numa expressão lógica para posterior validação.

5.3.1 Árvore de síntaxe abstracta dos comandos

Baseados na especificação sintática das tabelas 2.1, 2.2 e 2.3, foram formalizadas na linguagem

Haskell as estruturas de dados que seguidamente apresentamos. A interligação destas estruturas formaliza a AST que é percorrida para gerar as transformações do médodo BMC para verificação de *Software* e consequente condição de validação. Define também a DSL que o middle-end aceita como input.

```
1  {-- Structure: ExpInt --}
2  data ExpInt      = Number     Int
3                   | Variable   Int String
4                   | Array      Int String ExpInt
5                   | Function   String [ExpInt]
6                   | Neg        ExpInt
7                   | I_Plus     ExpInt ExpInt
8                   | I_Minus    ExpInt ExpInt
9                   | I_Times    ExpInt ExpInt
10                  | I_Div      ExpInt ExpInt
11                  | I_Mod      ExpInt ExpInt
12      deriving Eq
```

Figura 5.14: Representação Haskell da Classe de Expressões Inteiras

Na representação da classe de expressões inteiras definiram-se 5 (cinco) operações binárias sobre inteiros, uma unária e a representação dos números e variáveis. Estes são representados na sua forma textual, sendo que no âmbito deste projecto não é necessário atribuir significado para o cálculo das expressões.

Na representação da classe de expressões booleanas definiram-se 3 (três) operações binárias sobre os booleanos, 6 (seis) operações binárias de comparação sobre os inteiros, uma operação unária sobre booleanos e a representação de entidades de verdade ($\top$ e $\bot$), conforme apresentado na Figura 5.15. Procura-se assim simbolizar os valores de verdade $\top$, $\bot$; as operações de negação, conjunção e disjunção lógicas; as operações de comparação: **igualdade**, **desigualdade**, **menor ou igual a**, **menor que**, **maior que** e **maior ou igual a**, sobre os inteiros.

```
1  {-- Structure: ExpBool --}
2  data ExpBool     = B_Top
3                   | B_Bottom
4                   | B_Not    ExpBool
5                   | B_And    ExpBool ExpBool
6                   | B_Or     ExpBool ExpBool
7                   | B_Then   ExpBool ExpBool
8                   | B_Eq     ExpInt  ExpInt
9                   | B_Neq    ExpInt  ExpInt
10                  | B_Le     ExpInt  ExpInt
11                  | B_Lt     ExpInt  ExpInt
12                  | B_Gt     ExpInt  ExpInt
13                  | B_Ge     ExpInt  ExpInt
14      deriving Eq
```

Figura 5.15: Representação Haskell da Classe de Expressões Booleanas

A Figura 5.16 exprime a representação da classe dos *Command* na linguagem Haskell. Um Skip é uma construção unitária; um Assert identifica uma expressão booleana (que pretendemos atribuir significado); um Assignment relaciona uma variável a uma expressão inteira; a composição relaciona 2 (dois) comandos da linguagem; um If relaciona 2 (dois) comandos da linguagem a uma condição booleana; enquanto que um While associa uma condição booleana e um comando.

A invocação de procedimentos é representada pelo nodo CALL e respectiva identificação do procedimento. Este comando foi introduzido na linguagem para permitir a segmentação do código. O pré-processamento do código irá estender o código, criando um comando único onde todas as invocações ao comando **call** (p) são substituídas pelo seu comando correspondente. Assumem-se

boas práticas de segmentação na programação, por parte do programador, evitando invocações recursivas e consequentes erros de memória.

```
{-- Structure: Command --}
data Command     = Skip
                 | Assert ExpBool
                 | Assign Int String  ExpInt
                 | Comp               Command Command
                 | If     ExpBool     Command Command
                 | While  ExpBool     Command
                 | Call   String
    deriving Eq
```

Figura 5.16: Representação Haskell da Classe dos Comandos

5.3.2 Transformações

Iniciamos esta subsecção com a eliminação dos procedimentos. Embora não se tenha estudado esta transformação, ela é essencial devido às decisões tomadas nesta implementação. A aplicação do método BMC estudado, exige a construção de um comando $\mathcal{W}hile_{\mathbf{assert}}$. A função `step0` implementada realiza esta tarefa, substituindo as invocações de procedimentos pelo comando respectivo. As funções `step1`, `step2`, `step3` e `step4` apresentadas nas figuras 5.18, 5.19, 5.20 e 5.21, implementam as transformações $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ e $\mathcal{T}_4$ respectivamente.

```
step0::[Procedure]->Command->Command
step0 l w@(Skip)          = (Skip)
step0 l w@(Assert b)      = (Assert b)
step0 l w@(Assign i v e) = (Assign i v e)
step0 l w@(Comp c1 c2)    = (Comp (step0 l c1) (step0 l c2))
step0 l w@(If b c1 c2)    = (If b (step0 l c1) (step0 l c2))
step0 l w@(While b c)     = (While  b           (step0 l c))
step0 l w@(Call  pName)
        |(length procedures == 1) =
                (step0 l (head procedures))
        |(length procedures < 1) =
                error("[ERROR]: procedure ["++pName++"] NOT found")
        |(length procedures > 1) =
                error("[ERROR]: multiple decalarations of procedure ["++pName++"]")
    where procedures = (getProcedures pName l)
```

Figura 5.17: Pré-processamento do comando call

Apresenta-se seguidamente pequenos extractos de código com as implementações, na linguagem Haskell, das quatro transformações ($\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ e $\mathcal{T}_4$) apresentadas na Secção 3.3.

```
1  step1::Int->Command->Command
2  step1 n w@(Skip)          = (Skip)
3  step1 n w@(Assert b)      = (Assert b)
4  step1 n w@(Assign i v e) = (Assign i v e)
5  step1 n w@(Comp c1 c2)    = (Comp (step1 n c1) (step1 n c2))
6  step1 n w@(If b c1 c2)    = (If b (step1 n c1) (step1 n c2))
7  step1 n w@(While b c)     = (uwd (n,n) w)
8      where
9          uwd (n,0) w@(While b cw) = (Assert (B_Bottom));
10         uwd (n,m) w@(While b cw) = (If b (Comp (step1 n cw) (uwd (n,m-1) w)) Skip)
```

Figura 5.18: Transformação $\mathcal{T}_1$ (implementação)

```
1  step2::Command->Command
2  step2 c = cmd
3      where (tb,cmd) = dSAF [] c
4
5
6  type Memory = [(String,Int)]
7  dSAF::Memory->Command->(Memory,Command)
8  dSAF lIn w@(Skip)            = (lIn, (Skip))
9  dSAF lIn w@(Assert bIn)      = (lIn, (Assert bOut ))
10     where
11         bOut = (substituteExp substituteExpBool bIn lIn)    -- for all lIn, update
               values
12
13 dSAF lIn w@(Assign i x eIn) = (lOut, (Assign n x eOut ))
14     where
15         eOut = (substituteExp substituteExpInt eIn lIn);    -- for all lIn, update
               values
16         lOut = update(lIn,x);                               -- apply succ function
17         n    = (get x lOut) {-- get value from update --}
18
19 dSAF lIn w@(Comp c1 c2)      = (lOut, (Comp cmd1 cmd2))
20     where
21         (lAux,cmd1) = (dSAF lIn c1);
22         (lOut,cmd2) = (dSAF lAux c2)
23
24 dSAF lIn w@(If b c1 c2)   = (lIn,
25         (If (boolean)                                   -- updated boolean
26             (foldl (\a b -> (Comp a b)) cmd1 lt)        -- join true
27             (foldl (\a b -> (Comp a b)) cmd2 lf)  )     -- join false
28     )
29     where
30         (boolean)      = (substituteExp substituteExpBool b lIn);
31         (lTrue,cmd1)   = (dSAF lIn c1);
32         (lFalse,cmd2)  = (dSAF lIn c2);
33         (lt,lf)        = unzip(compile lTrue lFalse);
34         (lOut)         = (compose lTrue lFalse)         -- builds new table.
               Highest succ, win factor.
```

Figura 5.19: Transformação $\mathcal{T}_2$ (implementação)

```
1 step3::Command -> ExpBool -> CNF
2 step3 p@(Skip)           phi = (CNF_IF (phi) (Skip))
3 step3 p@(Assert b)       phi = (CNF_IF (phi) (Assert b))
4 step3 p@(Assign i str e) phi = (CNF_IF (phi) (Assign i str e))
5 step3 p@(Comp ci cj)     phi = (CNF_Comp (step3 ci phi) (step3 cj phi))
6 step3 p@(If b ct cf)     phi = (CNF_Comp
7         (step3 ct (B_And phi b))
8         (step3 cf (B_And phi (B_Not b)))
9     )
```

Figura 5.20: Transformação $\mathcal{T}_3$ (implementação)

```
1 step4::CNF->([ExpBool],[ExpBool])
2 step4 c = ((lefts.skim) c, (rights.skim) c)
3     where
4         skim(CNF_IF phi p@(Skip)           ) = [];
5         skim(CNF_IF phi p@(Assert b)       ) = [(Right (B_Then phi b) )]; -- array
              and funcion normalization ignored
6         skim(CNF_IF phi p@(Assign i str e)) = [(Left  (B_Then phi (B_Eq (Variable i
               str) e ) ) )];
7         skim(CNF_Comp   ci cj              ) = (skim ci)++(skim cj)
```

Figura 5.21: Transformação $\mathcal{T}_4$ (implementação)

5.4 Back-end: Validação das fórmulas com recurso a um SMT

Como explicado no Capítulo 3, o método BMCsw assenta na geração de uma fórmula lógica ($\bigwedge \mathcal{C} \rightarrow \bigwedge \mathcal{P}$) cuja validade garante a correcção do programa em questão, *i.e.*, garante que quando um qualquer comando **assert** (b) do programa for atingido, a consequente condição booleana é válida. Para testar a validade de fórmulas lógicas utilizamos um SMT *solver*. Os SMT's procuram a existência de uma valoração que satisfaça a fórmula apresentada. Assim, para sabermos se uma fórmula ϕ é válida, basta considerarmos a fórmula $\neg\phi$ e recorrer a um SAT para decidir se existe alguma valoração que satisfaça $\neg\phi$. Se não existir, podemos então considerar (em lógica clássica) que ϕ é uma fórmula válida. Caso haja valoração que satisfaça $\neg\phi$, então, por redução ao absurdo, ϕ não é uma fórmula válida.

No nosso caso o SMT *solver* usado foi o Z3. O Z3 é uma ferramenta de alta performance desenvolvida pela Microsoft® Research e lida com lógicas de primeira ordem tipadas. Além de trabalhar com quantificadores, reconhece as seguintes teorias lógicas por defeito:

- *"equality over free (aka uninterpreted) function"* e *"predicate symbols"*;
- *"real and integer arithmetic (with limited support for non-linear arithmetic)"*,
- *"bit-vectors"*,
- *"arrays"*,
- *"tuple/records/enumeration types"*
- *"algebraic (recursive) data-types"*.

A ferramenta Z3 fornece funcionalidades sobre a aritmética real ($\mathbb{R}$) e inteira ($\mathbb{Z}$), `bit-arrays` de tamanho fixo, `arrays` extensionais, *uninterpreted functions*, e quantificadores.

O Z3 tem integrado um conjunto de ferramentas da **Microsoft Research**. As ferramentas de análise, verificação e teste de programas, incluem: Spec#/Boogie, Pex, Yogi, Vigilante, SLAM, F7,

SAGE, VS3, FORMULA, e HAVOC. Além do formato nativo de *sintaxe*, o Z3 reconhece os formatos SMT-LIB, SMT-LIB *versão 2* e Simplify.

Além da tarefa fundamental de representar a informação necessária para um SAT *solver* moderno efectuar a decisão do problema de satifazibilidade inerente ao método BMC, o backend gera também documentação no formato LaTeX, relatando os resultados das transformações $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ e $\mathcal{T}_4$. Os aspectos mais técnicos da ferramenta (compilação, instalação e utilização) encontram-se descritos no Apêndice A.1.

Capítulo 6

Conclusões e trabalho futuro

Conclusões

O *Bounded Model Checking* (BMC) é uma técnica aplicada na verificação de sistemas computacionais, caracterizada por considerar apenas uma versão limitada do sistema computacional em causa. Essencialmente o BMC restringe o comprimento dos traços de execução que analisa, o que, ao nível da programação imperativa com *whiles*, se reflete na substituição de ciclos *while* por sequências de comandos if then else, e dá origem a uma fórmula lógica cuja satisfazibilidade pode ser tratada com um SAT *solver*. Sendo uma ferramenta de verificação, é fundamental ter garantias acerca da validade da própria técnica BMC. Apesar de ser um método muito difundido pela indústria na verificação formal de sistemas, em especial na área de *hardware*, na altura da realização desta tese de mestrado, constatou-se que a documentação acerca da validade da técnica BMCsw no contexto de programação imperativa com *whiles* e *asserts*, além de escassa, assentava em ideias intuitivas, não nos tendo sido possível encontrar trabalhos cujo objectivo se focasse num estudo rigoroso da validade do método neste contexto.

Esta tese de mestrado contribui academicamente com o desenvolvimento de uma abordagem para a formalização do método BMC no contexto de programação imperativa com *whiles* e *asserts*. Em particular, nesta tese de mestrado estabelece-se a validade de uma das transformações usadas pelo BMC neste contexto. Designadamente, no Capítulo 4, prova-se a correcção e a completude da transformação $\mathcal{T}_4$, que mapeia programas *Conditional Normal Form* (CNF) em fórmulas lógicas. Estes resultados exigiram a definição de conceitos adequados, como por exemplo o caso da função **Eval**, bem como a identificação de diversas propriedades acerca de programas CNF e de programas SA (algumas mais naturais, outras não tão óbvias). A demonstração destas propriedades, em alguns casos, atingiu uma certa complexidade e, em geral, as demonstrações revelaram a necessidade de combinar diversos resultados acerca de pequenos detalhes.

Apesar da linguagem $\mathcal{W}hile$ ser alvo de muitos estudos, o enriquecimento da linguagem $\mathcal{W}hile$ com *asserts* e com um estado de *erro*, parece ser um tema pouco estudado. De facto, não nos foi possível encontrar nenhuma fonte de onde pudessemos retirar uma semântica operacional para a linguagem $\mathcal{W}hile_{\mathbf{assert}}$. Deste modo o trabalho desenvolvido no Capítulo 2 em torno do comando da linguagem $\mathcal{W}hile_{\mathbf{assert}}$, não levantando dificuldades especiais, é também uma das contribuições desta tese de mestrado.

Esta tese de mestrado contribuiu ainda com o desenvolvimento de uma ferramenta de verificação de programas $\mathcal{W}hile_{\mathbf{assert}}$, conforme descrito no Capítulo 5. A ferramenta permite escrever programas $\mathcal{W}hile_{\mathbf{assert}}$ de forma modular, e recorre ao SMT *solver* Z3 para testar a validade da fórmula lógica produzida pela sequência de transformações que compõem o BMC, gerando documentação dos vários passos em formato LaTeX.

Como consequência das diversas experiências com a ferramenta desenvolvida, constatamos que a aplicação do método BMC, com *bounds*/limites elevados, no desenrolar dos ciclos *while*, produz um crescimento exponencial não só ao nível do programa sem ciclos que é gerado, mas também ao nível do tamanho da fórmula lógica gerada.

Trabalho futuro

A sequência natural para esta tese de mestrado seria a de desenvolver um estudo das propriedades das transformações $\mathcal{T}_1$, $\mathcal{T}_2$ e $\mathcal{T}_3$ análogo ao desenvolvido para a transformação $\mathcal{T}_4$. No que se refere à transformação $\mathcal{T}_3$, durante os trabalhos desta tese de mestrado, foram realizados alguns pequenos exercícios que pareciam apontar no sentido de ser possível mostrar, de uma forma não muito complexa, que dado c, um comando SA, $\mathcal{T}_3$ (c) é um programa equivalente a c. Em relação à transformação $\mathcal{T}_2$ é de esperar também que esta produza um programa SA equivalente ao programa (já sem comandos **while** (b) **do** c_w) que lhe for passado; contudo, a justificação desta equivalência parece uma questão mais complexa. Já no que se refere à transformação $\mathcal{T}_1$, o programa sem *whiles* produzido, não será necessariamente equivalente ao programa dado, visto que o *bound* usado na aproximação de *whiles* por sequências de *ifs* poderá não ser suficientemente grande (conforme discutido no Capítulo 3). No entanto, é de esperar que da correcção do programa gerado pela transformação $\mathcal{T}_1$ se possa inferir a correcção do programa original, uma vez que em $\mathcal{T}_1$ são adicionadas *unwinding assertions* (conforme discutido na Subsecção 3.3.1).

Tendo em atenção o detalhe apresentado nas demonstrações dos diversos resultados que conduzem aos teoremas de correcção e completude da transformação $\mathcal{T}_4$, a formalização destes resultados numa ferramenta de prova assistida por computador como o Coq [28], conceptualmente, não deveria constituir uma tarefa muito árdua e forneceria uma garantia adicional sobre este passo do método BMC. Naturalmente, seria então desejável proceder ao mesmo tipo de formalização no que se refere às restantes transformações.

Na linguagem $\mathcal{W}hile_{\mathbf{assert}}$ que estudamos, existe apenas um tipo de erro, que corresponde à não validação de um dos *asserts* do programa (que em alguns casos, poderá ser proveniente de uma *unwinding assertion*). Seria interessante considerar extensões à linguagem $\mathcal{W}hile_{\mathbf{assert}}$, com outras possíveis fontes de erros, como por exemplo, as excepções que resultam da avaliação de expressões, conforme discutido na Subsecção 2.2.2.

A *Lógica de Hoare* é uma ferramenta muito utilizada na verificação de programas $\mathcal{W}hile$. Assim, numa outra direcção, seria interessante estudar extensões da *Lógica de Hoare* para a linguagem $\mathcal{W}hile_{\mathbf{assert}}$ e procurar relações entre a verificação de programas $\mathcal{W}hile_{\mathbf{assert}}$ assente em *Lógica de Hoare* e a verificação de programas $\mathcal{W}hile_{\mathbf{assert}}$ assente no método BMC.

Considerações pessoais

A título pessoal, o mestrando aprofundou os seus conhecimentos sobre ferramentas e métodos de verificação de *software*; aprofundou também os seus conhecimentos sobre semântica operacional, demonstrações dedutivas e indutivas; e aprofundou os seus conhecimentos sobre as linguagens de programação (Noções de complexidade: Expressões Regulares e Gramáticas Regulares) bem como as técnicas de compilação existentes (sintaxe e semântica).

Devido ao seu background o mestrando necessitou de fortalecer os seus conhecimentos e realizar um esforço de aprendizagem no sentido de estruturar o raciocínio lógico em demonstrações matemáticas na aplicação de formalismos lógicos.

Bibliografia

[1] Yael Abarbanel, Ilan Beer, Leonid Gluhovsky, Sharon Keidar, and Yaron Wolfsthal. Focs – automatic generation of simulation checkers from formal specifications. In E.Allen Emerson and AravindaPrasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 538–542. Springer Berlin Heidelberg, 2000. 1

[2] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on sat-solvers. In *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, TACAS '00, pages 411–425, London, UK, UK, 2000. Springer-Verlag. 32

[3] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977. 60

[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag. 31

[5] Armin Biere, Alessandro Cimatti, Edmund Melson Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58, 2003. 30, 31

[6] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009. 2, 32, 33

[7] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986. 31

[8] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142 – 170, 1992. 31

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986. 1, 30

[10] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003. 2

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. 2, 32

[12] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *In Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003. 32

[13] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag. 30

[14] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999. 1, 30

[15] Edmund Melson Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. 30, 31

[16] Edmund Melson Clarke and Ernest Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Lecture Notes in Computer Science*, page 52–71, 1981. 30

[17] Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2006. 34

[18] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 436–453, London, UK, UK, 2001. Springer-Verlag. 33

[19] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *ICCAD*, pages 126–129, 1990. 31

[20] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer Aided Verification, 2nd International Workshop, CAV 90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*, volume 531 of *Lecture Notes in Computer Science*, pages 23–32. Springer, 1990. 31

[21] David W. Currie, Alan J. Hu, and et al. Automatic formal verification of dsp software. In *IN 37TH ACM/IEEE DESIGN AUTOMATION CONFERENCE*, pages 130–135. ACM Press, 2000. 33

[22] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011. 2

[23] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008. 32, 33

[24] Harry Foster. Assertion-based verification: Industry myths to realities (invited tutorial). In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 5–10. Springer Berlin Heidelberg, 2008. 1

[25] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. *Handbook of Knowledge Representation*, chapter Satisability Solvers, pages 89 – 134. Elsevier Science, San Diego, USA, 2007. 2

[26] J. Harrison. Formal verification at intel. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 45–54, 2003. 1

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. 1

[28] Informatiques Mathématiques Inria. Welcome ! — the coq proof assistant, July 2013. 76

[29] F. Ivancic, I. Shlyakhter, A. Gupta, M.K. Ganai, V. Kahlon, Chao Wang, and Zijiang Yang. Model checking c programs using f-soft. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 297–308, 2005. 34

[30] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 2*, AAAI'96, pages 1194–1201. AAAI Press, 1996. 33

[31] Daniel Kroening. The cbmc homepage, July 2013. 2, 34

[32] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003. 2

[33] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI 2003, pages 298–309, London, UK, UK, 2003. Springer-Verlag. 32

[34] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209. 31

[35] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992. 9

[36] Terence Parr. Antlr, July 2013. 60

[37] C. Pixley. A computational theory and implementation of sequential hardware equivalence. In *Proceedings of the DIMACS workshop on Computer Aided Verification*, pages 293–320. DIMACS (technical report 90-31), 1990. 31

[38] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag. 30

[39] Ofer Shtrichman. Tuning sat checkers for bounded model checking. pages 480–494. Springer-Verlag, 2000. 32, 33

[40] João P. Marques Silva and Karem A. Sakallah. Grasp – a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society. 32

[41] G. Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems 1990 (SAFECOMP '90). Safety, Security and Reliability Related Computers for the 1990s. Proceedings of the IFAC/EWICS/SARS Symposium*, pages 31–36. Pergamon Press, 1990. 32

[42] Zerksis D. Umrigar and Vijay Pitchumani. Formal verification of a real-time hardware design. In *Proceedings of the 20th Design Automation Conference*, DAC '83, pages 221–227, Piscataway, NJ, USA, 1983. IEEE Press. 1

[43] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and sat procedures for efficient symbolic model checking. 32

[44] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993. 9

[45] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *In Proc. 32Ç È POPL. ACM*, 2005. 34

[46] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *In Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 243–257, 2006. 34

[47] Hantao Zhang. Sato: An efficient propositional prover. In William McCune, editor, *CADE*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer, 1997. 32

Apêndice A

Apêndice

A.1 Aspectos técnicos sobre a ferramenta

Compilar a implementação ghc –make Main.hs -o engine.bin resulta numa ferramenta que reconhece um e um só *bound* e multiplos ficheiros em estilo DSL. Um ficheiro diz-se do estilo DSL se for reconhecido pela instancia de Read implementada (o parser gera este tipo de ficheiros). A ferramenta reconhece 3 argumentos de linha de comandos, -v -b e -f. Qualquer outro tipo de argumento que não seja reconhecido, as opções são disponibilizadas. Conforme Figura A.1.

```
1  User C:Install\Directory\> ./engine.bin --something
2  engine.bin: unrecognized option `--something'
3  Usage: main [OPTION...]
4    -v                          --version                          show version number
5    -b 34                       --bound=34                         BMC unWinding bound
       limit
6    -f /Absolute/Path/file.dsl  --file=/Absolute/Path/file.dsl   input DSL file to
       read
7
8  User C:Install\Directory\> ./engine.bin --bound=4 --file=/Users/jj/Dropbox/gbmc/dsl
     /04.dsl --something
9  engine.bin: unrecognized option `--something'
10 Usage: main [OPTION...]
11   -v                          --version                          show version number
12   -b 34                       --bound=34                         BMC unWinding bound
       limit
13   -f /Absolute/Path/file.dsl  --file=/Absolute/Path/file.dsl   input DSL file to
       read
14
15 User C:Install\Directory\>
```

Figura A.1: Argumentos reconhecíveis

A opção -v ou –version disponibiliza a informação sobre a implementação que está a ser executada. Várias invocações resultarão no mesmo resultado, conforme Figura A.2.

```
1 User C:Install\Directory\> ./engine.bin --bound=34 --version --version --version
2 Ferramenta Bounded Model Checking aplicado a software 2013
3 User C:Install\Directory\> engine.bin --version
4 Ferramenta Bounded Model Checking aplicado a software 2013
5 engine.bin: [error] please provide a bound
6 User C:Install\Directory\>
```

Figura A.2: Ilustração de versão

A ferramenta necessitará e exigirá sempre um *bound*. Mais do que um *bound* gerará um erro pelo que será necessário executar a ferramenta a um dado ficheiro ou conjunto de ficheiros, tantas vezes quantos os *bounds* necessários, conforme Figura A.3.

```
1 User C:Install\Directory\> ./engine.bin
2 engine.bin: [error] please provide a bound
3 User C:Install\Directory\> ./engine.bin --bound=4 --bound=5
4 engine.bin: [error] please provide just one bound
5 User C:Install\Directory\> ./engine.bin
6 engine.bin: [error] please provide a bound
7 User C:Install\Directory\> ./engine.bin --bound=7
8 User C:Install\Directory\>
```

Figura A.3: Ilustração de *bounds*

Sem um ficheiro de input, a ferramenta não produz output. É possivel fornecer vários ficheiros de input, como consequência, o método é aplicado a todos os ficheiros fornecidos e é gerado 1 relatório para cada ficheiro, conforme Figura A.4.

```
1  User C:Install\Directory\> ./engine.bin --bound=7
2  User C:Install\Directory\> ./engine.bin --bound=7 --file=./input/file01.dsl
3  Successfully generated tex file: ./input/file01.dsl.tex
4  User C:Install\Directory\> ./engine.bin --bound=7 --file=./input/file01.dsl \
5                                                    --file=./input/file02.dsl \
6                                                    --file=./input/file03.dsl
7  Successfully generated tex file: ./input/file01.dsl.tex
8  Successfully generated tex file: ./input/file02.dsl.tex
9  Successfully generated tex file: ./input/file03.dsl.tex
10 User C:Install\Directory\>
```

Figura A.4: Ilustração de input com ficheiros

É um processo iterativo, qualquer erro que suceda durante o processamento (erro de leitura, inexistência de ficheiro) não é tradado e os ficheiros criados/estados intermédios durante o processo não serão eliminados, conforme Figura A.5.

```
1 User C:Install\Directory\> ./engine.bin --bound=4   --file=./input/file01.dsl \
2                            --file=./input/file02.dsl \
3                            --file=./input/fileUnknown.dsl \
4                            --file=./input/file04.dsl
5 engine.bin: ./input/fileUnknown.dsl: openFile: does not exist (No such file or
     directory)
6 User C:Install\Directory\>
```

Figura A.5: Ilustração de erros gerados

Conforme expresso na gramática, Figura 5.13, cada ficheiro é considerado como uma sequência de procedimentos. A ferramenta ferramenta `gbmc` analiza cada ficheiro e para cada procedimento cria um ficheiro *".smt"* e introduz essa informaçao no relatório do ficheiro. Assim, um ficheiro com 1 procedimento resultará num relatório *".tex"* e 1 fórmula codificada no standard smt-liv v2 para o procedimento. Um ficheiro com 100 procedimentos resultará num relatório *".tex"* e 100 fórmulas, uma para cada procedimento.

Note-se no entanto que o comando **call** (p) apenas poderá invocar procedimentos definidos dentro do ficheiro invocador. Caso contrário um erro é gerado pois o procedimento não é encontrado, nem está definido, como tal não se sabe qual o significado de aplicar o comando **call** (p). A ferramenta não realiza ligações entre ficheiros.

A.2 Ficheiro SMT-LIB v2.0 resultante do exemplo ilustrativo

```
1  ; * Mestrado em Matematica e Computacao
2  ; * Projecto de Dissertacao - Ano lectivo de 2011/2013
3  ; *      - Bounded Model Checking para uma Linguagem Imperativa Simples
4  ; * Realizado por:
5  ; *      - Jose Joao Peixoto Pereira   - pg17312@alunos.uminho.pt
6  ; * Professores Orientadores:
7  ; *      - Luis Filipe Ribeiro Pinto   - Departamento de Matematica
8  ; *      - Jorge Sousa Pinto           - Departamento de Informatica
9  ; * Em conformidade com o standard SMT-LIB v2.0
10 ; *      - \url{goedel.cs.uiowa.edu/smtlib/papers/smt-lib-reference-v2.0-r10.03.30.
     pdf}
11
12 ; * Options:
13 (set-option :print-success true)
14
15 ; * Declarations:
16 (declare-const x1 Int)
17 (declare-const y0 Int)
18 (declare-const y1 Int)
19 (declare-const y2 Int)
20 (declare-const y3 Int)
21 (declare-const y4 Int)
22
23 ; * Assertions:
24 ; *     set(C) -> set(P) === not_set(C) or set(P)
25 ; *     is there a not (set(C)->set(P))? not(not_set(C) or set(P)) === set(C) and
     not_set(P)
26 (assert (and ; set(C), the Codification of the Program
27      (and
28          (or  (not  true )  (=  x1 2 ))
29      (and
30          (or  (not  (and  true  (<  y0  x1 )))  (=  y1  (+  y0 1 ) ))
31      (and
32          (or  (not  (and  (and  true  (<  y0  x1 )) (<  y1  x1 )))  (=  y2  (+  y1
             1 ) ))
33      (and
34          (or  (not  (and  (and  true  (<  y0  x1 )) (<  y1  x1 )))  (=  y3  y2 ))
35      (and
36          (or  (not  (and  (and  true  (<  y0  x1 )) (not  (<  y1  x1 )) ))  (=  y3
             y1 ))
37      (and
38          (or  (not  (and  true  (<  y0  x1 )))  (=  y4  y3 ))
39          (or  (not  (and  true  (not  (<  y0  x1 )) ))  (=  y4  y1 ))
40 ))))))
41  (not ; not_set(P) the Codification of the Conditions
42      (and
43          (or  (not  (and  (and  true  (<  y0  x1 )) (<  y1  x1 )))  (not  (<  y2
             x1 )) )
44          (or  (not  true )  (or  (=  y4  x1 ) (>  y4 2 )))
45 )
46 )))
47 ; *
48 ; * Computation:
49 ; *      - Check satisfiability
50 (check-sat)
51 (get-model) ; -- A formula is unsatisfiable if it does not have any model.
52 (exit)
```

A.3 Modelo obtido pela ferramenta Z3

```
1  (model
2    (define-fun y0 () Int
3      (- 1))
4    (define-fun y1 () Int
5      0)
6    (define-fun y4 () Int
7      1)
8    (define-fun y2 () Int
9      1)
10   (define-fun y3 () Int
11     1)
12   (define-fun x1 () Int
13     2)
14 )
```