Universidade Federal do Amazonas

Faculdade de Tecnologia

Programa de Pós-Graduação em Engenharia Elétrica

Verificação Baseada em Indução Matemática para Programas C++

Mikhail Yasha Ramalho Gadelha

Manaus – Amazonas

Dezembro de 2013

Mikhail Yasha Ramalho Gadelha

Verificação Baseada em Indução Matemática para Programas C++

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica. Área de concentração: Automação e Controle.

Orientador: Prof. Dr. Lucas Carvalho Cordeiro

Mikhail Yasha Ramalho Gadelha

Verificação Baseada em Indução Matemática para Programas C++

Banca Examinadora

Prof. Ph.D. Lucas Carvalho Cordeiro – Presidente e Orientador
Departamento de Eletrônica e Computação – UFAM

Prof. D.Sc. Raimundo da Silva Barreto
Instituto de Computação – UFAM

Prof. Dr-Ing Vicente Ferreira de Lucena Junior
Departamento de Eletrônica e Computação – UFAM

Manaus – Amazonas
Dezembro de 2013

À minha mãe e minha esposa.

Agradecimentos

Agradeço primeiramente a Deus por me ajudar a conseguir todas as minha vitórias.

Agradeço à minha mãe, Isley Maria da Conceição Ramalho Gomes, por estar sempre do meu lado, nos momentos de vitória e derrota, e cujo amor é incondicional. A meu pai, Aldamir Gadelha, por ter me acompanhado nessa jornada e à minha esposa, Loma Brito do Nascimento, por sua paciência, compreensão, ajuda e pelo seu amor.

Agradeço aos meus amigos e colegas de mestrado da UFAM, pela amizade e pela força nas longas horas de estudo.

Agradeço aos amigos e ex-colegas de graduação da UFAM e Fucapi, pela amizade e companheirismo.

Agradeço aos meus amigos da AIESEC, que muito me ensinaram e me tornaram uma pessoa melhor e mais consciente do mundo.

Agradeço aos Prof. Lucas Cordeiro e Waldir Sabino, pelo suporte, direcionamento, encorajamento e amizade essenciais para a realização desse trabalho.

Agradeço à equipe do projeto de verificação formal na UFAM, que contribuíram direta e indiretamente para a realização deste trabalho.

“Doing the same thing, and Expecting a different outcome is the definition of insanity”.

Albert Einstein (1879-1955)

Resumo

A utilização de sistemas embarcados, sistemas computacionais especializados para realizar uma função em sistemas maiores, eletrônicos ou mecânicos, vem crescendo no dia a dia das pessoas, e vem se tornando cada vez mais importante garantir a robustez desses sistemas. Existem diversas técnicas para garantir que um sistema seja lançado sem erros. Em especial, a verificação formal de programas está se mostrando efetiva na busca por falhas. Neste trabalho, serão descritos a verificação formal de programas C++ e a prova de corretude por indução matemática. Ambas as técnicas serão desenvolvidas utilizando a ferramenta *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC), um verificador de modelos que se baseia em teorias de satisfabilidade de fórmulas proposicionais e de lógica de primeira ordem. Os experimentos mostram que a ferramenta pode ser utilizada para verificar uma ampla gama de aplicações, de casos simples à aplicações comerciais. A ferramenta também mostrou-se superior em comparação com outros verificadores na verificação de programas C++, encontrando um maior número de erros e suportando um número superior das funcionalidades que a linguagem C++ tem a oferecer, além de ser capaz de provar diversas propriedades (por exemplo, laços invariantes), utilizando a técnica de indução matemática.

Palavras-chave: verificação formal, prova por indução, linguagem de programação C++.

Abstract

The use of embedded systems, computational systems specialized to do a function in larger systems, electronic or mechanical, is growing in the daily life, and it is becoming increasingly important to ensure the robustness of these systems. There are several techniques to ensure that a system is released without error. In particular, formal verification is proving very effective in finding bugs in programs. In this work, we describe the formal verification for C++ Programs and correctness proof by mathematical induction. Both techniques will be developed using the tool *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC), a model checker based on satisfiability modulo theories and first order logic. The experiments show that the tool can be used to check a wide range of applications, from simple test cases to commercial applications. The tool also proved to be more efficient than other models checkers to verify C++ programs, finding a greater number of bugs, and supporting a larger number of the features that the language C++ has to offer, in addition to being able to prove several properties, using the method of mathematical induction.

Keywords: formal verification, proof by induction, C++ programming language.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Abreviações

BMC - ***B**ounded **M**odel **C**hecking*

DMA - ***D**irect **M**emory **A**ccess*

TLM - ***T**ransaction **L**evel **M**odeling*

FPGA - ***F**ield-**P**rogrammable **G**ate **A**rray*

SAT - **SAT**isfatibilidade Booleana

SMT - ***S**atisfiability **M**odulo **T**heories*

VC - ***V**erification **C**condition*

CBMC - ***C B**ounded **M**odel **C**hecker*

ESBMC - ***E**fficient **S**MT-Based Context-**B**ounded **M**odel **C**hecker*

STL - ***S**tandard **T**emplate **L**ibrary*

LP **L**ógica *P*roposicional

CNF - ***C**onjective **N**ormal **F**orm*

DPLL - Algoritmo **D**avis-**P**utnam-**L**ogemann-**L**oveland

OO - **O**rientação a **O**bjetos

AST - ***A**bstract **S**yntax **T**ree*

IREP - ***I**ntermediate **REP**resentation*

VCG - ***V**erification **C**condition **G**enerator*

SSA - ***S**ingle **S**tatic **A**ssignment*

COM - ***C**++ **O**perational **M**odel*

CHG - ***C**lass **H**ierarchy **G**raph*

LLBMC - ***L**ow **L**evel **B**ounded **M**odel **C**hecker*

Capítulo 1

Introdução

Um sistema embarcado é um sistema computacional que tem o objetivo de controlar uma função ou um conjunto de funções em um sistema maior, elétrico ou mecânico [2]. As linguagens C e C++ são duas das principais linguagens de programação para desenvolvimento desses sistemas [3]. Por sua vez, os sistemas embarcados são utilizados em uma gama de aplicações, como controles automotivos e sistemas de entretenimento, de tal forma que os sistemas embarcados vêm se tornando indispensáveis para a vida do homem moderno [3].

Devido a importância dos sistemas embarcados é que se mostra evidente a importância de garantir a robustez na execução dos mesmos. A partir dessa preocupação, nota-se o crescimento da área de testes no processo de desenvolvimento desses sistemas.

Atualmente, Revisão por Pares e Teste de Programas são as duas principais técnicas para teste de programas [1]. Na Revisão por Pares, uma equipe de engenheiros experientes inspeciona o programa e, preferencialmente, não se envolve com o desenvolvimento do programa que está sendo revisado. Estudos empíricos mostram que a técnica é capaz de encontrar entre 31% a 93% do defeitos, com um média de 60% [1]. No teste de programas, a busca de defeitos é feita através de extensivas baterias de testes, que forçam o programa a executar nas mais diversas situações, seguida da análise dos resultados desses testes. Nessa abordagem, cerca de 70% do tempo de desenvolvimento é concentrado em encontrar e corrigir falhas no programa [4].

Existe porém, uma técnica que vem crescentemente sendo utilizada na verificação de programas, chamada verificação formal, que é uma técnica baseada em formalismos matemáticos para especificação, projeto e verificação de *software* e *hardware* [5]. Em especial, algoritmos *Bounded Model Checking* (BMC) baseados em Satisfatibilidade Booleana (SAT) ou

Teorias do Módulo da Satisfatibilidade (SMT, do inglês *Satisfiability Modulo Theories*) foram aplicados com sucesso para verificação de programas sequenciais e paralelos e na descoberta de defeitos sutis [6, 7, 8, 9].

1.1 Descrição do Problema

Esse trabalho visa dois problemas da área de verificação de modelos: (1) a verificação de modelos de programas C++ e (2) utilizar a técnica de indução matemática para provar corretude de programas.

O primeiro problema ocorre porque a verificação de modelos de programas escritos em linguagem C++ é complexa. A linguagem C++ possui diversas funcionalidades e conceitos de orientação a objetos (OO) que tornam difícil a codificação correta dos programas nas fórmulas matemáticas necessárias pelo solucionador SMT. Diversos pesquisadores [10, 11, 12, 13, 14] propuseram técnicas e desenvolveram algoritmos com o intuito de verificar programas C++. Porém, devido ao grande número de funcionalidades que a linguagem C++ provê, a implementação e verificação de programas C++ caminha a passos lentos [11]. O suporte das ferramentas para linguagem C++ está geralmente focada em funcionalidades específicas e acaba por ignorar funcionalidades igualmente importantes, por exemplo, processos de verificação que não suportam tratamento de exceções [15].

O segundo problema deriva do fato de que algoritmos BMC somente conseguem verificar a negação de propriedades até uma dada profundidade k, o que restringe a abrangência dos algoritmos (por exemplo, quando se quer provar propriedades sem que um limite superior seja conhecido). A Figura 1.1 apresenta um programa que exemplifica o problema da verificação utilizando a técnica BMC.

```
1 int main() {
2   long long int i, sn=0;
3   unsigned int n;
4   assume (n>=1);
5   for(i=1; i<=n; i++)
6     sn = sn + a;
7   assert(sn==n*a);
8 }
```

Figura 1.1: Código exemplo para prova por indução matemática.

No programa apresentado na Figura 1.1, a propriedade (representada pela assertiva da

linha 7) deve ser verdadeira para qualquer valor de n. A técnica BMC apresenta dificuldades em provar esse programa, devido ao fato do limite superior do laço ser não determinístico. Devido a esta condição, o laço será desdobrado $2^n - 1$ vezes, o que necessitará de uma grande quantidade de memória e tempo para encontrar uma solução.

1.2 Objetivos

O principal objetivo dessa dissertação é propor um algoritmo que use indução matemática em verificadores de modelos limitados (BMC), que possa ser executado em arquiteturas multiprocessadas de modo a tornar possível a prova de corretude de programas C++.

Os objetivos específicos são:

a) Comprovar experimentalmente que a adição de *templates* e tratamento de exceções, em uma ferramenta de verificação de modelos de programas C++, aumenta a cobertura de programas verificados com sucesso em suítes de teste públicas

b) Provar que a indução matemática é um meio eficaz para verificação de programas C++.

1.3 Descrição da Solução

A abordagem lida com os aspectos teóricos e práticos da verificação de programas C e C++ através da indução matemática. Para tal, foram desenvolvidos algoritmos em uma ferramenta de verificação estado da arte, que foi avaliada utilizando diversos conjuntos de casos de testes. A escolha do desenvolvimento em uma ferramenta de verificação baseia-se no fato de que o esforço foi concentrado na criação e implementação de técnicas de verificação para as funcionalidades da linguagem C++ e do algoritmo de indução matemática. A ferramenta escolhida foi o ESBMC [7], que utiliza o solucionador SMT Z3 [16] e o *front-end* do *framework* CProver, o qual é base dos verificadores CBMC [6] e SATABS [17].

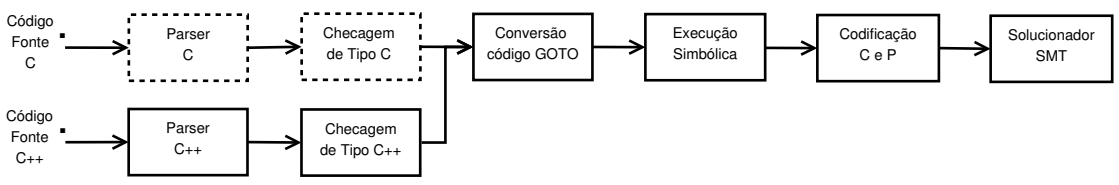


Figura 1.2: Arquitetura do ESBMC.

A Figura 1.2 mostra uma visão geral da arquitetura e do funcionamento da ferramenta ESBMC. Os dois primeiros blocos do processo de verificação, *parser* e checagem de tipo, funcionam de forma diferente no caso da verificação de programas em C e C++ (blocos pontilhados e contínuos, respectivamente). Ao final desses dois blocos, uma representação intermediária do programa é gerada e utilizada nas subsequentes fases da verificação. As etapas são descritas como:

***Parser* C++** Utiliza as ferramentas flex e bison [18] para gerar a maior parte da representação intermediária do programa. O *parser* é fortemente baseado no compilador GNU C++, pois isso permite ao ESBMC encontrar a maior parte dos erros de sintaxe que são reportados pelo GCC. Foram feitas alterações para criação de representação intermediária de *templates* e tratamento de exceções.

Checagem de Tipo C++ São feitas verificações iniciais na representação intermediária, incluindo verificação de atribuição, conversão de tipos, inicialização de ponteiros, chamadas de funções, além da criação da representação intermediária dos *templates*. As principais alterações referem-se a instanciação de *templates*, porém houveram alterações para oferecer suporte ao tratamento de exceções e alocação dinâmica utilizando o operador *new*.

Conversão código *GOTO* A representação intermediária é convertida em expressões *GOTO*. Essa conversão simplifica a representação do programa (por exemplo, substituição de *switch* e *while* por *if* e expressões *goto*). Além disso, são eliminadas recursões e as iterações são desenroladas. Em relação à linguagem C++, foram necessárias alterações para suportar as novas estruturas criadas na etapa anterior, especialmente no caso do tratamento de exceções, enquanto que para suportar indução matemática, as diversas transformações necessárias para provar propriedades (descritas no Capítulo 5) ocorrem nesse bloco.

Execução Simbólica o programa *GOTO* simplificado é convertido para expressões SSA e assertivas são inseridas nas expressões para verificar propriedades como estouro de limites de vetor, *overflow* e *underflow* aritmético, vazamento de memória, *double free* e divisão por zero. Nesse bloco, foram feitas alterações para implementar o comportamento do

fluxo de tratamento de exceções, como inserção de instruções *GOTO* para expressões do tipo *throw* e especificação de exceções para chamadas de funções.

Codificação C e P A partir das expressões SSA, dois conjuntos de fórmulas livres de quantificadores são criadas: *C* para as restrições e *P* para as propriedades. Para o trabalho proposto, não houve necessidade de alterações nesse bloco.

Solucionador SMT Os dois conjuntos de fórmulas *C* e *P* são utilizadas como entrada para o solucionador SMT (atualmente o ESBMC utiliza o Z3 [16]). Ao final, o solucionador irá produzir um contraexemplo se uma dada propriedade for violada ou uma resposta insatisfeita se a propriedade se manter [19]. Assim como no bloco anterior, não houve necessidade de alterações nesse bloco.

Além das alterações nos blocos citados, para oferecer suporte a verificação em paralelo por indução matemática, a estrutura de execução do ESBMC precisou ser alterada, de forma que até três sequências dos blocos anteriores operem simultaneamente.

1.4 Contribuições

Este trabalho apresenta duas contribuições. Primeiro, é descrito o processo de verificação de duas funcionalidades de programas C++: *templates* e tratamento de exceções. Em relação a *templates*, é apresentado o processo de criação e instanciação de *templates* durante a verificação de programas C++. Através de testes, nota-se que a implementação serve como base ao desenvolvimento de um modelo simplificado da Biblioteca de *Template* Padrão, que inclui todos os contêineres sequenciais e associativos, e que foi utilizado para verificação de aplicações comerciais. Na verificação de programas com tratamento de exceções, foram mostradas diversas regras de conexão entre expressões *throw* e *catch*, além de descrever a implementação do suporte a especificação de exceções, funcionalidade que somente a ferramenta ESBMC oferece suporte.

Na prova de corretude por indução matemática, são mostradas as transformações necessárias para provar as propriedades de um programa C++. Essas transformações são modeladas utilizando a Lógica Hoare. Além disso, é apresentado o algoritmo responsável pela indução matemática, chamado *k-induction*, que consiste de três passos (caso base, condição adiante e passo

indutivo), para provar a corretude de um programa. O algoritmo *k-induction*, desenvolvido neste trabalho, fornece a opção de ser executado de forma sequencial, onde cada passo inicia ao final do anterior, ou de forma paralela, utilizando uma abordagem multiprocessos, onde cada passo do algoritmo é executado de forma concorrente. O algoritmo *k-induction* apresentado neste trabalho é totalmente automático, requisitando apenas o código fonte de um programa para iniciar o processo de prova.

1.5 Organização da Dissertação

A introdução apresenta os objetivos e motivações da pesquisa feita durante a realização desse trabalho. O restante das seções estão divididas da seguinte forma:

O Capítulo 2, *Fundamentação Teórica*, apresenta uma revisão dos conceitos básicos por trás da verificação formal de programas, além da definição da Lógica Hoare.

O Capítulo 3, *Trabalhos Relacionados*, apresenta um resumo de diversas ferramentas e técnicas de verificação empregadas, para verificar programas C++ e provar corretude de tais programas utilizando técnicas de indução matemática.

O Capítulo 4, *Verificação de Programas C++*, apresenta o processo de criação e instanciação de *templates* e como foi utilizada como base para a implementação de um modelo simplificado da Biblioteca de *Template* Padrão. Além disso, são mostrados as regras de conexões e tratamento de especificação de exceções que podem estar presentes em programas C++. Por fim, são apresentados os resultados da verificação de mais de 1180 casos de teste e uma aplicação comercial, desenvolvidos em C++, e a comparação dos resultados com a ferramenta LLBMC [11].

O Capítulo 5, *Prova por Indução Matemática*, apresenta o algoritmo de verificação por indução matemática utilizando a técnica *k-induction*, bem como a formalização das transformações utilizando a lógica Hoare e lógica proposicional necessárias para a prova das propriedades. Por fim, são apresentados os resultados da verificação de mais de 90 casos de teste, desenvolvida em C++, e a comparação dos resultados da execução do algoritmo *k-induction* sequencial e paralelo.

Por fim, o Capítulo 6, *Conclusões*, aapresenta as contribuições do trabalho, além de apresentar sugestões de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Nesse capítulo, serão mostrados os conceitos e conhecimentos básicos para o entendimento de verificação formal e o funcionamento da ferramenta ESBMC. Será mostrada também a Lógica Hoare, utilizada para formalizar a prova por indução matemática.

2.1 Lógica Proposicional

Lógica pode ser definida através de símbolos e um sistema de regras para manipular esses símbolos [20]. A sintaxe da Lógica Proposicional (LP) consiste de símbolos e regras que combinam os símbolos para construir sentenças (ou fórmulas). Generalizando, a lógica proposicional é um lógica binária, baseada na suposição que toda sentença é verdadeira ou falsa. Um valor verdade (um valor lógico, representado pelo símbolo *tt* ou *ff*), é um valor que indica se a relação de uma proposição (por exemplo, o resultado de uma expressão) é verdadeira ou falsa. Os elementos básicos da LP são as constantes verdadeiro (também representado por $\top$ ou 1) e falso (também representado por $\bot$ ou 0), além das variáveis proposicionais: $x_1, x_2, \ldots, x_n$ (onde o conjunto é geralmente representado pela letra X e n é um número finito de variáveis proposicionais). Operadores lógicos (por exemplo, $\neg, \wedge$), também chamados operadores booleanos, fornecem o elemento de conexão para criação de expressões na LP [8].

Definição 2.1 *A sintaxe das fórmulas em LP é definida pela seguinte gramática:*

$$\begin{aligned} Fml &::= Fml \wedge Fml \,|\, \neg Fml \,|\, (Fml) \,|\, Atomo \\ Atomo &::= Variable \,|\, verdadeiro \,|\, falso \end{aligned}$$

Utilizando-se dos operadores lógicos conjunção ($\wedge$) e negação ($\neg$), pode-se construir as expressões em LP. Outros operadores lógicos como disjunção ($\vee$), implicação ($\Rightarrow$), equivalência lógica ($\Leftrightarrow$), ou exclusivo ($\oplus$) e expressões condicionais (ite) são definidos a seguir.

Definição 2.2 *Os operadores lógicos são definidos da seguinte forma:*

- $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$
- $\phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$
- $\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$
- $\phi_1 \oplus \phi_2 \equiv (\phi_1 \wedge \neg\phi_2) \vee (\phi_2 \wedge \neg\phi_1)$
- $ite(\theta, \phi_1, \phi_2) \equiv (\theta \wedge \phi_1) \vee (\neg\theta \wedge \phi_2)$

Fórmulas em LP são definidas em termos dos elementos básicos verdadeiro, falso, ou uma variável proposicional x; ou, a partir da aplicação lógica de um dos seguintes operadores em uma dada fórmula ϕ: "não" ($\neg\phi$), "e" ($\phi_1 \wedge \phi_2$), "ou" ($\phi_1 \vee \phi_2$), "implica" ($\phi_1 \Rightarrow \phi_2$), "se, somente se" ($\phi_1 \Leftrightarrow \phi_2$). "igualdade" ($\phi_1 \oplus \phi_2$) ou "ite" ($ite(\theta, \phi_1, \phi_2)$).

Cada operador em LP possui uma aridade (isto é, número de argumentos). O operador "não" é unário enquanto os outros operadores são binários, com exceção do "ite", que é terciário. Os argumentos da direita e da esquerda do operador $\Rightarrow$ são chamados antecedentes e consequentes, respectivamente. As variáveis proposicionais, e constantes proposicionais, verdadeiro e falso, são proposições indecomponíveis, conhecidas como átomos ou proposições atômicas. Um literal é um átomo β e sua negação é $\neg\beta$. Uma fórmula é um literal ou a aplicação de um operador lógico em uma ou mais fórmulas.

Fórmulas em LP são expressões sobre o alfabeto: $\{x_1, x_2, x_3, \ldots\} \cup \{\neg, \wedge, \vee, \Rightarrow\Leftrightarrow\} \cup \{(,)\}$. A expressão $\wedge(\neg) \vee x_1 x_2 \Leftrightarrow$ é uma expressão neste alfabeto, porém não possui sentido se considerarmos a sintaxe e semântica de expressões em LP.

A Figura 2.1 mostra um exemplo de dois códigos e os seus respectivos equivalentes em LP. Ambas as fórmulas 2.1 e 2.2 são equivalentes, segundo os operadores apresentados na Definição 2.2. A validade dessa igualdade pode ser verificada a partir da Fórmula 2.3.

Definição 2.3 *É dito que uma fórmula em LP é bem formada se a mesma utiliza as regras definidas em (2.1), dada que a negação tem prioridade sobre conjunções.*

```
1 if(!a&&!b) h();
2 else
3   if(!a) g();
4   else f();
```

(a) Exemplo de código.

$$(\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \vee a \wedge f) \tag{2.1}$$

(b) Fórmula LP do código na Figura 2.1a

```
1 if(a) f();
2 else
3   if(b) g();
4   else h();
```

(c) Exemplo de código.

$$a \wedge f \vee \neg a \wedge (b \wedge g \vee \neg b \wedge h) \tag{2.2}$$

(d) Fórmula LP do código na Figura 2.1c

Figura 2.1: Códigos C e equivalentes fórmulas LP.

$$(\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \vee a \wedge f) \Longleftrightarrow a \wedge f \vee \neg a \wedge (b \wedge g \vee \neg b \wedge h) \tag{2.3}$$

Definição 2.4 *As regras de precedência dos operadores lógicos são definidas na seguinte forma, do mais alto ao menor:* $\neg$, $\wedge$, $\vee$, $\Rightarrow$ *e* $\Leftrightarrow$.

Para verificar se uma dada LP é verdadeira ou falsa, é preciso definir um mecanismo para avaliar as variáveis proposicionais através de interpretação. Uma interpretação I atribui para todas as variáveis proposicionais um valor. Por exemplo, $I = \{x_1 \mapsto tt, x_2 \mapsto ff\}$ é uma interpretação atribuindo verdadeiro para x_1 e falso para x_2. Dada uma fórmula em LP e uma interpretação, o valor da fórmula pode ser calculado através de uma tabela verdade ou por indução. Considerando os possíveis valores de um variável proposicional x (por exemplo, *tt* ou *ff*), pode-se construir uma tabela verdade para os operadores lógicos $\neg, \wedge, \vee, \Rightarrow$, $\Leftrightarrow$ e $\oplus$ como mostrado na Tabela 2.1.

x_1	x_2	$\neg x_1$	$x_1 \wedge x_2$	$x_1 \vee x_2$	$x_1 \Rightarrow x_2$	$x_1 \Leftrightarrow x_2$	$x_1 \oplus x_2$
ff	*ff*	*tt*	*ff*	*ff*	*tt*	*tt*	*ff*
ff	*tt*	*tt*	*ff*	*tt*	*tt*	*ff*	*tt*
tt	*ff*	*ff*	*ff*	*tt*	*ff*	*ff*	*tt*
tt	*tt*	*ff*	*tt*	*tt*	*tt*	*tt*	*ff*

Tabela 2.1: Tabela verdade.

Pode-se descrever uma definição indutiva da semântica da LP que define o significado dos operadores básicos e do significado de fórmulas mais complexas em termos dos operadores

básicos. Pode-se escrever $I \models \phi$ se ϕ for avaliado em *tt* e $I \not\models \phi$ se ϕ for avaliado em *ff*, baseados em um dado I.

Definição 2.5 *Define-se a avaliação da fórmula ϕ baseada em uma interpretação I como:*

- $I \models x$ *sse* $I[x] = tt$
- $I \models \neg\phi$ *sse* $I \not\models \phi$
- $I \models \phi_1 \wedge \phi_2$ *sse* $I \models \phi_1$ *e* $I \models \phi_2$
- $I \models \phi_1 \vee \phi_2$ *sse* $I \models \phi_1$ *ou* $I \models \phi_2$

Lemma 2.1 *A semântica de fórmulas mais complexas são avaliadas como:*

- $I \models \phi_1 \Rightarrow \phi_2$ *sse, sempre que* $I \models \phi_1$ *então* $I \models \phi_2$
- $I \models \phi_1 \Leftrightarrow \phi_2$ *sse* $I \models \phi_1$ *e* $I \models \phi_2$, *ou* $I \not\models \phi_1$ *e* $I \not\models \phi_2$

Definição 2.6 *Uma fórmula LP é satisfatível em relação a uma classe de interpretações se existe uma atribuição para as variáveis na qual a fórmula seja avaliada em verdadeiro.*

A entrada do algoritmo para verificar a satisfabilidade é normalmente uma proposição lógica na forma normal conjuntiva (CNF, do inglês *Conjuctive Normal Form*).

Definição 2.7 *Formalmente, uma fórmula LP ϕ está na forma normal conjuntiva se ela consiste de um conjunção de uma ou mais cláusulas, onde cada cláusula é um disjunção de um ou mais literais. Possui a forma $\bigwedge_i \left(\bigvee_j l_{ij}\right)$, onde cada l_{ij} é um literal.*

Os BMC modernos, que verificam a satisfabilidade de uma fórmula LP, são baseados em uma variante do algoritmo *Davis-Putnam-Logemann-Loveland* (DPLL) [21]. No algoritmo DPLL, um valor verdade é escolhido para um literal e as implicações de fácil detecção dessa atribuição são propagadas, e retrocede em caso de conflitos, ou seja, se os valores propagados aos literais são conflitantes. O algoritmo retorna satisfatível (SAT, do inglês *satisfiable*), quando valores forem atribuídos para todas as variáveis e nenhum conflito for encontrado, e retorna insatisfatível (UNSAT, do inglês *unsatisfiable*), caso contrário.

A Figura 2.2 mostra o algoritmo DPLL. No estado *Atribui literal*, um valor verdade é atribuído a um literal. Caso valores tenham sido atribuídos a todos os literais e nenhum conflito

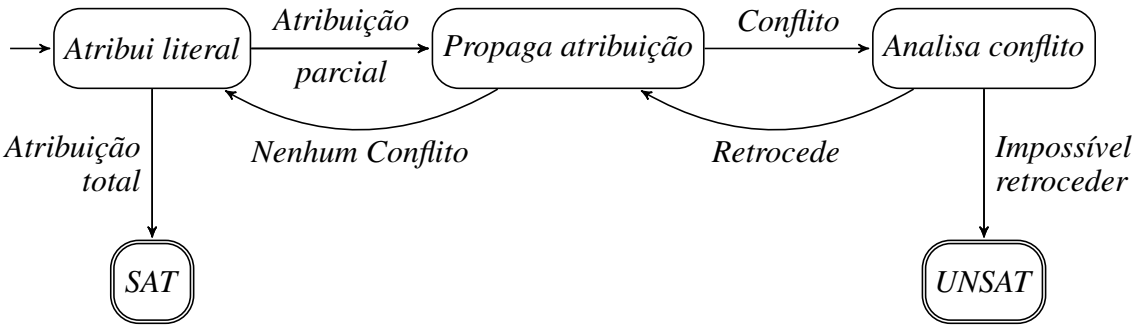


Figura 2.2: Algoritmo DPLL.

tenha sido encontrado, o algoritmo retorna *SAT* através da transição *Atribuição total*, ou seja, a fórmula é satisfatível. Caso valores tenham sido atribuídos a somente uma parte dos literais, a transição *Atribuição parcial* ocorre. No estado *Propaga atribuição*, a atribuição é propagada e é verificado se ocorreu um conflito. Se nenhum conflito ocorreu durante a propagação do valor atribuído a esse literal, o algoritmo volta ao estado inicial, *Atribui literal* através da transição *Nenhum conflito*. No estado inicial, é novamente verificado se valores foram atribuídos a todos o literais e assim por diante. Se houve um conflito na propagação do valor atribuído, a transição *Conflito* ocorre e o conflito é analisado no estado *Analisa conflito*.

No estado *Analisa conflito*, o algoritmo aprende a situação que houve o conflito, retrocede ao estado *Propaga atribuição*, e verifica se houve conflito a partir desse aprendizado. O algoritmo ficará caminhando entre esses dois estados até que não seja mais possível retroceder ou até que nenhum conflito seja encontrado e um novo valor de literal possa ser atribuído. Caso não seja mais possível retroceder, o algoritmo retorna *UNSAT*, através da transição *Impossível retroceder*.

Neste contexto, um solucionador SAT é um algoritmo (baseado em uma variante do DPLL) que recebe uma fórmula ϕ como entrada e decide se ela é satisfatível ou insatisfatível. A fórmula ϕ é dita satisfatível (*SAT*) se o solucionador SAT for capaz de encontrar uma interpretação que torne a fórmula verdadeira (Definição 2.6). A fórmula ϕ é dita insatisfatível (ou *UNSAT*) se nenhuma interpretação torne a fórmula verdadeira. Em caso satisfatível, solucionadores SAT podem fornecer um modelo, por exemplo, um conjunto de atribuições para as variáveis proposicionais da fórmula ϕ, ou seja, um contraexemplo.

2.2 Teorias do Módulo da Satisfabilidade

SMT decide a satisfabilidade de uma fórmula em lógica de primeira ordem utilizando uma combinação de teorias, generalizando satisfabilidade proposicional através do suporte de funções não interpretadas, aritmética linear e não linear, vetor de *bit*, tuplas e outras teorias de decisão de primeira ordem [20].

Definição 2.8 *Dada uma teoria $\mathscr{T}$ e uma fórmula livre de quantificadores ψ, é dito que ψ satisfaz $\mathscr{T}$ se e, se somente se, existe uma estrutura que satisfaz tanto a fórmula ψ quanto as sentenças de $\mathscr{T}$, ou equivalentemente, se $\mathscr{T} \cup \{\psi\}$ for satisfeita.*

Definição 2.9 *Dado um conjunto $\Gamma \cup \{\psi\}$ de fórmulas de primeira ordem sobre uma teoria $\mathscr{T}$, é dito que ψ é uma $\mathscr{T}$- consequência de Γ, e é escrito $\Gamma \models_{\mathscr{T}} \psi$, se e somente se, todo modelo de $\mathscr{T} \cup \Gamma$ for também um modelo de ψ. Verificando $\Gamma \models_{\mathscr{T}} \psi$ pode ser reduzido de forma usual para verificação da $\mathscr{T}$-satisfabilidade de $\Gamma \cup \{\neg\psi\}$.*

Lucas et al. propõe uma notação padrão que será utilizada para descrever a gramática abaixo [8]. A Figura 2.3 mostra a notação proposta.

$$
\begin{array}{rcl}
Fml & ::= & Fml\ con\ Fml \mid \neg Fml \mid Atom \\
con & ::= & \wedge \mid \vee \mid \oplus \mid \Rightarrow \mid \Leftrightarrow \\
Atom & ::= & Trm\, rel\, Trm \mid Var \mid verdadeiro \mid falso \\
rel & ::= & < \mid \leq \mid > \mid \geq \mid = \mid \neq \\
Trm & ::= & Trm\, op\, Trm \mid \sim Trm \mid Var \mid Const \\
 & & \mid select(Trm, i) \mid store(Trm, i, v) \\
 & & \mid Extract(Trm, i, j) \mid SignExt(Trm, k) \mid ZeroExt(Trm, k) \\
 & & \mid ite(Fml, Trm, Trm) \\
op & ::= & + \mid - \mid * \mid / \mid rem \mid \ll \mid \gg \mid \& \mid \mid \mid \oplus \mid @
\end{array}
$$

Figura 2.3: Notação Padrão.

Nessa notação, *Fml* denota uma expressão booleana, *Trm* denota os termos construídos sobre inteiros, reais e vetor de *bit* enquanto *op* denota operadores binários. O conector lógico *con* consiste de conjunções ($\wedge$), disjunções ($\vee$), ou exclusivo ($\oplus$), implicação ($\Rightarrow$) e equivalência lógica ($\Leftrightarrow$). A interpretação dos operadores relacionais (por exemplo, $<$, $\leq$, $>$, $\geq$) e operadores não lineares (por exemplo, $*$, $/$, *rem*) dependem se os argumentos passados são vetores de *bit* com sinal ou sem sinal, inteiros ou reais (o operador *rem* denota a operação de

resto, com ou sem sinal, dependendo do argumento [22]). Os operadores de deslocamento para a direita e esquerda (»), «) dependem se um vetor de *bit* com sinal ou sem sinal é utilizado. Os operadores *bit* a *bit* são e (&), ou (|), ou exclusivo ($\oplus$), complemento ($\sim$), deslocamento para direita (»), e deslocamento para esquerda («). $Extract(Trm,i,j)$ denota a extração a partir do *bit i* até o *bit j* para formar um novo vetor de *bit* de tamanho $i-j+1$ enquanto o operador @ denota a concatenação de vetores de *bit*. $SignExt(Trm,k)$ aumenta o vetor de *bit* para um vetor de *bit* equivalente, com sinal, de tamanho *w*+*k*, onde *w* é o tamanho original do vetor de *bit*. $ZeroExt(Trm,k)$ aumenta o vetor de *bit* com zeros para um vetor de *bit* equivalente, sem sinal, de tamanho *w*+*k*. A expressão condicional $ite(f,t_1,t_2)$ recebe como argumentos uma fórmula booleana *f*, e dependendo do seu valor, seleciona o primeiro ou segundo argumento.

As teorias de vetores de solucionadores SMT são baseadas nos axiomas de McCarthy [23]. A função *select(a, i)* denota o valor de um vetor *a* na posição *i* e *store(a,i,v)* denota um vetor que é exatamente o mesmo vetor *a* exceto que o valor na posição *i* é *v*. Formalmente, as funções *select* e *store* podem ser caracterizadas pelos seguintes axiomas [16, 24, 25]:

$$i = j \Rightarrow select(store(a,i,v),j) = v$$
$$i \neq j \Rightarrow select(store(a,i,v),j) = select(a,j)$$

A verificação dos limites de um vetor precisam ser codificadas separadamente, pois as teorias de vetores utilizam a noção de tamanho de vetor ilimitado, porém os vetores em programas possuem tamanho limitado.

A teoria de tuplas fornecem operações de *store* e *select* similares as aplicadas em vetores, mas trabalham com tuplas como argumentos. Cada campo da tupla é representado por um inteiro constante. Logo, a expressão $select(t,f)$ denota o campo f da tupla t enquanto a expressão $store(t,f,v)$ denota a tupla t que, no campo f possui o valor v e todos os outros campos permanecem os mesmos.

O funcionamento de um solucionador SMT é demonstrado no exemplo a seguir. Seja a um vetor, b, c e d vetores de bit com sinal de tamanho 16, 32 e 32, respectivamente, e g uma função unária, que implica que se $x = y$ então $g(x) = g(y)$.

$$g(select(store(a,c,12),b+3)) \neq g(b-c+4)$$
$$\wedge\ b = c-3$$
$$\wedge\ c+1 = d-4$$

Porém b possui tamanho menor que c e, para realizar as operações $b-c+4$ e $b=c-3$, deve ser expandido para um vetor de bit equivalente de tamanho 32.

$$g(select(store(a,c,12),SignExt(b,16)+3)) \neq g(SignExt(b,16)-c+4)$$
$$\wedge\ SignExt(b,16) = c-3$$
$$\wedge\ c+1 = d-4$$

O resultante de $b' = SignExt(b,16)$ possui tamanho 32.

$$g(select(store(a,c,12),b'+3)) \neq g(b'-c+4)$$
$$\wedge\ b' = c-3$$
$$\wedge\ c+1 = d-4$$

Nesse momento, b e c possuem o mesmo valor. Uma forma de verificar se a fórmula é satisfatível é substituir b' por $c-3$, de forma que se obtém:

$$g(select(store(a,c,12),c-3+3)) \neq g(c-3-c+4)$$
$$\wedge\ c-3 = c-3$$
$$\wedge\ c+1 = d-4$$

Utilizando fatos da aritmética de vetores de bit:

$$g(select(store(a,c,12),c)) \neq g(1)$$
$$\wedge\ c-3 = c-3$$
$$\wedge\ c+1 = d-4$$

Aplicando a teoria de vetores, é possível resolver as operações de *select* e *store*:

$$g(12) \neq g(1)$$
$$\wedge\ c-3 = c-3$$
$$\wedge\ c+1 = d-4$$

Por fim, a fórmula se mostra satisfatível, pois existe uma atribuição de valores para as variáveis (por exemplo, $c = 5, d = 10$), que torna a formula verdadeira.

2.3 Sistema de Transições

Um sistema de transição de estados *M = (S, T, S_0)* é uma máquina abstrata, que consiste em um conjunto de estados S, onde $S_0 \subseteq S$ representa um conjunto de estados iniciais e $T \subseteq S \times S$

é a relação de transição. Um estado $s \in S$ consiste no valor do contador de programa *pc* e também nos valores de todas as variáveis do programa. Um estado inicial s_0 atribui a localização inicial do programa. As transições são identificadas como $\gamma = (s_i, s_{i+1}) \in T$ entre dois estados s_i e s_{i+1}, com uma fórmula lógica $\gamma(s_i, s_{i+1})$ que contém as restrições dos valores das variáveis do programa e do contador de programa.

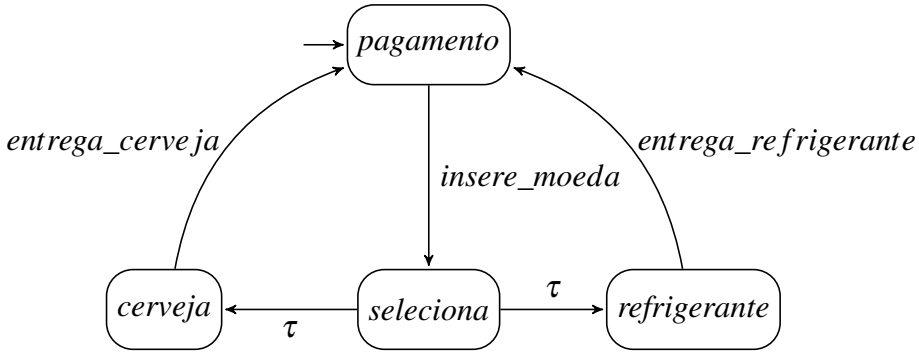


Figura 2.4: Sistema de transições de uma máquina de refrigerante [1].

Um sistema de transição é capaz de modelar o comportamento de diversos tipos de sistema. O exemplo clássico da máquina de bebidas é mostrado na Figura 2.4. A máquina de refrigerante representada pode entregar refrigerante ou cerveja. Neste sistema, estado inicial é representado por uma seta sem estado anterior. O espaço de estado é formado por $S = \{$*pagamento, seleciona, cerveja,refrigerante*$\}$ enquanto o estado inicial é $S_0 = \{$*pagamento*$\}$. A transição *insere_moeda* denota a ação do usuário de inserir uma moeda na máquina, e as transições *entrega_cerveja* e *entrega_refrigerante* representam as ações, por parte da máquina, de entregar uma cerveja e um refrigerante, respectivamente. A transição τ representa o funcionamento interno da máquina, a partir da escolha do usuário.

2.4 A Técnica *Bounded Model Checking*

No ESBMC, o programa que está sendo analisado é modelado por um sistema de transição de estados, que é gerado a partir de um grafo de fluxo de controle do programa (*Control-Flow Graph*, CFG) [26]. O grafo de fluxo de controle do programa é gerado automaticamente, durante o processo de verificação. Um nó no CFG representa uma atribuição (determinística ou não determinística) ou uma expressão condicional, enquanto que uma aresta representa uma mudança no fluxo do programa.

Dado um sistema de transição M, uma propriedade ϕ e um limite k, o ESBMC desdobra o sistema k vezes e transforma o resultado em uma condição de verificação (VC, do inglês *verification condition*) ψ, de tal forma que ψ é satisfeita se, e se somente se, ϕ possuir um contraexemplo de comprimento menor ou igual a k [8]. O problema da técnica BMC é então mostrado na Figura 2.4.

$$\psi_k = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{2.4}$$

onde ϕ é uma propriedade, I é o conjunto de estados iniciais de M e $\gamma(s_j, s_{j+1})$ é a relação de transição de M entre os passos j e $j+1$. Logo, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ representa a execução de M, i vezes, e a equação (2.4) só poderá ser satisfeita se, e somente se, para um $i \leq k$, existir um estado alcançável no passo em que ϕ é violada. Se a equação (2.4) é satisfeita, então o ESBMC mostra um contraexemplo, definindo quais os valores das variáveis necessários para reproduzir o erro. O contraexemplo para uma propriedade ϕ é uma sequência de estados $s_0, s_1, \cdots, s_k$ com $s_0 \in S_0$, e $\gamma(s_i, s_{i+1})$ com $0 \leq i < k$. Se a equação (2.4) não for satisfeita, pode-se concluir que nenhum estado com erro é alcançável em k ou menos passos.

A Figura 2.5 mostra um exemplo da aplicação da técnica BMC em um programa, mostrado na Figura 2.5a. A Figura 2.5b mostra os estados gerados a partir do código. Nesta, cada estado representa uma nova atribuição a uma variável. Os estados s_0, s_1 e s_2 apresentam a inicialização das variáveis do programa, N e M com valores não determinísticos sem sinal e $i = 0$. Em seguida, o laço *for* gera $2M$ estados, pois são gerados dois estados para cada iteração, um para a atribuição do vetor a e um para o incremento do contador i. Neste exemplo, são gerados $2M + 3$ estados, os três primeiros são inicialização das variáveis e o restante é gerado no laço *for*. A Figura 2.5c mostra o exemplo de uma propriedade que pode ser verificada com a técnica. Nesta, dado um estado s_k, o valor k deve ser maior que 0 e menor que o limite do vetor, que possui tamanho N. A técnica tenta provar que um programa tem defeitos então nega a propriedade $\phi(s_k)$ durante a verificação. Se $s_0 \wedge \ldots \wedge s_k \wedge \neg\phi(s_k)$ é satisfatível, então o programa possui um defeito (neste exemplo, ocorre um estouro do limite de vetor, se $M \geq N$).

```
1 int main() {
2   unsigned int N, a[N], M;
3   for(unsigned int i=0; i<M; ++i)
4     a[i] = 1;
5   return 0;
6 }
```

(a) Exemplo de programa.

$$
\begin{array}{rrcl}
s_0: & N & = & nondet\_uint \\
s_1: & M & = & nondet\_uint \\
s_2: & i_0 & = & 0 \\
s_3: & a[i_0] & = & 1 \\
s_4: & i_1 & = & i_0 + 1 \\
s_5: & a[i_1] & = & 1 \\
s_6: & i_2 & = & i_1 + 1 \\
\ldots & & & \\
s_{2M+1}: & a[i_{\frac{M-3}{2}}] & = & 1 \\
s_{2M+2}: & i_{\frac{M-2}{2}} & = & i_{\frac{M-3}{2}} + 1
\end{array}
$$

(b) Estados gerados a partir do programa da Figura 2.5a.

$$\phi(s_k): \quad i_k > 0 \wedge i_k \leq N$$

(c) Uma propriedade gerada a partir do programa da Figura 2.5a.

Figura 2.5: Exemplo da aplicação da Técnica BMC.

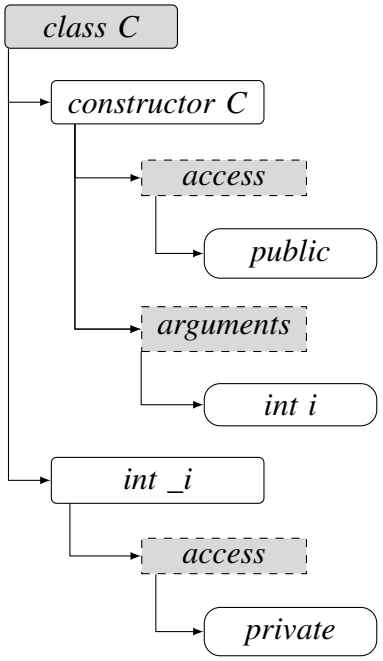
2.5 CBMC (*C Bounded Model Checker)*

O CBMC implementa BMC para programas C/C++ utilizando solucionadores SAT/SMT [6]. Ele processa programas C/C++ utilizando a ferramenta *goto-cc* [27], que compila o programa C/C++ em um programa *goto* equivalente ao original utilizando um estilo compatível com o GCC. O programa GOTO pode então ser processado por uma máquina de execução simbólica. Alternativamente, CBMC utiliza seu próprio *parser* interno, baseado em Flex/Bison [6], para processar os arquivos C/C++ e para construir a árvore sintática abstrata (AST, do inglês *abstract syntax tree*). O Verificador de Tipo no *front-end* do CBMC anota a AST com os tipos e gera uma tabela de símbolos. A classe IREP (do inglês *Intermediate Representation*), do CBMC então converte a AST anotada em um formato interno, independente de linguagem, que será utilizado nas fases subsequentes do *front-end*. O ESBMC modifica o *front-end* para

tratar as definições da biblioteca padrão do C++ enquanto outras funcionalidades (por exemplo, polimorfismo, *template* e tratamento de exceções) são tratadas internamente.

```
1   class C
2   {
3     public:
4       C(int i);
5     private:
6       int _i;
7   };
```

(a) Código exemplo.



(b) Representação da IREP gerada a partir do código na Figura 2.6a.

Figura 2.6: Exemplo de criação de IREP.

A Figura 2.6b mostra um exemplo de IREP gerada a partir do código apresentado na Figura 2.6a. Na Figura 2.6a é mostrada uma classe *C* composta por um construtor e uma variável do tipo *int*, chamada *_i*. O construtor da classe *C* possui um argumento, do tipo *int*, chamado *i*. A Figura 2.6b mostra uma representação gráfica da IREP gerada a partir do código da Figura 2.6a. É criada uma estrutura hierárquica, onde o nome da IREP é o nome da classe, e o construtor e a variável interna são IREPs internas da IREP da classe. Além disso, as IREP dos atributos e métodos devem possuir qual o tipo de acesso (público, privado ou protegido) e, para um método ou função, a IREP de conter ainda IREPs para os argumentos e para o retorno (porém, construtores e destrutores não possuem retorno). As IREPs também podem possuir informações extras, como o valor da variável e/ou arquivo e linha onde foi declarada.

CBMC e ESBMC usam duas funções *C* e *P*, que computam as restrições (***C**onstraints*), como suposições e atribuições de variáveis, e propriedades (***P**roperties*) como condições de segurança e assertivas definidas pelo usuário, respectivamente. A Figura 2.7 mostra um exemplo de restrições e propriedades geradas a partir do programa da Figura 2.5a. As restrições *C* são codificadas através da conjunção dos estados do programa e as propriedades *P* são codificadas

através da conjunção das propriedades do programa. Neste exemplo, deseja-se verificar estouro no acesso ao vetor logo,o solucionador SMT deverá procurar uma solução para a fórmula $C \wedge \neg P$. Caso seja satisfatível, um defeito de acesso ao vetor foi encontrado no programa.

$$C := \left[\begin{array}{l} N = nondet\_uint \wedge\ M = nondet\_uint \wedge\ i_0 = 0 \\ \wedge\ a[i_0] = 1 \wedge\ i_1 = i_0 + 1 \wedge\ a[i_1] = 1 \wedge\ i_2 = i_1 + 1 \\ \wedge \ldots \wedge\ a[i_{\frac{M-3}{2}}] = 1 \wedge\ i_{\frac{M-2}{2}} = i_{\frac{M-3}{2}} + 1 \end{array} \right]$$

$$P := \left[\ i_k > 0 \wedge i_k \leq N\ \right]$$

Figura 2.7: Restrições e propriedades geradas a partir do programa da Figura 2.5

Ambas as ferramentas, CBMC e ESBMC, geram condições de verificação, como por exemplo, estouros aritméticos, violação dos limites de um vetor e de-referência de ponteiro nulo (*NULL pointer dereference*). Um gerador de VC (VCG, do inglês *Verification Condition Generator*) então deriva VCs a partir delas.

2.6 LLBMC (*Low Level Bounded Model Checker)*

O LLBMC (*Low-Level Bounded Model Checker*) é um verificador de programas C/C++ sequenciais, aplicando a técnica BMC que utilizam solucionadores SMT [11]. Porém, diferentemente do CBMC e ESBMC, o processo de verificação não utiliza o código fonte do programa mas a representação intermediária gerada pelo compilador [28]. A ferramenta é capaz de verificar diversos defeitos, entre eles *overflow* e *underflow* aritmético, divisão por zero, deslocamento inválido de bits, acesso ilegal de memória (por exemplo, violação de limite de vetor), além de assertivas inseridas pelo desenvolvedor.

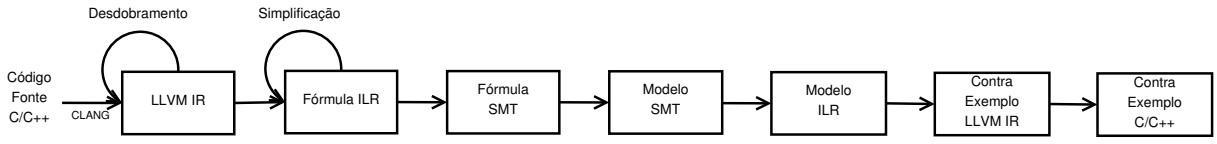


Figura 2.8: Arquitetura do LLBMC.

A Figura 2.8 mostra a abordagem utilizada pelo LLBMC para verificar programas C/C++ [28]. O programa C/C++ a ser verificado é primeiramente compilado em uma representação intermediária LLVM (LLVM IR), utilizando o compilador CLANG [29]. O desdobramento dos laços até um valor definido pelo usuário (ou determinado automaticamente) é realizado através de transformações diretamente na LLVM IR. Em seguida, a LLVM IR é transformada em uma

representação intermediária lógica (LLVM ILR), que estende a lógica QF_ABV [30] utilizando expressões para codificar as verificações feitas pelo LLBMC.

Na próxima etapa, a fórmula ILR é simplificada utilizando um conjunto de regras de simplificação. Essas regras são suficientes para descartar várias expressões "fáceis", antes serem passadas ao solucionador SMT. Ao final da simplificação, a fórmula ILR é transformada em fórmula SMT, que é solucionada utilizando o solucionador SMT STP [31].

Se a fórmula for satisfatível, então será gerado um modelo SMT correspondente contendo um erro. O modelo SMT é convertido em um modelo ILR, que é utilizado para construir um contraexemplo em LLVM IR. Atualmente, os desenvolvedores estão trabalhando para estender o LLBMC para apresentar um contraexemplo em C/C++, pois este depende das informações de *debug* inseridas pelo CLANG durante a conversão do programa em LLVM IR.

2.7 Laços Invariantes

Os laços invariantes são utilizados para provar a corretude de programas [32]. Um laço invariante é dividido em três etapas:

- **Inicialização**: Deve ser válido antes da primeira iteração.
- **Manutenção**: Se for verdadeiro antes de uma iteração, ele será verdadeiro antes da próxima iteração.
- **Término**: Quando o laço termina, o invariante fornece uma propriedade útil que ajuda a provar que o algoritmo está correto.

Quando as duas primeiras propriedades são verdadeiras, o laço invariante é verdadeiro antes de toda iteração do laço. Existe uma similaridade à indução matemática onde, para provar que uma propriedade é verdadeira, deve-se mostrar o caso base e o passo indutivo [32]. Nesse caso, a prova de que a propriedade é verdadeira antes da primeira iteração é equivalente ao caso base, e a prova de que a propriedade é verdadeira para qualquer iteração é equivalente ao passo indutivo. A terceira propriedade é a mais importante, pois o laço invariante tenta provar a corretude de um algoritmo. O laço invariante difere da indução matemática usual, onde o passo indutivo é utilizado indefinidamente. No laço invariante, a indução finaliza quando o

laço termina. Um exemplo de laço invariante é mostrado na Figura 2.9. Neste exemplo, o laço invariante pode ser verificado através do *assert*.

```
1 int main() {
2   unsigned int i, n=nondet_uint(), sn=0;
3   assume(n>=1);
4   for(i=0; i<=n; i++)
5     sn = sn + i;
6   assert(sn==(n*(n+1))/2)
7 }
```

Figura 2.9: Exemplo de laço invariante.

As etapas do laço invariante na Figura 2.9 são:

- **Inicialização**: a variável n será um valor não determinístico sem sinal, ou seja, $n > 0$. Dessa forma, a condição do laço *for* é sempre verdadeira.
- **Manutenção**: ao final de cada iteração, o valor de i é incrementado, até que $i = n + 1$.
- **Término**: quando $i = n + 1$, a condição do laço *for* será falsa, e será finalizado.

2.8 Lógica Hoare

A ideia básica por trás da Lógica Hoare é que um programa de computador é uma ciência exata e todas as consequências de execução em qualquer ambiente podem, em princípio, ser encontradas a partir do código desse programa, por meio de pensamento dedutivo [33].

Em diversos casos, a validade do resultado de um programa dependerá dos valores assumidos pelas variáveis antes do programa executar. A notação proposta por Hoare cria uma relação entre as pré-condições (P), um determinado código, composto por uma sequência de comandos (Q), e o resultado da execução (R), conforme a Fórmula 2.5.

$$\{ P \} Q \{ R \} \tag{2.5}$$

A sentença pode ser interpretada como "se a assertiva P é verdadeira antes de Q, então R será verdadeira ao término de Q". Em caso particular, onde não existem pré-condições, a sentença muda, conforme a Fórmula 2.6.

$$\{ \textbf{verdadeiro} \} Q \{ R \} \tag{2.6}$$

A atribuição é a funcionalidade mais comum em programas. O axioma da atribuição define que, o valor de uma variável x, depois de executar um comando de atribuição $x{:=}t$, se torna o valor da expressão t no estado antes da atribuição. O axioma de atribuição é definido na Fórmula 2.7.

$$\{\, p\,[t/x]\,\}\, x := t\,\{\, p\,\} \tag{2.7}$$

onde x é o identificador de uma variável, p é uma assertiva e a notação $p[t/x]$ denota o resultado de substituir o termo t por todas as ocorrências livres de x em p.

A regra da composição define uma sequência de operações, executadas uma após a outra. As expressões devem ser separadas por um símbolo ou equivalente de forma a denotar uma composição procedural: $(Q_1; Q_2; ...; Q_n)$. A regra da composição é definida na Fórmula 2.8.

$$\frac{\{\, p\,\}\, S_1\, \{\, r\,\},\, \{\, r\,\}\, S_2\, \{\, q\,\}}{\{\, p\,\}\, S_1, S_2\, \{\, q\,\}} \tag{2.8}$$

onde p, q, r são assertivas (na primeira expressão, p é a pré-condição e r é a pós-condição, enquanto que na segunda expressão, r é a pré-condição e q é a pós-condição) e S_1 e S_2 são códigos ou sequências de comandos.

A regra do se-então-senão define a operação lógica de decisão binária, em que uma condição define qual trecho de código deve ser executado. Esta regra é definida na Fórmula 2.9.

$$\frac{\{\, p \wedge e\,\}\, S_1\, \{\, q\,\}, \{\, p \wedge \neg\, e\,\}\, S_2\, \{\, q\,\}}{\{\, p\,\}\ \textbf{se}\ e\ \textbf{então}\ S_1\ \textbf{senão}\ S_2\ \textbf{fim-se}\ \{\, q\,\}} \tag{2.9}$$

onde p, q são assertivas, S_1 e S_2 são códigos ou sequências de comandos e e é um condicionante, de forma que assegura a execução de q após S_1, caso a fórmula $(p \wedge e)$ for satisfeita, e assegura a execução de q após S_2, caso a fórmula $\{p \wedge \neg e\}$ for satisfeita.

A regra do enquanto define as operações iterativas, compostas por um valor inicial, uma condição de parada e um incremento sobre a condição inicial. Esta regra é definida na Fórmula 2.10.

$$\frac{\{\, p \wedge e\,\}\, S\, \{\, p\,\}}{\{\, p\,\}\ \textbf{enquanto}\ e\ \textbf{faça}\ S\ \textbf{fim-faça}\ \{\, p \wedge \neg\, e\,\}} \tag{2.10}$$

onde p é uma assertiva, S é um código ou sequência de comandos e e é um condicionante, de forma que, caso S mantenha a expressão $\{p \wedge e\}$ satisfeita, então a execução de S manterá a expressão satisfeita em qualquer número de iterações. A regra do enquanto também expressa o fato de que, após o comando enquanto terminar, a expressão $\{\, p \wedge \neg e\,\}$ deverá ser satisfeita, ou seja, o condicionante e passa a ser falso.

2.9 Resumo

Neste capítulo, inicialmente foram explicados as teorias que servem de base para a verificação formal de programas. Na Lógica Proposicional são definidos a sintaxe da gramática utilizada pela lógica, assim como os operadores e operações lógicas, e regras de precedência. Em seguida, a Teoria do Módulo de Satisfabilidade apresentou a notação padrão da teoria, assim como as funções *select* e *store*, que são as formalizações das formas de acesso e atribuições de valores em vetores.

Após a Teoria do Módulo de Satisfabilidade, foi apresentado o conceito de sistema de transição. Os sistemas de transição são autômatos utilizados para modelagem do comportamento de um sistema. Em seguida, foi mostrada técnica BMC, modelada a partir de um sistema de transições, que realiza uma conjunção entre um estado inicial, os estados do sistema e a negação de uma propriedade. A negação da propriedade deriva do fato de que a técnica BMC tenta sempre provar a negação das propriedades em um programa. A próxima seção apresentou o CBMC, um *front-end* utilizado pelo ESBMC para verificação de programas C e C++ utilizando solucionadores SAT/SMT.

Em seguida, foi mostrado o conceito de laço invariante, e as três propriedades para corretude da invariância, inicialização, manutenção e término. Por fim, é mostrada a Lógica Hoare, que cria uma relação entre pré- e pós-condições, assim como o axioma e as três regras da lógica, regra da composição, regra do se-então-senão e a regra do enquanto.

A compreensão dos assuntos contidos neste trabalho requerem o entendimento dos assuntos neste capítulo. A técnica BMC e solucionadores SMT são a base da verificação aplicada pelo ESBMC, ferramenta utilizada no desenvolvimento deste trabalho. O entendimento da lógica Hoare é crucial para o trabalho desenvolvido no Capítulo 5, visto que o algoritmo de indução matemática, bem como as transformações efetuadas no programa, são formalizadas utilizando essa teoria.

Capítulo 3

Trabalho Relacionados

Nesse capítulo, serão mostrados trabalhos com objetivos semelhantes ao desenvolvidos nesta dissertação. Está dividido em duas seções, na Seção 3.1 serão apresentados trabalhos relacionados à verificação de programas C++, enquanto que na Seção 3.2, serão apresentados trabalhos relacionados à prova de corretude de programas por indução matemática.

3.1 Verificação de Programas C++

Os trabalhos descritos nesta seção, serão avaliados quanto à necessidade de conversão do programa C++ em outra linguagem antes do início do processo de verificação e quanto às funcionalidades suportadas no processo de verificação.

A necessidade de conversão do programa C++ em outra linguagem facilita o processo de verificação, pois os verificadores C++ ainda estão nos estágios iniciais de desenvolvimento e existem ferramentas de verificação de outras linguagens, que vem sendo desenvolvidas ao longo dos anos e são mais estáveis (por exemplo, ferramentas de verificação de programas C). Apesar disso, a conversão pode inserir ou remover defeitos presentes no programa original e o resultado da verificação do programa convertido pode apresentar resultados diferentes da verificação direta do programa C++.

Em relação às funcionalidades, nota-se que a grande parte dos trabalhos sobre verificadores de programas C++, focam em funcionalidades específicas, como tratamento de exceção, e acabam por negligenciar outras funcionalidades de igual importância, como a verificação da Biblioteca de *Template* Padrão (STL). A Tabela 3.1 mostra um comparativo entre os trabalhos

relacionados e o trabalho proposto nesse documento.

Trabalho relacionado	Conversão em outra linguagem	Funcionalidades da linguagem C++ suportadas	
		Exceções	STL
Prabhu et al. [14]	Linguagem C	Sim	Não mencionado
Blanc et al. [13]	Não	Não mencionado	Sim
Merz et al. [11]	Linguagem intermediária LLVM	Não	Sim
Trabalho proposto	Não	Sim	Sim

Tabela 3.1: Tabela comparativa entre os trabalhos relacionados.

Prabhu et al. [14] apresentam uma análise de exceção interprocedural e um *framework* de transformação para programas C++, que registra o fluxo do programa criado pelas exceções e cria um programa C livre de exceções. A criação do programa livre de exceções inicia com a geração de um grafo de fluxo de controle, denominado *modular interprocedural exception control-flow graph* ou IECFG. O IECFG é então analisado por um algoritmo desenvolvido pelos autores que modela o conjunto das possíveis exceções que podem se conectar a blocos *catch* utilizando uma representação compactada, chamada *Signed-TypeSet*. O resultado da modelagem é utilizado então para gerar o programa C livre de exceções, que simula o comportamento do lançamento e captura de exceções através da atribuição do objeto lançado a um objeto local (que é atribuído ao objeto da declaração do bloco *catch*), e do uso de instruções GOTO. Ao final do processo de conversão, o programa C é verificado utilizando a ferramenta F-SOFT [10]. Como experimentos, foram utilizados 18 programas C++ [34] e 4 aplicações comerciais. A verificação desses casos de teste é focada em somente duas propriedades: "*no throw*" (a porcentagem do código que não gera exceções) e "*no leak*" (o número de vazamento de memória nos blocos *try* [14]). Os autores não indicam se a abordagem é capaz de verificar outros defeitos além das duas propriedades descritas. São mostradas as regras de conexão e formalização da verificação utilizando tratamento de exceções porém, além do trabalho focar somente nessa funcionalidade, um importante aspecto do tratamento de exceções não é citado, a especialização de exceções. Na especificação de exceções, é possível definir as exceções que uma função ou método pode lançar. Neste trabalho, são mostradas todas as regras de conexão e formalização da verificação de programas que utilizam tratamento de exceções além do processo de verificação de programas que utilizem especificação de exceções.

Blanc et al. descrevem a verificação de programas C++ que utilizam contêineres através de abstração de predicado [13]. Para oferecer suporte a programas C++ que façam uso da Biblioteca *Template* Padrão é proposto um modelo operacional simplificado, o qual é modelado utilizando a lógica Hoare. O objetivo do modelo operacional é simplificar o processo de verificação utilizando a ferramenta SATABS [35]. O SATABS é uma ferramenta de verificação de programas C e C++ que oferece suporte a objetos, sobrecarga de operadores, referências e *templates* (porém sem suporte a especificação parcial). Os autores mostram que é suficiente para verificar a corretude de um programa utilizando um modelo operacional através da prova de que, se as pré- e pós-condições se sustentarem, o modelo se sustenta. A abordagem é eficiente em encontrar erros triviais em programas C++. São modeladas as pré-condições necessárias para a verificação dos diversos contêineres da biblioteca utilizando um modelo operacional, semelhante ao modelo utilizado pela ferramenta ESBMC para a mesma finalidade. Os dois trabalhos formalizam os modelos operacionais utilizando lógicas diferentes, enquanto Blanc et al. utilizam a lógica Hoare, o presente trabalho utiliza lógica de primeira ordem. Em relação ao modelo operacional, enquanto o descrito por Blanc et al. apenas apresentam as pré-condições, o modelo operacional da ferramenta ESBMC implementa tanto as pré- quanto as pós-condições, o que aumenta o espectro de aplicações que podem ser corretamente verificadas pela ferramenta

Merz et al. descrevem a ferramenta LLBMC, que aplica BMC para verificação de programas C++ [11]. A ferramenta precisa converter o programa para a representação intermediária do LLVM, utilizando o compilador CLANG [29]. Essa conversão inicial acaba por remover informações de alto nível sobre a estrutura do programa C++ (por exemplo, o relacionamento entre classes), porém os trechos de códigos que utilizam a Biblioteca de *Template* Padrão são tornados explícitos, o que facilita o processo de verificação. A partir da representação intermediária do LLVM, é gerada uma fórmula lógica interna da ferramenta LLBMC. Esta fórmula é simplificada e passada a um solucionador SMT para verificação. O LLBMC é a ferramenta que mais se aproxima de oferecer a verificação de todas as funcionalidades da linguagem C++, porém a ferramenta não verifica programas com tratamento de exceções, o que torna difícil a verificação realística de programas C++, pois devem ser desabilitadas durante a geração para a representação intermediária do LLVM. A maior diferença a ferramenta descrita pelos autores e a proposta neste trabalho, se apresenta no início do processo de verificação. No LLBMC, é necessária a conversão do programa original em uma representação intermediária LLVM. O maior entrave desta abordagem é a necessidade de adequação constante da ferramenta para as novas

versões da representação intermediária LLVM, gerada pelo compilador CLANG, que não mantido pelos autores da ferramenta LLBMC. Devido a sua grande abrangência, esta ferramenta será utilizada para comparação de resultados com a ferramenta descrita neste trabalho, e uma análise comparativa mais detalhada pode ser encontrada na Seção 4.4.

Dos trabalhos citados, apenas o trabalho proposto não requer qualquer tipo de transformação do programa original e oferece suporte programas com tratamento de exceção, herança e polimorfismo e STL.

3.2 Prova de Corretude por Indução Matemática

Os trabalhos descritos nesta seção, serão avaliados quanto ao suporte de linguagem, se a ferramenta utiliza o algoritmo *k-induction* e se o processo de verificação é automatizado.

O suporte de linguagem está relacionado ao tipo de programa que está sendo verificado. Grande parte dos trabalhos relacionados à prova por indução matemática, estão ligados a prova de corretude em projetos de *hardware*, que são escritos utilizando linguagens como SystemC [36]. A segunda característica avaliada está ligada ao algoritmo de prova por indução, que pode ser ou não o algoritmo *k-induction* e, por fim, será avaliado se o processo é totalmente automatizado, sem que haja necessidade do desenvolvedor alterar o programa para que o processo de verificação possa ocorrer. A Tabela 3.2 mostra um comparativo entre os trabalhos relacionados e o trabalho proposto nesta dissertação.

Trabalho relacionado	Linguagem	Algoritmo *k-induction*	Processo automatizado
Niklas et al. [37]	SMV	Não (*temporal induction*)	Sim
Daniel et al [36]	SystemC	Sim	Sim
Alastair et al. [38]	C	Sim	Sim
Alastair et al. [39]	C/Dafny	Sim	Não
Trabalho proposto	C/C++	Sim	Sim

Tabela 3.2: Tabela comparativa entre os trabalhos relacionados.

Niklas et al. propõem uma forma de verificação baseada em solucionador SAT para checar propriedades em projetos de *hardware* (modelados como máquinas de estado finitos), utilizando indução temporal [37]. Foram implementadas políticas de otimização para diminuir o espaço de estados explorados durante a prova por indução matemática, o que se diminuiu consideravelmente o tempo de verificação de modelos. A abordagem de verificação das pro-

priedades foi implementada na ferramenta TIP e foram verificados casos de teste escritos no formato SMV (*symbolic model verification*) [40]. O algoritmo de indução matemática utiliza dois passos para prova (caso base e passo indutivo). A principal conclusão do trabalho é que a verificação iterativa, a partir de um limite de iterações baixo é, na maioria da vezes, mais eficaz do que a verificação utilizando um limite de iterações aleatório [37]. A diferença entre o algoritmo proposto pelos autores está no passo indutivo. Neste, existe uma junção do que esta dissertação chama de condição adiante. O passo indutivo não faz nenhuma transformação no programa original e a condição de prova do algoritmo é uma conjunção entre a negação do resultado passo indutivo e o resultado da condição adiante. A proposta do artigo foca na verificação de programas SMV, o que diminui o espectro de aplicações da ferramenta proposta.

Daniel et al. descrevem uma forma de provar propriedades de projetos TLM (*Transaction Level Modeling*) na linguagem SystemC [36]. A abordagem consiste de três passos, iniciando com a transformação do programa em SystemC para programa C, seguido do mapeamento das propriedades TLM em lógica para monitoração, utilizando assertivas no programa C e máquinas de estado finitos e, verificação do programa C utilizando a ferramenta de verificação CBMC [6]. No CBMC, a prova das propriedades é feita por indução matemática, utilizando o algoritmo *k-induction*. O algoritmo *k-induction* descrito neste artigo é dividido em três passos (caso base, condição adiante e passo indutivo). Um programa produtor-consumidor é testado porém com diversas restrições. O aumento do número de produtores e consumidores faz com que a ferramenta não finalize a verificação no tempo determinado, e a técnica implementada somente consegue provar corretude para um número limitado de iterações. O algoritmo *k-induction* utilizado pelos autores é semelhante ao descrito nesta dissertação, utilizando três passos (caso base, condição adiante e passo indutivo). A diferença entre os trabalhos está nas transformações feitas no algoritmo proposto pelos autores e o proposto neste trabalho, durante a condição adiante. No algoritmo proposto no artigo, durante a condição adiante, transformações semelhantes as que ocorrem no passo indutivo são introduzidas no código, a fim de verificar se existe um caminho entre um estado inicial e o estado atual k, enquanto que no algoritmo proposto nesta dissertação, uma assertiva é inserida ao final do laço a fim de verificar se todos os estados foram alcançados em k passos. Além disso, a ferramenta converte o programa SystemC em programa C, antes de iniciar o processo de verificação, etapa que não acontece no trabalho proposto.

Alastair et al. descrevem uma ferramenta de verificação chamada Scratch [38], para

detectar corrida de dados durante acesso direto de memória (DMA, do inglês *Direct Memory Access*) em processadores CELL BE da IBM [41]. A abordagem utilizada para verificação de programas, faz uso da técnica de *k-induction*. A ferramenta insere assertivas no programa verificado para modelar o comportamento do controle de fluxo da memória, e tenta provar a corretude do programa, utilizando o algoritmo *k-induction*. O algoritmo *k-induction* implementado na ferramenta Scratch utiliza dois passos, caso base e passo indutivo. A ferramenta é capaz de provar a ausência de corridas de dados utilizando a técnica, porém está restrita a verificar essa classe específica de problemas para um tipo específico de *hardware*. Diferentemente do algoritmo proposto neste trabalho, é proposto pelos autores o algoritmo *k-induction* com dois passos, caso base e passo indutivo. Porém, os passos do algoritmo são semelhantes ao proposto nesta dissertação, onde transformações são feitas durante o passo indutivo para provar a corretude do programa. A ferramenta oferece suporte à linguagem C, a qual é utilizada para programar os processadores, porém está restrita a uma classe bastante restrita de problemas. No trabalho proposto, além de ser possível verificar programas C e C++, uma classe mais genérica de problemas é avaliada, de laços invariantes.

Alastair et al. descrevem duas ferramentas para provar corretude de programas: K-Boogie e K-Inductor [39]. O K-Boogie é uma extensão para o verificador de linguagem Boogie [42] e permite a prova de corretude utilizando *k-induction* de diversas linguagens de programação, incluindo Boogie, Spec, Dafny, Chalice, VCC e Havoc. K-Inductor é um verificador de programas C, construído sobre a ferramenta de verificação CBMC [6]. Os autores utilizam um algoritmo chamado *combined-case k-induction*, que utiliza o caso base e passo indutivo, porém são feitas otimizações no código de forma a remover laços aninhados, transformando-os em programas com somente um laço. O K-Inductor foi utilizado para verificar os mesmos casos de teste apresentados em um trabalho anterior dos autores, na qual a ferramenta Scratch é apresentada [38]. O K-inductor apresentou resultados semelhantes em termos de verificação correta dos programas, porém com ganhos na velocidade de verificação. A diferença entre o trabalho dos autores do artigo e o trabalho proposto surge ao analisar o algoritmo de indução matemática, que utiliza dois passos, caso base e passo indutivo. A falta da condição adiante no algoritmo proposto pelos autores não interfere no processo de verificação dos casos descritos no artigo pois, além de utilizar os mesmos casos de teste de um trabalho anterior dos autores [38], os casos de teste são alterados manualmente, de forma a inserir o modelo de corridas de dados.

Dos trabalhos citados, apenas o trabalho proposto suporta as linguagens C e C++, uti-

liza o algoritmo *k-induction* e realiza um processo totalmente automatizado de verificação de programas com laços invariantes.

3.3 Resumo

Neste capítulo, as características de trabalhos relacionados foram estabelecidas e comparadas. Na primeira seção, foram avaliados os trabalhos relacionados à Linguagem de programação C++. Os parâmetros avaliados foram: se há necessidade de conversão da linguagem e quais as funcionalidades suportadas na verificação de programas C++. Em seguida, os trabalhos foram avaliados e mostrou-se que as ferramentas estão fortemente voltadas à verificação de funcionalidades específicas e que a ferramenta estado da arte que mais se aproxima do ESBMC, peca no suporte de tratamento de exceções.

Na segunda seção, foram avaliados trabalhos relacionados a prova de corretude de programas por indução matemática. Os parâmetros avaliados foram: qual linguagem suportada no processo de verificação, se a ferramenta utiliza o algoritmo *k-induction* e se o processo de verificação é totalmente automatizado (i.e., se não há necessidade de alterações manuais no programa). Em seguida, os trabalhos foram avaliados e mostrou-se que as ferramentas de prova de corretude estão fortemente ligadas a verificação de programas ligados a *hardware* e a ferramenta estado da arte que mais se aproxima do ESBMC, não possui um processo totalmente automatizado, o que torna difícil a verificação de programas grandes, pois necessita que o desenvolver insira as propriedades a serem verificadas diretamente no programa.

Esta dissertação apresenta uma ferramenta que mais se aproxima das ferramentas LLBMC (devido ao número de funcionalidades suportadas na verificação de programas C++) e K-Inductor (devido ao suporte de verificação de laços invariantes), porém oferecendo suporte à prova por indução matemática de programas C e C++, de forma totalmente automatizada e oferecendo suporte a tratamento de exceções e *templates* (focando principalmente no suporte a verificação da Biblioteca de *templates* Padrão).

Capítulo 4

Uma Abordagem para Verificação de Programas C++

A linguagem C++ é uma linguagem de nível intermediário (ou seja, pode ser utilizada para programação tanto alto nível quanto baixo nível), estaticamente tipada e multiparadigmas [43]. Foi criada em 1979 por Bjarne Stroustrup na Bell Labs, com o objetivo de estender a linguagem C, introduzindo conceitos de orientação a objeto, como classes, herança e polimorfismo, entre outras funcionalidades [44].

Nesse capítulo, serão mostradas certas funcionalidades providas pela linguagem e como foram desenvolvidas para oferecer o suporte adequado ao processo de verificação utilizando a ferramenta ESBMC. Ao final, serão mostrados os resultados da verificação de diversos casos de teste, desenvolvidos em C++, utilizando a ferramenta ESBMC e a ferramenta LLBMC, outra ferramenta estado da arte de verificação [11].

4.1 *Templates*

Os *templates* são uma importante funcionalidade da linguagem C++. Eles são utilizados para definir funções ou classes de tipo de dado genérico, que podem ser instanciados com tipos de dados quaisquer no desenvolvimento de um programa. A maior vantagem do uso de *templates* é que não se faz necessário o desenvolvimento de diversas funções ou classes, para cada tipo de dado requeridos por um programa.

Templates são o núcleo da Biblioteca de *Template* Padrão (STL). Eles são usados para

definir as classes e funções da biblioteca, para que todos os contêineres e algoritmos possam ser utilizados com uma vasta gama de tipos de dados e classes, por exemplo, utilizando um *vector* para armazenar objetos do tipo *int* e outro *vector* para armazenar objetos do tipo *float*.

Os *templates* não são objetos de tempo de execução [43]. Uma vez que o programa é compilado, os *templates* geram funções e classes com os tipos de dados instanciados e são removidos do executável final. No ESBMC o processo é similar, *templates* são somente utilizados até a fase de checagem de tipo, onde todos os *templates* são instanciados. No processo de instanciação de um *template*, uma nova função ou classe é gerada, com um tipo de dado específico. As funções e classes instanciadas não são *templates* e as representações intermediárias (IREP) criadas, serão utilizadas nas próximas fases da verificação. Ao final da fase de checagem de tipo, todos os *templates* são descartados.

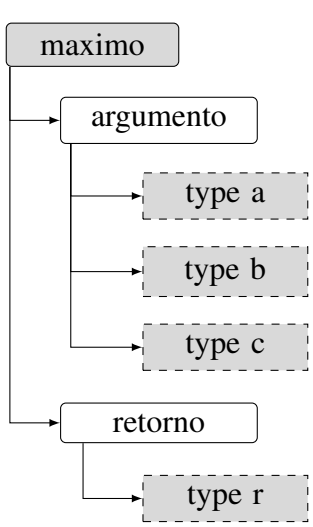
A verificação de programas C++ contendo *templates* é essencialmente dividida em duas etapas: criação de *templates* e instanciação de *templates*. A criação de *templates* ocorre na etapa de *parser*. Nesta, a IREP da função ou classe é criada e o tipo de dado é marcado como genérico. Neste momento, nenhuma outra representação é criada, pois neste ponto o ESBMC não sabe com quais tipos o *template* será instanciado.

A instanciação de *templates* ocorre quando, no programa C++, é feito o uso do *template*, utilizando um tipo de dado. Nesse momento, o ESBMC faz uma busca se já foi criada uma IREP com o tipo de dado utilizado. Em caso positivo, essa IREP é utilizada no processo de instanciação (isso evita a criação de IREPs duplicadas, diminuindo os requisitos de memória da ferramenta). Caso não haja uma IREP para o tipo de dado, uma nova IREP é criada, utilizada no processo de instanciação e salva para eventuais buscas.

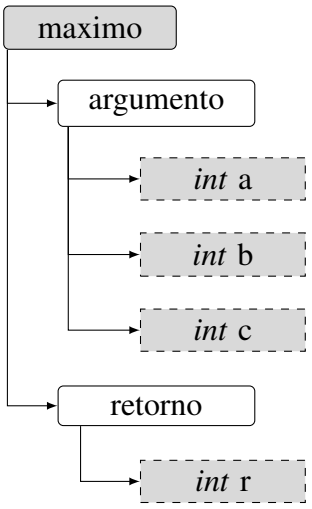
```
1 // Primeiro passo: criação de templates
2 template<class T>
3 T maximo(T a, T b, T c) {
4   T max = a > b? a : b;
5   return max > c? max : c;
6 }
7
8 int main() {
9   // Segundo passo: instanciação de templates
10  assert(maximo(1, 2, 3) == 3);
11  assert(maximo<float>(3.0f, 2.0f, 1.0f) == 3.0f);
12  return 0;
13 }
```

Figura 4.1: Exemplo do uso de *templates* de função.

A Figura 4.1 mostra um exemplo do uso de *templates*. A primeira etapa, criação de *templates*, ocorre quando é feito o *parser* da declaração de uma função *template* (linhas 3–6). Neste ponto, a IREP do *template* é criada com tipo genérico. A instanciação de *templates* ocorre no uso do *template*. Na Figura 4.1, o *template* é instanciado duas vezes (linhas 10–11). É possível ainda determinar o tipo de forma implícita (linha 10), ou de forma explícita (linha 11). Na instanciação implícita, o tipo de dado é determinado através do tipo dos parâmetros utilizados, enquanto que na instanciação explícita, o tipo de dado é determinado através do valor passado entre os símbolos menor que e maior que (< e >, respectivamente).



(a) Representação da IREP da função *maximo*, mostrada na Figura 4.1 (tipos genérico).



(b) Representação da IREP gerada ao instanciar o *template* da função *maximo* implicitamente com o tipo *int*.

Figura 4.2: Exemplo de criação de IREP.

A Figura 4.2 mostra um exemplo de IREP gerada a partir do código na Figura 4.1. A Figura 4.2a mostra a IREP gerada a partir do função *template maximo*. A IREP é construída com o nome da função (que é utilizada para busca, se necessário), e duas IREPs internas: uma que contém os tipos dos argumentos e outra que contém o tipo do retorno. Note que o tipo de dado é *type*, o que representa que o tipo é genérico. A Figura 4.2b mostra a IREP criada durante a instanciação do *template* com tipo de dado *int* (linha 10). Note que os tipos de dado não são do tipo *type* nesta IREP, mas do tipo instanciado. Como descrito nesta seção, ao final da fase de checagem de tipo, a IREP mostrada na Figura 4.2a é descartada.

4.2 Modelo Operacional C++

A linguagem C++ é construída sobre uma coleção de poderosas bibliotecas padrões para fornecer a maior parte das funcionalidades requeridas pelo programador [43]. Grande parte das bibliotecas padrões é feita utilizando *templates*, oferecendo assim uma grande flexibilidade para o uso das mesmas. Porém, essas bibliotecas padrões podem complicar desnecessariamente a verificação do programa C++, pois contém trechos de código que não são importantes para verificação (por exemplo, trechos de códigos para escrever mensagens na tela e códigos de otimização a nível de *assembly*). Para contornar o problema, o ESBMC utiliza uma representação simplificada da biblioteca chamada Modelo Operacional de C++ (COM, do inglês *C++ Operational Model*) que representa as classes, métodos, e outras funcionalidades semelhantes às estruturas reais [45].Uma técnica semelhante já foi aplicada para verificar pré-condições em programas [13]. No entanto, o ESBMC verifica tanto pré- quanto pós-condições, o que apresenta melhores resultados, pois as propriedades definidas pelas pós-condições são verdadeiras após o final da execução da função ou método. No processo de verificação, as bibliotecas COM substituem as bibliotecas reais de C++. O COM consiste de quatro grupos de bibliotecas, como mostrado na Figura 4.3.

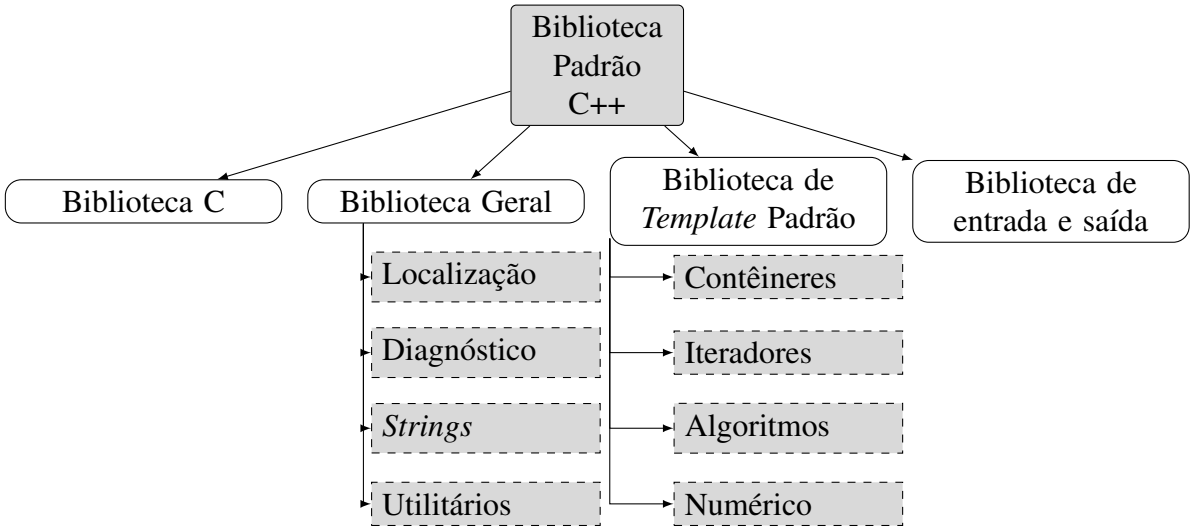


Figura 4.3: Visão geral do modelo operacional.

Note que o COM também inclui bibliotecas de ANSI-C já suportadas pelo ESBMC. Porém, como a verificação de programas C++ utiliza um *front-end* diferente, é preciso uma nova representação das bibliotecas ANSI-C no COM; caso contrário, o ESBMC não reconheceria os métodos dessas bibliotecas e falharia na verificação de programas C++ que as utilizassem.

A maior parte do COM é a Biblioteca de *Template* Padrão (STL), onde todas as funções e classes são *templates*. O processo de verificação de programas que utilizem a STL é descrita na Seção 4.1.

4.2.1 Linguagem Descritiva de Contêineres

Para formalizar a verificação de contêineres STL, é definida uma linguagem descritiva e são estendidas as funções C e P, restrições e propriedades, para essa linguagem [46] (como definidas no Capítulo 2). A linguagem descritiva é então utilizada para implementar o modelo operacional dos contêineres.

$$
\begin{aligned}
T &::= t \mid *It \mid *P \\
It &::= i \mid C.begin \mid C.end \\
&\quad \mid C.insert(It,T,N) \mid C.insert(It,It,It) \\
&\quad \mid C.erase(It) \mid C.erase(It,It) \mid C.search(T) \\
P &::= p \mid P(+\mid -)P \mid C.array \\
Int &::= n \mid Int(+\mid *\mid \ldots)Int \mid It.pos \mid C.size \mid C.capacity \\
C &::= c \mid It.source
\end{aligned}
$$

Figura 4.4: Sintaxe da Linguagem Descritiva dos Contêineres

A linguagem descritiva é composta por diversos domínios sintáticos (símbolos básicos para formar uma expressão lógica), começando com elementos básicos T, iteradores It, ponteiros P e índices inteiros Int, além das expressões de contêiner C. A Figura 4.4 resume a sintaxe da linguagem descritiva, onde t, i, p e c são variáveis do tipo T, It, P e C, respectivamente, e n é uma variável ou constante do tipo Int.

A notação $*It$ é extrapolada para denotar o valor armazenado no contêiner apontado pelo iterador It; isso é um abreviação para $(It.source.array)[It.pos]$, onde *source* é o contêiner apontado, *array* é o vetor interno que armazena os elementos e *pos* é a posição no vetor *array* que armazena o elemento apontado. Além disso, $*P$ é o valor armazenado na posição apontada por P na memória.

$C.begin$ e $C.end$ são métodos que retornam iteradores que apontam para o início e fim do contêiner, respectivamente. A maior parte das operações em contêineres também retornam um iterador apontando para o novo elemento ao invés de simplesmente retornar um contêiner atualizado. Por exemplo, para contêineres do tipo *vector*, a operação $C.erase$ retorna um iterador

apontando para o vizinho da direita do elemento apagado. Note que a única forma da linguagem descritiva acessar o contêiner resultante é através do campo *source* do iterador retornado.

Finalmente, *C.array* é a posição da memória que armazena o início do vetor do contêiner, *It.pos* é o índice (dentro do vetor) do elemento apontado pelo iterador e *C.size* e *C.capacity* retorna o valor atual e o valor máximo, respectivamente, do contêiner *C*.

4.2.2 Modelo Operacional de Contêineres

Assim como cada contêiner é diferente entre si, os seus métodos também variam, mudando inclusive os modelos internos, por exemplo, o contêiner *list* não possui operadores de referência e os seus elementos são somente acessados através de iteradores, devido a sua natureza dinâmica.

Para simular os contêineres apropriadamente, COM faz uso de três variáveis: uma variável do tipo *P* chamada *array* que aponta para o primeiro elemento do vetor, um número natural *size* que armazena a quantidade de elementos no contêiner, e um número natural *capacity* que armazena a capacidade total do contêiner. Similarmente, iteradores são modelados utilizando duas variáveis: um número natural *pos*, que contém o valor do índice apontado pelo iterador e uma variável do tipo *P*, chamada *source* que aponta para o contêiner. A Figura 4.5 apresenta uma visão geral do modelo operacional criado para contêineres sequenciais.

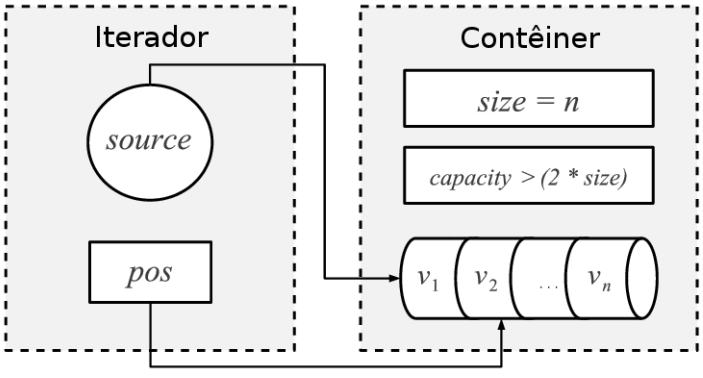


Figura 4.5: Modelo Operacional de Contêineres Sequenciais.

A linguagem descritiva suporta somente os métodos listados na Figura 4.4. Outros métodos como *push_back()*, *pop_back()*, *front()*, *back()*, *push_front()*, e *pop_front()* são somente variações dos métodos principais, que são otimizados para cada contêiner (por exemplo, remover um elemento do topo de uma *stack*). Como parte da transformação SSA, como descrito no Capítulo 2, efeitos secundários (como por exemplo, chamadas de funções ou métodos) dos

iteradores e contêineres são feitos explícitos, de tal forma que essas operações retornem novos iteradores e contêineres como resultado. Por exemplo, considere o contêiner C com a chamada do método *c.insert* que retorna um iterador como resultado e faz uso de um iterador i que aponta para a posição de inserção desejada, um valor de *template* t com um elemento para ser inserido e um inteiro n que denota o número de vezes que o elemento será inserido no contêiner. A expressão $C.insert(i,t,n)$; (onde é descartado o iterador retornado) se torna $(C',i') = C.insert(i,t,n)$, onde os efeitos secundários são explícitos.

Em particular, o algoritmo do método $C.insert(pos,t,n)$ é descrito no Algoritmo 4.1.

```
1 Passo 1: Verifique pré-condições.
2 Passo 2: Inicialize a variável counter com valor zero.
3 Passo 3: Copie cada elemento no intervalo [C.pos-1,C.size-1] para
4          [C.pos,C.size].
5 Passo 4: Incremente C.size em uma unidade.
6 Passo 5: Atribua t para C.array[i.pos].
7 Passo 6: Incremente counter em uma unidade.
8 Passo 7: Se counter < n, vá para Passo 2.
9 Passo 8: Se C.size >= C.capacity, duplique C.capacity.
```

Algoritmo 4.1: Algoritmo do método C.insert(pos, t, n)

O primeiro passo realizado no Algoritmo 4.1 é a verificação das pré-condições. É importante fazer esse tipo de verificação pois, caso essas condições não sejam satisfeitas, não é possível prever o comportamento do algoritmo, o que poderá acarretar em uma falha no programa verificado. Neste algoritmo, é checado se *pos* aponta para um contêiner válido, se t é um valor do mesmo tipo do contêiner e se n é um valor dentro dos limites do contêiner.

A Tabela 4.1 mostra um exemplo de inserção utilizando o Algoritmo 4.1. Suponha um contêiner, instanciado com o tipo *int*, e que contém os elementos $\{1,2,3\}$. Neste caso, o contêiner possui $size = 3$ e $capacity = 4$ (pois deve ser sempre múltiplo de 2). No exemplo, deseja-se inserir 2 elementos com valor 5 na segunda posição do contêiner (o método é executado como $C.insert(2,5,2)$).

Após a verificação das pré-condições (que, para este exemplo, assume-se que sejam verdadeiras), o algoritmo inicia atribuindo o valor 0 para a variável *counter*. No passo seguinte, os elementos de $[C.pos-1, C.size-1]$ são copiados para $[C.pos, C.size]$, ou seja, serão copiados

contêiner	*counter*	*size*	*capacity*	*pos*	*t*	*n*	Passo
1,2,3	0	3	4	2	5	2	2
1,2,2,3	0	3	4	2	5	2	3
1,2,2,3	0	4	4	2	5	2	4
1,5,2,3	0	4	4	2	5	2	5
1,5,2,3	1	4	4	2	5	2	6
1,5,5,2,3	1	4	4	2	5	2	3
1,5,5,2,3	1	5	4	2	5	2	4
1,5,5,2,3	1	5	4	2	5	2	5
1,5,5,2,3	2	5	4	2	5	2	6
1,5,5,2,3	2	5	8	2	5	2	8

Tabela 4.1: Exemplo de inserção do Algoritmo 4.1 ($C.insert(2,5,2)$). Células cinza representam uma mudança na linha anterior.

os elementos de $[1,2]$ para $[2,3]$. Em seguida, a variável *size* é incrementada e o valor 5 é atribuído para a posição 1. Por fim, a variável *counter* é incrementada e o processo se repete enquanto $counter < n$. Quando essa condição for falsa, o algoritmo terminou de inserir os elementos no *contêiner* e checa o tamanho da variável *capacity*, duplicando-a se necessário.

Existe outra versão do método *insert*. Neste, é possível inserir uma sequência de elementos a partir de uma posição desejada, usando tanto iteradores quanto ponteiros para selecionar trechos de outros contêineres. Por exemplo, seja *begin* um iterador que marca o primeiro elemento a ser inserido na posição desejada, *end* seja outro iterador apontando para o último elemento da sequência a ser inserido. O algoritmo do método $C.insert(pos, begin, end)$ é definido no Algoritmo 4.2.

```
1 Passo 1: Verifique pré-condições.
2 Passo 2: Declare a variável elem.
3 Passo 3: Atribua o valor apontado pelo iterador begin à variável elem.
4 Passo 4: Execute o método C.insert(pos, elem, 1).
5 Passo 5: Incremente o iterador begin.
6 Passo 6: Se begin != end, vá para Passo 3.
```

Algoritmo 4.2: Algoritmo do método C.insert(pos, begin, end)

Novamente, o Algoritmo 4.2 inicia com a verificação das pré-condições. Para este método, $[begin, end)$ não deve ser vazia e *begin* e *end* devem ser iteradores de um mesmo contêiner

(por exemplo, $begin.source.array = end.source.array$ deve ser verdadeira), mesmo que o contêiner seja diferente do contêiner que esteja sofrendo a inserção.

contêiner 1	contêiner 2	*elem*	*begin*	**begin*	*end*	**end*	*pos*	Passo
1,2,3	4,5,6	4	0	4	2	6	2	3
1,4,2,3	4,5,6	4	0	4	2	6	2	4
1,4,2,3	4,5,6	4	1	5	2	6	2	5
1,4,2,3	4,5,6	5	1	5	2	6	2	3
1,4,5,2,3	4,5,6	5	1	5	2	6	2	4
1,4,5,2,3	4,5,6	5	2	6	2	6	2	5

Tabela 4.2: Exemplo de inserção do Algoritmo 4.2 ($C.insert(2,0,2)$). Células cinza representam uma mudança na linha anterior.

A Tabela 4.2 mostra um exemplo de inserção utilizando o Algoritmo 4.2. Suponha um contêiner, instanciado com o tipo *int*, e que contém os elementos $\{1,2,3\}$ e, um segundo contêiner, instanciado com o tipo *int*, e que contém os elementos $\{4,5,6\}$. No exemplo, deseja-se inserir o primeiro e segundo elemento do segundo contêiner, na segunda posição do primeiro contêiner (o método é executado como $C.insert(2,0,2)$).

Após a verificação das pré-condições (que, para este exemplo, assume-se que sejam verdadeiras), o algoritmo inicia atribuindo o valor apontado pelo ponteiro *begin* para a variável $elem$. Note que na tabela, o a coluna *begin* representa a posição apontada no segundo contêiner e $*begin$ representa o valor. A variável $elem$ é sempre atribuída com o valor apontado por *begin*. Em seguida, o método $C.insert(pos, elem, 1)$ é executado, ou seja, é inserido na posição *pos*, o valor *elem*, uma única vez, como descrito pelo Algoritmo 4.1. Ao final, o endereço apontado pelo iterador *begin* é incrementado e o processo se repete até que o endereço apontado por *begin* seja igual o endereço apontado pelo iterador *end*. Neste algoritmo, não é necessário ajustar *size* ou *capacity* do contêiner, já que isso é feito durante o passo 4, na execução do método $C.insert(pos, elem, 1)$.

O método *erase* funciona de forma similar ao método *insert*. O método também usa iteradores, valores inteiros ou ponteiros, para definir posição, e remove o elemento independente do valor. O método retorna um iterador apontando para a próxima posição válida dentro do contêiner. O algoritmo do método $C.erase(pos)$, que deleta um único elemento, é definido no Algoritmo 4.3.

```
1 Passo 1: Verifique pré-condições.
2 Passo 2: Copie cada elemento no intervalo [C.pos,C.size-1] para
3          [C.pos-1,C.size-2].
4 Passo 3: Decremente C.size em uma unidade.
5 Passo 4: Se C.size <= C.capacity/2, divida C.capacity pela metade.
```

Algoritmo 4.3: Algoritmo do método C.erase(pos)

O método possui apenas uma condição a ser verificada: se *pos* é um iterador válido sobre *C* (por exemplo, que $i.source = C$ seja verdadeira).

contêiner	*size*	*capacity*	*pos*	Passo
1,5,5,2,3	5	8	3	-
1,5,2,3	5	8	3	1
1,5,2,3	4	8	3	2
1,5,2,3	4	4	3	3

Tabela 4.3: Exemplo de remoção do Algoritmo 4.3 ($C.erase(3)$). Células cinza representam uma mudança na linha anterior.

A Tabela 4.3 mostra um exemplo de remoção utilizando o Algoritmo 4.3. Suponha o contêiner do exemplo da Tabela 4.1, ao final da inserção, composto pelos elementos $\{1,5,5,2,3\}$. No exemplo, deseja-se remover o terceiro elemento do contêiner. O método é executado como $C.erase(3)$.

Após a verificação das pré-condições (que, para este exemplo, assume-se que sejam verdadeiras), o algoritmo inicia copiando os elementos de $[C.pos, C.size-1]$ para $[C.pos-1, C.size-2]$, o que sobrescreve o elemento na posição *pos* com o elemento seguinte e assim sucessivamente, até o último elemento do contêiner (no exemplo, serão copiados elementos de $[3,4]$ para $[2,3]$). Em seguida, a variável *size* é decrementada e, caso *size* seja menor ou igual a metade de *capacity*, o valor da *capacity* é dividido pela metade.

Também é possível apagar um número de elementos do contêiner, utilizando iteradores para definir um trecho da sequência no contêiner. Por exemplo, seja *begin* um iterador que marca o primeiro elemento a ser removido, *end* outro iterador que aponta para o último elemento da sequência a ser removido. O algoritmo do método $C.erase(begin, end)$ é definido no Algoritmo 4.4.

```
1 Passo 1: Verifique pré-condições.
2 Passo 2: Execute o método C.erase(begin).
3 Passo 3: Decremente o iterador end.
4 Passo 4: Se begin != end, vá para Passo 2.
```

Algoritmo 4.4: Algoritmo do método C.erase(begin, end)

As condições verificadas são iguais às condições do algoritmo $C.insert(pos, begin, end)$: $[begin, end)$ não deve ser vazia, e ambos os iteradores devem apontar para elementos de um mesmo contêiner.

contêiner	*begin*	*end*	Passo
1,5,5,2,3	2	4	-
1,5,2,3	2	4	2
1,5,2,3	2	3	-
1,2,3	2	3	-
1,2,3	2	2	-

Tabela 4.4: Exemplo de remoção do Algoritmo 4.4 ($C.erase(2,4)$). Células cinza representam uma mudança na linha anterior.

A Tabela 4.4 mostra um exemplo de remoção utilizando o Algoritmo 4.4. Suponha o contêiner do exemplo da Tabela 4.1, ao final da inserção, composto pelos elementos $\{1,5,5,2,3\}$. No exemplo, deseja-se remover o segundo e terceiro elementos do contêiner. O método é executado como $C.erase(2,4)$.

Após a verificação das pré-condições (que, para este exemplo, assume-se que sejam verdadeiras), o algoritmo inicia executando o método $C.erase(begin)$, para remover elemento na posição apontada pelo iterador *begin*. O elemento é removido, porém *begin* ainda aponta para o próximo elemento a ser removido. Em seguida, o endereço apontado pelo iterador *end* é decrementado e o processo se repete, até que os endereços apontados pelos iteradores *begin* e *end* sejam iguais. Assim como o Algoritmo 4.2, não é necessário ajustar *size* ou *capacity* do contêiner, já que isso é feito durante o passo 2, na execução do método $C.erase(begin)$.

Por fim, o método de busca *C.search* faz uso de um elemento $elem$ e percorre o contêiner até encontrar o primeiro elemento igual ao procurado. Neste caso, o método retorna um iterador apontando para o elemento. Caso um elemento não seja encontrado, é retornado o iterador *C.end*. O algoritmo do método $C.search(elem)$ é definido no Algoritmo 4.5.

```
1 Passo 1: Verifique pré-condições.
2 Passo 2: Inicie novo iterador p apontando para C.begin.
3 Passo 3: Se *p = elem, retorne p.
4 Passo 4: Incremente o ponteiro p.
5 Passo 5: Se p != end, vá para Passo 2.
6 Passo 6: Se p = end, retorne p.
```

Algoritmo 4.5: Algoritmo do método C.search(el)

Neste método, a única condição verificada é se o contêiner *C* não é vazio.

contêiner	*p*	**p*	*elem*	Passo
1,5,5,2,3	0	1	3	2
1,5,5,2,3	1	5	3	2
1,5,5,2,3	2	5	3	2
1,5,5,2,3	3	2	3	2
1,5,5,2,3	4	3	3	2

Tabela 4.5: Exemplo de busca do Algoritmo 4.5 ($C.search(3)$). Células cinza representam uma mudança na linha anterior.

A Tabela 4.5 mostra um exemplo de busca utilizando o Algoritmo 4.5. Suponha o contêiner do exemplo da Tabela 4.1, ao final da inserção, composto pelos elementos $\{1,5,5,2,3\}$. No exemplo, deseja-se fazer uma busca se o contêiner possui um elemento com valor 3.

Após a verificação das pré-condições (que, para este exemplo, assume-se que sejam verdadeiras), o algoritmo inicia copiando o endereço apontado pelo iterador *begin* para um novo iterador *p*. Em seguida, é verificado se o valor apontado por *p* é igual ao valor procurado. O processo repete até que um elemento com mesmo valor seja encontrado ou o algoritmo tiver percorrido todo o contêiner. O algoritmo deve sempre percorrer todos os elementos do contêiner pois os elementos podem ser inseridos de forma não ordenada.

4.3 Tratamento de Exceção

As exceções são situações inesperadas que podem acontecer durante a execução de um programa. Em C++, o tratamento de exceções está dividido em três partes: um bloco *try*, onde uma exceção pode ocorrer; um bloco *catch*, onde uma exceção pode ser tratada e uma expressão

throw, que conecta os dois blocos. O código na Figura 4.6 mostra um exemplo de programa com tratamento de exceções.

```
1 int main() {
2   try { // bloco try
3     throw 20; // expressão throw
4   }
5   // catch block
6   catch (int i) { /* tratamento de exceções do tipo int */ }
7   catch (float f) { /* tratamento de exceções do tipo float */ }
8   return 0;
9 }
```

Figura 4.6: Exemplo de tratamento de exceções: lançando uma exceção do tipo inteira.

O processo de verificação de programas que contém tratamento de exceções ocorre em duas etapas: durante a checagem de tipo e a execução simbólica. Na checagem de tipo, uma IREP é construída baseada no código contido no bloco *try*, com algumas adaptações: antes do código contido no bloco *try* iniciar, uma instrução CATCH é inserida com um mapa vazio (que será preenchido adiante), seguido pelo código contido no bloco *try*, uma instrução CATCH (para representar o fim do bloco *try*) e uma instrução GOTO apontando para o código após o bloco *catch*. Após a checagem de tipo do bloco *try*, é verificado o bloco *catch*, que pode conter um ou mais *catchs*. Novamente, uma IREP será criada baseada no código contido no bloco *catch*, com uma alteração: uma instrução GOTO será inserida no final de cada código *catch*, apontando para o código após os *catchs*. Para cada código *catch*, será atribuído um rótulo, dessa forma será possível decidir para qual *catch* deverá ser feito o chaveamento da instrução *throw* durante a fase de execução simbólica. Ao final, o mapa do primeiro *catch* é preenchido com os tipos das exceções indexados pelos seus rótulos. O preenchimento do mapa é somente feito nesse momento pois quando a primeira instrução CATCH é inserida, não é possível saber quais os *catchs* estão presentes no programa. A Figura 4.7 mostra o código GOTO gerado a partir do código mostrado na Figura 4.6.

Durante a execução simbólica, quando a primeira instrução CATCH é encontrada, o mapa presente nessa instrução é empilhado. A ideia de usar uma pilha para os mapas vem do fato que é possível ter blocos de *try* e *catchs* aninhados, e o processo de verificação sempre trata a exceção em um bloco mais interno, antes de tratá-la externamente. Seguindo com a execução simbólica, o processo de verificação continuará até encontrar uma instrução THROW. Caso uma instrução THROW seja encontrada, será procurado no mapa se existe um *catch* válido para

```
1  main():
2      CATCH signed_int->1, float->2
3      THROW 20
4      CATCH
5      GOTO 3
6  1: int i;
7      /* tratamento de exceções do tipo int */
8      GOTO 3
9  2: float f;
10     /* tratamento de exceções do tipo float */
11 3: return 0;
12 END_FUNCTION
```

Figura 4.7: Conversão do código de tratamento de exceção em código GOTO.

a exceção lançada. Caso seja encontrado um *catch* válido, o rótulo correspondente será salvo para futura utilização, caso não seja encontrado nenhum *catch* válido, uma mensagem de erro será mostrada ao usuário. Quando a segunda instrução CATCH é encontrada, o que significa que o código contido no bloco *try* terminou, o mapa é desempilhado e a instrução GOTO após o segundo *catch* atualizada.

4.3.1 A Linguagem de Tratamento de Exceções

Para formalizar a verificação de tratamento de exceções, é necessário definir uma linguagem de tratamento de exceções. A linguagem de tratamento de exceção é similar a linguagem descritiva de contêineres, mostrada na Figura 4.4, porém é composta somente pelos elementos básicos T e ponteiros P. A sintaxe para valores de T é:

$$T ::= t_T \mid cvr\, t_T \mid *P$$

onde t_T é uma variável do tipo T, $cvrt_T$ é uma variável do tipo T com qualificadores *const*, *volatile* ou *restrict* e $*P$ é o valor armazenado na posição P da memória. Para os ponteiros P (valores de posição na memória), a sintaxe é:

$$P ::= p \mid P_{vetor}$$

onde p é uma variável do tipo P e P_{vetor} é a posição da memória que armazena o início de um vetor.

4.3.2 Lançando e Capturando uma Exceção

Além da expressão *throw*, existem diversas situações que uma exceção pode ser lançada: (a) uma exceção do tipo *bad_alloc* pode ser lançada pelo operador *new*, (b) o operador *dynamic_cast* pode lançar uma exceção do tipo *bad_cast* e (c) a função *typeid* pode lançar uma exceção do tipo *bad_typeid*. Essas exceções estão definidas internamente na linguagem C++ e devem ser tratadas pelo programa. No documento de definição da linguagem C++, diversas regras são definidas sobre como uma expressão *throw* e um *catch* podem ser conetados [47].

Para modelar o comportamento do tratamento de exceções durante o processo de verificação, foram definidas as funções $rule$. Cada função $rule$ retorna o *catch* correspondente, se a regra for verdadeira. Caso contrário, retorna um *catch* inválido, definido como $C_{invalido}$. Além disso, a função *rule* aceita como argumentos um tipo de exceção lançado e um conjunto de *catchs*.

Funções	Comportamento
$rule_1$	Retornará o *catch* se o tipo da exceção lançada for igual ao tipo do *catch*.
$rule_2$	Retornará o *catch* se o tipo da exceção lançada for igual ao tipo do *catch*, ignorando os qualificadores *const*, $volatile$ e $restrict$.
$rule_3$	Se uma expressão *throw* lançar uma exceção do tipo "vetor do tipo T", a função retornará o *catch* se o tipo dele for "ponteiro para T".
$rule_4$	Se uma expressão *throw* lançar uma exceção do tipo "função retornando o tipo T", a função retornará o *catch* se o tipo dele for "ponteiro para função retornando tipo T".
$rule_5$	Retornará o *catch* se o tipo dele for uma base não ambígua do tipo da exceção lançada.
$rule_6$	Retornará o *catch* se a exceção lançada for do tipo ponteiro *P* e for possível fazer uma conversão, por qualificação ou por conversão padrão de ponteiros [47], para o tipo do *catch*.
$rule_7$	Retornará o *catch* se a exceção lançada for do tipo ponteiro *P* e o *catch* for do tipo *void** (ponteiro *void*)
$rule_8$	Retornará o *catch* se o tipo dele for reticências, independentes do tipo da exceção lançada.
$rule_9$	Se a expressão *throw* não possuir argumentos, então a função retornará o *catch* conectado à última exceção lançada, caso exista.

Tabela 4.6: Funções *rule* de conexão entre expressões *throw* e blocos *catch*.

Existem nove funções $rule$, como mostrado na Tabela 4.6, e cada uma delas representa uma regra de conexão entre um *throw* e um *catch*. Para cada exceção lançada, todas as regras são verificadas, antes de escolher qual *catch* deve ser selecionado. A expressão abaixo representa as regras $rule$ sendo aplicadas k vezes. No total, para cada *throw*, são chamadas as nove funções $rule$, logo $k = 9N$, onde k é o número total de aplicação das funções *rule* e N é o número de

expressões *throw*.

$$\begin{aligned} & C_1 = rule_1(T_{t1}, T_{c1}, T_{c2}, \ldots, T_{cM}) \\ & \wedge \ldots \\ & \wedge C_k = rule_9(T_{t1}, T_{c1}, T_{c2}, \ldots, T_{cM}) \end{aligned} \tag{4.1}$$

Onde M é o número de *catchs*.

Após a aplicação das funções *rule*, a função *match* é executada para retornar o *catch* escolhido. A função *match* retorna o *catch* C que pode ser conectado à exceção lançada e aceita como argumentos o resultado da aplicação das funções *rule*.

$$\begin{aligned} match(C_1, C_2, \ldots, C_k) \Longrightarrow \quad & ite(C_1 \neq C_{invalido}, C_1, \\ & ite(C_2 \neq C_{invalido}, C_2, \ldots \\ & \quad ite(C_k \neq C_{invalido}, C_k, C_{invalido}) \ldots) \end{aligned} \tag{4.2}$$

A função *match* irá selecionar o primeiro *catch* válido. Caso, não exista nenhum *catch* válido, a função *match* seleciona o $C_{invalido}$ e um erro é apresentado como resultado da verificação. Na prática, isso acontece quando o programa tenta lançar um exceção porém, não existe um *catch* correspondente.

$$(T_t = T_C) \Longrightarrow T_C \tag{4.3}$$

$$(T_t = cvr\,T \wedge T_C = T) \Longrightarrow T_C \tag{4.4}$$

As funções $rule_1$ e $rule_2$, são modeladas segundo as Fórmulas 4.3 e 4.4, respectivamente. Para implementar esse comportamento, é procurado no mapa por um rótulo com o mesmo tipo da exceção lançada e a instrução GOTO é atualizada se existir um rótulo correspondente (ou lança um erro, caso contrário).

$$(T_t = T[] \wedge T_C = T*) \Longrightarrow T_C \tag{4.5}$$

$$(T_t = T_{f()} \wedge T_C = T_{f()}) \Longrightarrow T_C \tag{4.6}$$

As funções $rule_3$ e $rule_4$, são modeladas segundo as Fórmulas 4.5 e 4.6, respectivamente. Durante o processo de verificação, a conversão é feita na checagem de tipo. Após a conversão, a expressão *throw* irá lançar duas exceções: "vetor do tipo T" e "ponteiro do tipo T", no caso da exceção lançada ser do tipo "vetor do tipo T"; e "função retornando tipo T" e "ponteiro para função retornando tipo T", no caso da exceção lançada ser do tipo "função retornando tipo T".

Para modelar a função $rule_5$, é necessário definir a função binária $base(T_2, T_1)$.

$$base(T_2, T_1) = \begin{cases} 1, & se\ T_2\ é\ base\ não\ ambígua\ de\ T_1; \\ 0, & caso\ contrário. \end{cases} \tag{4.7}$$

A função binária $base(T_2, T_1)$ retorna verdadeiro se T_2 for base não ambígua de T_1 ou falso, caso contrário.

$$(T_t = T_1 \wedge T_C = T_2 \wedge base(T_2, T_1)) \Longrightarrow T_C \tag{4.8}$$

A função $rule_5$ é modelada segundo a Fórmula 4.8. No processo de verificado, a conversão é semelhante à da regra anterior, mas nesse caso, diversas exceções são lançadas: o tipo do objeto e os tipos das suas classes base.

Para modelar a $rule_6$, deve-se definir a função binária $implicit\_conv(P_1, P_2)$.

$$implicit\_conv(P_1, P_2) = \begin{cases} 1, & se\ P_1\ pode\ ser\ convertido\ implicitamente\ para\ P_2; \\ 0, & caso\ contrário. \end{cases} \tag{4.9}$$

A função $implicit\_conv(P_1, P_2)$ retorna verdadeiro se o ponteiro P_1 pode ser convertido implicitamente para o ponteiro P_2, ou falso, caso contrário.

$$(T_t = T_1 * \wedge T_C = T_2 * \wedge implicit\_conv(T_1, T_2)) \Longrightarrow T_C \tag{4.10}$$

A função $rule_6$ é modelada segundo a Fórmula 4.10. Novamente, durante a checagem de tipo, as possíveis conversões baseadas nos tipos dos *throw* serão lançadas, juntamente com o tipo original dos ponteiros.

$$(T_t = P * \wedge T_C = void*) \Longrightarrow T_C \tag{4.11}$$

A função $rule_7$ é modelada segundo a Fórmula 4.11. Durante a execução simbólica, se nenhum *catch* correspondente for encontrado no mapa e o tipo lançado for um ponteiro, então um catch do tipo *void** é procurado, e caso exista, a instrução GOTO é atualizada.

$$(T_t = T \wedge T_C = T_{...}) \Longrightarrow T_C \tag{4.12}$$

A função $rule_8$ é modelada segundo a Fórmula 4.12. Esta regra funciona de forma semelhante à regra anterior, porém para qualquer tipo lançado. Caso nenhum *catch* correspondente

seja encontrado, é procurado por um *catch* do tipo reticências e, se for encontrado, a instrução GOTO é atualizada.

$$(T_t = NULL \wedge T_{t-1} = T) \Longrightarrow MATCH(T, T_{C1}, T_{C2}..., T_{CN}) \tag{4.13}$$

A função $rule_9$ é modelada segundo a Fórmula 4.13. O processo de verificação sempre mantém uma referência para a última exceção lançada e atualiza uma instrução THROW vazia se essa referência não for nula.

4.3.3 Especificação de Exceções

A especificação de exceções define quais exceções podem ser lançadas por uma função ou método (incluindo construtores e destrutores). Ela é formada por uma lista de exceções, podendo ser vazia (o que implica que a função ou método não poderá lançar nenhuma exceção). Vale notar que exceções lançadas e tratadas dentro da função ou métodos são independentes da especificação de exceção, de tal forma que é possível definir a especificação de exceções como uma restrição das exceções que função e método podem lançar. O código na Figura 4.8 mostra alguns exemplos do uso da especificação de exceções.

```
1 // função exemplo1 pode lançar exceções do tipo int e float
2 void func1() throw(int, float) {
3   ...
4 }
5 // função exemplo2 não pode lançar nenhuma exceção
6 void func2() throw() {
7   try {
8     throw 1; // OK, exceção lançada e tratada no escopo da função
9   }
10  catch(int) {
11    /* tratamento de exceção do tipo int */
12  }
13 }
```

Figura 4.8: Exemplo de especificação de exceções.

A especificação de exceções é tratada durante o processo de verificação, utilizando uma instrução chamada THROW_DECL, inserida após a declaração de cada função ou método. Na fase de execução simbólica, as instruções THROW_DECL contendo a lista de exceções permitidas são empilhadas e removidas na instrução END_FUNCTION (que é inserida ao final de cada função ou método). A ideia de usar uma pilha para as especificações de exceções é

semelhante à ideia de empilhar blocos *try* e *catchs*, pois podem existir funções aninhadas que possuam suas próprias especificações de exceções. Finalmente, quando a exceção é lançada, é verificado se existe alguma especificação de exceção presente, e caso exista, é verificado se o tipo de exceção que está sendo lançada é permitida de acordo com a especificação de exceções. Se for permitida, o fluxo de verificação segue normalmente, caso contrário, uma mensagem de erro é mostrada ao usuário, indicando a linha no código onde o erro ocorreu.

4.4 Resultados Experimentais

Esta seção está dividida em três partes. O ambiente onde os experimentos foram realizados é descrito na Seção 4.4.1 enquanto que a Seção 4.4.2 descreve a comparação entre o ESBMC [48] e o LLBMC (*Low-Level Bounded Model Checker*) [49] usando um conjunto de casos de teste de programas C++. Alguns detalhes sobre o LLBMC são descritos também na Seção 4.4.2. Por fim, a Seção 4.4.3 apresenta a descrição da verificação de uma aplicação comercial, na área de telecomunicações.

4.4.1 Ambiente do Experimento

Para verificação das funcionalidades da linguagem C++, foram utilizados 1183 programas C++ como casos de teste. O resultado da verificação desses casos de teste utilizando a ferramenta ESBMC foram comparados aos resultados da verificação utilizando a ferramenta LLBMC. Desses, cerca de 290 programas foram extraídos do livro Deitel [44], 16 programas foram obtidos da NEC [34], 16 programas foram obtidos do LLBMC [14], 19 programas foram obtidos da suíte de teste do GCC [50] e os outros foram desenvolvidos por uma equipe de desenvolvedores para testar as diversas funcionalidades que a linguagem C++ fornece [46]. O casos de teste são divididos em 11 suítes de teste: *algorithm* contém casos de teste que envolvem os métodos na classe Algorithm; *cpp* contém os casos de teste gerais envolvendo as bibliotecas de uso geral do C++, *multi-threading* e *templates*. Além disso, nesta suíte de teste estão contidos os casos de teste do LLBMC e a maior parte dos casos de teste da NEC. As suítes de teste *deque*, *list*, *queue*, *stack*, *stream*, *string* e *vector* contém os casos de teste dos respectivos contêineres. A suíte de teste *try_catch* contém casos de teste relacionados a tratamento de exceção. Finalmente, a suíte de teste *templates* contém os casos de teste da suíte de teste do GCC, específicos

para teste de *templates*.

Todos os experimentos foram conduzidos em um computador com processador Intel Core i7-2600, 3.40Ghz com 24GB de memória RAM com Ubuntu 11.10 64-bits. Na comparação do ESBMC com o LLBMC, foram ajustados um limite de tempo de verificação e um limite de memória, de 3600 segundos (60 minutos) e 22GB de memória, respectivamente. Os casos de teste foram verificados com o ESBMC v1.22 e o LLBMC 2013.1.

4.4.2 Comparação ao LLBMC

Nessa seção, está descrita a comparação do ESBMC e LLBMC, ferramenta de verificação desenvolvida por Merz et al. [11]. As Tabelas 4.7 e 4.8 mostram os resultados. Nestas, *N* é o número de programas C++, *L* é o número de linhas em todos os casos de teste, *Tempo* é o tempo total de verificação de cada suíte de teste, *P* é o número de casos de teste reportados como corretos, *N* é o número de casos de teste reportados como incorretos, *FP* é o número de casos de teste reportados como corretos, mas que estão incorretos, *FN* é o número de casos de teste reportados como incorretos, mas que estão corretos, *Falha* é o número de casos de teste onde ocorreu um erro interno nas ferramentas, *TO* é o número de *time-outs* (a ferramenta foi abortada após 3600 segundos) e *MO* é o número de *memory-outs* (a ferramenta tentou consumir mais do que 22GB de memória).

As ferramentas foram executadas utilizando dois *scripts*: um para o ESBMC, que lê os parâmetros de um arquivo e executa a ferramenta[1], e outro para o LLBMC que primeiro compila o programa para *bytecode* usando o compilador CLANG[2] [29], em seguida, lê os parâmetros de um arquivo e executa a ferramenta. O número de iterações definidas para cada ferramenta (isto é, o valor de *B*) depende de cada caso de teste. Atualmente, o LLBMC não suporta tratamento de exceções e todos os *bytecodes* foram gerados sem o suporte a exceções (com a *flag -fno-exceptions*) na verificação com LLBMC. Se o tratamento de exceção for habilitado, o LLBMC aborta durante a verificação da maioria dos casos de teste.

Como pode ser visto na Tabela 4.8, o LLBMC aborta por limite de tempo em 13 programas na suíte de teste *algorithm*. Se os casos de teste forem analisados, a grande maioria utiliza iteradores, o que pode estar causando o aumento do tempo de verificação; situação que também

[1] esbmc --unwind *B* --no-unwinding-assertions -I /libraries/ –timeout 60m

[2] /usr/bin/clang++ -c -g -emit-llvm *.cpp -fno-exceptions
/usr/bin/llvm-link *.o -o main.bc

				ESBMC							
	Suíte de teste	*N*	*L*	Tempo	P	N	FP	FN	Falha	TO	MO
1	algorithm	144	4354	6360	65	67	1	11	0	0	0
2	deque	43	1239	47	20	21	0	2	0	0	0
3	vector	146	6853	2825	92	39	1	14	0	0	0
4	list	72	2292	1463	27	29	0	16	0	0	0
5	queue	14	328	18	7	7	0	0	0	0	0
6	stack	14	286	15	7	7	0	0	0	0	0
7	try_catch	81	4743	29	23	43	4	11	0	0	0
8	stream	66	1831	85	47	12	1	6	0	0	0
9	string	233	4921	5433	107	124	0	2	0	0	0
10	cpp	350	33112	3901	279	41	6	23	1	0	0
11	templates	19	563	2	11	0	0	8	0	0	0
		1182	60522	20162	685	390	13	93	1	0	0

Tabela 4.7: Resultados da ferramenta ESBMC v1.22

				LLBMC							
	Suíte de teste	*N*	*L*	Tempo	P	N	FP	FN	Falha	TO	MO
1	algorithm	144	4354	48419	62	62	1	3	3	13	0
2	deque	43	1239	33068	16	17	0	0	1	9	0
3	vector	146	6853	23900	90	39	1	4	2	6	4
4	list	72	2292	3555	7	30	5	30	0	0	0
5	queue	14	328	56	6	7	0	1	0	0	0
6	stack	14	286	61	7	7	0	0	0	0	0
7	try_catch	81	4743	-	-	-	-	-	-	-	-
8	stream	66	1831	11	17	13	0	35	1	0	0
9	string	233	4921	34	6	121	4	102	0	0	0
10	cpp	350	33112	9935	250	26	11	53	6	2	2
11	templates	19	563	2	15	0	0	4	0	0	0
		1182	60552	119046	476	322	22	232	94	30	6

Tabela 4.8: Resultados da ferramenta LLBMC v2013.1

acontece em outras suítes de teste. Nas suítes de teste *deque* e *vector*, a ferramenta também aborta devido a *time-out*, porém em menor número. Da classe dos contêineres, a suíte de teste que teve o maior número de verificações incorretas foi a *list* e a maior parte dos erros foram relacionados ao tamanho do contêiner (por exemplo, assertivas se um contêiner está vazio ou se ele possui um tamanho em particular). No ESBMC, a maior parte dos erros foi causado pela

falta de modelagem dessas bibliotecas.

Na suíte de teste *queue*, o LLBMC falha no programa que utiliza o tamanho de uma lista como parâmetro de um construtor, enquanto na suíte de teste *stack*, todos os programas são verificados corretamente. O ESBMC é capaz de verificar todos os casos de teste corretamente, em ambas as suítes de teste.

Na suíte de teste *stream*, a maior parte dos erros estão relacionadas a assertivas do tamanho do *stream* (utilizando o método *gcount* ()) e a *flags* internas (como *ios::hex* e *iostream::hex*). No ESBMC, a maior parte dos erros está relacionada a modelagem incompleta de *flags* internas. Na suíte de teste *string*, os erros estão relacionados a assertivas de *strings*, geralmente se uma *string* é igual a outra *string*. Na suíte de teste *try_catch*, o LLBMC não é capaz de verificar os casos de teste que contém tratamento de exceção, e por esse motivo, a ferramenta não será avaliada nessa suíte de teste. O ESBMC foi capaz de verificar a maior parte dos casos de teste nesta suíte de teste. Os erros relatados estãorelacionados a falta de implementação de especificação de exceções em construtores de classe. Na suíte de teste *cpp*, que possui casos de teste envolvendo todas as outras suítes de teste (porém sem redundância), a maior parte dos erros apresentados foram citados durante a verificação das outras suítes de teste. Finalmente, na suíte de teste *templates*, o ESBMC verifica menos casos de teste; dos casos de teste verificados de forma errada, quatro casos de testes estão relacionados à escolha de um *template* baseado no valor que ele está sendo instanciado, ao invés do tipo; dois casos de teste estão relacionados à especialização de *templates* durante a instanciação de outro *template* e, dois estão relacionados à herança de *templates* de classe.

Para melhor apresentar os resultados finais entre as ferramentas, as suítes de teste apresentados nas Tabelas 4.7 e 4.8 foram agrupadas quatro categorias:

- **STL**: composto pelas suítes de teste *algorithm, deque, vector, list, queue* e *stack*;
- ***Streams***: composto pelas suítes de teste *stream* e *string*.
- **Deitel**: composto pela suíte de teste cpp.
- **GCC**: composto pela suíte de teste *templates*.

Além disso, os resultados foram agrupados em resultados verdadeiros (positivos, P, e negativos, N), resultados falsos (falsos positivos, FP, e falsos negativos, FN) e falhas de verificação (*memory-out*, MO, *time-out*, TO, e $Crash$). As Figuras 4.9, 4.10 e 4.11 apresentam

os resultados verdadeiros, falsos e as falhas de verificação das ferramentas ESBMC e LLBMC, respectivamente.

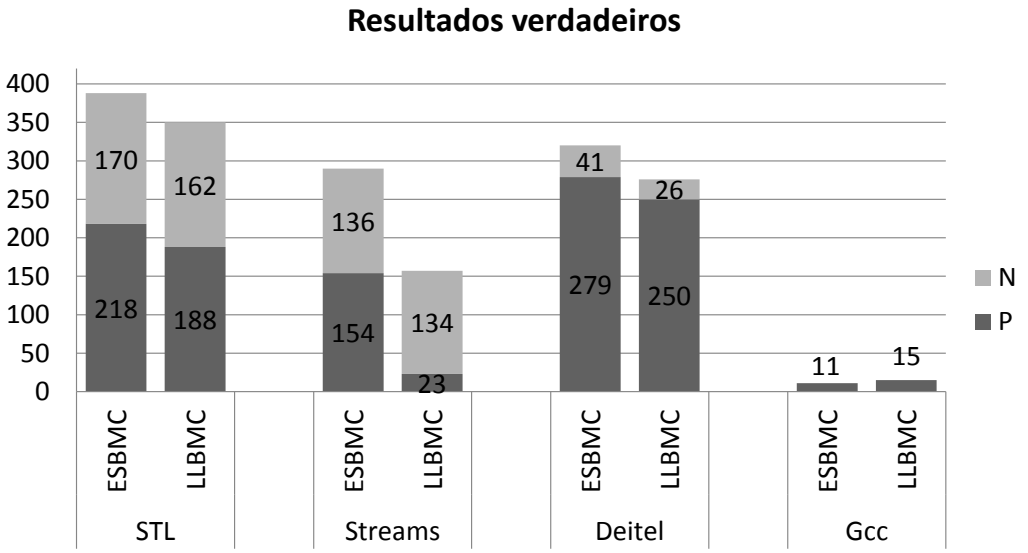


Figura 4.9: Comparativo dos resultados verdadeiros entre ESBMC e LLBMC.

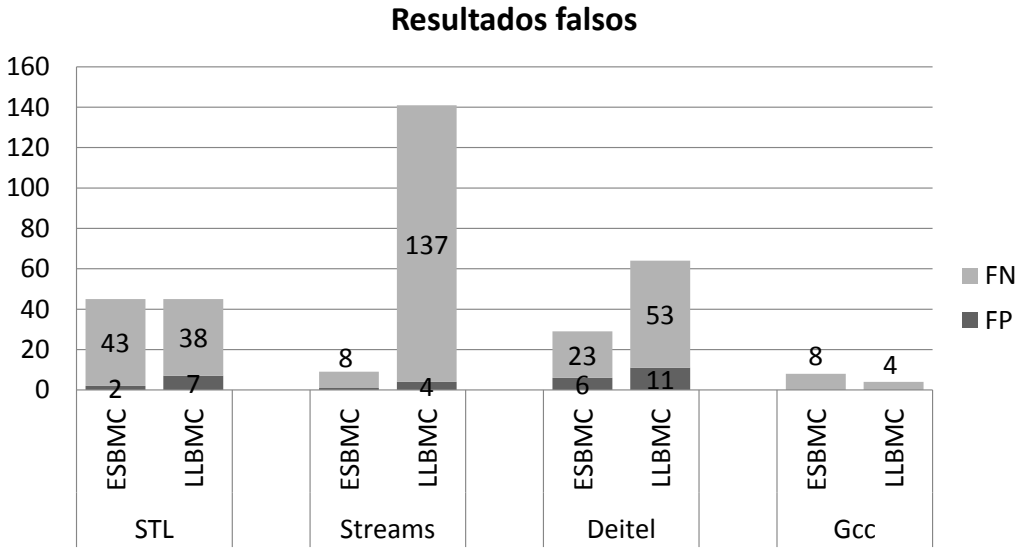


Figura 4.10: Comparativo dos resultados falsos entre ESBMC e LLBMC.

Como pode ser visto na Figura 4.9, o ESBMC verifica corretamente mais casos de teste do que o LLBMC em quase todas as categorias, com exceção da categoria Gcc. Um resultado

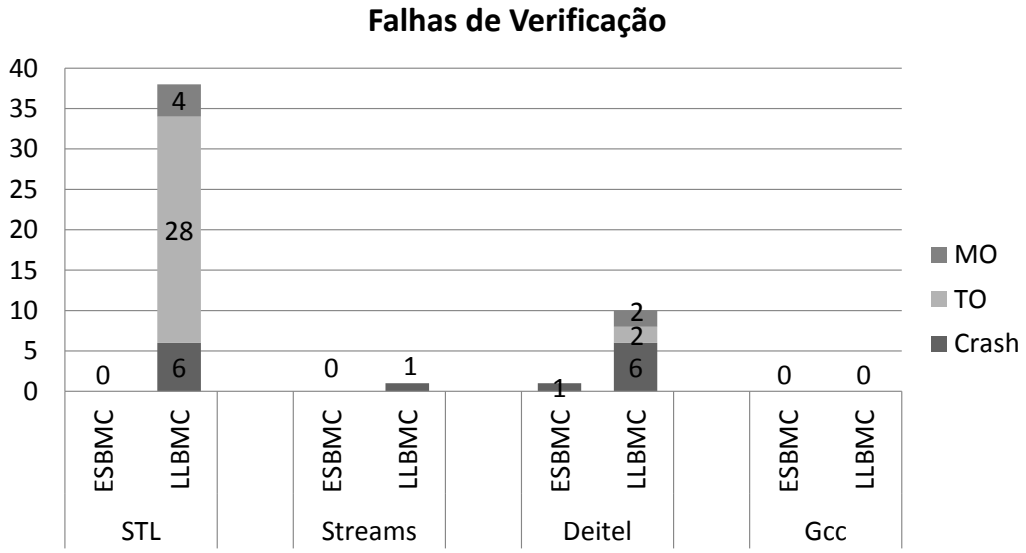


Figura 4.11: Comparativo das falhas de verificação entre ESBMC e LLBMC.

semelhante é mostrado na Figura 4.10, onde o ESBMC apresenta menos resultado incorreto, com exceção das categorias STL, onde ambas as ferramentas apresentam a mesma quantidade de erros, e a categoria Gcc, onde o LLBMC apresenta menos erros. Por fim, na Figura 4.11, que apresenta as falhas de verificação, ou seja, se a ferramenta abortou durante a verificação. O ESBMC somente falha em um dos casos de teste, enquanto o LLBMC apresenta falhas em quase todas as categorias.

As Figuras 4.12 e 4.13 apresentam os tempos de verificação de ambas as ferramentas, por categoria e total, respectivamente. Pode-se notar que o LLBMC leva mais tempo para verificar a categoria STL e Deitel, principalmente quando os casos de teste apresentam ponteiros e iteradores. Por outro lado, o tempo de verificação na categoria *streams* é menor. Porém, se for feita a análise das Figuras 4.9 e 4.10, é possível ver que o LLBMC verifica corretamente menos casos de teste e apresenta maior número de erros nesta categoria, diminuindo sua precisão em favorecimento do tempo. Vale ressaltar também que no tempo total de verificação, o ESBMC foi 5,9 vezes mais rápido do que o LLBMC.

O ESBMC apresentou melhores resultados do que o LLBMC, sendo capaz de verificar corretamente mais casos de teste, em menos tempo. No resultado geral, o ESBMC verificou todas as categorias em 20162 segundos (aproximadamente 5,6 horas) e verificou corretamente 1075 de 1182 casos de teste (91%) enquanto que o LLBMC verificou todas as categorias em

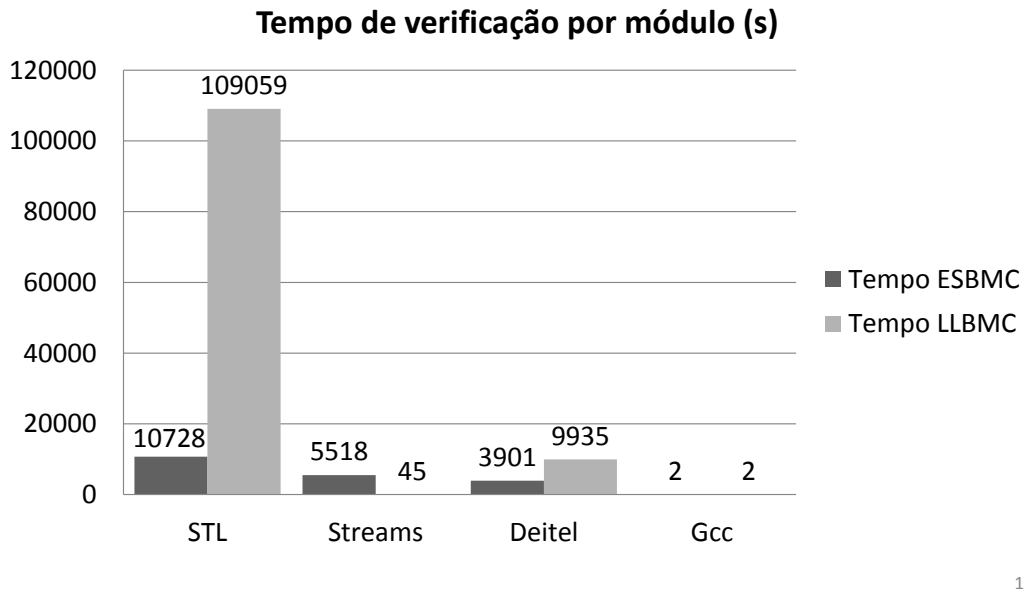


Figura 4.12: Comparativo dos tempos de verificação entre ESBMC e LLBMC por categoria.

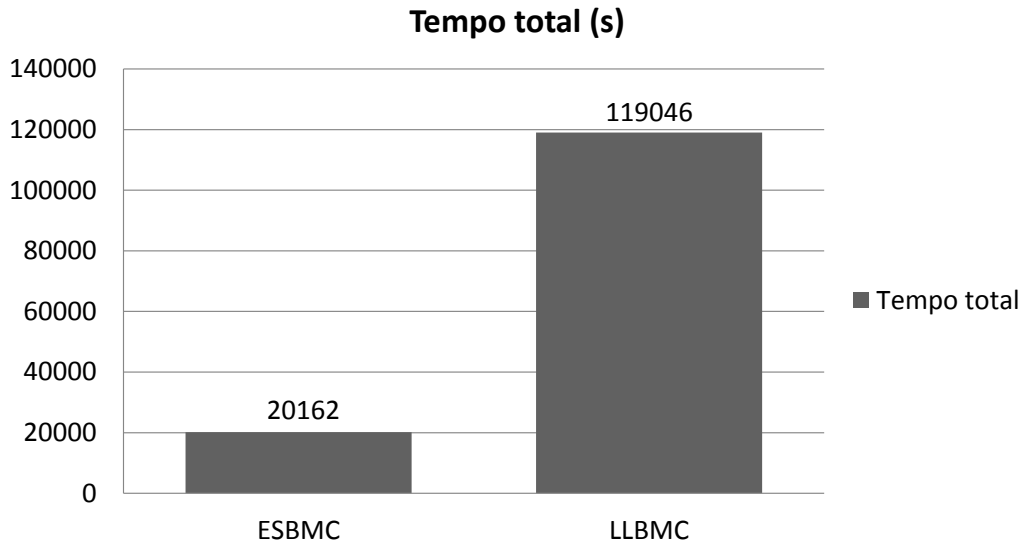


Figura 4.13: Comparativo dos tempo total de verificação entre ESBMC e LLBMC.

119045 segundos (aproximadamente 33 horas) e verificou corretamente 798 de 1182 casos de teste (67%). Por fim, note que o ESBMC não aborta devido a *memory-out* ou *time-out* em nenhuma categoria.

Um dos objetivos deste trabalho é alcançado, e superado, nesta seção. A ferramenta

proposta neste trabalho obteve melhores resultados do que a ferramenta estado da arte LLBMC, tanto em termos de cobertura quanto tempo de verificação. Além disso, a ferramenta LLBMC somente consegue verificar os casos de teste, se o tratamento de exceções for desabilitado, o que não acontece quando a ferramenta ESBMC é utilizada na verificação. Desta forma, pode-se concluir que a ferramenta ESBMC apresenta uma maior cobertura quando utilizada para verificação de casos de teste que utilizem STL (que faz uso de *templates*) e tratamento de exceções. Porém, em uma categoria que contém uso altamente especializados dos *templates*, a ferramenta LLBMC conseguiu verificar corretamente mais casos de teste.

4.4.3 Aplicação *Sniffer*

Essa seção descreve os resultados do processo de verificação utilizando o ESBMC e o LLBMC, em um programa *sniffer*, disponibilizado pelo Instituto Nokia de Tecnologia (INdT), que é responsável por captura e monitoração do tráfico de uma rede que suporta *Message Transfer Part Level 3 User Adaptation Layer* (M3UA) [46]; ele oferece o transporte de protocolos SS7 (*Signaling System 7*) e usa os serviços disponibilizados pelo *Stream Control Transmission Protocol* (SCTP). O código fonte do programa *sniffer* contém 20 classes, 85 métodos e 2800 linhas de código em C++.

As seguintes propriedades foram verificadas no programa *sniffer*: violação de limites de vetor, divisão por zero e *underflow* e *overflow* aritmético. Devido a questões de confidencialidade, o ESBMC somente foi capaz de verificar 50 dos 85 métodos (pois não foram disponibilizados acessos a algumas classes externas requeridas pelo código fonte do programa *sniffer*). Do código verificado, o ESBMC foi capaz de identificar cinco erros, relacionados a *underflow* e *overflow* aritmético enquanto que o LLBMC foi capaz de identificar somente três deles. Todos os erros foram reportados aos desenvolvedores que confirmaram os erros.

Como exemplo dos erros encontrados, a Figura 4.14 mostra um trecho de código do método *getPayloadSize* da classe *PacketM3UA*. Neste, um *overflow* aritmético pode ocorrer, pois o método retorna *ntohs*, um inteiro sem sinal, mas o método *getPayloadSize* deve retornar um valor do tipo inteiro com sinal. Uma possível solução para o erro é trocar o tipo do retorno do método *getPayloadSize* para inteiro sem sinal.

```
1 int PacketM3UA::getPayloadSize() {
2   return ntohs(m3uaParamHeader->paramSize)
3             - (M3UA_PROTOCOL_DATA_HEADER_SIZE
4             + M3UA_PARAMETER_HEADER_SIZE);
5 }
```

Figura 4.14: *Overflow* aritmético na operação de *typecast* no método *getPayloadSize*.

4.5 Resumo

Neste capítulo, inicialmente foi mostrado o funcionamento de *templates* em programas C++, além dos passos no processo de verificação de programas C++ que utilizem esse recurso. Compreender essa poderosa funcionalidade da linguagem é fundamental para o desenvolvimento deste trabalho pois, a grande maioria dos casos de teste verificados utilizam *templates* para facilitar o desenvolvimento de programas. Em especial, *templates* são a base da Biblioteca de *Templates* Padrão (STL), responsável por oferecer ao desenvolvedor um conjunto de contêineres (e.g., vetores, listas e pilhas) e algoritmos (e.g., *quick sort* e *heap sort*), que são utilizados para os mais diversos objetivos.

Para facilitar a verificação de programas C++ que contenham STL, o ESBMC utiliza um modelo operacional, chamado COM. Esse modelo é uma versão simplificada da STL, onde trechos de códigos que não são importantes do ponto de vista da verificação, são removidos, a fim de diminuir a complexidade do processo de verificação. Neste trabalho, o COM é modelado utilizando uma linguagem descritiva que, além de definir dos tipos básicos para operações, define um conjunto de métodos para inserção, remoção e busca nos contêineres.

Em seguida, é mostrado o processo de verificação de tratamento de exceções em programas C++. O tratamento de exceções é uma funcionalidade oferecida pela linguagem C++ que tem como objetivo evitar que um programa finalize abruptamente em caso de erros. Faz-se óbvio o fato de que o tratamento de exceções é outra funcionalidade essencial no desenvolvimento de programas C++ robustos. Para verificar programas que contenham tratamento de exceções foi necessário definir uma linguagem descritiva, similar à linguagem descritiva de contêineres, que foi utilizada para modelar o comportamento dos componentes dessa funcionalidade, expressões *throw* e blocos *try* e *catch*. O documento de referência da linguagem C++ define várias regras de conexão entre esses elementos, que foram formalizadas e desenvolvidas neste trabalho.

Por fim, foi mostrado o resultado da verificação de mais de 1100 casos de teste, incluindo

uma aplicação comercial, utilizando a ferramenta ESBMC, e a ferramenta LLBMC, outra ferramenta de verificação de programas C++. Nos casos de teste, o ESBMC mostrou-se superior à ferramenta comparada, sendo capaz de verificar corretamente mais casos de teste, em menor tempo. Além disso, o ESBMC foi capaz de verificar programas que continham tratamento de exceções, atualmente não suportado pelo LLBMC, e programas com especificação de exceções, que nenhuma ferramenta de verificação disponível atualmente é capaz de suportar.

Capítulo 5

Prova por Indução Matemática

A indução matemática pode ser aplicada para provar propriedades em BMC. O algoritmo descrito neste capítulo é chamado *k-induction*, baseia-se em indução matemática para a prova de corretude de propriedades. Algoritmo de indução matemática já foi utilizado para a prova de corretude de propriedades [37, 36, 38, 39, 41] e alguns estão descritos na Seção 3.2. A forma mais simples de *k-induction* consiste de dois passos: caso base e passo indutivo. No caso base, é verificado se uma dada propriedade P é verdadeira a partir de um estado inicial em k passos. No passo indutivo, é verificado se a propriedade P é verdadeira para os próximos estados do sistema. Para formalizar a implementação do *k-induction*, será utilizada a Lógica Hoare [51], descrita na Seção 2.8. Neste capítulo, além de apresentar o algoritmo *k-induction*, será mostrada a explicação de cada passo e as transformações realizadas por cada um deles durante a execução do algoritmo. Ao final, será mostrado o algoritmo *k-induction* paralelo, onde cada passo executa de forma concorrente, seguido dos resultados experimentais do algoritmo.

5.1 Algoritmo *K-Induction*

No ESBMC, a verificação utilizando o *k-induction* consiste de três passos: caso base, condição adiante e passo indutivo [52].

O Algoritmo 5.1 é o algoritmo *k-induction*. No caso base, o algoritmo tenta encontrar um contraexemplo de até um dado número máximo de iterações k. Na condição adiante, o algoritmo verifica se todos os estados foram alcançados dentro das k iterações e, no passo indutivo, o algoritmo verifica que, se a propriedade é verdadeira em k iterações, então deve ser

verdadeira para as próximas iterações. O algoritmo executa até um limite máximo de iterações e somente incrementa o valor de *k* caso não consiga provar a negação da propriedade durante o caso base.

```
1  k = 1
2  enquanto k <= max_iteracoes faca
3    se caso_base(k) entao
4      retorne contraexemplo s[0..k]
5    senao
6      k=k+1
7      se condicao_adiante(k) entao
8        retorne verdadeiro
9      senao
10       se passo_indutivo(k) entao
11         retorne verdadeiro
12       fim-se
13     fim-se
14   fim-se
15 fim-enquanto
16 retorne falso
```

Algoritmo 5.1: O algoritmo *k-induction*.

Para os três passos do algoritmo, uma série de transformações são realizadas. Em especial, os laços *for* e *do while* são transformados em laços *while*, e são convertidos segundo as Fórmulas 5.1 e 5.2, respectivamente.

$$for(A;c;B) \wedge D \wedge E \iff A \wedge enquanto(c) \wedge D \wedge B \wedge E \tag{5.1}$$

onde A é a condição inicial do laço, c é a condição de parada do laço, B é o incremento de cada iteração sobre A, D é o código dentro do laço *for* e E é um código após o laço *for*. O laço *do while* é transformado segundo a Fórmula 5.2.

$$do\,A\,while(c) \wedge B \iff A \wedge enquanto(c) \wedge B \tag{5.2}$$

onde A é o código dentro do laço *do while*, c é a condição de parada do laço e B é um código após o laço *do while*. Os laços *do while* são convertidos em laços *while*, com uma única diferença,

o código dentro do laço deve executar pelo menos uma vez antes da condição de parada ser verificada.

Além disso, as expressões *if-else* também são transformadas segundo a Fórmula 5.3.

$$if(c)\,A\,else\,B \Longleftrightarrow (\neg c \wedge B) \vee (c \wedge A) \tag{5.3}$$

onde c é a condição da expressão *if*, A é o código a ser executado caso c seja verdadeiro e B é o código a ser executado caso c seja falso.

Além dessas transformações, os três passos do algoritmo *k-induction* inserem suposições e assertivas com o intuito de provar a corretude das propriedades. Como exemplo de aplicação do algoritmo *k-induction*, será utilizado um programa extraído dos *benchmarks* do SV-COMP [53], conforme mostrado na Figura 5.1.

```
1  int main() {
2    long long int i, sn=0;
3    unsigned int n;
4    assume (n>=1);
5    i=1;
6    while(i<=n) {
7      sn = sn + a;
8      i++;
9    }
10   assert(sn==n*a);
11 }
```

Figura 5.1: Código exemplo para prova por indução matemática.

As variáveis *i* e *sn* são declaradas com um tipo de dado maior que o tipo de dado da variável *n* para evitar *overflow* aritmético. Matematicamente, o código acima representa simplesmente a implementação do somatório dado pela seguinte fórmula:

$$S_n = \sum_{i=1}^{n} a = na, n \geq 1 \tag{5.4}$$

Note que no código da Figura 5.1, a propriedade (representada pela assertiva da linha 10) deve ser verdadeira para qualquer valor de n. Este código pode ser modelado através de um sistema de transições (mostrado na Figura 5.2). O estado inicial representa o trecho de código antes do laço, contendo a inicialização das variáveis e suposições sobre as mesmas. Os dois estados seguintes representam o trecho de código que é executado no laço, o cálculo da variável *sn* e o incremento da variável *i*. A transição τ representa a mudança imediata de estado, nesse

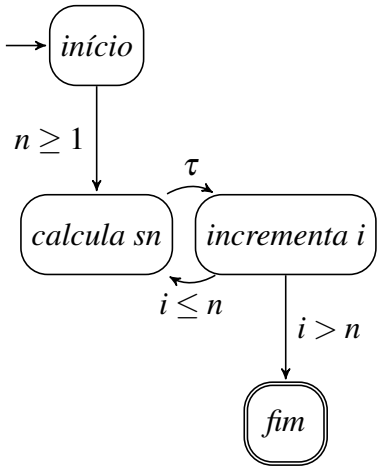


Figura 5.2: Sistema de transições do código mostrado na Figura 5.1

caso, após o cálculo de *sn*. O estado final, representa o trecho de código após o laço, contendo a assertiva sobre o valor da variável *n*.

Note que a técnica BMC apresenta dificuldades em provar esse programa, devido ao fato do limite superior do laço ser não determinístico. Devido a esta condição, o laço será desdobrado $2^n - 1$ vezes, o que necessitará de uma grande quantidade de memória e tempo para encontrar uma solução. Basicamente, a verificação ficaria executando simbolicamente o incremento da variável *i* e o cálculo da variável *sn*.

Para solucionar o problema de desdobrar o laço $2^n - 1$ vezes, são realizadas as transformações descritas anteriormente. O caso base inicializa os limites do condicionante do laço com valores não determinísticos com o intuito de explorar todos os possíveis estados implicitamente. Sendo assim, as pré- e pós-condições do código presente na Figura 5.1 tornam-se:

$$\begin{aligned} P &\iff n_1 = nondet\_uint \wedge n_1 \geq 1 \wedge sn_1 = 0 \wedge i_1 = 1 \\ R &\iff i_k > n_1 \Rightarrow sn_k = n_1 \times a \end{aligned} \tag{5.5}$$

onde P e R são as pré- e pós-condições, respectivamente, e *nondet_uint* é uma função não determinística, que pode retornar qualquer valor do tipo *unsigned int*. Nas pré-condições, n_1 representa a primeira atribuição para a variável *n*, de um valor não determinístico maior do que ou igual a um. Note que, esta atribuição de valores não determinísticos garante que o verificador explore todos os possíveis estados implicitamente. Além disso, sn_1 representa a primeira atribuição para a variável *sn* com valor zero e i_1 representa a condição inicial do laço. Nas pós-condições, sn_k representa a atribuição $n+1$ para a variável *sn* da Figura 5.1, que é verdadeira se $i_k > n_1$. O trecho de código, que não é pré- ou pós-condição, é representado pela

variável Q e este não sofre nenhuma alteração durante as transformações do caso base.

O código resultante das transformações do caso base pode ser visto na Figura 5.3. A instrução *assume* (linha 10), contendo a condição de parada, elimina todos os caminhos de execução que não satisfazem a restrição $i > n$. Isso assegura que o caso base encontre um contraexemplo de profundidade k sem reportar nenhum falso negativo.

```
1  int main() {
2    long long int i, sn=0;
3    unsigned int n;
4    assume (n>=1);
5    i=1;
6    while(i<=n) {
7      sn = sn + a;
8      i++;
9    }
10   assume(i>n); //suposicao de desdobramento
11   assert(sn==n*a);
12 }
```

Figura 5.3: Código exemplo para prova por indução, durante o caso base.

Na condição adiante, o algoritmo *k-induction* tenta provar se o laço foi suficientemente desdobrado e se a propriedade é válida em todos os estados alcançáveis até a profundidade k. As pré- e pós-condições do código presente na Figura 5.1, durante a condição adiante, são definidas como:

$$\begin{aligned} P &\iff n_1 = nondet\_uint \wedge n_1 \geq 1 \wedge sn_1 = 0 \wedge i_1 = 1 \\ R &\iff i_k > n_k \wedge sn_k = n_k \times a \end{aligned} \tag{5.6}$$

As pré-condições do caso adiante são idênticas à do caso base. Porém, nas pós-condições R, existe a verificação se o laço foi suficientemente expandido, representado pela comparação $i_k > n_1$, onde i_k representa o valor da variável i na iteração $n+1$ (devido às atribuições que ocorrem no laço *while* da Figura 5.1 que possui n iterações). O código resultante das transformações do caso adiante pode ser visto na Figura 5.4. O caso adiante tenta provar se o laço foi suficientemente desdobrado (verificando o laço invariante na linha 10) e se a propriedade é válida em todos os estados alcançáveis dentro de k desdobramentos (através da assertiva na linha 11).

No passo indutivo, o algoritmo *k-induction* tenta provar que, se a propriedade é válida até a profundidade k, a mesma deve ser válida para o próximo valor de k. Diversas transformações são feitas no código original. Primeiramente, é definido um tipo de estrutura chamada

```
1  int main() {
2    long long int i, sn=0;
3    unsigned int n;
4    assume (n>=1);
5    i=1;
6    while(i<=n) {
7      sn = sn + a;
8      i++;
9    }
10   assert(i>n); // checa laço invariante
11   assert(sn==n*a);
12 }
```

Figura 5.4: Código exemplo para prova por indução, durante o caso adiante.

statet, contendo todas as variáveis presentes dentro do laço e na condição de parada desse laço. Em seguida, é declarado uma variável do tipo *statet*, chamada *cs* (*current state*), que é responsável por armazenar os valores das variável presentes no laço em uma determinada iteração. É declarado também um vetor de estados de tamanho igual ao número de iterações do laço, chamado *sv* (*state vector*), que irá armazenar os valores de todas as variáveis do laço em cada iteração.

Antes do início do laço, todas as variáveis são inicializadas com valores não determinísticos e armazenadas no vetor de estados, durante a primeira iteração do laço. Dentro do laço, após o armazenamento do estado atual e o código do laço, todas as variáveis do estado atual são atualizadas com os valores da iteração atual. Uma instrução *assume* é inserida com a condição de que o estado atual é diferente do estado anterior, para evitar que estados redundantes sejam inseridos no vetor de estados (neste caso a comparação entre todos os estados é feita de forma incremental). Por fim, após o laço, tem-se uma instrução *assume* igual à instrução inserida no caso base. As pré- e pós-condições do código presente na Figura 5.1, durante o passo indutivo, são definidas como:

$$\begin{aligned} P \iff & n_1 = nondet\_uint \wedge n_1 \geq 1 \wedge sn_1 = 0 \wedge i_1 = 1 \\ & \wedge cs_1.v_0 = nondet\_uint \\ & \ldots \\ & \wedge cs_1.v_m = nondet\_uint \\ R \iff & i_k > n_1 \Rightarrow sn_k = n_1 \times a \end{aligned} \tag{5.7}$$

Nas pré-condições P, além da inicialização dos valores das variáveis, é necessário iniciar o valor de todas as variáveis contidas no estado atual *cs*, com valores não determinísticos, onde

m é o número de variáveis (automáticas e estáticas) que são utilizadas no programa. As pós-condições não mudam em relação ao caso base, e contém somente a propriedade que se deseja provar. No conjunto de instruções Q, são feitas alterações no código para salvar o valor das variáveis na iteração i atual, da seguinte maneira:

$$\begin{aligned} Q \Longleftrightarrow\ & sv[i-1] = cs_i \wedge S \\ & \wedge\, cs_i.v_0 = v_{0i} \\ & \ldots \\ & \wedge\, cs_i.v_m = v_{mi} \end{aligned} \tag{5.8}$$

No conjunto de instruções Q, $sv[i-1]$ representa a posição no vetor para salvar o estado atual cs_i, S representa o código dentro do laço e a série de atribuições, semelhantes às da pré-condição, que representam o estado atual cs_i salvando o valor das variáveis na iteração i. O código modificado pelo passo indutivo pode ser visto na Figura 5.5. Assim como no caso base, o passo indutivo também inclui uma instrução *assume*, contendo a condição de parada. Diferentemente do caso base, que tenta provar a negação da propriedade, o passo indutivo tenta provar que a propriedade contida no *assert* é verdadeira para qualquer valor de n.

5.2 Algoritmo *K-Induction* Paralelo

A partir do entendimento de cada passo do algoritmo *k-induction* (Figura 5.1), nota-se que as transformações e a aplicação da técnica BMC em cada passo é feita de forma totalmente independente. Portanto, além da forma sequencial de execução do algoritmo, é possível que a execução de cada passo seja feita de forma paralela.

A abordagem escolhida para a implementação do algoritmo paralelo foi a utilização de quatro processos ao invés de quatro *threads* (essa escolha baseia-se no fato do solucionador Z3 não ser *thread safe*). O processo pai é responsável pela inicialização dos três processos filhos, pela lógica de execução do algoritmo *k-induction* e a exibição do resultado final da verificação. Cada processo filho será responsável por um passo do algoritmo *k-induction*, ou seja, um processo para o caso base, um para a condição adiante e um para o passo indutivo.

Para criação de novos processos filhos, fez-se uso do comando *fork*, que cria uma nova cópia do processo pai e inicia a execução a partir do ponto onde foi executado [54], e para a comunicação interprocessos utilizou-se dois *pipes* em cada processo. *Pipes* permitem comunicação sequencial entre um processo e outro processo relacionado [55]. A necessidade de 2

```
1 //variaveis presentes no laço
2 typedef struct state {
3   long long int i, sn;
4   unsigned int n;
5 } statet;
6 int main() {
7   long long int i, sn=0;
8   unsigned int n=nondet_uint();
9   assume (n>=1);
10  i=1;
11  //declara estado atual e vetor de estados
12  statet cs, sv[n];
13  //atribui valores nao deterministicos
14  cs.i=nondet_uint(); cs.sn=nondet_uint();
15  cs.n=n;
16  while(i<=n) {
17    sv[i-1]=cs;   //armazena estado atual
18    sn = sn + a;  //codigo dentro do loop
19    //atualiza as variaveis do estado atual
20    cs.i=i; cs.sn=sn; cs.n=n;
21    //remove estados redundantes
22    assume(sv[i-1]!=cs);
23    i++;
24  }
25  assume(i>n);//suposicao de desdobramento
26  assert(sn==n*a);
27 }
```

Figura 5.5: Código exemplo para prova por indução, durante o passo indutivo.

pipes deriva do fato que há a necessidade de comunicação bi-direcional entre os processos nos diversos cenários da execução em paralelo do algoritmo *k-induction*.

As Figuras 5.6, 5.7, 5.8 e 5.9 mostram o exemplo dos quatro possíveis cenários provenientes da execução em paralelo do algoritmo *k-induction*. Note que todos os processos filhos comunicam o resultado de cada iteração ao processo pai e este por sua vez decide o rumo da execução. No primeiro cenário, mostrado na Figura 5.6, o processo pai inicia os três processos filhos que iniciam o processo de verificação. Durante a verificação do caso base, um defeito é encontrado para $k = 4$. Nesse momento, o processo do caso base comunica o processo pai, que manda sinais para os dois outros processos filhos finalizarem. Ao final, o processo pai exibe que foi encontrado um defeito no programa.

Os cenários dois e três (mostrados nas Figuras 5.7 e 5.8) são semelhantes, com a diferença de se tratar dos processos da condição adiante e do passo indutivo, respectivamente. Nesses casos, o processo em questão comunica ao processo pai que encontrou uma solução e finaliza. Nesse momento, o processo pai manda sinais para o processo do caso base e para os

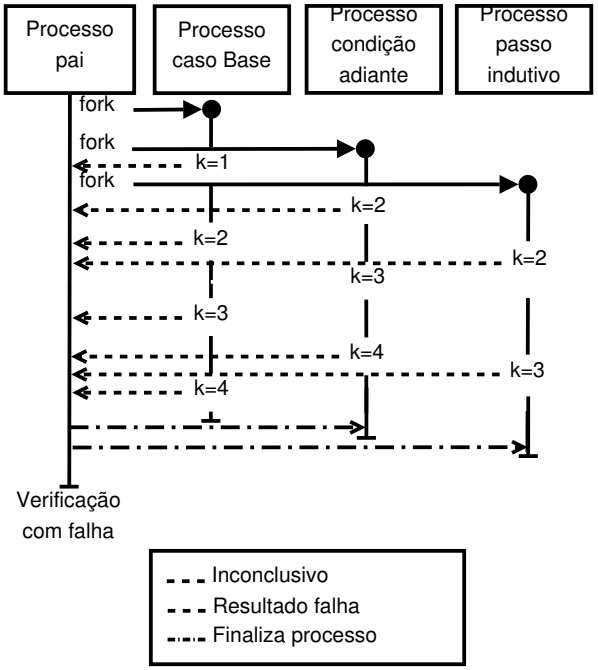


Figura 5.6: Caso Base provou a negação da propriedade.

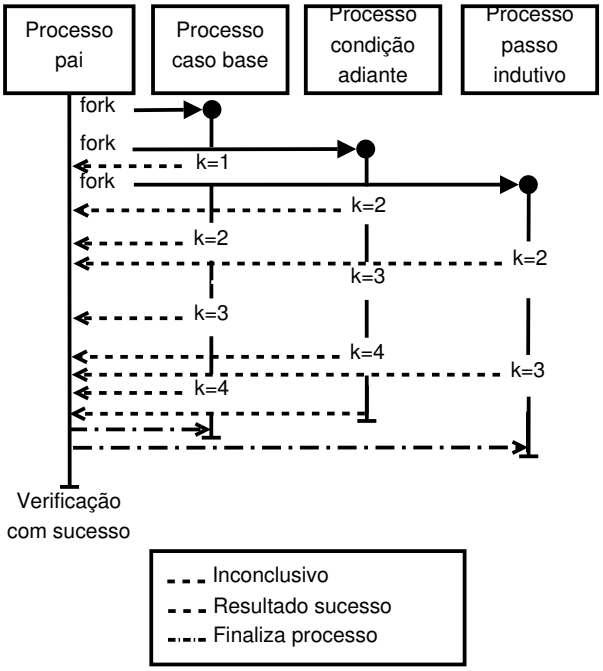


Figura 5.7: Condição adiante provou a corretude da propriedade.

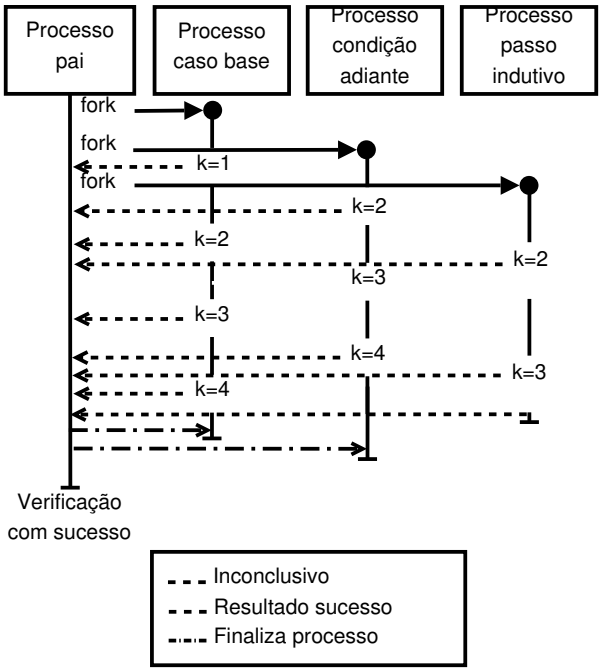


Figura 5.8: Passo indutivo provou a corretude da propriedade.

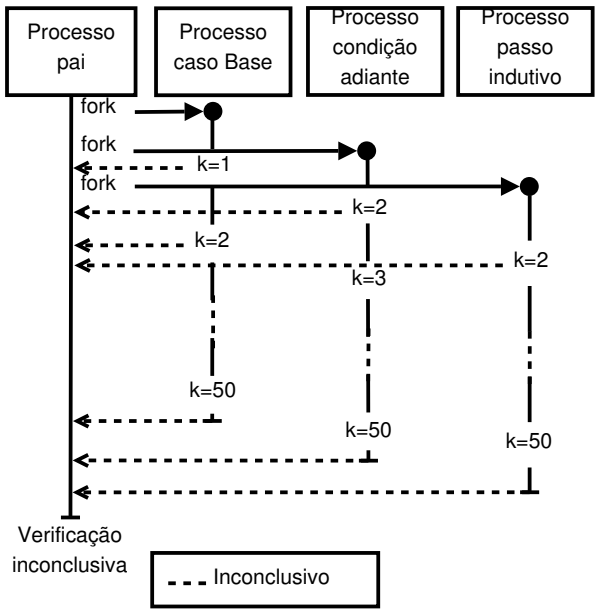


Figura 5.9: O Algoritmo não foi possível provar a corretude ou negação da propriedade.

processos passo indutivo e processo condição adiante, nos cenários dois e três, respectivamente. Ao final, o processo pai exibe que conseguiu provar a propriedade com sucesso.

O quarto e último cenário, apresentado na Figura 5.9, acontece quando os três processos filhos atingiram o valor máximo de k e não encontraram uma solução. Nesse caso, os processos filhos comunicam ao processo pai que finalizaram sem encontrar um defeito e o processo pai exibe que a ferramenta ESBMC não foi capaz de provar ou negar a propriedade.

5.3 Resultados Experimentais

Esta seção está dividida em três partes. O ambiente onde os experimentos foram realizados é descrito na Seção 5.3.1 enquanto que a Seção 5.3.2 apresenta os resultados das duas formas do algoritmo *k-induction*, sequencial e paralela, usando um conjunto de casos de teste de programas C++.

5.3.1 Ambiente do Experimento

Para avaliar o algoritmo *k-induction*, foram utilizados 75 casos de teste da categoria *loops*[1] do SV-COMP [53] e 24 casos de teste feitos em C++, envolvendo funcionalidades como classes, templates e exceções. A categoria *loops* foi escolhida pelo fato de possuir diversos programas que requerem análise da condição de parada dos laços. O conjunto de programas, está dividido da seguinte forma: 49 casos de testes contém propriedades válidas, ou seja, o ESBMC deve conseguir provar a propriedade; 50 casos de testes contém propriedades inválidas, ou seja, o ESBMC deve ser capaz de provar a negação da propriedade.

Todos os experimentos foram conduzidos em um computador com processador Intel Core i7-2600, 3.40Ghz com 24GB de memória RAM com Ubuntu 11.10 64-bits. Para avaliar o algoritmo *k-induction*, foram ajustados um limite de tempo de verificação e um limite de memória, de 900 segundos (15 minutos) e 12GB de memória, respectivamente. Os casos de teste foram verificados com o ESBMC v1.22.

N	Caso de Teste	Sequencial		Paralelo	
		Passo	T (s)	Passo	T (s)
1	array	FC	0.17	FC	0.11
2	array_bug	BC	0.18	BC	0.11
3	bist_cell	-	900	-	900
4	bubble_sort	IS	0.25	IS	0.2
5	byte_add_bug	†	1.38	BC	0.13
6	check_if	IS	0.18	IS	0.08
7	check_if_bug	BC	0.19	BC	0.11
8	count_down	IS	0.18	IS	0.07
9	count_down_02	IS	0.18	IS	0.07
10	count_up_down	IS	0.17	IS	0.07
11	count_up_down_bug	BC	0.16	BC	0.1
12	digital-controller	IS	0.34	FC	0.13
13	eureka_01	IS	0.33	FC	0.36
14	eureka_01_bug	BC	91.62	BC	0.22
15	eureka_05	FC	0.28	FC	0.13
16	for_bounded_loop1	BC	0.19	BC	0.08
17	for_infinite_loop_1	IS	0.17	IS	0.06
18	for_infinite_loop_2	IS	0.17	IS	0.07
19	gcd-with-time	BC	0.26	BC	0.1
20	insertion_sort	IS	0.35	IS	0.24
21	insertion_sort_bug	BC	2.42	BC	0.34
22	invert_string	FC	0.45	FC	0.09
23	invert_string_bug	BC	0.31	BC	0.09
24	kinductor_01	IS	0.18	IS	0.06
25	kundu	-	TO	-	TO
26	kundu_bug	BC	5.97	BC	0.54
27	linear_search	IS	0.2	IS	0.09
28	linear_search_bug	BC	6.65	BC	0.66
29	ludcmp	IS	0.68	FC	0.19
30	ludcmp_bad	-	TO	BC	3.23
31	matrix	FC	0.18	FC	0.07
32	matrix_bug	BC	0.29	BC	0.12
33	mem_slave_tlm	-	TO	-	TO
34	nec11	IS	0.18	IS	0.07
35	nec11_bug	BC	0.17	BC	0.1
36	nec20_bug	BC	0.19	BC	0.11
37	nec24	-	TO	-	TO
38	nec40	FC	0.17	IS	0.07
39	nec40_bug	FC	0.17	FC	0.06
40	pc_sfifo_1	-	TO	-	TO
41	pc_sfifo_2_BUG	BC	0.26	BC	0.08
42	problem01_20_unsafe	BC	26.43	BC	9.68
43	string	IS	1.48	FC	0.63

44	string_bug	BC	0.5	BC	0.16
45	sum01	IS	0.18	IS	0.12
46	sum01_bug	BC	0.47	BC	0.18
47	sum01_bug02	BC	0.32	BC	0.12
48	sum02	IS	0.19	IS	0.13
49	sum03	IS	0.17	IS	0.11
50	sum03_bug	BC	0.33	BC	0.08
51	sum04	IS	0.21	IS	0.15
52	sum04_bug	BC	0.23	BC	0.12
53	sum05_bug	BC	0.19	BC	0.08
54	sum_array	IS	0.21	IS	0.16
55	sum_array_bug	BC	0.77	BC	0.09
56	terminator_01	IS	0.17	IS	0.07
57	terminator_01_bug	BC	0.17	BC	0.09
58	terminator_02	IS	0.18	IS	0.1
59	terminator_02_bug	BC	0.17	BC	0.08
60	terminator_03	IS	0.17	IS	0.07
61	terminator_03_bug	BC	0.17	BC	0.07
62	terminator_04	IS	0.19	IS	0.08
63	terminator_04_bug	BC	0.17	BC	0.07
64	terminator_05	IS	0.19	IS	0.08
65	terminator_05_bug	BC	0.17	BC	0.08
66	terminator_06	IS	0.19	IS	0.1
67	terminator_06_bug	BC	0.17	BC	0.11
68	token_ring01	-	TO	-	TO
69	token_ring01_bug	BC	1.47	BC	0.21
70	toy	-	TO	-	TO
71	transmitter_bug	BC	0.64	BC	0.12
72	trex01	IS	0.21	IS	0.14
73	trex01_bug	BC	0.18	BC	0.07
74	trex02	IS	0.19	IS	0.07
75	trex02_bug	BC	0.17	BC	0.08
76	trex03	IS	0.2	IS	0.12
77	trex03_bug	BC	0.18	BC	0.12
78	trex04	IS	0.19	IS	0.08
79	verisec_NetBSD_bad	BC	0.2	BC	0.09
80	verisec_NetBSD_ok	IS	0.18	FC	0.11
81	verisec_OpenSER_bad	BC	0.31	BC	0.2
82	verisec_OpenSER_ok	FC	1.26	FC	0.74
83	verisec_sendmail_bad	BC	2.98	BC	2.16
84	verisec_sendmail_ok	IS	0.35	FC	0.23
85	vogal	IS	0.29	FC	0.18
86	vogal_bug	BC	3.74	BC	0.8
87	while_loop_1	IS	0.17	IS	0.06
88	while_loop_2	IS	0.17	IS	0.06

89	while_loop_3	BC	0.17	BC	0.07
90	while_loop_4	BC	0.17	BC	0.1
91	znec_ex3-dtor-throw	†	1.61	BC	0.14
92	znec_ex6-std-uncaught-dtor	†	0.57	BC	0.15
93	zpriority_queue_size_bug	†	3.22	BC	1.05
94	zqueue_front_bug	†	3.82	BC	1.29
95	zqueue_pop_bug	†	3.8	BC	1.27
96	zstack_empty_bug	†	104.08	BC	0.69
97	zstack_top_bug	†	101.42	BC	0.73
98	zsum_class	IS	0.19	IS	0.12
99	zsum_class_bug	BC	1.71	BC	0.31
		Total Corretos	Tempo Total	Total Corretos	Tempo Total
	Total	79	7582.85	92	6332.78

Tabela 5.1: Resultados comparativos entre execução sequencial e paralela do algoritmo *k-induction*

5.3.2 Resultados da Categoria *Loops*

Nessa seção, está descrita a avaliação da forma sequencial e paralela do algoritmo *k-induction* utilizando a ferramenta ESBMC. A comparação dos resultados do ESBMC (utilizando o algoritmo *k-induction*) com outras ferramentas, não será apresentada neste trabalho, uma vez que a edição do SV-COMP 2013 já compara os resultados da categoria *loops* usando diferentes ferramentas de verificação (conforme apresentado por Dirk Beyer [53]). Dentre seis ferramentas participantes do SV-COMP, o algoritmo *k-induction* sequencial obteve a segunda posição no *ranking* da categoria *loops*[2] [53]. A forma paralela não foi utilizada por que não estava pronta a tempo da competição. Será comparado o resultado dos casos de teste da ferramenta ESBMC utilizando as duas formas do algoritmo, sequencial e paralela. Não foi feita a comparação da ferramenta ESBMC com outra ferramenta, pois a que mais se aproxima do algoritmo desenvolvido nesta dissertação, K-Inductor [39], necessita de alteração manual do caso de teste, o que pode inserir defeitos no processo de verificação e não apresentar resultados válidos.

A Tabela 5.1 mostra os resultados dos experimentos. Nesta, *N* representa o número do caso de teste, *Caso de Teste* é o nome do caso de teste, *Passo* apresenta o último passo executado pelo algoritmo para provar a corretude ou a negação das propriedades no programa,

[1]https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp13
[2]http://sv-comp.sosy-lab.org/2013/results/index.php

T apresenta o tempo necessário pelo algoritmo, em segundos, até encontrar uma solução, e *Sequencial* e *Paralelo*, apresentam os resultados obtidos para cada uma das abordagens de execução do algoritmo *k-induction*. Na coluna *Passo*, as siglas *BC*, *FC* e *IS* significam caso base, condição adiante e passo indutivo, respectivamente. Além disso, o símbolo † representa que a ferramenta abortou durante a verificação e, na coluna tempo, a sigla TO significa que o caso de teste extrapolou o tempo de verificação definido.

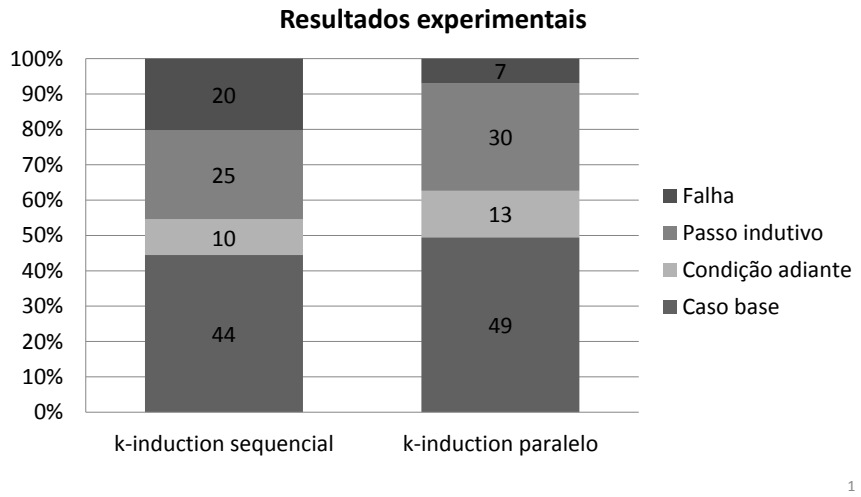


Figura 5.10: Resultados comparativos da verificação, utilizando a forma sequencial e paralela.

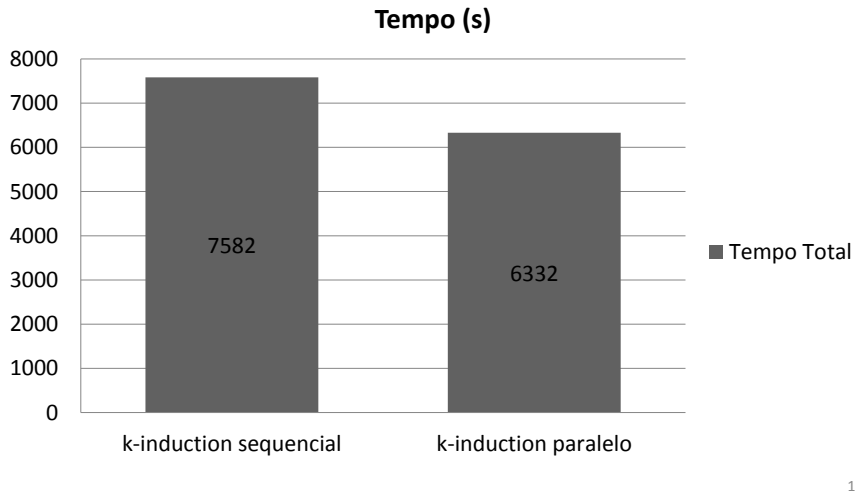


Figura 5.11: Resultados comparativos do tempo verificação, utilizando a forma sequencial e paralela.

As Figuras 5.10 e 5.11 apresentam o resultado comparativo e o tempo total de verifica-

ção entre as duas formas do algoritmo *k-induction*, respectivamente. Como pode ser visto na Figura 5.10, utilizando a forma sequencial, 20 casos de teste falharam, enquanto que utilizando a forma paralela, 7 casos de teste falharam. Grande parte das falhas ocorreram devido a extrapolamento de tempo pois os casos de teste possuem diversos laços aninhados, o que aumenta a complexidade das VCs geradas e, consequentemente, o tempo necessário para o solucionador verificar a satisfabilidade das VCs. Em especial, no caso *ludcmp_bad*, utilizando a forma paralela, o caso base encontra o defeito na iteração 7, enquanto que utilizando a forma sequencial, o algoritmo não alcança essa iteração antes do tempo limite. Os casos em que a ferramenta abortou, ocorreram durante as transformações do passo indutivo, durante a execução simbólica do programa utilizando a nova representação intermediária introduzida recentemente na ferramenta [56]. As falhas não afetaram o funcionamento da abordagem paralela porque somente o processo do passo indutivo abortou nessas situações, e o processo de verificação continou normalmente pelos processos do caso base e condição adiante, sendo falsificados pelo caso base. Na forma sequencial, o caso base foi o passo que encontrou a solução em 44 casos de teste, enquanto o condição adiante encontrou a solução em 10 casos de teste e o passo indutivo encontrou em 25 casos de teste. Na forma paralela, o caso base foi o passo que encontrou a solução em 49 casos de teste, enquanto a condição adiante encontrou a solução em 13 casos de teste e o passo indutivo encontrou a solução em 30 casos de teste.

Em certos casos de teste, por exemplo, *string, verisec_sendmail_ok* e *vogal*, cada forma do algoritmo encontrou a solução em um passo diferente do algoritmo *k-induction*. Nesses casos, utilizando a forma sequencial, a prova ocorre em uma determinada profundidade k de desdobramento dos laços no passo indutivo. Na forma paralela, a prova ocorre em uma profundidade $k+1$ na condição adiante. Como o processo de verificação da condição adiante é mais rápida do que a verificação do passo indutivo, a forma paralela encontra a solução em um passo diferente do encontrado pela forma sequencial.

Como mostrado na Figura 5.11, forma sequencial, todos os casos de teste foram verificados em 7582 segundos, sendo verificados corretamente 79 de 99 (79%)[3], enquanto que na forma paralela, todos os casos de teste foram verificados em 6332 segundos, sendo verificados corretamente 92 de 99 (93%)[4]. Note que a forma paralela apresenta maior casos de verificação

[3]Os casos de teste foram verificados usando o comando: esbmc file.c −−k-induction −−k-step 100 −−memlimit 12g −−timeout 900s

[4]Os casos de teste foram verificados usando o comando: esbmc file.c −−k-induction-parallel −−k-step 100 −−memlimit 12g −−timeout 900s

correta e em menor tempo.

5.4 Resumo

Neste capítulo, inicialmente foi apresentado o algoritmo *k-induction*, que foi desenvolvido como alternativa ao algoritmo BMC. Enquanto o BMC tenta provar a negação de uma propriedade, o algoritmo *k-induction* tenta provar a corretude de uma propriedade. O algoritmo *k-induction* baseia-se em indução matemática e possui três passos: caso base, condição adiante e passo indutivo. No caso base, o algoritmo funciona da mesma forma que o BMC e tenta encontrar um contraexemplo até uma dada profundidade k. Na condição adiante, o algoritmo verifica se todos os estados do programa foram alcançados em k iterações e, no passo indutivo, o algoritmo tenta provar que, se a propriedade é verdadeira em k iterações, deve ser verdadeira para qualquer número de iterações.

Durante a execução do algoritmo, cada passo aplica diversas transformações na tentativa de provar a corretude de uma propriedade. Comum aos três passos, são as transformações aos laços *for*, *do while* e *while* e a expressões *if*, que são formalizadas utilizando Lógica de Primeira Ordem (como definida na Seção 2.1). Em seguida, são apresentadas as formalizações das transformações específicas de passo, que são modeladas segundo a lógica Hoare (como definida na Seção 2.8).

Cada passo do algoritmo faz transformações diferentes no programa original, que são utilizados de maneira independente no processo de verificação. Partindo dessa estrutura separada, é proposto uma abordagem paralela para a execução de cada passo do algoritmo. Nesta nova forma de execução, utilizou-se uma forma multiprocesso, onde são criados três processos filhos, um para cada passo do algoritmo. Além disso, o processo pai dos três processos é responsável pelo gerenciamento e tomada de decisão a partir da solução encontrada por cada processo. Os processos filhos somente se comunicam com o processo pai, que decide quando uma solução foi encontrada e pode encerrar os processos filhos.

Por fim, foi mostrado o resultado da verificação de mais de 95 casos de teste, utilizando a ferramenta ESBMC, com ambas as formas do algoritmo *k-induction*, sequencial e paralela. Nos casos de teste, a forma paralela do algoritmo se mostrou superior à forma sequencial, sendo capaz de verificar corretamente mais casos de teste, em menor tempo.

Capítulo 6

Conclusões

Nesse trabalho, dois problemas foram abordados: a verificação de programas C++, focando na verificação de programas C++ que contêm *templates* e tratamento de exceções; e a prova por indução matemática de programas C++, utilizando a técnica de *k-induction*, para provar a corretude de programas.

Para a verificação de programas C++, foi inicialmente descrita uma forma de verificar programas que contenham *templates*, que é dividida em duas etapas e, que ao final da etapa de checagem de tipo, os *templates* são removidos do processo de verificação de forma similar ao processo de compilação de programas [43]. Utilizando os *templates* como base, foi descrito e modelado o modelo operacional (COM) utilizado pelo ESBMC para verificação de programas que façam uso da Biblioteca de *Template* Padrão (STL).

Além disso, foi descrito o processo de verificação de programas C++ que contenham tratamento de exceções. Para a verificação destes, foram formalizadas as regras de conexão entre os componentes do processo de tratamento de exceções, descritas no documento de definição da linguagem C++ [47]. Em especial, foi descrito o processo de verificação de programas que contenham especificação de exceções, funcionalidade não suportada por outros verificadores estado da arte [6, 49]).

Os experimentos utilizados para validar o desenvolvimento da ferramenta de verificação de programas C++ foram compostos por programas que continham grande parte das funcionalidades que a linguagem C++ tem a oferecer, além de uma aplicação comercial, utilizada na área de telecomunicações. Os resultados mostram que a ferramenta ESBMC supera a ferramenta LLBMC, outra ferramenta estado da arte na verificação de códigos C++. Em particular,

a ferramenta ESBMC é capaz de verificar a maior parte dos casos de teste corretamente e em menos tempos, além de ser capaz de verificar casos de teste com tratamento de exceções (uma funcionalidade ausente no LLBMC, que diminui a precisão da verificação de programas C++). Em contra partida, o LLBMC apresentou melhores resultados na verificação de casos de teste que apresentam usos específicos de *templates*. Além disso, o ESBMC foi capaz de encontrar defeitos no programa *sniffer*, a aplicação comercial utilizada nos testes, que foram confirmados pelos desenvolvedores.

Na prova por indução matemática de programas C++, foi desenvolvido o algoritmo *k-induction* para prova de corretude de propriedades no programa C++. O algoritmo consiste de três passos, o caso base, condição adiante e passo indutivo, que transformam de maneira automática o programa original, na tentativa de provar a corretude do mesmo. Essas transformações foram modeladas utilizando lógica de primeira ordem e lógica Hoare. O algoritmo ainda foi melhorado para fazer uso de arquitetura paralela de processamento, e permite executar cada passo do algoritmo em um processo diferente.

Os experimentos utilizados para validar o algoritmo *k-induction*, sequencial e paralelo, foram compostos por programas de uma competição internacional de verificação, SV-COMP [53], além de programas desenvolvidos para validar a prova de programas que contenham *templates* e tratamento de exceções. Ambas as abordagens apresentaram bons resultados na verificação dos casos de teste, porém a abordagem paralela mostrou-se superior, sendo capaz de verificar um número maior de casos de teste, em menor tempo.

6.1 Trabalhos Futuros

Para trabalhos futuros, no que diz respeito à verificação de programas C++, a verificação de programas contendo *templates* pode ser melhor explorada, oferecendo suporte a usos específicos de *templates*, como os disponíveis nos teste de regressão do compilador GCC [50]. A Figura 6.1 mostra um exemplo de uso de *template* que atualmente não é suportado pelo ESBMC. Neste caso de teste, o *template* escolhido, baseia-se no sinal do valor passado como tipo. No ESBMC, ao invés de escolher o *template* da linha 10 (devido a instanciação com valor -1 na linha 24), a ferramenta escolhe o *template* da linha 17.

Além das alterações nos *templates*, um outro trabalho futuro seria a modelagem e avaliação do novo modelo operacional da ferramenta ESBMC, que já oferece suporte a contêineres

```
1  template <int N> struct HoldInt
2  {
3  };
4
5  template <class A, class B> struct Add
6  {
7  };
8
9  template <int N>
10 struct Add<HoldInt<N>, HoldInt<-N> >
11 {
12   typedef int type;
13   int f() { return 0; }
14 };
15
16 template <int N, int M>
17 struct Add<HoldInt<N>, HoldInt<M> >
18 {
19   typedef HoldInt<N+M> type;
20   int f() { return 1; }
21 };
22
23 int main() {
24   Add<HoldInt<1>, HoldInt<-1> > a;
25   Add<HoldInt<1>, HoldInt<-2> > b;
26   if (a.f() != 0 || b.f() != 1)
27     assert(0);
28 }
```

Figura 6.1: Exemplo do uso de *templates* de função.

associativos (por exemplo, *map* e *multimap*), além dos contêineres sequenciais mostrados nesta dissertação.

Para a prova por indução matemática, o algoritmo ainda não suporta alocação dinâmica de memória (por exemplo, programas que utilizem os operadores *malloc/new* e *free/delete*). O trabalho consistiria na avaliação do comportamento de uma variável dinâmica durante o processo de transformação do código original e desenvolvimento de algoritmo, visando maior abrangência da técnica.

Bibliografia

[1] BAIER, Christel, KATOEN, Joost-Pieter. *Principles of Model Checking*. Cambridge, Reino Unido: MIT Press, 2008. 975 p.

[2] HEATH, Steven. *Embedded Systems Design*. Oxford, Reino Unido: Newnes, 2003. 430 p.

[3] KOPETZ, Hermann. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Nova York, Estados Unidos: Springer, 2011. 376 p.

[4] GOLDSTEIN, Harry. Checking the play in plug-and-play. *Spectrum*, v. 39, n. 4, p. 957–974, 2012.

[5] BUTLER, Ricky. *What is Formal Methods?* 2001. Disponível em: http://shemesh.larc.nasa.gov/fm/fm-what.html. Acesso em: 10 abril 2013.

[6] CLARKE, Edmund, KROENING, Daniel, LERDA, Flavio. A tool for checking ANSI-C programs. In: *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*. Lancaster, Reino Unido: Springer, 2004. v. 2988, p. 168–176.

[7] CORDEIRO, Lucas, FISCHER, Bernd, MARQUES-SILVA, João. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transaction of Software Engineering*, v. 38, n. 6, p. 50–55, 2002.

[8] CORDEIRO, Lucas. *SMT-Based Bounded Model Checking of Multi-threaded Software in Embedded Systems*. Southampton, Reino Unido: University of Southampton, 2011. 197 p.

[9] Qadeer, Shaz, REHOF, Jakob. Context-Bounded Model Checking of Concurrent Software. In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*. Redmond, Estados Unidos: Microsoft Research, 2005. p. 93–107.

[10] IVANCIC, Franjo. Model Checking C programs using F-Soft. In: *Proceedings of International Conference on Computer Design*. San Jose, Estados Unidos: IEEE Computer Society, 2005. p. 297–308.

[11] MERZ, Florian, FALKE, Stephan, SINZ, Carsten. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In: *Proceedings of Verified Software: Theories, Tools and Experiments*. Philadelphia, Estados Unidos: Springer, 2012. p. 146–161.

[12] YANG, Jing, BALAKRISHNAN, Gogul, MAEDA, Naoto, et al. Object Model Construction for Inheritance in C++ and its Applications to Program Analysis. In: *Lecture Notes in Computer Science*. Tallinn, Estônia: Springer, 2012. v. 7210, p. 144–164.

[13] BLANC, Nicolas, GROCE, Alex, KROENING, Daniel. Verifying C++ with STL containers via predicate abstraction. In: *Proceedings of International Conference on Automated Software Engineering*. Zurich: ETH, Department of Computer Science, 2007. p. 521–524.

[14] PRABHU, Prakash, MAEDA, Naoto, BALAKRISHNAN, Gogul, IVANCIC, Franjo, GUPTA, Aarti. Interprocedural Exception Analysis for C++. In: *Proceedings of European Conference on Object-Oriented Programming*. Lancaster, Reino Unido: Springer, 2011.

[15] NEGGERS, Joseph, KIIM, Hee. *Basic Posets*. Singapura: World Scientific Pub Co Inc, 1999. 178 p.

[16] MOURA, Leonardo de, BJØRNER, Nikolaj. Z3: An efficient SMT solver. In: *Lecture Notes in Computer Science*. Budapeste, Hungria: Springer, 2008. v. 4963, p. 337–340.

[17] CLARKE, Edmund, KROENING, Daniel, SHARYGINA, Natasha, YORAV, Karen. Predicate Abstraction of ANSI-C Programs using SAT. In: *Formal Methods in System Design*. Estados Unidos: Springer US, 2003. p. 105–127.

[18] LEVINE, John. *flex & bison*. Estados Unidos: O'Reilly, 2009. 292 p.

[19] CORDEIRO, Lucas, FISCHER, Bernd. Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. In: *Proceedings of International Conference on Software Engineering*. Waikiki, Havaí: [s.n.], 2011. p. 331–340.

[20] BRADLEY, Aaron, ZOHAR, Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Stanford, Estados Unidos: Springer, 2007. 366 p.

[21] KROENING, D.; STRICHMAN, O. *Decision Procedures: An Algorithmic Point of View*. 1. ed. Oxford, Reino Unido: Springer Publishing Company, Incorporated, 2008.

[22] PATTERSON, David, HENNESSY, John. *Computer Organization and Design — The Hardware/Software Interface*. Berkeley, Estados Unidos: Morgan Kaufmann, 1994. xxiv + 648 p.

[23] MCCARTHY, John. Towards a mathematical science of computation. In: *Proceedings of International Federation for Information Processing Congress*. Munique, Alemanha: North-Holland, 1962. p. 21–28.

[24] BARRETT, Clark, TINELLI, Cesare. CVC3. In: *Lecture Notes in Computer Science*. Berlin, Alemanha: Springer, 2007. v. 4590, p. 298–302.

[25] BRUMMAYER, Robert, BIERE, Armin. Boolector: An efficient SMT solver for bit-vectors and arrays. In: *Lecture Notes in Computer Science*. York, Reino Unido: Springer, 2009. v. 5505, p. 174–177.

[26] MUCHNICK, Steven. *Advanced compiler design and implementation*. Massachusetts, Estados Unidos: Morgan Kaufmann Publishers Inc., 1997.

[27] WINTERSTEIGER, Christoph. *Compiling GOTO-Programs*. 2009. Disponível em: http://www.cprover.org/goto-cc/. Acesso em: 5 março 2013.

[28] FALKE, Stephan, MERZ, Florian, SINZ, Carsten. LLBMC: The Bounded Model Checker LLBMC. In: *Proceedings of 28th International Conference on Automated Software Engineering (ASE)*. Vale do Silício, Estados Unidos: IEEE Computer Society, 2013. p. 706–709.

[29] LLVM. *LLVM Tools*. 2013. Disponível em: http://llvm.org/releases/. Acesso em: 5 março 2013.

[30] The Satisfiability Modulo Theories Library. *SMT-LIB*. 2014. Disponível em: http://combination.cs.uiowa.edu/smtlib. Acesso em: 9 março 2014.

[31] GANESH, Vijay, DILL, David. A Decision Procedure for Bit-Vectors and Arrays. In: *Lecture Notes in Computer Science*. Berlin, Alemanha: Springer, 2007. v. 4590, p. 519–531.

[32] CORMEN, Thomas, LEISERSON, Charles, RIVEST, Ronald, STEIN, Clifford. *Algoritmos: Teoria e Prática*. São Paulo, Brasil: Elseveir, 2001.

[33] HOARE, Tony. An axiomatic basis for computer programming. In: *Proceedings of Association for Computing Machinery*. Nova York, Estados Unidos: ACM, 1969. p. 576–580, 583.

[34] NEC Labs. *NEC*. 2013. Disponível em: http://www.nec-labs.com/research/system/. Acesso em: 5 março 2013.

[35] CLARKE, Edmund, KROENING, Daniel, SHARYGINA, Natasha, YORAV, Karen. Satabs: Sat-based predicate abstraction for ansi-c. In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*. Edimburgo, Reino Unido: Springer, 2005. p. 570–574.

[36] GROβE Daniel, LE, Hoang, DRECHSLER., Rolf. Proving Transaction and System-level Properties of Untimed SystemC TLM Designs. In: *International Conference on Formal Methods and Models for Codesign, Lecture Notes in Computer Science*. Grenoble, França: IEEE Computer Society, 2000. v. 1954, p. 108–125.

[37] EÉN, Niklas, SÖRENSSON, Niklas. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, v. 89, n. 4, 2003.

[38] DONALDSON, Alastair F. *Scratch*. 2013. Disponível em: http://www.cprover.org/scratch/. Acesso em: 6 junho 2013.

[39] DONALDSON, Alastair, HALLER, Leopold, KROENING, Daniel, RÜMMER, Phillip. Software Verification using k-Induction. In: *Static Analysis*. Veneza, Itália: Springer, 2011. p. 351–268.

[40] BURCH, Jerry R., CLARKE, Edmund, MCMILLAN, Kenneth L., DILL, David L., HWANG, Larry J. *Symbolic Model Checking: 10 20 States and Beyond*. 1990.

[41] DONALDSON, Alastair, KROENING, Daniel, RÜMMER, Phillip. Automatic Analysis of Scratch-pad Memory Code for Heterogeneous Multicore Processors. In: *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*. Paphos, Chipre: Springer, 2010. v. 6015, p. 280–295.

[42] LEINO, Rustan. *Boogie*. 2013. Disponível em: http://boogie.codeplex.com/. Acesso em: 15 junho 2013.

[43] STROUSTRUP, Bjarne. *The C++ Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[44] DEITEL, Harvey, DEITEL, Paul. *C++ How to Program*. Estados Unidos: Prentice Hall, 2006. 1536 p.

[45] CPP Reference. *Reference of the C++ Language Library*. 2013. Disponível em: http://www.cplusplus.com/reference/. Acesso em: 5 março 2013.

[46] RAMALHO, Mikhail, FREITAS, Mauro, SOUSA, Felipe, MARQUES, Hendrio, CORDEIRO, Lucas C., FISCHER, Bernd. SMT-Based Bounded Model Checking of C++ Programs. In: *Proceedings of International Conference on the Engineering of Computer Based Systems*. Phoenix, Estados Unidos: IEEE Computer Society, 2013. p. 147–156.

[47] INTEL Corporation. *Standard for Programming Language C++*. [S.l.], 2012, 1324 p.

[48] ESBMC. *Efficient SMT-Based Context-Bounded Model Checker*. 2013. Disponível em: http://esbmc.org. Acesso em: 5 março 2013.

[49] LLBMC. *The Low-Level Bounded Model Checker*. 2013. Disponível em: http://llbmc.org. Acesso em: 5 março 2013.

[50] Free Software Foundation. *GCC Test Suite*. 2013. Disponível em: https://github.com/mirrors/gcc/tree/master/gcc/testsuite/. Acesso em: 1 dezembro 2013.

[51] APT, Krzysztof R. *Ten Years of Hoare's Logic: A Survey – Part l*. 1981.

[52] MORSE, Jeremy, CORDEIRO, Lucas, NICOLE, Denis, FISCHER, Bernd. Handling Unbounded Loops with ESBMC 1.20. In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*. Roma, Itália: Springer, 2013. p. 619–622.

[53] BEYER, D. Second competition on software verification - (summary of sv-comp 2013). In: *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*. Roma, Itália: Springer, 2013. v. 7795, p. 594–609.

[54] JONES, T. *GNU/Linux Application Programming*. Boston, Estados Unidos: Charles River Media, 2005. (Charles River Media programming series).

[55] MITCHELL, J. O. M.; SAMUEL, A. *Advanced Linux Programming*. Estados Unidos: New Riders, 2001.

[56] MORSE, Jeremy, RAMALHO, Mikhail, CORDEIRO, Lucas, NICOLE, Denis, FISCHER, Bernd. ESBMC 1.22 (Competition Contribution). In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*. Grenoble, França: Springer, 2014. p. 619–622.

Apêndice A

Publicações

A.1 Referente à Pesquisa

- **Mikhail Ramalho**, Mauro Freitas, Felipe Sousa, Hendrio Marques, Lucas Cordeiro e Bernd Fischer. *SMT-Based Bounded Model Checking of C++ Programs*. **20th IEEE International Conference and Workshops on the Engineering of Computer Based Systems**, Phoenix, 2013. p. 147-156.

- **Mikhail Ramalho**, Lucas Cordeiro, André Cavalcante e Vicente Lucena. *Verificação Baseada em Indução Matemática para Programas C/C++*. **III Simpósio Brasileiro de Engenharia de Sistemas Computacionais**. Niterói, Rio de Janeiro.

A.2 Contribuições em outras Pesquisas

- Mauro L. de Freitas, **Mikhail Y. R. Gadelha**, Lucas C. Cordeiro, Waldir S. S. Júnior e Eddie B. L. Filho. *Verificação de Propriedades de Filtros Digitais Implementados com Aritmética de Ponto Fixo*. **XXXI Simpósio Brasileiro de Telecomunicações - SBrT**, 2013.