Universidade Federal do Amazonas

Faculdade de Tecnologia

Programa de Pós-Graduação em Engenharia Elétrica

Verificação de Programas Embarcados ANSI-C Baseada em Indução Matemática e Invariantes

Raimundo Williame Rocha de Melo

Manaus – Amazonas

Agosto de 2017

Raimundo Williame Rocha de Melo

Verificação de Programas Embarcados ANSI-C Baseada em Indução Matemática e Invariantes

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

Orientador: Lucas Carvalho Cordeiro

Raimundo Williame Rocha de Melo

Verificação de Programas Embarcados ANSI-C Baseada em Indução Matemática e Invariantes

Banca Examinadora

Prof. D.Sc. Eddie Batista de Lima Filho – Presidente e Coorientador
TP Vision Indústria e Eletrônica Ltda

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

Prof. D.Sc. Herbert Oliverira Rocha
Departamento de Ciência da Computação – UFRR

Manaus – Amazonas
Agosto de 2017

À minha mãe e minha família.

Agradecimentos

Agradeço primeiramente à minha mãe, Iracema Luisa Zurra Rocha, por me apoiar nas mais importantes decisões da minha vida, por seu incondicional amor e por estar ao lado meu em todos os momentos difíceis que passamos juntos. Ela é minha maior motivação para acordar todos os dias e seguir em frente. Agradeço aos meus irmãos (Ewerton Rocha, Suellem Rocha e Patrícia Rocha) por todo amor, paciência e apoio incondicional.

Ao meu tio, Orlando Zurra Rocha, que me encorajou a seguir na vida mesmo quando parecia não haver sinais de dias melhores, seu amor e carinho foram muito importantes para chegar até aqui.

Aos meus amigos do laboratório de Verificação Formal pela amizade e pela força nas longas horas de estudo (João Paulo Mendes, Phillipe Pereira, Mário Praia, Higo Albuquerque e Hussama Ismail).

Agradeço aos Professores Lucas Cordeiro e Herbert Rocha, pelo suporte, direcionamento, encorajamento e amizade que foram essenciais para a realização desse trabalho, onde longas horas de *brainstoming* para a evolução da ferramenta foram dispendidas.

“A ignorância gera mais frequentemente confiança do que o conhecimento: são os que sabem pouco, e não aqueles que sabem muito, que afirmam de uma forma tão categórica que este ou aquele problema nunca será resolvido pela ciência”.

Charles Darwin (1809-1882)

Resumo

O uso de sistemas embarcados, sistemas computacionais especializados para execução em sistemas eletrônicos ou mecânicos tem crescido de forma vertiginosa devido a utilização cada vez mais intensa de sensores, interfaces de rede e protocolos de comunicação em diversas áreas. Por isso, é cada vez mais importante garantir a robustez desses sistemas, uma vez que estão se tornando mais complexos e integrados. Existem várias técnicas para garantir que um sistema seja entregue ao cliente sem erros, em particular, a verificação formal dos programas tem se revelado eficaz na busca de falhas. Neste trabalho é descrito um algoritmo de indução matemática conhecido como *k*-induction combinado ao uso de invariantes para verificar e refutar propriedades de segurança em programas desenvolvidos na linguagem ANSI-C. Em particular, a abordagem proposta infere invariantes no programa para auxiliar na verificação de programas ANSI-C através da técnica de indução matemática através do refinamento de restrição (*i.e, poliédrico*) para especificar pré- e pós-condições.

No método proposto, adotamos dois geradores de invariantes para produzir e alimentar o algoritmo de indução matemática ao qual é implementado na ferramenta Efficient SMT-Based Context-Bounded Model Checker. A motivação para a combinação de invariantes com o algoritmo de indução matemática é fechar um *gap* na verificação formal de programas que possuam variáveis globais, além de programas com *loops* que possuem desvios condicionais e o número de iterações é desconhecido. PIPS e PAGAI são as ferramentas utilizadas para analisar o código e produzir invariantes indutivas responsáveis por guiar o algoritmo de indução matemática na verificação do *benchmark*, sendo este o principal desafio do método proposto.

Para avaliar a eficácia da abordagem proposta neste trabalho, além de aplicações de Sistemas Embarcados foram utilizados *benchmarks* públicos disponibilizados pela Competição Internacional de Verificação de Software onde participam Universidades, pesquisadores, estudandantes de doutorado de várias partes do mundo, e fornece amplo conjunto de casos de teste

para verificação. Além disso, foram utilizadas ferramentas estado-da-arte para a comparação dos resultados e, assim mensurar a eficácia do método proposto.

Os resultados experimentais foram positivos e mostraram que o algoritmo de indução matemática com invariantes pode verificar uma grande variedade de propriedades de segurança em programas com *loops* e aplicações de sistemas embarcados de telecomunicações, sistemas de controle e dispositivos médicos.

Palavras-chave: verificação formal, indução matemática, invariantes, PIPS, PAGAI.

Abstract

The use of embedded systems, *i.e.*, computer systems focused on performing specific functions in larger (electronic or mechanical) systems, has been growing lately, and ensuring the robustness of such systems has become increasingly important. There are several techniques to ensure that a system is released without errors. In particular, formal verification of programs is proving itself to be effective in the search for failures. In this work, an induction-proof algorithm is described, which combines k-induction and invariants to verify and refute safety properties in embedded ANSI-C software. Moreover, the proposed k-induction-based approach infers invariants in the program to assist in verification tasks, using constraint refinement (*i.e., polyhedral*) to specify pre- and post-conditions.

We adopted two invariant generators to produce such and feed the k-induction algorithm, which is implemented in the Efficient SMT-Based Context-Bounded Model Checker tool.

Public benchmarks were used to assess the effectiveness of our approach. In addition, a comparison to other state-of-the-art verification tools using a set of benchmarks from the International Competition for Software Verification in addition to embedded systems applications.

Experimental results have shown that the proposed approach, with and without invariants, can verify a wide variety of safety properties in programs with loops and embedded software from telecommunications, control systems, and medical domains.

Keywords: formal verification, mathematical induction, invariants, PIPS, PAGAI.

Índice

Índice de Figuras

Índice de Tabelas

Abreviações

2LS - ***2****and order* ***L****ogic* ***S****olving*

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

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

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

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

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

LLVM - ***L****ow* ***L****evel* ***V****irtual* ***M****achine*

LP **L**ógica *P*roposicional

LTL - ***L****ogic* ***T****emporal* ***L****inear*

SAT - **SAT**isfatibilidade Booleana

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

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

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

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

Capítulo 1

Introdução

Sistemas embarcados desenvolvidos em ANSI-C têm sido aplicados a diferentes domínios (*e.g*, industrial, militar, educacional e *wearable*), os quais essencialmente exigem alta qualidade e confiabilidade. Particularmente, em sistemas embarcados críticos, como aeronáuticos e hospitalares, várias restrições (*e.g*, tempo de resposta e precisão dos dados) devem ser atendidas e aferidas de acordo com as necessidades dos utilizadores; caso contrário, as falhas podem levar à situações catastróficas, onde seus impactos podem levar a perdas monetárias ou de vidas humanas. Como um exemplo do impacto causado por falhas em software embarcados críticos, em 2001 o Minitério da Saúde do Panamá encomendou à Organização Pan-americana de saúde uma investigação relacionada a pacientes do Instituto Oncológico Nacional (ION), onde 23 pacientes morreram devido à exposição de altas doses radiação. Durante a investigação, testes indicaram que o software responsável pela dosagem da radiação, em determinadas configurações, calculava até o dobro da dose recomendada [1]. Este erro de software poderia ter sido evitado por meio da aplicação de métodos formais que tem por finalidade contribuir para a confiabilidade e robustez de sistemas de *software*.

Um método formal que pode ser utilizado para a verificação de software é a técnica conhecida como Verificação de Modelos Limitados (do inglês, *Bounded Model Checking* - BMC) que pode ser baseado na Satisfatibilidade Booleana (do inglês, *Boolean Satisfiability* - SAT) [2] ou em Teorias de Módulo de Satisfatibilidade (do inglês, *Satisfiability Modulo Theories* - SMT) [3], que tem sido aplicado com sucesso para verificar programas sequenciais e multi-tarefas [4] a fim de expor *bugs* difíceis de encontrar e reproduzir manualmente [5, 6]. A idéia da técnica BMC é, dado um sistema de transição e uma propriedade de segurança, veri-

ficar se existe um caminho de execução que viole a propriedade de segurança e então gerar um contra-exemplo que contém todos os estados explorados até a violação da propriedade.

A técnica BMC limita as regiões de estruturas dos dados que podem ser exploradas (*i.e., arrays*) além de restringir o número de iterações de *loop* e o espaço de estado a ser explorado durante a verificação [5, 6, 4, 7]. Ainda que a técnica BMC seja muito eficaz, a mesma é limitada por não ser capaz de desdobrar *loops* e funções recursivas indefinidamente, pois é suscetível ao esgotamento dos limites de tempo ou de memória [4].

Por exemplo, no programa mostrado na Fig. 1.1a, o *loop* (da linha 3 até a linha 5) será desenrolado enquanto $i < N$. No entanto, N recebe um valor não-determinístico (linha 1) enquanto o resultado do *assert* (linha 6) for verdadeiro, se, e somente se, todos os estados possíveis forem alcançados e verificados. No entanto, a técnica BMC tipicamente não consegue verificar este tipo de programa, devido à natureza não-determinística do limite superior do *loop* [8], onde o k pode crescer indefinidamente.

```
1 unsigned int N = unondet_int()
    ;
2 unsigned int i = 0;
3 while( i < N ){
4   ++i;
5 }
6 assert( i == N );
```

(a) *Loop* com Limite Superior Não-Determinístico

```
1 unsigned int N = unondet_int()
    ;
2 unsigned int i = 0;
3 if( i < N ){
4   ++i;
5 }
6 ...
7 assert( !(i < N) );
8 assert( i == N );
```

(b) Programa Livre de *Loops*

Figura 1.1: *Loop* com limite superior baseado em variável não-determinística.

Ao combinar a técnica BMC com a técnica de indução matemática, é possível provar a corretude de programas que contém *loops* com limite superior desconhecido. A prova por indução é uma técnica muito bem estabelecida na verificação de *hardware*, onde é fácil de ser aplicada, devido à relação de transição monolítica presente em projetos de hardware [9, 10, 11].

O algoritmo de indução matemática consiste de 3 passos: No caso base, o algoritmo tenta encontrar um contraexemplo até uma profundidade k . Na condição adiante, o algoritmo checa se todos os estados foram alcançados até k iterações. No passo indutivo, o algoritmo checa, se a propriedade é verdadeira em k iterações, então deve ser verdadeira para as próximas [8].

Sabendo que, *Loops* que possuem no seu corpo variáveis globais, desvios condicionais que podem gerar dois ou mais caminhos de execução e que isso pode elevar o número de espaços de estados a serem verificados, faz-se necessário a restrição do conjunto de espaços de estados a serem explorados durante a verificação. Para alcançar este objetivo, em todos os *benchmarks* foram utilizadas invariantes estaticamente inferidas através da análise de código. Invariantes são assertivas que devem ser verdadeiras em todos os estados ao longo da verificação do programa [12]. Este trabalho é uma extensão da ferramenta *DepthK* que utiliza o PIPS como gerador de invariantes [13]. A ferramenta *Path Analysis for Invariant Generation by Abstract Interpretation* (PAGAI) [14] foi utilizada para análise estática de código e geração de invariantes sobre variáveis numéricas com o propósito de diminuir o conjunto de espaços de estados e, consequentemente, reduzir o tempo de verificação aumentando a cobertura de programas verificados, foi adotada por implementar o método poliedral, realizar análise estática de programas ANSI C e C++ e, por utilizar-se da infraestrutura do compilador LLVM, não oferece complexidade na instalação.

Ao final da verificação é gerado um arquivo *XML* conhecido como arquivo de testemunha que possui os estados explorados juntamente com os valores atribuídos às variáveis. Em seguida, é aplicada a técnica de validação que consiste em avaliar e reproduzir o resultado da ferramenta de verificação de forma automática, eliminando a necessidade de inspeção manual [15, 16, 17].

A metodologia descrita e avaliada neste trabalho foi implementada na ferramenta DepthK [13], cuja finalidade se comprova na tentativa de provar que a combinação das técnicas descritas são capazes de verificar uma ampla gama de *benchmarks* e fornecer resultados confiáveis.

Muitos foram os desafios deste trabalho, dentre os quais é possivel destacar que a integração com as ferramentas geradoras de invariantes, integração com validadores de testemunha e refino do algoritmo de tradução das invariantes em *assumes* para limitar espaço de estados.

1.1 Descrição do Problema

A problemática deste trabalho encontra-se centralizada nas variáveis globais e nos desvios condicionais de *loops* que podem gerar muitos estados, como exemplo, o programa extraído do trabalho apresentado por Henry, Monniaux e Moy [14] mostrado na Figura 1.2. O corpo de *loop* possui dois caminhos viáveis. Estes são executados alternadamente dependendo de uma variável chamada *phase*. Tais programas, por sua vez, com caminhos de execução que dependem de variáveis globais muitas vezes ocorrem em sistemas reativos que são orientados a eventos, por exemplo, onde "um evento α ocorre no estado **A**, se a condição **C** é verdadeira em um determinado tempo, então o sistema passa para o estado **B**" [18].

```
1  void main(){
2     int x = 0 ;
3     int t = 0 ;
4     int phase = 0 ;
5
6     while ( t < 100) {
7       if( phase == 0 ){
8        x = x + 2;
9       }
10      if( phase == 1 ){
11       x = x - 1;
12      }
13      phase = 1 - phase;
14      ++t;
15    }
16    assert( t <= 100 );
17 }
```

Figura 1.2: Exemplo para verificação com Indução Matemática.

Na Figura 1.2, a propriedade representada pelo *assert* na linha 16 deve ser mantida para o limite superior do *loop* ($t == 100$). Em contraste com o método proposto, a técnica BMC tem dificuldades em comprovar a corretude deste tipo de programa devido à ausência de condições que são verdadeiras antes da execução do *loop* e se mantém verdadeira até o fim da execução do mesmo.

A técnica BMC tipicamente desdobra este tipo de *loop* até um limite superior (neste caso, 100), para verificar a corretude da propriedade. No mesmo caso mencionado, se a condição de saída de *loop* (*i.e.*, $t < 100$) for substituído por $t < N$, onde N é um valor não determinístico, então o limite superior vinculado pode variar de acordo com o tipo de dado escolhido e a arquitetura sob a qual o *benchmark* será executado [4], a técnica BMC tentará

desenrolar o *loop* $2^{31} - 1$ vezes (em máquinas de arquitetura 32 e 64 *bits*).

Devido a recursos limitados (*i.e.*, tempo e memória), pode ser impossível verificar todos os estados alcançáveis através da técnica BMC.As principais ferramentas de verificação que implementam a técnica de indução matemática combinada com invariantes são capazes de provar o exemplo acima, entretanto, as mesmas ferramentas falham ao utilizar a técnica BMC (*e.g*, *CBMC* [19], *CPAchecker* [20] e 2*LS* [21]).

Projetos de software voltados para a plataforma embarcada (*e.g.*, controladores digitais) normalmente exigem demasiado tempo e esforço, logo, é essencial o desenvolvimento de ferramentas que auxiliem durante os testes do mesmo. Atualmente, existem ferramentas que auxiliam nesta tarefa, no entanto, sem a utilização de métodos formais falhas sutis podem não ser detectadas. A utilização de métodos formais é promissora nesta área uma vez que falhas catastróficas podem ser evitadas e, desta forma, pode-se poupar perdas financeiras e humanas [22, 23, 24, 25], se considerarmos a geração de invariantes em ferramentas de verificação de software de controle com o intuito de analisar profundamente o espaço de estado do sistema.

1.2 Objetivos

O objetivo geral deste trabalho é propor um algoritmo que seja capaz de complementar a técnica de indução matemática em verificadores baseados na técnica BMC utilizando-se da geração de invariantes para determinar a corretude de programas ANSI-C.

Os objetivos específicos deste trabalho são listados a seguir:

a) Demonstrar experimentalmente que as invariantes atuam reduzindo o espaço de estados a serem explorados.

b) Desenvolvimento de protótipo combinando a indução matemática do ESBMC com invariantes geradas por ferramentas externas.

c) Integração com validadores de arquivos de testemunha.

1.3 Descrição da Solução

A metodologia proposta utiliza aspectos teóricos e práticos da verificação de modelos através da técnica de indução matemática e geração de invariantes. Como resultado, a abor-

dagem aplicada foi desenvolvida como uma extensão da ferramenta *DepthK* [13]. A geração de invariantes foi realizada pelas ferramentas PAGAI [14] e PIPS [26].

Para a verificação dos *benchmarks* foi escolhida a ferramenta estado-da-arte ESBMC (do inglês, *Efficient SMT-Based Context-Bounded Model Checker*) que além de implementar a técnica de indução matemática [27], oferece suporte total a programas escritos na linguagem ANSI-C que utilizam operação *bit* a *bit*, matrizes, ponteiros, alocação de memória, ponto flutuante, concorrência e etc [28, 29, 30]. O ESBMC destaca-se pela qualidade e número de tarefas verificadas com sucesso na Competição Internacional de Verificação de Software (do inglês, *International Competition on Software Verification* - SVCOMP) [31] que é um dos indicadores de vivacidade da área de verificação formal na qual participam ferramentas que implementam várias técnicas de verificação.

A Figura 1.3 mostra os passos executados pelo algoritmo implementado neste trabalho bem como a atuação de cada ferramenta utilizada [17].

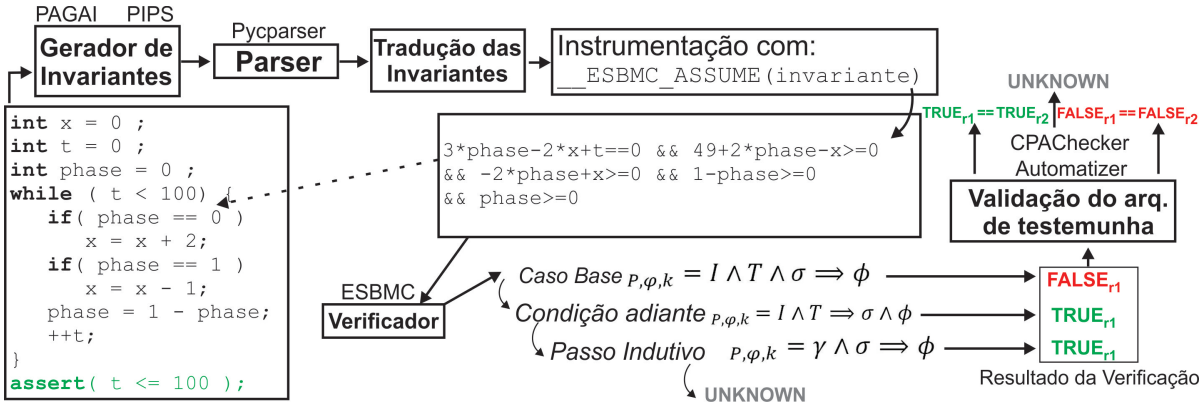

Figura 1.3: Passos da verificação executada através do DepthK.

a) Gerador de Invariantes: O método proposto utiliza-se das ferramentas PIPS [26] e PAGAI [14] que são ferramentas inte-precedurais de transformação *source-to-source* para programas C e contam com a abstração poliedral do comportamento de programas. O PAGAI utiliza-se da análise do código fonte para gerar invariantes para cada ponto do fluxo de controle de um programa C usando a infra-estrutura do LLVM (veja http://llvm.org), focando na distinção do caminho dentro do grafo de fluxo de controle enquanto evita uma enumeração de caminho exponencial sistemática [14]. Enquanto isso, o PIPS executa a análise em dois passos [26]: (1) Cada instrução de programa está associada a um transformador, representando a sua função de transferência subjacente. Trata-se de um pro-

cedimento de instruções elementares então trabalhando em declarações compostas e até definições de funções. (2) As invariantes poliedrais são propagadas junto com instruções, usando os transformadores previamente calculados.

b) Tradução das Invariantes: O *benchmark* é posteriormente submetido ao PAGAI ou PIPS (o usuário pode selecionar uma das duas ferramentas) que geram como código de saída um arquivo ANSI-C modificado, o mesmo contém as invariantes escritas como comentários em torno de instruções matemáticas. Estas invariantes são traduzidas em declarações *ESBMC_ASSUME* para restringir os possíveis valores das variáveis relacionadas a estas invariantes. Esta etapa de tradução é necessária uma vez que o PIPS e PAGAI geram invariantes representadas como expressões matemáticas que, naturalmente, não são aceitas pela sintaxe de programas ANSI-C.

c) *Pycparser*[1]: Após as transformações de código e a geração das invariantes em forma de comentário, o arquivo é submetido a validação de sintaxe a fim de evitar possíveis falhas na verificação por parte do ESBMC caso o código não esteja compatível com o padrão ANSI-C.

d) ESBMC: Logo após as transformações de código e a tradução das invariantes em instruções *ESBMC_ASSUME*, o *benchmark* é submetido a esta ferramenta que executará a verificação utilizando a técnica de indução matemática onde o resultado pode ser *TRUE*, *FALSE* ou *UNKNOWN*.

e) Validação do Arquivo de Testemunha: Após chegar a um resultado conclusivo (*true* ou *false*), o ESBMC gera um arquivo XML (do inglês, *eXstensible Markup Language*) que contém os estados verificados bem como os valores das variáveis em cada estado. Este arquivo é conhecido como arquivo de testemunha [32]. Atualmente, existem duas ferramentas estado-da-arte capazes de realizar a validação do arquivo de testemunha, CPAchecker [16] e Ultimate Automizer [33]. O DepthK manipula este arquivo da seguinte maneira:

   1. O arquivo de testemunha é submetido ao CPAchecker [16] para validação, caso o resultado do ESBMC seja confirmado, o mesmo será disponibilizado ao usuário;

[1] https://github.com/eliben/pycparser

2. Se o CPAchecker não for capaz de chegar um resultado que ratifique ou aquele exposto pela ferramenta, ou sofrer falha interna, o arquivo de testemunha é submetido ao Ultimate Automizer [33], e caso o resultado do ESBMC seja confirmado, o mesmo será disponibilizado ao usuário;
3. Se nenhuma das duas ferramentas for capaz de avaliar os estados contidos no arquivo de testemunha e ratificá-lo, o resultado será *UNKNOWN*;

A validação do arquivo de testemunha [32] tornou-se regra no SV-COMP, como forma de avaliar mais profundamente a ferramenta de verificação que forneceu o resultado, uma vez que é possível confirmar ou não, de forma totalmente automática, se os valores utilizados na exploração dos espaços de estado, e que estão disponíveis no contra-exemplo, de fato guiam ao resultado correto.

1.4 Contribuições

A ferramenta DepthK [13], anteriomente mencionada, foi ampliada de forma a incluir novas funcionalidades que permitem geração e inclusão de invariantes. As principais contribuições deste trabalho são:

- Adição da ferramenta de geração de invariantes PAGAI.
- Criação automática de assertivas baseadas nas invariantes geradas.
- Adição da ferramenta para validar o contraexemplo obedecendo o formato padrão do arquivo de testemunha.
- Combinação da técnica de indução matemática com invariantes.

O capítulo 4 define e exemplifica os passos e as transformações necessárias para provar as propriedades de um programa ANSI-C. Essas transformações são geradas pela ferramenta adotada para a geração de invariantes. Além disso, é apresentado o algoritmo responsável pela indução matemática, chamado *k-induction* [27], que consiste de três passos (caso base, condição adiante e passo indutivo), para provar a corretude de um programa acrescido de invariantes. A ferramenta apresentada neste trabalho necessita, como entrada, de um arquivo fonte de um programa ANSI-C para iniciar o processo de prova.

É importante ressaltar a existência de outros algoritmos na literatura que implementam o algoritmo de indução matemática acrescido da geração de invariantes, no entanto, o método neste trabalho se difere dos outros por combinar as invariantes geradas por uma ferramenta independente [34, 35, 36, 21] e pela facilidade de integração do código anotado com invariantes e o verificador, não sendo necessário código adicional no ESBMC para lidar com *benchmarks* que passaram por transformação através do método proposto.

1.5 Organização da Dissertação

A introdução apresenta os objetivos e motivações da pesquisa que resultou na realização desse trabalho. As demais seções estão divididas da seguinte forma:

O Capítulo 2, *Fundamentação Teórica*, apresenta uma revisão dos conceitos básicos por trás da verificação formal de programas, interpretação abstrata, transformações de código aplicadas aos programas e geração de invariantes.

O Capítulo 3, *Trabalhos Relacionados*, apresenta um resumo de diversas ferramentas de verificação que, a fim de provar a corretude de programas ANSI-C, utilizam a indução matemática acrescida da geração de invariantes.

O Capítulo 4, *Método Proposto*, apresenta as ferramentas PIPS e PAGAI, ambas utilizadas no processo de geração de invariantes através da análise estática de código . Além disso, detalhes do algoritmo de indução matemática atuando em conjunto com pré- e pós-condições baseadas nas invariantes são apresentados.Por fim, são apresentados os resultados da verificação baseada em *benchmarks* do SV-COMP e Sistemas Embarcados, e a comparação dos resultados com as ferramentas CPAchecker [37], CPAchecker com indução matemática e invariantes [38], CBMC com indução matemática [10] e 2LS [21].

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

Capítulo 2

Fundamentação Teórica

Nesse capítulo, serão abordados conceitos gerais sobre fundamentos essenciais, a fim de agregar conhecimentos básicos que norteiam a compreensão da verificação formal bem como a lógica proposicional, técnica de indução matemática aplicada a programas ANSI-C, definição da técnica de Interpretação Abstrata para análise estática de código e geração de invariantes.

2.1 Lógica Proposicional

A Lógica Proposicional é uma linguagem capaz de expressar sentenças naturais declarativas [39] sendo expressa de forma afirmativa ou negativa além de ajudar a determinar a validade do conteúdo de um argumento. Podemos chegar a uma conclusão através de proposições lógicas que podem ser escritas de maneira declarativa como na sentença "se p e não q, então r". Podemos notar que a forma de representar os argumentos está de acordo com a lógica natural ou pode ser facilmente entendida intuitivamente.

Para que seja possível garantir a consistência da Lógica Proposicional foi introduzido o formalismo matemático que consiste de símbolos e regras. A partir deste formalismo é possível construir setenças (proposições), conhecidas como fórmulas. Podem ser formadas através da utilização de proposições atômicas usando conectivos lógicos. Os seguintes símbolos são utilizados para a definição e construção das expressões:

- $\neg$: Representa o operador *not*, utilizado para expressar uma sentença ou uma proposição atômica além de ser um operador unário. Por exemplo, $\neg p$ é lido como “não p”.

- $\wedge$: Representa o operador *and*, conhecido como conjunção. É um operador binário. Por exemplo, $p \wedge q$ é lido como "p e q".

- $\vee$: Representa o operador *or*, conhecido como disjunção. É um operador binário. Por exemplo, $p \vee q$ é lido como "p ou q".

Além do formalismo matemático as proposições são regidas pelos seguintes princípios:

- **Princípio da Identidade**: Uma proposição Verdadeira é Verdade enquanto que uma proposição Falsa é Falsa.

- **Princípio da Terceiro Excluído**: Uma proposição é verdadeira ou falsa, desta forma não existe uma terceira opção.

- **Princípio da Não-Contradição**: Uma proposição não pode assumir mais de um valor simultaneamente, ou seja, não pode ser verdeira e falsa.

Um valor verdade, representado pelo símbolo *tt* ou *ff*, é um valor que indica se o resultado de uma expressão é verdadeira ou falsa. Os elementos básicos da lógica proposicional são as constantes verdadeiro (representado por $\top$ ou 1) e falso (representado por $\bot$ ou 0).

Operadores lógicos ou booleanos (por exemplo, $\neg, \wedge$), fornecem o elemento de conexão para criação de expressões na lógica proposicional [40].

Definição 2.1 *Gramática que define a sintaxe das fórmulas:*

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

Para a construção de expressões lógicas, utiliza-se os operadores lógicos conjunção ($\wedge$) e negação ($\neg$). Há outros operadores lógicos como disjunção ($\vee$), implicação ($\Rightarrow$), equivalência lógica ($\Leftrightarrow$), ou exclusivo ($\oplus$) e expressões condicionais (*ite*) sendo definidos a seguir.

Definição 2.2 *Operadores lógicos:*

- $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$

- $\phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$

- $\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$

```
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 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 do código na Figura 2.1c

Figura 2.1: Códigos ANSI-C e fórmulas equivalentes.

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

Basicamente, as fórmulas são definidas em termos dos elementos verdadeiro, falso, ou uma variável proposicional x; ou, a partir da aplicação 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" ($ite(\theta, \phi_1, \phi_2)$).

Como mostrado na Figura 2.1, é possível representar um código fonte em forma de lógica proposicional. 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 *Para que uma fórmula em lógica proposicional seja considera bem formada, é necessário que a mesma utilize as regras apresentadas na definição (2.1), onde a negação tem prioridade sobre conjunções.*

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

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* $\Leftrightarrow$.

A tabela verdade é utilizada como mecanismo para avaliar se uma fórmula é verdadeira ou falsa. Uma interpretação I atribui para todas as variáveis proposicionais um valor. Considerando os possíveis valores de uma variável proposicional x (por exemplo, *tt* ou *ff*), é pos-

sível construir a 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.

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

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

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* $I \models \phi_2$
- $I \models \phi_1 \Leftrightarrow \phi_2 \quad sse\ I \models \phi_1\ e\ I \models \phi_2,\ ou\ I \not\models \phi_1\ e\ I \not\models \phi_2$

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

A Lógica Proposicional é utilizada, por exemplo, em solucionadores SAT para verificar a corretude de sistemas e tem ampla utilização na Verificação Formal.

2.2 Verificação de Modelos Limitada

A difícil tarefa de garantir que os requisitos de *software* sejam desenvolvidos e entregues sem defeitos tem levado à utilização da verificação formal de software. Uma das principais

vantagens é garantir que o procedimento de teste seja executada de forma automática com a intenção de diminuir a possibilidade de falhas que podem levar a situações catastróficas como no caso do foguete *Ariane 5* que explodiu devido a conversão de um número de 64 *bits* em um inteiro de 16 *bits*.

Falhas como do foguete *Ariane 5* poderiam ser evitadas através do uso da técnica de verificação de modelos limita (do inglês, *Bounded Model Checking* - BMC), uma vez que, a mesma tem sido aplicada com sucesso para verificar programas sequenciais [5, 6, 4] e multi-tarefas [41, 42], além de ser útil para encontrar e expor *bugs* difíceis de localizar manualmente. A técnica BMC pode ser baseada em satisfatibilidade booleana (do inglês, *Boolean Satisfatibility* - SAT) ou em teorias do módulo de satisfatibilidade (do inglês, *Satisfiability Modulo Theories* - SMT) [43]. A ideia desta técnica é checar a negação de uma determinada propriedade de segurança até uma determinada profundidade, através da utilização de sistemas de transição de estados para representar os possíveis comportamentos do sistema.

A técnica BMC é capaz de verificar a negação de uma determinada propriedade até uma profundidade k. Esta técnica não é capaz de verificar a corretude do sistema, a menos que um limite k seja previamente conhecido, isto é, um limite que desdobra todos *loops* e funções recursivas até a profundidade máxima. Uma das principais características da técnica BMC é limitar regiões de estruturas de dados (por exemplo, *arrays*) e o número de iterações de *loop* até um limite k. Desta forma, os espaços de estado que necessitam ser explorados são limitados, e assim, erros em aplicações podem não ser detectados [5, 6, 40, 44]; As ferramentas BMC são suscetíveis a limite de tempo ou de memória, esta vulnerabilidade fica evidente em programas com *loops* cujos limites podem crescer indefinidamente e que, consequentemente, não podem ser determinados estatisticamente.

A ferramenta adotada, neste trabalho, para verificação dos *benchmarks* é o ESBMC, ao submeter um programa a esta ferramenta, o mesmo é modelado por um sistema de transição de estados, que é gerado a partir de um grafo de fluxo de controle do programa (do inglês, *ControlFlow Graph* - CFG) [45]. O ESBMC automaticamente gera o grafo de fluxo de controle do programa 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.

A Fig. 2.2 mostra a representação de um Sistema de Transição M onde existe uma propriedade ϕ, e um limite de iterações k, a técnica BMC desdobra o sistema k vezes e converte-o

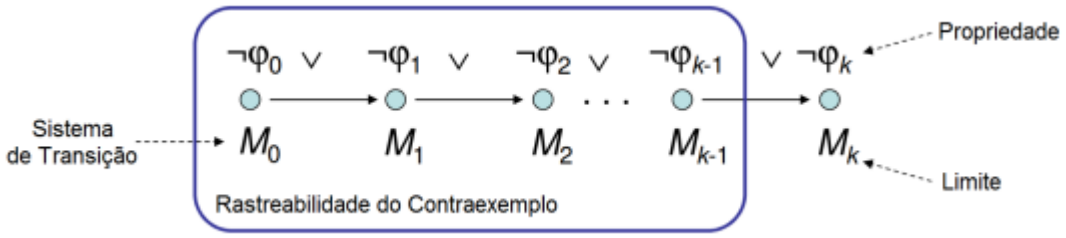

Figura 2.2: Representação Gráfica do Sistema de Transição de Estados.

em uma condição de verificação (do inglês, *Verification Condition* - VC) ψ, tal que ψ é *satisfatível* se e somente se ϕ houver um contra-exemplo de profundidade menor ou igual a k.

```
1 int main() {
2   int a[2], i, x;
3   if (x==0)
4     a[i]=0;
5   else
6     a[i+2]=1;
7     assert(a[i+1]==1);
8 }
```

(a) Exemplo de programa.

$g_1 = x_1 == 0$
$a_1 = a_0$ WITH $[i_0:=0]$
$a_2 = a_0$
$a_3 = a_2$ WITH $[2+i_0:=1]$
$a_4 = g_1 \ ? \ a_1 : a_3$
$t_1 = a_4[1+i_0] == 1$

(b) Estados gerados a partir do programa da Figura 2.3a.

Figura 2.3: Exemplo da aplicação da Técnica BMC.

A Figura 2.3 mostra um exemplo da aplicação da técnica BMC em um programa, mostrado na Figura 2.3a. A Figura 2.3b mostra os estados gerados a partir do código. Cada estado representa mudança nos valores das variáveis. Os estados a_1 e a_3 representam a atribuição de valores a posição 0 do vetor a. Primeiramente os possíveis estados alcançados são mapeados e no a_4 é realizada uma operação ternária que avalia a condicional do estado g_1 e, dependendo do resultado, o fluxo do programa será desviado para o estado a_1 ou a_3. A assertiva é executada no estado t_1. A técnica BMC tenta provar que o programa tem defeitos através da negação das propriedades de segurança geradas a partir dos estados durante a verificação. Neste exemplo, ocorre um estouro do limite de vetor no estado a_3.

2.3 *Loops* Invariantes

Um dos principais motivadores deste trabalho é a aplicação de métodos formais de verificação para identificar e gerar invariantes, a fim de restringir os espaços de estados a serem explorados, evitando o esgotamento dos recursos de memória. Pelos motivos citados, é de vital importância que este tipo de estrutura seja suportada pelas ferramentas de geração de invariantes, como o PAGAI. *Loops* invariantes são estruturas que podem ser utilizadas para provar a corretude de programas [46]. Para identificar um *loop* invariante é necessário que o mesmo obedeça aos seguintes critérios:

- **Inicialização**: Deve ser válido antes da primeira iteração.
- **Manutenção**: Ser válido durante a execução do *loop*.
- **Término**: Válido após a execução do *loop*. Esta validação é executada através de um *assert* após o término do *loop* e indicará a corretude do programa.

Quando os dois primeiros critérios são obedecidos, então, é válido afirmar que o *loop* invariante é verdadeiro antes e depois do desdobramento do laço. Existe uma similaridade à indução matemática, na qual para provar que uma propriedade é verdadeira, deve-se mostrar o caso base e o passo indutivo [46]. Um exemplo de *loop* invariante é mostrado na Figura 2.4.

```
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(i == n+ 1);
7 }
```

Figura 2.4: Exemplo de laço invariante.

Com relação ao código da Figura 2.4 podemos observar que os 3 critérios anteriormente descritos são obedecidos. Desta forma, é possível realizar as seguintes observações:

- **Inicialização**: a variável *n* recebe um valor não determinístico maior que zero, logo, a condição é verdadeira antes da execução do *loop*.
- **Manutenção**: durante o desdobramento do *loop* a condição será verdadeira até que $i = n+1$.

- **Término**: a validação do *loop* invariante será uma assertiva verificando se $i = n+1$. Caso o resultado seja verdadeiro, o *loop* invariante está correto, caso contrário há um bug na manutenção do mesmo.

2.4 Programa sem *Loops*

Na indução matemática, o desdobramento dos *loops* é realizado incrementalmente de um até *max_iterations*, *i.e.*, k cópias do respectivo *loop* são criadas e a estrutura de controle *while* é removida (veja a Fig. 1). O número de desdobramentos é medido pela contagem do número de *backjumps* [45], que é, um *bracktracking* para uma decisão anterior. Em cada passo do algoritmo de indução matemática, uma instância de programa que contém k cópias do corpo do *loop* corresponde à verificação do programa sem *loops*. Apenas instruções *if* são utilizadas a fim de evitar sua execução caso um *loop* termine antes de k iterações.

Definition 1 ***(Programa sem Loops)****um programa sem loops é representado por um programa sequencial, através do fornecimento um operador ite* $ite(\theta, \rho_1, \rho_2)$*, ao qual é fornecida uma fórmula booleana* θ *e, dependendo do seu valor, seleciona* ρ_1 *ou* ρ_2*, onde* ρ_1 *representa o corpo do loop e* ρ_2 *representa o outro operador ite, que codifica uma quantidade k de cópias do corpo do loop, ou uma assertiva.*

Cada passo do algoritmo de indução matemática tranforma um programa com *loops* em um programa sem *loops*, de forma que provar a corretude de um programa sem *loops*, consequentemente, prova a corretude do programa com *loops*. Caso o programa de entrada possua múltiplos *loops* aninhados, a quantidade de desdobramento é global, ou seja, todos os *loops* são considerados e as traduções são aplicadas recursivamente. No entanto, cada passo do algoritmo de indução matemática realiza diferentes transformações, no final do *loop*: ou para encontrar *bugs* (caso base) ou para provar que o *loop* foi suficientemente desdobrado (condição adiante).

2.5 A Técnica de Indução Matemática

A Figura 1 mostra uma visão geral do algoritmo de indução matemática [13, 8]. É importante mencionar que seu fluxo de execução é como na linguagem ANSI-C, nesse sentido,

a segunda parte da segunda instrução *if* é executada somente se a primeira é avaliada como *false*. Detalhes adicionais sobre as transformações, em cada parte algoritmo não foram adicionados para tornar mais fácil o entendimento; em vez disso, o mesmo foi mantido de forma simples e os detalhes são descritos nos próximos capítulos. A entrada do algoritmo é um programa ANSI-C P e seus invariantes, junto com uma propriedade de segurança ϕ.

O resultado da verificação é *true* se não houver caminho que viole ϕ, *false* se houver um caminho que viole ϕ ou *unknown* caso não seja capaz de chegar a um resultado conclusivo.

```
   Input: Programa P' com invariantes e a propriedade de segurança φ
   Output: TRUE, FALSE, ou UNKNOWN
 1 begin
 2     k = 1;
 3     while k <= max_iterations do
 4         if baseCase (P', φ, k) then
 5             show the counterexample s[0...k] ;
 6             return FALSE ;
 7         end
 8         else
 9             k=k+1
10             if forwardCondition (P', φ, k) or inductiveStep (P', φ, k) then
11                 return TRUE;
12             end
13         end
14     end
15     return UNKNOWN;
16 end
```

Algorithm 1: O algoritmo k-induction com re-checagem no caso base.

No caso base, o algoritmo tenta encontrar um contraexemplo até uma profundidade k, caso não seja possível obter uma resposta satisfatível através da negação da propriedade ϕ e, por consequência, a não geração do contra-exemplo, então, o algoritmo executará o próximo passo conhecido como condição adiante. Neste passo, a corretude global do *loop* com respeito a propriedade ϕ é verdadeira até a profundidade k. Caso a corretude do programa não seja provada até uma profundidade k, então o passo *inductive step* será executado. O *inductive step* verifica se a propriedade é válida para k iterações, então, assume-se que a mesma será válida para as próxima iterações. O algoritmo executará os 3 passos descritos até chegar a um resultado satisfatível ou k igual a *max_iterations*.

2.6 Interpretação Abstrata de Programas

A interpretação abstrata de programas consiste no uso de denotações para descrever computações em outro universo de objetos abstratos. Por exemplo, o cálculo -1515×17 pode denotar uma computação sobre o universo abstrato $(+), (-), (\pm)$ onde a semântica dos operadores aritméticos é definida pela regra de sinais. A execução abstrata $-1515 \times 17 \Rightarrow -(+) \times (+) \Rightarrow (-) \times (+) \Rightarrow (-)$ provê que -1515×17 é um número negativo.

Assim, a interpretação abstrata é fundamental com resultados incompletos permitindo ao programador ou compilador responder questões, as quais não se necessitam o conhecimento completo das execuções de um programa ou quais se tolera respostas imprecisas, por exemplo, prova de corretude parcial de programas que ignoram problemas de terminação [47].

Segundo Cousot [48], um entendimento mais restrito de interpretação abstrata é vê-la como uma teoria de aproximação do comportamento dinâmico de sistemas discretos (no qual um sistema discreto é um sistema dirigido à eventos, isto é, seu estado de evolução depende inteiramente da ocorrência de eventos discretos assíncronos ao longo do tempo [49], dado este contexto, um exemplo seria a semântica formal de programas. Formalmente segundo Cousot (1996), uma semântica S de uma linguagem de programação Ł associa um valor semântico em um domínio semântico *D* para cada programa p de Ł.

O domínio semântico *D* pode ser composto por sistemas de transição [50], traços, relações, funções de alta ordem e assim por diante. *D* é geralmente definido, em termos de composição, por indução sobre estruturas de objetos em tempo de execução (computações, dados e outros). $\mathscr{S}$ é geralmente definido, em termos de composição, por indução sobre estruturas sintáticas de programas, as recursões.

A semântica da linguagem de programação é mais ou menos precisa de acordo com o nível da observação considerada da execução do programa. A teoria da interpretação abstrata formaliza esta noção de aproximação e abstração em uma configuração matemática na qual é independente de aplicações particulares.

Um domínio abstrato é uma abstração da semântica concreta em forma de propriedades abstratas (aproximações das propriedades concretas - Comportamentos) e operações abstratas. Pode-se citar os seguintes tipos de abstração, para isto assuma, por exemplo, que se deve abstrair os seguintes conjuntos de rastreamentos (em geral um conjunto infinito de sequências finitas ou infinitas de estados) [51].

Ao coletar o conjunto de todos os estados que aparecem, ao longo de qualquer desses estados, se tem uma invariante global que é um conjunto de estados alcançáveis durante a execução. Assim, de um modo geral se tem um conjunto infinito de pontos $(xi, yi) : i \in \Delta$, que formaliza o método de prova de Floyd/Naur e a lógica de Hoare [52].

- **Abstração Simples:** é uma abstração por trimestres de plano que cobrem todos os estados. Isto proporciona uma análise de sinal que consiste em descobrir os possíveis sinais que as variáveis numéricas possuem para cada ponto do programa [53]. Assim invariantes locais são expressas com a seguinte forma $x \geq 0, y \geq 0$.

- **Abstração de Intervalos:** Uma abstração mais refinada que salva somente os valores mínimos e os máximos das variáveis e assim ignorando suas relações mútuas. Esta abstração fornece invariantes da forma $a \leq x \leq b, c \leq y \leq d$, onde a, b, c, d são constantes numéricas descobertas pela análise efetuada [47].

- **Abstração de Octógonos**: outra abstração refinada que identifica invariantes da seguinte forma $x \leq a, x - y \leq b$ ou $x + y \leq c$ e suas inversas, onde a, b, c são constantes numéricas descobertas pela análise efetuada [54].

- **Abstração de Poliedros Convexos**: esta abstração resulta em desigualdades lineares tais como $a.x + b.y \leq c$, onde a, b, c são constantes numéricas automaticamente descobertas pela análise efetuada [55].

- **Abstração de Elipsoides**: é um típico exemplo de abstração não linear, esta abstração é capaz de descobrir invariantes da forma $(x-a)^2 + (y-b)^2 \leq c$, onde a, b, c são constantes numéricas automaticamente descobertas pela análise efetuada [56].

- **Abstração Exponencial**: Esta é outro tipo de abstração não linear, onde as invariantes tem a forma $a^x \leq y$ [57].

2.7 Transformações Aplicadas aos Programas

Em termos de transformação de programa (realizadas automaticamente pelo método proposto), o caso base insere uma suposição (*assert*) de desdobramento, consistindo da condição

de terminação σ após o *loop*, isto é, uma pós-condição que será utilizada no caso base para encontrar um contra-exemplo. Isto é necessário devido ao fato de ser uma condição de início para a validação de σ. De fato, $I \wedge T \wedge \sigma \Rightarrow \phi$, onde I é a condição incial, T é a relação de transição de P' (programa sem *loops*) e ϕ é a propriedade de seguranção a ser verificada.

A condição adiante insere uma assertiva (*assume*) em oposição a suposição, após o *loop*, como $I \wedge T \Rightarrow \sigma \wedge \phi$. Especificamente, a condição adiante proposta por Große *et al.* [10] introduz uma sequência de comandos a fim de verificar se existe um caminho entre o estado inicial e o estado inicial k, à medida que o algoritmo proposto, automaticamente insere uma asssertiva (*i.e.,* o *loop* invariante), sem intervenção manual. Isto é feito no final do *loop*, a fim de verificar se todos os estados são alcançáveis em k iterações.

É importante notar que as transformações aplicadas ao caso base propostos e condição adiante, podem facilmente ser implementadas na técnica BMC simples.

Contudo, referente ao passo indutivo, várias transformações são realizadas. Um *loop* ANSI-C $while(c)\{E;\}$ é convertido em

$$A; while(c)\{S;E;U;\}R; \tag{2.4}$$

onde A (um bloco de comando) é responsável por atribuir valores não-determinísticos a todas as variáveis de *loop* [13], *i.e.*, o estado é *havocked* antes do *loop*, c é a condição de saída da estrutura *while*, S armazena o estado atual das variáveis do programa, antes de executar as instruções de E, E é o código atual dentro do *while*, U atualiza todas as variáveis do programa com valores locais e depois de executar E, e R remove os estados redundantes. A hipótese de indução do passo indutivo consiste na conjunção entre pós-condiçoes (*Post*) e uma condição de término do *loop* (σ).

Em contraste com a abordagem proposta, Große *et al.* [10] elimina todas as variáveis, dificultando a aplicação à programas variados, desde que estes não forneçam informações suficientes, o que restringiria a destruição de variáveis.

Uma estrutura *for* pode facilmente ser convetida em uma estrutura *while*:

$$for(B;c;D)\{E;\} \tag{2.5}$$

pode ser reescrito como

$$B; while(c)\{E;D;\}, \tag{2.6}$$

onde *B* é a condição inicial do *loop*, *c* é a condição de saída, *D* é o incremento de cada iteração sobre *B*, e *E* é o código atual do corpo da estrutura *for*. Nenhuma outra transformação é aplicada à estrutura *for* durante o passo indutivo. Além disso, uma estrutura *do while* pode ser facilmente convertida *while*, com uma diferença: Código do corpo do *loop* dele deve executar ao menos uma vez antes de verificar a condição de saída.

2.8 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, operações lógicas e regras de precedência. Em seguida, a Verificação de Modelos Limitada apresentou o conceito da técnica BMC, a representação de um programa em sistema de transição de estados que são autômatos utilizados para modelagem do comportamento de um sistema a fim de checar a negação das propriedades de segurança (havendo violação será gerado um contraexemplo) , ao final é apresentado um exemplo de código e as transformações nele aplicadas.

Após a Verificação de Modelos Limitada, o conceito de *loop* invariante foi apresentado, e as três propriedades para corretude da invariância: inicialização, manutenção e término. Em seguida, foi mostrada a conversão de programa com *loops* em programas livre de *loops*. Esta conversão consiste em realizar *k* cópias do corpo do *loop* representando-o de forma sequencial através de instruções *if*, provar a corretude de um programa livre de *loops*, consequentemente, prova a corretude do programa original com *loops*. Em seguida, foi mostrado o conceito de indução matemática e seus três passos (caso base, condição adiante e passo indutivo). No caso base, o algoritmo tenta gerar um contraexemplo (se houver violação da propriedade de segurança). Na condição adiante, o algoritmo tenta provar que todos os estados foram alcançados em *k* iterações. Por fim, no passo indutivo o algoritmo tenta provar que se a propriedade é válida até uma profundidade *k* então é válida para a próxima iteração.

Após a contextualização da técnica de indução matemática, é apresentada a Interpretação Abstrata de Programas que consiste em denotações para representar objetos abstratos, desta forma, permite responder a questões sem a necessidade de execução completa de um programa. Com base nestas abstrações é possível gerar valores que limitem os espaços de estados de um determinado programa. Por fim, as tranformações aplicadas aos programas pelo algoritmo de

indução matemática em cada um dos três passos e a remoção de estados redundantes.

A compreensão dos assuntos contidos neste trabalho, exige o entendimento dos assuntos neste capítulo. O entendimento da técnica de indução matemática é essencial para o entendimento do papel das invariantes no processo de verificação.

Capítulo 3

Trabalhos Relacionados

A aplicação da técnica de indução matemática está se tornando cada vez mais popular na área de verificação de software. Por um lado, trabalhos anteriores exploraram provas por indução matemática de hardware e software com algumas limitações, como a necessidade de mudança de código com o objetivo de introdução invariantes de *loop* [58, 59, 10]. Este tipo de intervenção manual complica a automatização de um determinado processo de verificação, a menos que outros métodos sejam usados na combinação como o cálculo automático de identificação de *loop* invariantes [60, 61]. Similarmente ao método proposto por Hagen e Tinelli [62], o método proposto nesta dissertação é completamente automático e não exige que o usuário identifique manualmente o *loop* invariante e nem a assertiva de validação ao final de cada *loop*. Por outro lado, ferramentas BMC estado-da-arte na verificação de sistemas têm sido amplamente utilizadas, mas dado que eles tipicamente analisam o programa até uma determinada profundidade k [5, 6], a completude só pode ser assegurada se for conhecido um limite superior relativo à profundidade do espaço de estados, geralmente não é o caso.

Este trabalho preenche a lacuna mencionada acima, fornecendo evidência clara de que o algoritmo de indução matemática pode ser aplicado a uma variedade de programas ANSI-C sem intervenção manual. Em resumo, o usuário não precisa guiar a ferramenta durante a exploração dos estados da aplicação a fim de eliminar redundâncias, gerar invariantes e, em última instância, provar a corretude do programa.

3.1 CPAchecker com Indução Matemática e Invariantes

Atualmente, o CPAchecker é a ferramenta que possui melhores resultados na difícil tarefa de combinar a geração de invariantes com o algoritmo de indução matemática. Beyer *et al.* [35] introduziu uma abordagem diferente em relação aos trabalhos com invariantes: foi desenvolvido um algoritmo responsável pela geração de invariantes que é separado do algoritmo de verificação, denominado Invariantes Continuamente Refinadas (do inglês, *Continuously-Refined Invariants*). Este algoritmo é executado paralelamente à verificação atuando como um assistente para o algoritmo de indução matemática do CPAchecker. O algoritmo refina as invariantes com maior precisão conforme a quantidade de estados desenrolados aumenta, por este motivo, as invariantes iniciais geradas pela própria ferramenta (*i.e.*, sem um gerador invariável como PAGAI ou PIPS) são descritas como fracas, ou seja, não são fortes o suficiente para restringir o espaço de estados e, assim guiar o algoritmo de indução matemática na determinação da corretude de um programa. O tipo de interpretação abstrata utilizada para analisar o código do programa e gerar invariantes é a Abstração de Intervalos. Este método ajusta e refina continuamente a precisão das invariantes durante o processo de verificação com a finalidade de gerar invariantes mais fortes [35]. Diante disso, a tarefa de verificação tira vantagem dos valores gerados anteriormente, de modo que o algoritmo de indução matemática tem a hipótese indutiva fortalecida.

3.2 2LS

Brain *et al.* [21] apresenta uma abordagem similar ao método proposto neste trabalho. O método adota as técnicas de Interpretação Abstrata, Verificação de Modelos Limitada e Indução matemática em uma ferramenta chamada *k*I*k*I. Esta ferramenta não é apenas uma combinação das técnicas citadas, mas permite a interação entre elas de forma que uma reforce a outra, resultando em uma única ferramenta de verificação. Para criar um robusto sistema de verificação, é necessário combinar uma variedade de técnicas, onde uma opção seria a execução em série de ferramentas independentes de forma sequencial, no entanto, sem interação entre os algoritmos de verificação. Brain *et al.* propõe nesta abordagem, a utilização da indução matemática para verificar a corretude de programas, a interpretação abstrata para a geração de invariantes e a técnica BMC para validar o resultado da indução matemática quando um contra-exemplo é

gerado. A geração de invariantes é baseada em templates, onde as mesmas são geradas progressivamente para fortalecer a hipótese indutiva. O método proposto não necessita de interação com o usuário para guiar o algoritmo.

Os experimentos executados mostram que os resultados apresentados pela ferramenta são promissores, mas ainda estão abaixo dos resultados apresentados pelo ESBMC e CPAchecker. O principal problema do método proposto por Brain *et al.* é a quantidade de resultados inconclusivos e resultados incorretos, esta é uma falha grave, uma vez que impacta diretamente na confiabilidade da ferramenta.

3.3 CBMC com Indução Matemática

Große *et al.* descreve um método para provar propriedades de projetos TLM (do inglês, *Transaction Level Modeling*), em SystemC [10]. Essa abordagem basicamente converte um programa SystemC em um programa ANSI-C e depois realiza a prova de propriedades por indução matemática utilizando a ferramenta CBMC [5].

A conversão do programa SystemC é seguida pela geração e adição de lógicas para monitorar propriedades TLM (utilizando *asserts* e máquinas de estado finito). A verificação do programa ANSI-C é realizada através da ferramenta CBMC [5]. No CBMC, a prova das propriedades é realizada pelo algoritmo de indução matemática implementado pelos autores. A ferramenta possui severas limitações, por exemplo, não consegue finalizar a verificação em tempo hábil. O algoritmo de indução matemática implementado pelos autores é semelhante ao descrito neste trabalho pois, utiliza três etapas: caso base, condição adiante e passo indutivo.

3.4 Outros

Madhukar *et al.* [36] descreve um método para acelerar a geração de invariantes adotando a análise do código fonte em relação aos *loops*. A idéia básica é identificar invariantes e seus desvios a fim de acelerar o processo de verificação. Seu artigo compara alguns verificadores de modelo (*e.g.*, UFO, CPAChecker, CBMC, e IMPARA) sem a utilização do algoritmo de indução matemática em relação à geração invariante, a fim de acelerar a verificação de programas com *loops*. Em resumo, esta técnica proposta por Madhukar *et al.* [36] é baseada em *loops* de aproximação para detecção rápida de contraexemplos e funciona como um pré-processador

para substituir *loops* e melhorar o processo de verificação o que reduz o tempo de verificação, diminui a quantidade de erros além de melhorar a sua confiança. Em contraste com Madhukar *et al.*, o método proposto neste trabalho não modifica os *loops* existentes, mas sim inclui *asserts* baseados em invariantes para guiar o algoritmo de indução do ESBMC.

Em relação ao grande número de trabalhos que utilizam o método de indução matemática onde apenas o caso base e o passo indutivo são utilizados, Gadelha *et al.* [8, 29] apresenta uma versão que acrescenta a condição adiante. Este passo adotado entre o caso base e o passo indutivo verifica se o *loop* foi completamente desenrolado e se todos os estados possíveis foram alcançados dentro de *k* iterações. Outro diferencial é a geração de invariantes, de forma completamente automática, para as variáveis internas do *loop* e verificar a corretude do *loop* invariante, onde normalmente, não se sabe o limite superior. É neste momento que a geraçao de invariantes se mostra importante pois, não necessita que o usuário as crie manualmente. Enquanto Gadelha gera invariantes relacionadas ao escopo do *loop*, o método proposto neste trabalho gera invariantes para todas as variáveis do programa.

Finalmente, Donaldson *et al.* [58] descreveu uma ferramenta de verificação chamada *Scratch*, com o objetivo de detecção de corridas de dados durante o acesso directo à memória (do inglês, *Dynamic Memory Access*, DMA), no processador CELL BE da IBM [58], através da técnica de indução matemática. A ferramenta insere afirmações no programa para modelar o comportamento do controle de fluxo da memória e, desta forma, tenta provar a corretude do programa. Sua abordagem foi implementada na ferramenta Scratch utilizando apenas o caso base e o passo indutivo. Ainda assim, a ferramenta é capaz de provar a ausência de corrida de dados; entretanto, a abordagem proposta é direcionada a uma classe específica de problemas para um determinado tipo de hardware. Os passos de seu algoritmo são semelhantes aos propostos neste artigo, no entanto, é necessário utilizar anotações no código fonte de entrada a fim de identicar *loops* invariantes.

3.5 Diferenças Entre o Método Proposto e as Abordagens Apresentadas

Com a gama de ferramentas que trabalham com indução matemática e invariantes, é necessário conhecer algumas das características mais importantes de cada uma.

Ferramenta	Invariantes	Identifica Loop Invariante	Geração Contínua de Inv.	Método	Em Desenv.
ESBMC + Ind. Mat. [8, 29]	Sim	Sim	Não	Indutivo	Sim
CPAchecker [35]	Sim	Sim	Sim	Abst. de Intervalos	Sim
2LS [21]	Sim	Sim	Sim	Abst. de Intervalos	Sim
DepthK [17]	Sim	Sim	Não	Poliedral	Sim
CBMC + Ind. Mat. [10]	Não	Sim	Não	N/A	Não

Tabela 3.1: Comparativo entre trabalhos similares

A Tabela 3.1 mostra a comparação de características entre as principais ferramentas descritas neste capítulo e o método proposto, onde:

- O ESBMC é a ferramenta de verificação utilizada neste trabalho, o algoritmo de indução matemática implementado na mesma, apesar de estar em constante evolução não gera invariantes para todas as variáveis do programa, apenas para aquelas utilizadas no corpo do *loop* e para verificar a condição do *loop* invariante, é exatamente neste ponto que o método proposto fecha este *gap* adicionando esta característica de forma completamente automática e propagando invariantes a todas as variáveis do programa.

- Em relação ao CPAchecker [35], ferramenta estado-da-arte na geração de invariantes combinada com a técnica de indução matemática, a diferença está no domínio abstrato utilizado. A abordagem proposta neste trabalho utiliza o domínio poliedral enquanto que o CPAchecker utiliza a Abstração de Intervalos continuamente refinados (gera invariantes continuamente, conforme a quantidade de estados verificados aumenta o CPAchecker gera novas invariantes), enquanto que a abordagem utilizada neste trabalho gera as invariantes uma única vez através da análise estática do código.

- O 2LS [21] é uma ferramenta que apresenta características muito próximas do CPAchecker, mas os resultados apresentados pela mesma mostram que há um longo caminho a percorrer até que esta ferramenta transforme-se em uma alternativa viável ao CPAchecker ou mesmo ao método proposto neste trabalho.

- O CBMC [10] com indução matemática é a ferramenta que possui mais características que desencorajam sua utilização, principalmente por esta característica não mais estar em desenvolvimento.

Finalmente, quanto ao método proposto neste trabalho, a principal diferença em relação à abordagem, até aqui descrita reside na geração automática de invariantes que são introduzidas no código fornecido pelo usuário, a fim de verificar se há um caminho de violação entre o estado inicial e um determinado limite superior k ou não.

3.6 Resumo

Neste capítulo, as características de trabalhos relacionados foram estabelecidas e comparadas. Na primeira seção, uma descrição de trabalhos que implementam a técnica de indução matemática e que necessitam de intervenção do usuário, tornando o processo custoso e lento. Em seguida, a versão do CPAchecker utilizando indução matemática e invariantes é comparada ao método proposto, a principal diferença reside no fato o CPAchecker gerar invariantes continuamente enquanto que o método realiza a tarefa de geração de invariantes apenas uma única vez.

Uma abordagem promissora, mas ainda experimental, é a ferramenta 2LS que combina invariantes com indução matemática e a técnica BMC de forma a produzir invariantes mais fortes durante a verificação, similar ao CPAchecker. Em seguida, uma variante do CBMC com indução matemática é descrito, esta versão possui muitas restrições, por exemplo, o tempo limite de verificação (15 minutos) é excedido em muitos *benchmarks*.

Em seguida, outros métodos similares ao apresentado neste trabalho são descritos, como por exemplo, Madhukar *et al.* [36] que gera invariantes através da análise estática de código, e Donaldson *et al.* que descreve um método verificação baseado em anotações inseridas manualmente no código fonte. Por outro lado, Gadelha *et al.* [8, 29] apresenta uma abordagem completamente diferente, ao qual se faz presente neste trabalho. Por fim, na segunda seção foram avaliados os principais trabalhos relacionados onde suas principais características foram comparadas em forma de Tabela.

Capítulo 4

Método Proposto

Neste capítulo serão descritas duas ferramentas adotadas para análise estática de código e geração de invariantes voltadas a uma ampla variedade de programas ANSI-C. A primeira ferramenta descrita é o PIPS [63] que é um *framework* interprocedural *source-to-source* para programas C e Fortran. A segunda ferramenta é o PAGAI [14]. Esta ferramenta serve para análise estática e automática de código fonte, bem como possui a capacidade de gerar invariantes indutivas. As ferramentas selecionadas para integrar o método proposto foram escolhidas devido à documentação disponível, facilidade de uso e eficiência durante experimentos preliminares.

4.1 Geração de Invariantes Usando PIPS

O PIPS é uma ferramenta em pleno desenvolvimento há quase 20 anos, seu objetivo é analisar automaticamente programas extensos [26], realizando a análise em dois passos. Primeiramente, cada instrução do programa é associada à um transformador afim, representando a sua função de transferência subjacente. Trata-se de um procedimento de análise de código que o analisa de baixo para cima, partindo de instruções mais simples, então passa por instruções compostas, até a definição de funções. Em segundo lugar, as invariantes são propagadas com instruções, usando transformadores previamente calculados.

O PIPS recebe um programa como entrada e gera as invariantes, as mesmas são escritas no código fonte em forma de comentário em um arquivo de saída ANSI-C. Estas invariantes são então traduzidas e instrumentadas para o programa mencionado, como declarações de su-

posição. Em particular, a função `assume` (*expr*) é adotada a fim de limitar possíveis valores de variáveis que estão relacionadas a estas invariantes. Esta etapa é completamente necessária, uma vez que todos os invariantes gerados pelo PIPS contêm o sufixo #*init*, que inclui expressões matemáticas (textit ex., $2j < 5t$). De fato, tal sintaxe não é aceita nos programas ANSI-C, que então leva à necessidade de transformações em texttt assume (*expr*).

O método proposto é um aprimoramento e sua complexidade assintótica e é $O(n^2)$ [13], onde n é o tamanho do código com invariantes geradas pelo PIPS e está dividido em três partes: (1) identificação das estruturas `#init`, (2) geração do código de suporte a tradução das estruturas `#init` invariantes, e, finalmente, (3) tradução das expressões matemáticas em código ANSI-C.

A linha 6 do algoritmo 2 realiza a primeira parte da tradução das invariantes, a qual consiste da leitura de cada linha do código analisado, com invariantes, e identificar se um determinado comentário é uma invariante gerada pelo PIPS (linha 4) ou não. Se uma invariante é identificada e contém a estrutura `#init`, então o número da linha é armazenado, bem como o tipo de nome da variável associada que possui o prefixo da estrutura `#init` (linha 6).

Após a identificação das estruturas `#init` nas invariantes, a segunda parte do algoritmo 2 executa as instruções começando na linha 11, que também executa a análise de cada linha do código com invariantes (`PIPSCode`), mas com o objetivo de identificar a declaração e o início de cada função. Para cada função identificada, o algoritmo verifica se existe a estrutura `#init` (linha 14) e, caso verdadeiro, uma nova linha é gerada para cada variável no início da função com a declaração de uma variável auxiliar que contém o valor antigo, *i.e.*, inicializada no início da função. A variável recém-criada possui o formato `type var_init = var` , onde `type` é o tipo da variável, e `var` é o nome da variável. Durante a execução deste algoritmo, é gerada uma nova instância do código (`NewCodeInv`).

Finalmente, cada linha que possui uma invariável gerada pelo PIPS é transformada em expressões suportadas pelo padrão ANSI-C. Estas transformações consistem na aplicação de expressões regulares (linha 22) para identificar operadores matemáticos (*e.g.*, from $2j$ to $2 * j$) e substituir `#init` por `_init`, o que indica que uma nova variável auxiliar deverá ser gerada e seu valor será utilizado para inicializá-la.

Para cada comentário que representa uma invariante, uma nova linha é gerada no novo formato, onde esta linha é concatenada com o operador `&&` e adicionada à função `__ESBMC_assume`.

A Figura 4.1 apresenta a saída do algoritmo 2 referente ao programa ANSI-C uti-

Input: PIPSCode - Código ANSI-c com invariantes
Output: NewCodeInv - Novo código com invariantes suportadas por programas ANSI-C

```
// dicionário para suportar a estrutura #init
dict_varinitloc ← { }
// lista para o código gerado após a tradução das invariantes
NewCodeInv ← { }
// Part 1 - identificando a flag #init nas invariantes
foreach linha do código PIPSCode do
    if existe um comentário do PIPS no padrão // P(w,x) {w == 0, x#init > 10} then
        if o comentário está no padrão ([a-zA-Z0-9_]+)#init then
            dict_varinitloc[line] ← a variável sufixada #init
        end
    end
end
// lista para as invariantes traduzidas
listinvpips ← { }
// Part 2 - geração de código para suportar a estrutura #init e
   correções no formato das invariantes
foreach linha do PIPSCode do
    NewCodeInv ← line
    if é o início de uma função then
        if tem alguma linha desta função ∈ dict_varinitloc then
            foreach variable ∈ dict_varinitloc do
                NewCodeInv ← Declarar uma variável com este padrão Declare
                 type var_init = var;
            end
        end
    end
    if existe um comentário do PIPS neste padrão // P(w,x) {w == 0, x#init > 10} then
        foreach expression ∈ {w == 0, x#init > 10} do
            listinvpips ← Reformula a expressão de acordo com a sintax dos
              programas e substitui #init by _init
        end
        NewCodeInv ← __ESBMC_assume(concatena com && as invariantes que
          estão em listinvpips)
    end
end
```

Algorithm 2: Algoritmo de tradução das invariantes geradas pelo PIPS.

lizado como exemplo motivador que foi mostrado na Figura 1.2, com código PIPS e comentários. Após a geração e tradução das invariantes, o DepthK utilizando o algoritmo de indução matemática foi capaz de provar a corretude deste programa.

```
1 //PIPS invariants
2 void main(){
3     int x = 0 ;
4     int t = 0 ;
5     int phase = 0 ;
6     //  P(phase,t,x) {phase==0, t==0, x==0}
7     __ESBMC_assume( phase==0 && t==0 && x==0 );
8
9     while ( t < 100) {
10      if( phase == 0 ){
11      //  P(phase,t,x) {phase==0, t<=99, x<=2t, 0<=t+x}
12      __ESBMC_assume( t<=99 && x<=2*t && 0<=t+x );
13       x = x + 2;
14      }
15      if( phase == 1 ){
16      //  P(phase,t,x) {phase==1, 0<=t, t<=99, 0<=t+x, x<=2t+2}
17      __ESBMC_assume( 0<=t && t<=99 && 0<=t+x && x<=2*t+2 );
18       x = x - 1;
19      }
20      phase = 1 - phase;
21      //  P(phase,t,x) {0<=t, t<=99, x<=2t+2, 0<=t+x+1}
22      __ESBMC_assume( 0<=t && t<=99 && x<=2*t+2 && 0<=t+x+1 );
23      t++;
24    }
25    //  P(phase,t,x) {t==100, 0<=x+100, x<=200}
26    __ESBMC_assume( t==100 && 0<=x+100 && x<=200 );
27    assert( x <= 100 );
28 }
```

Figura 4.1: Exemplo com invariantes geradas pelo PIPS em forma de comentários.

4.2 Geração de Invariantes Usando PAGAI

A ferramenta de geração de invariantes PAGAI [14] é um analisador estático que utiliza a estrutura do compilador conhecido como LLVM [64] e, como resultado, calcula invariantes indutivas em variáveis números do programa analisado. A ferramenta PAGAI adota um algoritmo de análise de código fonte que utiliza o método de interpretação abstrata que permite a inferência de invariantes para programas ANSI-C/C++ utilizando a estrutura LLVM.

A ferramenta PAGAI realiza análise de relação linear que obtém invariantes como conjunções de desigualdades lineares ou, de forma equivalente, poliedros convexos; no entanto, também suporta outros domínios abstratos (*e.g.*, octágonos [54] e produtos de intervalos [47]). O domínio abstrato a ser utilizado pelo PAGAI é parametrizado pelo usuário em tempo de execução, com base nisto a ferramenta analisa o código e infere invariantes.

Na avaliação experimental apresentada por Henry, Monniaux e Moy [14], PAGAI tem sido aplicado a exemplos do mundo real (*e.g.,* códigos industriais) e programas da plataforma

GNU com no máximo 954 linhas de código, para as quais a habilidade de executar qualquer sob código ANSI-C e C++ por meio da infra-estrutura do LLVM, é particularmente útil. De acordo com Henry, Monniaux e Moy [14], o *front-end* de muitas ferramentas de análise possuem restrições (*e.g.,* não existem instruções de goto e nenhuma aritmética de ponteiro) que são necessárias para programas críticos.

A ferramenta PAGAI, por sua vez, não sofre essas restrições, mas pode aplicar abstrações grosseiras a alguns programas ANSI-C e C++ que pode levar a invariantes fracas e, consequentemente, guiar a ferramenta de verificação a um resultado incorreto.

Neste trablaho, o PAGAI é adotao para inferir invariantes, a fim de fornecer valores para restrição de espaço de estados do algoritmo de indução matemática.

O PAGAI recebe como entrada o programa a ser analisado e gera as invariantes, as mesmas são fornecidas como comentários no código de saída. Essas invariantes são traduzidas e instrumentados no programa como declarações *assume*, como mostrado no algoritmo 3.

Diferentemente do PIPS, estas invariantes necessitam de pequenas alterações em relação a expressões matemáticas, dado que estão no formato suportado pelo padrão ANSI-C. O PAGAI é capaz de verificar se algum erro ocorreu durante o processo de geração de invariantes (linha 4). Finalmente, o algoritmo acrescenta as invariantes do programa sobre o ponto de controle original identificado pela ferramenta. Estas etapas são repetidas até que todas as invariantes do programa estejam presentes no novo programa analisado (`NewCodeInv`).

```
   Input: Código Analisado - Código ANSI-C sem invariantes do PAGAI
   Output: NewCodeInv - Novo código suportado por programas ANSI-C com
           invariantes
 1 NewCodeInv ← { cria uma cópia do código analisado }
 2 Compilar o código fonte original em bitcode LLVM
 3 Executa o PAGAI adotando o bitcode LLVM que está em NewCodeInv
 4 if um erro ocorrer durante a geração de invariantes then
 5 |   abort();
 6 end
 7 Lê o o arquivo de saída para identificar as invariantes do programa
 8 foreach programa com as invariantes geradas pelo PAGAI do
 9 |   Traduzir a invariante para a instrução assume do ESBMC
10 |   Salvar a invariante atual sobre o ponto de controle correspondente em
   |    NewCodeInv
11 end
12 NewCodeInv pronta para o ESBMC
```

Algorithm 3: Algoritmo de tradução das invariantes geradas pelo PAGAI.

Na Figura 4.2 é possível visualizar a saída do algoritmo 3 em relação ao programa ANSI-C adotado como exemplo de execução e mostrado na Figura 1.2 com os comentários de código gerados pelo PAGAI.

```
1 //PAGAI invariants
2 void main(){
3     int x = 0 ;
4     int t = 0 ;
5     int phase = 0 ;
6
7     while ( t < 100) {
8
9       /* invariant:
10       -2 * x + t + 3 * phase = 0
11       3 - 2 * x + t >= 0
12       -x + 2 * t >= 0
13       147 + x -2 * t >= 0
14       2 * x - t >= 0
15      */
16      __ESBMC_assume( -2*x+t+3*phase == 0 );
17      __ESBMC_assume( 3-2*x+t >= 0 );
18      __ESBMC_assume( -x+2*t >= 0 );
19      __ESBMC_assume( 147+x-2*t >= 0 );
20      __ESBMC_assume( 2*x-t >= 0 );
21      if( phase == 0 ){
22       x = x + 2;
23      }
24      if( phase == 1 ){
25       x = x - 1;
26      }
27      phase = 1 - phase;
28      t++;
29    }
30    assert( x <= 100 );
31 }
```

Figura 4.2: Exemplo de verificação com invariantes geradas pelo PAGAI para o algoritmo de indução matemática.

4.3 Exemplo para Verificação

Esta seção mostra como o algoritmo de indução matemática (ver Algoritmo 1) usa invariantes para provar a corretude de programas ANSI-C, o qual é mostrado na Figura 1.2. Este programa foi levemente modificado, a fim de substituir a condição de término do *loop* $t < 100$ por $t < N$, onde N é um valor não-determinístico que varia de zero até o valor máximo suportado

pelo tipo *unsigned int*. Além disso, a propriedade a ser verificada *i.e.*, $x <= N$, foi substituída por $t <= N$, como explicado na Seção 1.1.

Vale ressaltar que, com uma pequena modificação neste programa para reduzir o espaço de estados a ser verificado e facilitar a verificação é possível alcançar o resultado com menos desdobramentos de *loop*. A técnica BMC necessitaria desdobrar o *loop* relacionado ao limite superior ($t < N$) e, o no algoritmo de indução matemática com invariantes, é possível alcançar o resultado da verificação com k (número de iterações de *loop*) igual a 2. Posto que, se um *loop* é executado, a condição associada é verificada pelo menos duas vezes. Na primeira vez, o corpo do laço será executado e a verificação de parada é executada mais uma vez; No entanto, o limite k será incrementado, mesmo se a condição de parada for verdadeira.

4.3.1 Caso Base

No caso base, o objetivo é encontrar um contra-exemplo de ϕ (neste caso, a propriedade $t <= N$ é um *assert*, linha 28), com até k iterações do *loop*. Assim, para o caso base, as pré- e pós-condições do *loop* mostradas na Figura 1.2, em linguagem chamada Atribuíção Única Estática (do inglês, *Static Single Assignment* - SSA) [45] que é utilizada pelo solucionador SMT utilizado no ESBMC, como mostrado a seguir:

$$Pre := \left[\; N_1 = nondet\_uint \wedge t_1 = 0 \wedge phase_1 = 0 \wedge \; x_1 = 0 \; \right]$$

e

$$Post := \left[\; N_1 \leq t_k \wedge x_k \leq 2 * t_k \wedge 0 \leq t_k + x_k \Rightarrow t_k \leq N_1 \; \right],$$

onde *Pre* e *Post* são as pré- e pós-condições, que devem ser verdadeiras antes e depois da execução do *loop* do programa, respectivamente. Nas pré-condições, N_1 é o limite do *loop*, enquanto x_1, t_1, e $phase_1$ representam a primeira atribuição às variáveis do programa x, t, e *phase*, respectivamente. Nas pós-condições, t_k e x_k representam a atribuição $N_1 + 1$ para as variáveis t e x, respectivamente. O código resultante das transformações do caso base pode ser visto na Figura 4.3).

Nota-se que na Figura 4.3, há duas expressões *assume* idênticas (linhas 4 e 6). A primeira está relacionada à declaração das variáveis x, t e *phase* e o PIPS assume que seus valores iniciais são iguais a zero, o que resulta na criação da primeira expressão *assume*. O

segundo *assume* refere-se ao resultado da análise de código realizada, dado que não há alteração nos seus valores, a nova expressão *assume* repete a comparação com zero.

```
1 void main(){
2  int x, t, phase;
3  int N=nondet_int();
4  __ESBMC_assume(phase==0 && t==0 && x==0);
5  assert(N>=0);
6  __ESBMC_assume(phase==0 && t==0 && x==0);
7  L1:if !(t < N) goto L4
8   __ESBMC_assume(phase+t==0 && 1<=N+phase && phase<=x &&
9                  2*phase+x<=0);
10  if !(phase == 0) goto L2
11  __ESBMC_assume(phase==0 && t==0 && x==0 && 1<=N);
12  x = x + 2;
13 L2:__ESBMC_assume(phase+t==0 && 1<=N+phase && phase<=x &&
14                   2*phase+x<=2 && 3*phase + 2<=x);
15  if !(phase == 1) goto L3
16  x = x - 1;
17
18 L3:__ESBMC_assume(phase+t==0 && 1<=N + phase && phase<=x &&
19                   2*phase+x<=2 && 3*phase+2<=x);
20  phase = phase -1;
21  __ESBMC_assume(phase+t==-1 && 0<=N+phase&& 2*phase+x<=0 &&
22                 phase+1<=x && 3*phase+5<=x);
23  t = t + 1;
24  goto L1
25
26 L4:__ESBMC_assume(phase+t==0 && N+phase<=0 && phase<=x &&
27                   2*phase+x<= 0);
28  __ESBMC_assert(t<=N);
29  __ESBMC_assume(phase+t==0 && N+phase <= 0 && phase<=x &&
30                 2*phase+x<=0 => t<=N);
31 return 0;
32 }
```

Figura 4.3: Código de exemplo para a prova por indução matemática, durante o caso base considerando invariantes de PIPS.

4.3.2 Condição Adiante

Na condição adiante, o algoritmo de indução matemática do ESBMC tenta provar que o respectivo *loop* é suficientemente desdobrado e se a propriedade é válida em todos os estados alcançados dentro de k iterações.

As pré- e pós-condições do *loop* mostradas na Figura 4.3 no formato da linguagem SSA, podem ser definidas como:

$$Pre := \left[\; N_1 = nondet\_uint \wedge t_1 = 0 \wedge phase_1 = 0 \wedge \; x_1 = 0 \; \right]$$

and

$$Post := \left[\; N_1 \leq t_k \wedge x_k \leq 2 * t_k \wedge 0 \leq t_k + x_k \Rightarrow t_k \leq N_1 \; \right]$$

As pré-condições da condição adiante são idênticas as do caso base. Nas pós-condições *Post* há uma assertiva para verificar se o *loop* foi suficientemente desdobrado, o que é representado pela constraint $t_k \leq N_1$, onde t_k é o valor da variável t na iteração $N_1 + 1$.

O código resultante, em relação às transformações realizadas na condição adiante, podem ser vistas na Figura 4.4. A condição adiante tenta provar que o *loop* está suficientente desdobrado (verificando o *loop* invariante 29) e se a propriedade é válida em todos os estados alcançados dentro de k iterações (pela verificação do *assert* na linha 31).

4.3.3 Passo Indutivo

No passo indutivo, o algoritmo de indução matemática tenta provar que se uma propriedade é válida até uma profundidade k, a mesma deve ser válida para o próximo valor de k. Várias alterações são realizadas durante esta etapa. Primeiramente, uma estrutura chamada *statet* é definida, esta contém todas as variáveis dentro de um *loop* e sua condição de saída. Então, uma variável do tipo *statet*, chamada *cs* (estado atual), é declarada e passa a ser responsável por armazenar o valor de cada variável em cada estado verificado pelo ESBMC.

Antes de iniciar um *loop*, todas as suas variáveis [13] são inicializadas com valores não-determinísticos e armazenadas em *sv*, na primeira iteração do *loop* para que o modelo verificador possa, implicitamente, explorar todos os estados possíveis.

Dentro de um *loop*, depois de armazenar o estado atual e executar o código dentro do corpo, todas as variáveis de estado são atualizadas com os valores da iteração atual.

Uma instrução *assume* é então inserida, com a condição de que o estado atual seja diferente do anterior a fim de evitar que estados redundantes sejam inseridos no vetor de estados. Quando isso acontece, $sv_j[i]$ é comparado com cs_j, para $0 < j \leq k$ e $0 \leq i < k$. No exemplo mencionado, as restrições são:

```
1 void main(){
2  int x, t, phase;
3  int N=nondet_int();
4  __ESBMC_assume(phase==0 && t==0 && x==0);
5  assert(N>=0);
6  __ESBMC_assume(phase==0 && t==0 && x==0);
7  L1: if !(t < N) goto L4
8   __ESBMC_assume (phase+t==0 && 1<=N+phase && phase<=x
9   && 2*phase+x<=0)
10   if !(phase == 0) goto L2
11   __ESBMC_assume(phase==0 && t==0 && x==0 && 1<=N);
12   x = x + 2;
13
14  L2: __ESBMC_assume(phase+t==0 && 1<=N+phase && phase<=x
15  && 2*phase+x<=2 && 3*phase+2<= x);
16   if !(phase == 1) goto L3
17   x = x - 1;
18
19  L3: __ESBMC_assume(phase+t==0 && 1<=N+phase && phase<=x
20  && 2*phase+x<=2 && 3*phase+2<=x);
21   phase = -phase + 1;
22   __ESBMC_assume(phase+t==-1 && 0<=N+phase && 2*phase+x<=0
23   && phase+1<=x && 3*phase+5<= x);
24   t = t + 1;
25   goto L1
26
27  L4: __ESBMC_assume(phase+t==0 && N+phase<=0 && phase<=x
28  && 2*phase+x<=0);
29   assert( t < N ); // check loop invariant
30   __ESBMC_assume(N>=0&&phase+t==0 && N+phase<=0 && phase<=x
31   && 2*phase+x<=0 => t<=N);
32 return 0;
33 }
```

Figura 4.4: Código de exemplo para a prova por indução matemática, durante a condição adiante.

$$
\begin{aligned}
& sv_1[0] \neq cs_1 \\
& sv_1[0] \neq cs_1 \wedge sv_2[1] \neq cs_2 \\
& \ldots \\
& sv_1[0] \neq cs_1 \wedge sv_2[1] \neq cs_2 \wedge \ldots sv_k[i] \neq cs_k.
\end{aligned}
\tag{4.1}
$$

Embora $sv_k[i]$ possa ser comparada com todos cs_k, para $i < k$, pode mostrar também que a codificação na equação (4.1) é mais eficaz e leva menos tempo quando aplicada aos *benchmarks* do SV-COMP [16].

Ao final, uma instrução *assume* é inserida no repectivo *loop*, que é similar àquela inserida no caso base. As pré- e pós-condições do *loop* são mostradas na Figura 4.3 no formato

da linguagem SSA, podem ser definidas como:

$$Pre := \left[\begin{array}{l} N_1 = nondet\_uint \wedge t_1 = 0 \wedge phase_1 = 0 \wedge\ x_1 = 0 \\ \wedge\ cs_1.v_0 = nondet\_int \\ \wedge\ \ldots \\ \wedge\ cs_1.v_m = nondet\_int \end{array} \right]$$

e

$$Post := \left[\ N_1 \leq t_k \wedge x_k \leq 2 * t_k \wedge 0 \leq t_k + x_k \Rightarrow x_k \leq N_1\ \right].$$

Além do passo de inicialização da variável, nas precondições *Pre*, os valores de todas as variáveis que estão contidas no estado atual *cs* devem ser inicializados com valores não-determinísticos, onde *m* é o número de variáveis (automáticas e estáticas) que são utilizadas no programa. Dessa forma, pode-se notar que a proposta deste trabalho se concentra na redução dos espaços de estado com a intenção de eliminar ou reduzir a redundância, o que resulta em tempos de verificação mais curtos. As pós-condições não mudam, como no caso base; eles contêm somente a propriedade que o algoritmo está tentando provar.

A fim de remover estados redundantes, a variável Q foi empregada, o qual representa um conjunto de instruções adicionadas pelo ESBMC permitindo assim a validação de uma pós-condição, a fim de reduzir o tempo de verificação. As alterações são realizadas no código fonte de entrada, com o objetivo de salvar os valores das variáveis, antes e depois da iteração atual de t:

$$Q := \left[\begin{array}{l} sv[t] = cs_t \wedge S \\ \wedge\ cs_t.v_0 = v_{0t} \\ \wedge\ \ldots \\ \wedge\ cs_t.v_m = v_{mt} \end{array} \right],$$

onde $sv[t]$ é o vetor responsável por armazenar o estado atual representado por cs_t, S é o código atual dentro do *loop*, e os valores das variáveis são representados por $cs_t.v_0 = v_{0t} \wedge \ldots \wedge cs_t.v_m = v_{mt}$, na iteração t, sendo salvas no estado atual cs_t. O código modificado para o passo indutivo pode ser visto na Figura 4.5. Pode-se notar que a instrução *if* (linhas 20–49) na Figura 4.5, é copiada k vezes, de acordo com a Definição 1. O passo indutivo também insere uma instrução *assume* (line 36), como no caso base, que contém a condição de término com a intenção de provar que uma propriedade especificada por uma assertiva, é válida para qual-

quer valor de $t \geq 0$ dentro de os estados alcançáveis, até uma profundidade k. Desta forma, assume-se que o valor também será verdadeiro para o próximo valor de k.

Lemma 1 *Se a hipótese indutiva* $\{Post \wedge \neg(t < N)\}$ *for válida a iteraçã* $k+1$*, então também é verdadeira para as k iterações anteriores.*

Depois que a condição do *loop while* é finalizada, a hipótese indutiva é $\{Post \wedge \neg(t < N)\}$ é satisfeita em qualquer número de iterações; O solver SMT pode facilmente verificar o Lemma 1 e concluir que $x \leq N$ é indutivo, em relação a $t < N$.

```
1 //variables inside the loop
2 typedef struct state {
3   long long int t, x;
4   unsigned int phase;
5 } statet;
6
7 void main(){
8  int x, t, phase;
9  int N=nondet_int();
10  __ESBMC_assume(phase==0 && t==0 && x==0);
11  int phase = 0 ;
12  __ESBMC_assume( phase==0 && t==0 && x==0 );
13  assert(N>=0);
14  __ESBMC_assume( phase==0 && t==0 && x==0 );
15  //A: assign non-determinstc values
16  cs.x = nondet_uint();
17  cs.t = nondet_uint();
18  cs.phase = nondet_uint();
19  kindice = 0;
20  L1: if !(t < N) goto L4
21   //S: store current state
22   s[kindice] = cs$1;
23   __ESBMC_assume(phase+t==0 && 1<=N+phase && phase<=x && 2*phase+x<=0);
24   __ESBMC_assume(cs.phase+cs.t==0 && 1<=cs.N+phase && cs.phase<=cs.x && 2*
       cs.phase+cs.x<=0);
25   if !(cs.phase == 0) goto L2
26   __ESBMC_assume(phase==0 && t==0 && x==0 && 1<=N);
27   __ESBMC_assume(cs.phase==0 && cs.t==0 && cs.x==0 && 1<=cs.N);
28   x = x + 2;
29   x = NONDET(int);
30  ...
31   __ESBMC_assume( s$1[kindice$1] != cs$1 );
32   kindice = kindice + 1;
33   goto 1;
34  ...
35
36  L4: __ESBMC_assume ( !(t < N) );
37   __ESBMC_assume(phase+t==0 && N+phase<=0 && phase<=x && 2*phase+x<=0);
38   assert(t<=N);
39   __ESBMC_assume(N>=0 && (1<=N && cs.phase+cs.t== 0 && 1<=cs.N &&
40                   cs.phase<=cs.x && 2*cs.phase+cs.x<=0 &&
41                   (1<=N && cs.phase==0 && cs.t==0 && cs.x==0 && 1<=cs.N) &&
42                   1<=N && 0<=x && x<=2 && 2<=x && cs.phase+cs.t==0 && 1<=cs.
                       N &&
43                   cs.phase<=cs.x && 2*cs.phase+cs.x<=2 && 3*cs.phase+2<=cs.x
                        &&
44                   1<=N && 0<=x && x<=2 && 2<=x && cs.phase+cs.t==0 &&
45                   1<=cs.N && cs.phase<=cs.x && 2*cs.phase+cs.x<=2 &&
46                   3*cs.phase+2<=cs.x && 0<=N+4294967295 && -2+x<=0 && 0<=x
                       &&
47                   2<=x && cs.phase+cs.t==1 && 0<=cs.N+4294967295 && 2*cs.
                       phase+cs.x<=0 &&
48                   cs.phase+1<=cs.x && 3*cs.phase+5<=cs.x) && !(t<N) && phase
                       +t==0 && N+phase<=0 &&
49                   phase<=x && 2*phase+x<=0 => t<= N);
50
51 return 0;
52 }
```

Figura 4.5: Exemplo de prova por indução matemática durante o passo indutivo.

Capítulo 5

Avaliação Experimental

Esta seção está dividida em duas partes. A configuração do ambiente experimental na Subseção 5.1, enquanto que a Subseção 5.2 mostra a comparação entre abordagens similares com a abordagem proposta neste trabalho, ESBMC [4] utilizando apenas a indução matemática, CBMC [19] utilizando indução matemática, 2LS [21], CPAchecker [20] com indução matemática e CPAchecker [38] com indução matemática e invariantes. O método proposto neste trabalho foi implementado na ferramenta DepthK[1] que utiliza o ESBMC versão 3.0 com indução matemática, e as invariantes geradas pelo PIPS [26] e o PAGAI [14].

O principal objetivo é realizar uma comparação entre ferramentas de verificação estado-da-arte, medir a efetividade de cada ferramenta no que diz respeito à verificação de programas ANSI-C e programas para plataformas embarcadas que contenham *loops*. Avaliar a efetividade do método proposto com invariantes, a partir de um conjunto de *benchmarks* do SV-COMP [16] e aplicações embarcadas [65, 66, 67].

5.1 Configuração do Ambiente de Testes

A avaliação experimental foi conduzida em um computador com um processador Intel Core $i7-4790$ CPU @ 3.60GHz e 32 GB de memória, executando sob a plataforma Linux Ubuntu 14.04 LTS x64. Cada tarefa de verificação foi limitada ao tempo de 15 minutos e ao consumo máximo de memória de 15 GB. Além disso, a ferramenta DepthK adota o valor 100 como limite da variável *max_iterations* relacio1nada à profundidade de verificação do algoritmo de indução matemática (ver Algoritmo 1). A fim de avaliar a efetividade da metodologia

[1] https://github.com/hbgit/depthk

proposta, é necessário comparar com outras ferramentas que também implementam a técnica de indução matemática (ESBMC, CBMC e CPAchecker), para isso, *benchmarks* do SV-COMP 2017 foram utilizados [2], dado que são programas escritos no padrão ANSI-C devido ao pré-processamento. Estes *benchmarks* estão disponíveis publicamente para *download* e constituem a principal referência na área de verificação de software.

Um conjunto de *benchmarks* escritos no padrão ANSI-C e relacionados à plataforma embarcada também foram adotados. São 3 categorias: *Powerstone* [66] que é utilizada para aplicações de controle automotivo, *real-time SNU* [67] que contém um conjunto de programas para manipulação de matrizes e processamento de sinais, tais como multiplicação e decomposição de matrizes, equações de segundo grau, verificação de redundância cíclica, transformada de Fourier e codificação JPEG. Por último e, completando a terceira categoria adotada, utilizou-se a *WCET* [65] que consiste de programas para executar análise de tempo de execução no pior caso.

A eficácia das ferrementas foi realizada através da medição de resultados corretos, incorretos e situações onde a ferramenta não chegou a um resultado conclusivo e como resposta a saída foi *unknown*. O somatório adequado gerou uma pontuação para cada ferramenta, onde a ferramenta que possui o maior número de acertos é considerada mais eficaz. Os experimentos foram realizados com as seguintes versões de ferramentas:

- DepthK v2.0 com indução matemática e invariantes, através do PIPS e PAGAI, onde os parâmetros estão definidos no arquivo de execução que está disponível no repositório do DepthK no GitHub;

- ESBMC v4.0 utilizando apenas o algoritmo de indução matemática. o arquivo de execução utilizado foi do SV-COMP 2017[3];

- CBMC v5.0 com indução matemática, foi utilizado o arquivo de execução disponibilizado por Beyer *et.al.* [38];

- CPAChecker[4] (revisão 24048, adquirida diretamente do repositório SVN). Versão escolhida por implementar a indução matemática juntamente com a geração de invariantes. As opções utilizadas foram $k-induction.properties$ e $sv-comp17--invariantGeneration-$

[2] http://sv-comp.sosy-lab.org/2017/
[3] http://sv-comp.sosy-lab.org/2017/
[4] https://svn.sosy-lab.org/software/cpachecker/trunk

seq (indução matemática e invariantes) e para executar somente a indução matemática foi utilizada a opção $k - induction$;

- 2LS v0.5.0 utilizando indução indução matemática e invariantes. foi utilizado o arquivo de execução do SV-COMP 2017[5];

5.2 Resultados Experimentais

A Tabela 5.1 mostra resultados referentes aos *benchmarks* de sistemas embarcados. As seguintes colunas são utilizadas: **Ferramenta** - nome da ferramenta utilizada nos experimentos, **Resultados Corretos** - quantidade de tarefas de verificação avaliadas corretamente, **Falso Incorreto** - número de programas onde a respectiva ferramenta encontra um caminho de violação, no entanto, o mesmo não existe, **Verdadeiro Incorreto** - número de programas onde a respectiva ferramenta aponta que não há caminho de violação, **Desconhecido** - número de programas que a ferramenta não consegue chegar a um resultado conclusivo devido à falta de recursos de computação, falha na ferramenta ou pelo limite de tempo de verificação não ser o suficiente (15 minutos) e **Tempo** - tempo de execução em minutos para verificar a totalidade conjunto de *benchmarks*.

Os resultados para todas as ferramentas utilizadas nos experimentos, em relação aos *benchmarks* do SV-COMP 2017, são mostrados na Tabela 5.2. As seguintes colunas são utilizadas: **Categoria** - nome da categoria no SV-COMP, **Ferramenta** - nome da ferramenta utilizada nos experimentos, **Verdadeiro Correto** - número de programas onde a respectiva ferramenta confirmou a inexistência do caminho de violação, **Falso Correto** - número de programas onde o a respectiva ferramenta encontrou corretamente o caminho de violação da propriedade, **Verdadeiro Incorreto** - número de programas onde a respectiva ferramenta não identifica o caminho de execução que leva à falha e informa ao usuário que o mesmo não existe, **Falso Incorreto** - número de programas onde a respectiva ferramenta encontra um caminho de violação, no entanto, o mesmo não existe.

Quanto à Tabela 5.1, é possível realizar as seguintes observações e comentários:

[5]https://sv-comp.sosy-lab.org/2017/

Ferramenta	DepthK + PIPS	DepthK + PAGAI	ESBMC + Ind. Mat.	CPAchecker + Ind. Mat.	CPAchecker + Ind. Mat. + Invariantes	CBMC + Ind. Mat.	2LS
Resultados Corretos	16	14	29	27	27	15	34
Falso Incorreto	0	0	0	0	0	0	0
Verdadeiro Incorreto	0	0	0	0	0	0	0
Desconhecido	18	20	5	7	7	19	0
Tempo(min)	55.51	56.13	54.18	1.8	1.95	286.06	10.6

Tabela 5.1: Resultados experimentais das categorias Powerstone, SNU e WCET.

- A ferramenta que mais resolveu tarefas de verificação com sucesso foi 2LS, a ferramenta foi capaz de devolver resposta satisfatória em todas as tarefas de verificação. A diferença para a abordagem proposta é forma de geração de invariantes, o método proposto gera invariantes antes do início da verificação enquanto a ferramenta 2LS gera invariantes conforme a exploração de espaço de estados aumenta.

- Os resultados apresentados pelo método proposto utilizando a ferramenta PIPS foram inferiores aos apresentados pelo ESBMC e CPAchecker, no entanto, o método proposto superou a ferramenta CBMC. Embora não existam falhas no processo de verificação, muitos resultados inconclusivos foram apresentados, a razão é que ambas as ferramentas de geração de invariantes (PIPS e PAGAI) não são preparadas para análise de código referente à plataforma embarcada que apresenta maior complexidade.

- Os resultados apresentados pelas duas abordagens do método proposto estão diretamente relacionados diretamente relacionados a maturidade de cada ferramenta de geração de invariantes. É possível melhorar os resultados apresentados pela ferramenta PAGAI pelo fato de a mesma possuir uma ampla variedade de configurações que podem ser exploradas pelo usuário. A ferramenta PAGAI não permite configuração por parte do usuário e esta é a principal dificuldade em gerar invariantes mais fortes para os *benchmarks* relacionados a sistemas embarcados.

- A ferramenta ESBMC v4.0 recebeu melhorias significativas no algoritmo de indução matemática onde falhas foram corrigidas, melhorias na geração de guarda e correção de

memory leaks. Como resultado, a técnica de indução matemática foi suficiente para superar o método proposto. O tempo de execução foi inferior ao do método proposto devido a implementação das melhorias citadas e pela não utilização de invariantes externas.

- A ferramenta CPAchecker utilizando apenas o algoritmo de indução matemática foi um pouco inferior ao ESBMC também com indução matemática; o algoritmo de geração de invariantes do CPAchecker trabalha em segundo plano e por este motivo o tempo de execução das duas abordagens desta ferramenta são similares e inferiores àqueles apresentados pelo ESBMC que executa suas funções sequencialmente sem utilizar nenhum algoritmo adicional em *background*.

- A ferramenta CBMC utilizando indução matemática foi a mais lenta devido a complexidade das tarefas de verificação e pelo fato desta funcionalidade ser experimental, além disso, o algoritmo de indução matemática não mais recebe melhorias e atualizações por parte da equipe de desenvolvimento.

Para avaliar a eficácia do método proposto, os *benchmarks* do SVCOMP'17 foram escolhidos devido à sua alta cobertura, enorme quantidade e diversidade de casos de teste a serem resolvidos, além de ser um dos principais indicadores de área de verificação de software.

Quanto à Tabela 5.2, é possível realizar as seguintes observações e comentários por categoria:

- **ReachSafety**:
  - As duas abordagens do método proposto têm o menor número de respostas corretas porque as invariantes geradas aumentaram o tempo de verificação e o consumo de memória, portanto, o ESBMC não chegou a um resultado conclusivo em tempo hábil ou excedeu a quantidade de memória permitida (15GB).
  - A ferramenta CPAchecker com indução matemática e invariantes tem um número significativo de casos de teste verificados com sucesso. Ao contrário das invariantes geradas pelo método proposto, a geração invariantes do algoritmo CPAchecker não foi capaz de gerar valores confiáveis e resultaram em uma expressiva quantidade de resultados incorretos.

Categoria	Ferramenta	Verdadeiro Correto	Falso Correto	Verdadeiro Incorreto	Falso Incorreto
ReachSafety	DepthK + PIPS	700	350	0	1
	DepthK + PAGAI	650	330	0	2
	ESBMC + Ind. Mat.	550	685	1	0
	CPAchecker + Ind. Mat.	906	325	0	5
	CPAchecker + Ind. Mat. + Inv.	850	300	7	10
	2LS	846	543	9	27
	CBMC	311	733	0	0
MemSafety	DepthK + PIPS	155	49	0	7
	DepthK + PAGAI	129	45	0	15
	ESBMC + Ind. Mat.	153	81	4	8
	CPAchecker + Ind. Mat.	0	0	0	0
	CPAchecker + Ind. Mat. + Inv.	0	0	0	0
	2LS	65	66	3	68
	CBMC	193	100	0	0
ConcurrencySafety	DepthK + PIPS	600	584	1	0
	DepthK + PAGAI	532	500	2	1
	ESBMC + Ind. Mat.	654	495	0	0
	CPAchecker + Ind. Mat.	0	0	0	0
	CPAchecker + Ind. Mat. +Inv.	0	0	0	0

	2LS	0	0	0	0
	CBMC	172	791	0	0
Overflows	DepthK + PIPS	85	170	0	0
	DepthK + PAGAI	90	154	0	1
	ESBMC + Ind. Mat.	304	21	0	0
	CPAchecker + Ind. Mat.	25	143	0	9
	CPAchecker + Ind. Mat. + Inv.	24	140	0	10
	2LS	95	138	0	0
	CBMC	32	156	0	0
Termination	DepthK + PIPS	365	0	5	0
	DepthK + PAGAI	300	0	5	0
	ESBMC + Ind. Mat.	0	0	0	0
	CPAchecker + Ind. Mat.	0	0	0	0
	CPAchecker + Ind. Mat. + Inv.	0	0	0	0
	2LS	580	347	4	30
	CBMC	0	0	0	0
SoftwareSystems	DepthK + PIPS	420	10	0	0
	DepthK + PAGAI	394	11	0	2
	ESBMC + Ind. Mat.	700	0	0	0

	CPAchecker + Ind. Mat.	1062	15	0	0
	CPAchecker + Ind. Mat. +Inv.	150	17	0	0
	2LS	1081	16	5	0
	CBMC	26	10	1	0
Total	DepthK + PIPS	2325	1163	6	8
	DepthK + PAGAI	2095	1017	7	21
	ESBMC + Ind. Mat.	2361	1029	5	8
	CPAchecker + Ind. Mat.	1993	483	0	14
	CPAchecker + Ind. Mat. +Inv.	1024	457	7	20
	2LS	2667	1110	21	125
	CBMC	734	1756	1	0

Tabela 5.2: Resultados Experimentais do SV-COMP 17.

  - A ferramenta 2LS foi a ferramenta que verificou corretamente o maior número de casos de teste, isso mostra a evolução do algoritmo de verificação desta ferramenta, no entanto, o algoritmo de geração invariantes é deficiente e guiou a ferramenta para a maior quantidade de resultados incorretos porque não foi possível avaliar adequadamente os casos de teste.

- **Memory Safety**:

  - A ferramenta ESBMC possui suporte total a esta categoria que não é suportada pelo CPAchecker. O algoritmo de indução matemática padrão do ESBMC foi capaz de verificar com sucesso uma grande variedade de casos de teste, no entanto, o número de resultados incorretos, embora relativamente baixo, é o principal problema. O método proposto usando o PIPS foi capaz de minimizar esse problema reduzindo-o significativamente, já que esta ferramenta geradora de invariante possui recursos voltados a análise de propriedades de segurança da memória.
  - A ferramenta 2LS possui pior desempenho porque o algoritmo de indução matemática foi induzido ao erro devido invariantes fracas que não levaram em conta a natureza dessa categoria e suas propriedades de segurança.
  - A ferramenta CBMC resolveu a maioria das tarefas de verificação corretamente devido a melhorias implementadas especificamente para esta categoria .

- **ConcurrencySafety**:

  - As ferramentas CPAchecker e 2LS não possuem suporte nativo para casos de teste que usam *multi-thread* e concorrência.
  - A ferramenta ESBMC possui suporte total a *multi-thread* e concorrência. O algoritmo de indução matemática padrão é capaz de resolver uma ampla gama de tarefas de verificação sem o auxílio de invariantes externas, no entanto, o método proposto usando PIPS foi capaz de aumentar a quantidade de casos de teste verificados com sucesso devido ao suporte oferecido por esta ferramenta a este tipo de caso de teste. A ferramenta PAGAI tem suporte limitado a *multi-thread* e concorrência, por isso o número de casos de teste verificados com sucesso foi menor e a quantidade de resultados incorretos foi ligeiramente superior. Ambas as abordagens do método

proposto e o ESBMC padrão foram capazes de superar o CBMC que também possui suporte a esta categoria.

- **Overflows**:
  - As duas abordagens do método proposto guiaram o ESBMC para um alto número de resultados não conclusivos porque as invariantes restringem os espaços de estados a serem verificados e os casos de teste desta categoria estão relacionados a *overflows* em tempo que poderão ocorrer em tempo de execução, assim, as invariantes adicionadas aos casos de teste limitou os espaços de estados a serem verificados pelo ESBMC.
  - A ferramenta ESBMC usando o algoritmo de indução matemática padrão foi a ferramenta que resolveu maior parte das tarefas de verificação com sucesso porque o algoritmo de redução de espaço de estados padrão não utiliza invariantes externas, este algoritmo tem evoluído constantemente e foi aplicado com sucesso em casos de teste desta categoria.
  - A ferramenta 2LS que também utiliza invariantes indutivas sofreu o mesmo problema de limitar muitos estados a serem verificados, o número de casos de teste verificados com sucesso foi inferior ao do método proposto.
  - A ferramenta CBMC possui a menor quantidade de casos de teste verificados com sucesso uma vez que o algoritmo de redução de espaço de estado não é tão eficaz quanto ao implementado no ESBMC.
- **Termination**:
  - As ferramentas CPAchecker e CBMC não possuem suporte a esta categoria, enquanto o ESBMC tem suporte parcial que resulta em muitos erros de verificação. Durante o SVCOMP, esta categoria é ignorada pelo ESBMC para evitar uma quantidade excessiva de resultados incorretos que impactam diretamete na pontuação geral.
  - Uma das principais contribuições deste trabalho foi a geração de invariantes suficientemente fortes que guiaram o ESBMC para o resultado correto em uma ampla gama de casos de teste. Nesta categoria, o verificador deve avaliar se o caso de

teste possui execução finita, devido esta característica as invariantes tiveram forte impacto e foram importantes para eliminar os estados que normalmente induzem o ESBMC ao resultado incorreto.

- A ferramenta 2LS foi a ferramenta que mais verificou casos de teste com sucesso, no entanto, o problema da geração de invariantes fracas que foi observado em outras categorias ocorreu novamente e o alto número de falhas reduz drasticamente a confiança nos resultado fornecidos por esta ferramenta.

- **SoftwareSystems**:

  - Enquanto a ferramenta CPAchecker utilizando apenas indução matemática foi superior, a versão desta ferramenta que adiciona invariantes gerou muitos resultados inconclusivos que resultaram em uma quantidade muito baixa de tarefas de verificação analisadas com sucesso.
  - O método proposto superou a versão do CPAchecker com invariantes e CBMC porque suporta estruturas com ponteiros e considera variáveis deste tipo na análise estática e, conseqüentemente, na geração de invariantes.

As Figuras 5.1 e 5.2 foram criadas com base na pontuação do SV-COMP, incluindo resultados comparativos para a sub-categoria *loops* do SV-COMP e o comparativo para os *benchmarks* de sistemas embarcados. Pode-se notar que para os *benchmarks* de sistemas embarcados, foram utilizados programas seguros [4] onde o objetivo principal é provar a segurança das propriedades, o método proposto utilizou valores constantes a frim de produzir invariantes indutivas que condições que permanecem verdadeiras ao longo de todo o programa a fim de provar propriedades.

É notório que, a abordagem DepthK + PAGAI obteve uma pontuação mais baixa quando submetida aos *benchmarks* de sistemas embarcados e a categoria *Loops* do SV-COMP (Figura 5.1 e Figura 5.2). Isto, em virtude do PAGAI não gerar invariantes suficientemente fortes e capazes de guiar o ESBMC a alcançar um resultado *true* ou *falso* . De fato, o PAGAI é uma ferramenta relativamente nova em comparação ao PIPS, a mesma está em desenvolvimento, principalmente no que diz respeito à previsão e geração de invariantes onde há *bugs* que necessitam ser corrigidos. Estes *bugs* ficam evidentes quando o programa de entrada possui muitas linhas código, *loops* e funções recursivas.

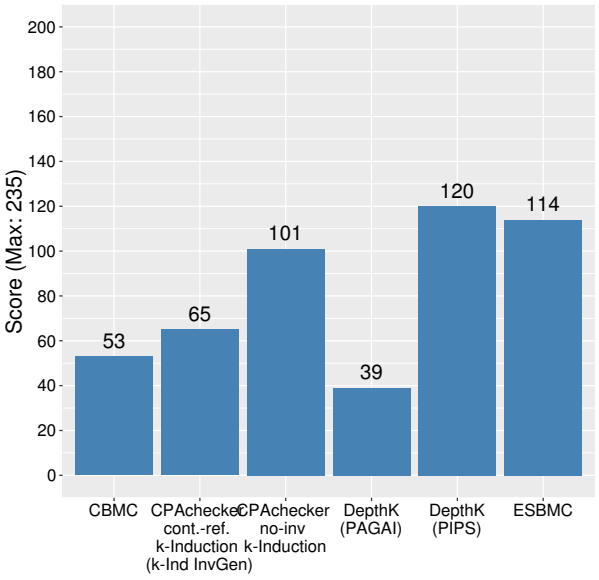

Figura 5.1: Pontuação da subcategoria *Loops*

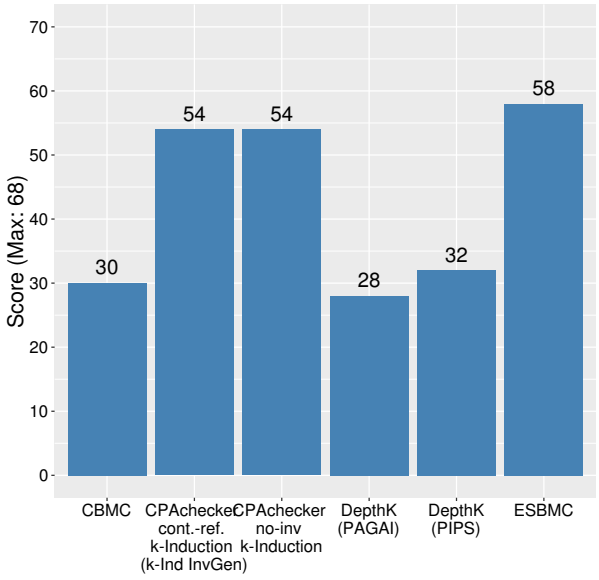

Figura 5.2: Pontuação de Sistemas Embarcados

Ainda referente aos *benchmarks* de sistemas embarcados, o CPAchecker utilizando indução matemática ao mesmo tempo que foi melhor que o CPAchecker utilizando indução matemática e invariantes (que é uma abordagem mais recente e ainda em desenvolvimento), mostrou que o algoritmo de indução matemática é tão maduro quanto do ESBMC, a quantidade de resultados corretos é similar. O principal problema com as duas abordagens desta ferramenta é a quantidade de erros que afetam diretamente a pontuação final.

CBMC utilizando indução matemática foi a ferramenta que gerou mais inconclusivos inconclusivos na categoria *Loops* e o tempo de verificação é visivelmente superior a todas as ferramentas utilizadas na execução dos experimentos; No entanto, pode-se apontar que o número de erros não é tão diferente de outras abordagens. O CBMC é uma ferramenta em constante evolução , apesar de o algoritmo de induçao matemática estar em fase inicial, a técnica BMC é muito aplicada por esta ferramenta.

Para medir o impacto da aplicação de invariantes em esquemas de verificação baseados na indução matemática, a distribuição dos resultados de DepthK + PIPS / PAGAI e ESBMC foi classificada por etapa de verificação (caso base, condição adiante e passo indutivo). Nesta análise, apenas os resultados relacionados ao DepthK e ESBMC foram avaliados, pois são parte da abordagem proposta e não é possível identificar as etapas do algoritmo de indução matemática nas outras ferramentas.

A Figura 5.3 mostra a distribuição dos resultados, para cada etapa de verificação, sobre

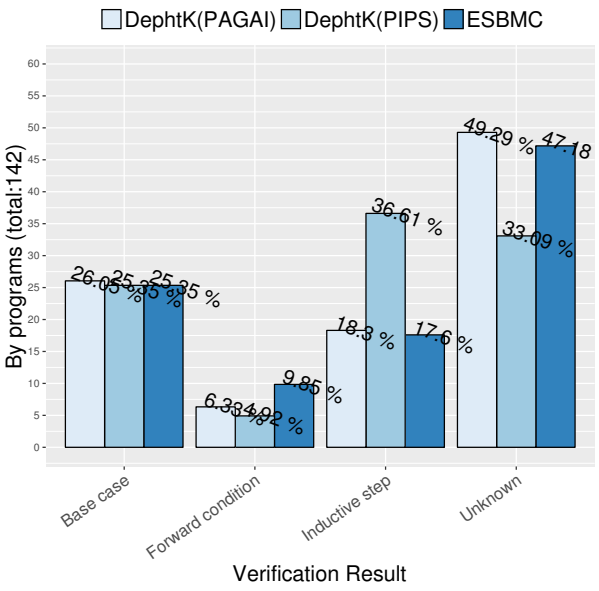

Figura 5.3: Categoria *Loops*.

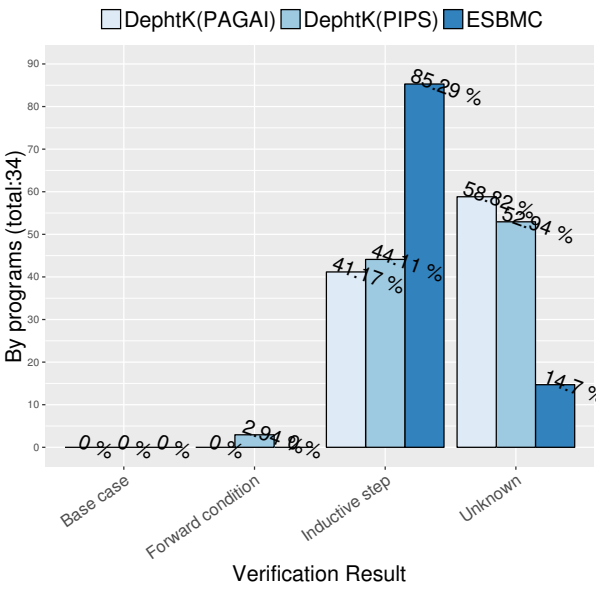

Figura 5.4: Sistemas Embarcados.

a subcategoria loops SVCOMP, enquanto a Figura 5.4 apresenta resultados para os *benchmarks* de sistemas embarcados.

A distribuição dos resultados nas Figuras 5.3 e 5.4 mostra que, durante a etapa indutiva na subcategoria *Loops* e sistemas embarcados:

Na categoria *Loops*, invariantes geradas pelo PIPS ajudaram o algoritmo de indução matemática a aumentar a quantidade de resultados corretos, justamente na categoria onde o ESBMC possui dificuldades pois, nesta categoria, há *benchmarks* onde o k pode crescer indefinidamente.

Em sistemas embarcados, as invariantes geradas por PIPS e PAGAI não foram capazes de limitar corretamente o espaço de estado a ser explorado pelo algoritmo de indução matemática do ESBMC, aumentando o tempo de verificação e, assim, aumentando consideravelmente o número de resultados inconclusivos.

O resultado zerado do caso base deu-se porque o espaço de estado da aplicação é reduzido em comparação com os programas convencionais, assim o algoritmo de indução matemática padrão se beneficia, conforme o limite k aumenta, o ESBMC gera invariantes indutivas mais fortes podendo atingir um resultado conclusivo na condição adiante ou passo indutivo.

Estes resultados levam à conclusão de que os invariantes ajudaram o algoritmo de indução matemática do ESBMC a desenrolar suficientemente os *loops* quando k cresce indefinidamente, e assim, chegar a um resultado conclusivo sem a necessidade de desenrolar todas as pos-

sibilidades. Também identificamos que o DepthK + PIPS e DepthK + PAGAI não encontraram uma solução (`Unknown` e `Timeout` em 33.09% e 49.29% dos casos na categoria *Loops* (ver Figura 5.3). Nos *benchmarks* de sistemas embarcados, o DepthK (PIPS) não encontrou uma solução em 52.94% e DepthK (PAGAI) em 58.82% (ver Figura 5.4). Isto se deve a explicação das invariantes geradas pelo PIPS e PAGAI não se apresentarem fortemente suficientes para guiar o algoritmo de indução matemática.

Na Tabela 5.1, identificamos que o tempo de verificação de DepthK (PIPS e PAGAI) é apenas mais rápido do que CBMC, como mostrado na Figura 5.5. Isto, porque o tempo gasto pelas ferramentas na geração de invariantes pode variar significativamente dependendo da complexidade do código, tamanho, quantidade de funções recursivas, *loops* e etc.

Acreditamos que o tempo de verificação de nossa ferramenta pode ser significativamente melhorado em duas direções: Corrigir erros relacionados à implementação da ferramenta, porque alguns resultados gerados como `Unknown` estão relacionados a falhas no DepthK, como por exemplo, o algoritmo responsável para identificar as funções e variáveis no código original pode ser otimizado e aprimorado. Realizar ajustes nos parâmetros do script PIPS responsável por gerar invariantes, uma vez que o PIPS possui um amplo conjunto de comandos para a transformação de código, o que pode ter um impacto positivo.

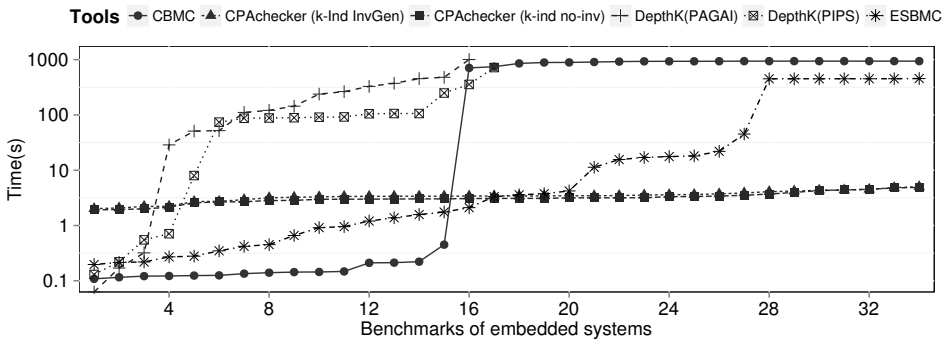

Figura 5.5: Tempo de verificação da categoria de sistemas embarcados.

5.3 Resumo

Neste capítulo, foi apresentada inicialmente a ferramenta de geração de invariantes PIPS [26], a forma de geração análise de código executada por esta ferramenta. Bem como, detalhado o algoritmo que transforma as invariantes geradas em assertivas compatíveis com

a linguagem ANSI-C e exemplificado o código com as invariantes geradas pelo PIPS. Em seguida, a ferramenta PAGAI [14] é apresentada juntamente com a explicação do código responsável pela integração desta ferramenta ao método proposto, o ponto fraco desta ferramenta é que *benchmarks* que possuem muitas linhas de código e funções recursivas normalmente travam a ferramenta. Ambas as ferramentas aplicam transformações no código original, estas transformações são úteis para identificar as invariantes geradas por cada uma das ferramentas.

Após a contextualização das ferramentas de geração de invariantes utilizadas neste trabalho, a próxima seção é destinada a mostrar detalhes da aplicação da geração de invariantes combinada ao algoritmo de indução matemática. Por fim, foi mostrado o resultado da verificação dos *benchmarks* do SV-COMP, utilizando a ferramenta ESBMC em conjunto com as ferramentas responsáveis por gerar as invariantes. Finalmente, os testes mostraram que a combinação de invariantes ao algoritmo de indução matemática é uma alternativa viável, uma vez que foi possível provar a corretude de 3.488 *benchmarks* com PIPS e 3.112 *benchmarks* com PAGAI, ambos relacionados ao SV-COMP. Em relação a sistemas embarcados, foi possível provar a corretude de 16 *benchmarks* com PIPS e 14 com PAGAI.

Capítulo 6

Conclusões

Neste trabalho, descrevemos e avaliamos uma abordagem de verificação baseada em indução matemática, que adota a abstração poliédrica do comportamento do programa para inferir invariantes indutivas. A abordagem proposta, implementada no DepthK [13], é aplicada à verificação automática de propriedades de alcançabilidade a partir de *benchmarks* do SV-COMP [32] que é um dos principais indicadores de vivacidade da área de Verificação de Sistemas, além de *benchmarks* de sistemas embarcados.

Em particular, foram avaliados 8.908 *benchmarks* do SV-COMP 2017 e 34 programas ANSI-C de aplicações de sistemas embarcados. Além disso, foi descrita a comparação entre DepthK (*i.e.*, adotando ferramentas de geração invariantes PIPS e PAGAI) e verificadores ESBMC, CBMC, 2LS e CPAChecker, com indução matemática e invariantes.

A técnica de indução matemática do ESBMC acrescida das invariantes geradas pelo PIPS foi capaz de determinar resultados tão precisos quanto àqueles obtidos com a indução matemática sem invariantes. Além disso, a combinação da indução matemática com as invariantes inferidas pelo PAGAI, determinou menos resultados precisos do que o obtido com o método padrão de indução matemática do ESBMC.

As configuracoes utilizadas no PIPS e no PAGAI (foi utilizado o domínio poliedral em ambas as ferramentas), tanto para sistemas embarcados quanto para o SVCOMP, não sofreram alterações e isso nos mostra que nossa abordagem não é suficientemente abrangente. Melhorias na geração de invariantes devem ser feitas dependendo do tipo de aplicação que será verificada. As melhorias vão desde refinamento das configuracoes do PIPS e PAGAI até modificar o DepthK para lidar com problemas distintos de forma a identificar qual a melhor abordagem a ser

utilizada para cada *benchmark*. Desta forma, a ferramenta seria capaz de lidar com uma série de problemas de verificação adotando estratégias específicas tanto na geração de invariantes quanto decidindo se a indução matemática seria ou não a melhor abordagem a ser utilizada.

Em relação aos *benchmarks* do SVCOMP, os resultados apresentados pelo método proposto (tanto com PAGAI quanto com PIPS) foram superiores àqueles apresentados pela ferramenta 2LS que possui o mesmo propósito e está em constante evolução, superando também a ferramenta CPAchecker. Este resultado é encorajador pois mostra que o método descrito neste trabalho é promissor e confiável.

Para os *benchmarks* de sistemas embarcados, falhas no DepthK foram identificadas, e estão relacionadas a defeitos na execução da ferramenta e a necessidade de ajustes na geração invariantes. Baseando-se no fato de que resultados abaixo da expectativa foram obtidos, quando comparados com outras ferramentas de verificação, em que se tem a superação do método sobre o CBMC, mostra que o trabalho com a geração de invariantes está apenas no começo e pode, se melhorado da forma adequada, obter resultados promissores.

Ainda assim, é importante salientar que com as invariantes geradas pelo método proposto (PIPS e PAGAI) é possível provar ou refutar uma grande variedade de propriedades de segurança. Entretanto, o algoritmo de indução matemática, adotando invariantes geradas pelo PIPS resolve mais tarefas de verificação.

Capítulo 7

Trabalhos Futuros

Para trabalhos futuros, no que diz respeito à geração de invariantes, é possível expandir o método proposto de tal forma que possibilite o desenvolvimento uma abordagem híbrida, que combina PIPS e PAGAI, *i.e.*, uma abordagem que combina invariantes produzidos por ambas as ferramentas. Como resultado, pode-se esperar a geração de invariantes que sejam indutivas e também o aumento da eficácia do algoritmo de indução matemática utilizando estas invariantes combinadas onde o principal desafio será combiná-las de forma que a verificação não seja prejudicada por excesso de restrição de espaço de estados.

Além disso, a integração de outras ferramentas geradoras de invariantes como a Parma Polyhedra Library (PPL) que é a base de outras ferramentas de verificação que utilizam invariantes, como o CPAchecker, fornecendo abstrações númericas especialmente voltadas para verificação de sistemas complexos [68].

Dominar a geração de invariantes para plataformas embarcadas pode possibilitar a combinação da mesma com modernas técnicas de aprendizado de máquina voltada a redução de espaço de estados, facilitar a verificação de sistemas desenvolvidos em novas tecnologias e etc [69]. Esta combinação pode ser útil em uma ampla gama de aplicações, uma vez que o desenvolvimento de *software*, em muitos casos, não acompanha a evolução do *hardware* e código legado pode ser reutilizado inúmeras vezes. A combinação destas duas técnicas pode tornar menos dispendioso testar um software que sofreu pouca ou nenhuma alteração ao longo do seu tempo de vida útil.

Finalmente, pode-se acrescentar suporte para a geração de invariantes a fim de verificar propriedades expressas em Lógica Temporal Linear (do inglês, *Linear Temporal Logic* -

LTL). Tais propriedades são úteis em especificar o comportamento de um sistema através de propriedades de segurança e vivacidade [70, 71].

Referências Bibliográficas

[1] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. F. Siok. Recent catastrophic accidents: Investigating how software was responsible. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 14–22, June 2010.

[2] Armin Biere. Bounded Model Checking. In *Handbook of Satisfiability*, pages 457–481. IOS Press, 2009.

[3] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885. IOS Press, 2009.

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

[5] 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*, volume 2988, pages 168–176, Lancaster, Reino Unido, 2004. Springer.

[6] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 146–161. Springer-Verlag, 2012.

[7] Franjo Ivancic, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai. Model Checking C Programs Using F-SOFT. In *23rd International Conference on Computer Design (ICCD)*, pages 297–308. IEEE Computer Society, 2005.

[8] Mikhail Y. Gadelha, Hussama I. Ismail, and Lucas C. Cordeiro. Handling loops in bounded model checking of c programs via k-induction. *Int. J. Softw. Tools Technol. Transf.*, 19(1):97–114, February 2017.

[9] EÉN, Niklas, SÖRENSSON, Niklas. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003.

[10] Daniel Große, Hoang M. Le, and Rolf Drechsler. Induction-Based Formal Verification of SystemC TLM Designs. In *10th International Workshop on Microprocessor Test and Verification (MTV)*, pages 101–106, 2009.

[11] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 108–125, 2000.

[12] Shikun Chen, Zhoujun Li, Xiaoyu Song, and Mengjun Li. An iterative method for generating loop invariants. In *Proceedings of the 5th Joint International Frontiers in Algorithmics, and 7th International Conference on Algorithmic Aspects in Information and Management*, FAW-AAIMÍ1, pages 264–274, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Herbert Rocha, Hussama Ismail, Lucas Cordeiro, and Raimundo Barreto. Model checking embedded c software using k-induction and invariants. In *V Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–6, 2015.

[14] Julien Henry, David Monniaux, and Matthieu Moy. Pagai: A path sensitive static analyser. *Electron. Notes Theor. Comput. Sci.*, pages 15–25, 2012.

[15] Herbert Rocha, Raimundo S. Barreto, Lucas C. Cordeiro, and Arilo Dias Neto. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, pages 128–142, 2012.

[16] Dirk Beyer. Software Verification and Verifiable Witnesses - (Report on SV-COMP 2015). In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 401–416, 2015.

[17] Williame. Rocha, Oliveira Herbert. Rocha, Lucas C Cordeiro, and Bernd. Fischer. Depthk: A k-induction verifier based on invariant inference for c programs. *In 23th Intl. Conf. on Tools and Algorithms for the Construction and Analysis for Systems (TACAS)*, pages 1–4, 2017.

[18] Robert Kowalski and Fariba Sadri. *A Logic-Based Framework for Reactive Systems*, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[19] Daniel Kroening and Michael Tautschnig. CBMC - C Bounded Model Checker - (Competition Contribution). In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 389–391, 2014.

[20] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Conference on Computer-Aided Verification (CAV)*, pages 184–190, 2011.

[21] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety verification and refutation by k-invariants and k-induction (extended version). *CoRR*, abs/1506.05671, 2015.

[22]

[23] Renato B. Abreu, Mikhail Y. R. Gadelha, Lucas C. Cordeiro, Eddie B. de Lima Filho, and Waldir S. da Silva. Bounded model checking for fixed-point digital filters. *Journal of the Brazilian Computer Society*, 22(1):1, 2016.

[24] Iury V. Bessa, Hussama I. Ismail, Lucas C. Cordeiro, and JoÃ£o E. Filho. Verification of fixed-point digital controllers using direct and delta forms realizations. *Des. Autom. Embedded Syst.*, 20(2):95–126, June 2016.

[25] Hussama I. Ismail, Iury V. Bessa, Lucas C. Cordeiro, Eddie B. de Lima Filho, and João E. Chaves Filho. *DSVerifier: A Bounded Model Checking Tool for Digital Systems*, pages 126–131. Springer International Publishing, Cham, 2015.

[26] Mines ParisTech. PIPS: Automatic Parallelizer and Code Transformation Framework. Available at http://pips4u.org, 2013.

[27] Mikhail Gadelha, Hussama Ismail, and Lucas Cordeiro. Handling loops in bounded model checking of c programs via k-induction. In *International Journal on Software Tools for Technology Transfer (STTT)*, 2015.

[28] Lucas Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer. *Context-Bounded Model Checking with ESBMC 1.17*, pages 534–537. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[29] 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*, pages 619–622, Roma, Itália, 2013. Springer.

[30] 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*, pages 619–622, Grenoble, FranÃ§a, 2014. Springer.

[31] Dirk Beyer. 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*, volume 7795, pages 594–609, Roma, ItÃ¡lia, 2013. Springer.

[32] Dirk Beyer. Software Verification and Verifiable Witnesses (Report on SV-COMP 2015). In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2015.

[33] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. *Ultimate Automizer with SMTInterpol*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[34] Temesghen Kahsai and Cesare Tinelli. Pkind: A parallel k-induction based model checker. In *Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*, pages 55–62, 2011.

[35] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-induction with continuously-refined invariants. In *27th International Conference on Computer Aided Verification (CAV)*, pages 622–640, 2015.

[36] Kumar Madhukar, Björn Wachter, Daniel Kroening, Matt Lewis, and Mandayam K. Srivas. Accelerating invariant generation. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 105–111, 2015.

[37] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *CAV*, pages 184–190. Springer-Verlag, 2011.

[38] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-Induction with Continuously-Refined Invariants. Available at http://www.sosy-lab.org/d̃beyer/cpa-k-induction/, 2015.

[39] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

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

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

[42] Phillipe Pereira, Higo Albuquerque, Hendrio Marques, Isabela Silva, Celso Carvalho, Lucas Cordeiro, Vanessa Santos, and Ricardo Ferreira. Verifying cuda programs using smt-based context-bounded model checking. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1648–1653, New York, NY, USA, 2016. ACM.

[43] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability modulo theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. 1 edition, 2009.

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

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

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

[47] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[48] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *« Informatics — 10 Years Back, 10 Years Ahead »*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[49] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[50] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[51] P. Cousot and R. Cousot. *A gentle introduction to formal verification of computer systems by abstract interpretation*. NATO Science Series III: Computer and Systems Sciences.

[52] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1):47 – 103, 2002.

[53] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[54] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[55] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[56] Jérôme Feret. Numerical abstract domains for digital filters.

[57] Jérôme Feret. *The Arithmetic-Geometric Progression Abstract Domain*, pages 42–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[58] DONALDSON, Alastair, KROENING, Daniel, RUMMER, 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*, volume 6015, pages 280–295, Paphos, Chipre, 2010. Springer.

[59] DONALDSON, Alastair, HALLER, Leopold, KROENING, Daniel, RUMMER, Phillip. Software Verification using k-Induction. In *Static Analysis*, pages 351–268, Veneza, ItÃ¡lia, 2011. Springer.

[60] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *Proceedings of the 23rd International Conference on Computer Aided verification (CAV)*, pages 703–719. Springer-Verlag, 2011.

[61] Corinne Ancourt, Fabien Coelho, and François Irigoin. A Modular Static Analysis Approach to Affine Loop Invariants Detection. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 3–16. Elsevier Science Publishers B. V., 2010.

[62] George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 109–117. IEEE, 2008.

[63] Vivien Maisonneuve, Olivier Hermant, and François Irigoin. Computing Invariants with Transformers: Experimental Scalability and Accuracy. In *5th International Workshop on Numerical and Symbolic Abstract Domains (NSAD), Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 17–31. Elsevier, 2014.

[64] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. Available at http://llvm.cs.uiuc.edu.

[65] MRTC. WCET Benchmarks. Mälardalen Real-Time Research Center. Available at http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2012.

[66] Jeff Scott, Lea Hwang Lee, John Arends, and Bill Moyer. Designing the Low-Power M*CORE Architecture. In *Power Driven Microarchitecture Workshop*, pages 145–150, 1998.

[67] SNU. SNU Real-Time Benchmarks. Available at http://www.cprover.org/goto-cc/examples/snu.html, 2012.

[68] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[69] L. Cordeiro. Automated Verification and Synthesis of Embedded Systems using Machine Learning. *ArXiv e-prints*, February 2017.

[70] Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Model checking ltl properties over ANSI-C programs with bounded traces. *Softw. Syst. Model.*, 14(1):65–81, February 2015.

[71] Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Context-bounded model checking of ltl properties for ANSI-C software. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, SEFM'11, pages 302–317, Berlin, Heidelberg, 2011. Springer-Verlag.

Apêndice A

Publicações

A.1 Referente à Pesquisa

- **Rocha, W.**, Rocha, H., Ismail, H., Cordeiro, L., Fischer, B. **DepthK: A k-Induction Verifier Based on Invariant Inference for C Programs.** In 23th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 1-4, 2017