UNIVERSIDADE FEDERAL DO AMAZONAS - UFAM
INSTITUTO DE COMPUTAÇÃO - ICOMP
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA - PPGI

Verificação de Sistemas de *Software* baseada em Transformações de Código usando *Bounded Model Checking*

por

Herbert Oliveira Rocha

Manaus - Amazonas

03 de julho de 2015

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO AMAZONAS - UFAM
INSTITUTO DE COMPUTAÇÃO - ICOMP
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA - PPGI

Verificação de Sistemas de *Software* baseada em Transformações de Código usando *Bounded Model Checking*

por

Herbert Oliveira Rocha

Tese de doutorado apresentada ao Programa de Pós-Graduação em Informática, da Universidade Federal do Amazonas, como parte dos requisitos necessários à obtenção do título de Doutor em Informática, na área de concentração em Sistemas Embarcados e Engenharia de Software.

Orientador: Raimundo da Silva Barreto, D.Sc.

Manaus - Amazonas
03 de julho de 2015

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Oliveira Rocha, Herbert
O48v Verificação de Sistemas de Software baseada em Transformações de Código usando Bounded Model Checking / Herbert Oliveira Rocha. 2015
177 f.: il. color; 29,7 cm.

Orientador: Raimundo da Silva Barreto
Orientador: Lucas Carvalho Cordeiro
Tese (Doutorado em Informática) - Universidade Federal do Amazonas.

1. Verificação e Teste de Software. 2. Model Checking. 3. Criação de Casos de Teste. 4. Invariantes de Programas. I. Barreto, Raimundo da Silva II. Universidade Federal do Amazonas III. Título

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO AMAZONAS - UFAM
INSTITUTO DE COMPUTAÇÃO - ICOMP
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA - PPGI

TERMO DE APROVAÇÃO

Verificação de Sistemas de *Software* baseada em Transformações de Código usando *Bounded Model Checking*

por

Herbert Oliveira Rocha

Esta Tese foi julgada adequada para obtenção do Título de Doutor em Informática e aprovada em sua forma final pelo Programa de Pós-Graduação em Informática da Universidade Federal do Amazonas.

Aprovada em: **03/07/2015**

BANCA EXAMINADORA

Raimundo da Silva Barreto, D.Sc.(Orientador)
Universidade Federal do Amazonas - UFAM

Lucas Carvalho Cordeiro, D.Sc.(Coorientador)
Universidade Federal do Amazonas - UFAM

Arilo Claudio Dias Neto, D.Sc.
Universidade Federal do Amazonas - UFAM

Djones Vinicius Lettnin, D.Sc.
Universidade Federal de Santa Catarina - UFSC

Leandro Buss Becker, D.Sc.
Universidade Federal de Santa Catarina - UFSC

Esta tese é dedicada a minha mãe que me impulsionou nesta jornada, concedendo a mim o apoio para sempre persistir em busca de minhas metas.

Agradecimentos

Agradeço primeiramente a Deus que tem me abençoado com capacidade intelectual neste processo de aprendizagem contínua, onde em cada etapa em busca pelo conhecimento, me permitiu deslumbrar novos ambientes, pelo privilégio de compartilhar tamanha experiência, contribuindo assim na consolidação do conhecimento em diferentes vertentes da minha vida.

A minha querida mãe Atenilza Oliveira, um exemplo de mulher forte, honesta, e trabalhadora que me incentiva a romper e superar desafios. Eu agradeço pelo seu apoio e paciência em cada momento, por estar presente quando necessitei, mesmo que em alguns momentos longe pela distância entre as cidades (Boa Vista - Manaus), por ser um referencial de vida para mim. Muito obrigado por tudo mãe.

Agradeço a minha esposa Ivaneide Amazonas, por estar ao meu lado nesta jornada, tanto nos momentos bons, mas especialmente nos difíceis. Obrigado por toda a sua paciência, compreensão, e amor, por sempre me incentivar a ser uma pessoa melhor, e, sobretudo, por me fazer feliz, a cada dia.

Ao professor Raimundo Barreto, meu orientador, muito obrigado pelos ensinamentos, por todo o seu apoio e presteza no auxílio às atividades e discussões sobre o andamento deste trabalho e principalmente por acreditar no desenvolvimento do meu trabalho, proporcionando-me novas experiências que contribuíram significativamente no meu processo de aprendizagem e para a minha vida.

Agradeço ao professor Lucas Cordeiro, meu coorientador, pela orientação, incentivo, conselhos objetivos e amizade ao longo dos anos da minha pesquisa. Pela troca de saberes nas discussões e esclarecimentos na execução deste trabalho.

Aos professores Arilo Claudio Dias Neto, Djones Vinicius Lettnin e Leandro Buss Becker, por participarem de minha banca de defesa de doutorado.

Agradeço também aos amigos, colegas de laboratório que ajudaram com inúmeras discussões frutíferas sobre o meu trabalho, colegas de classe pela espontaneidade e alegria na troca de informações e materiais numa rara demonstração de amizade e solidariedade. Aos professores que tanto me ajudaram e incentivaram nesta longa jornada que agora conclui mais uma etapa e aos demais idealizadores, coordenadores e funcionários da Universidade Federal do Amazonas (UFAM), em especial ao grupo do departamento do Programa de Pós-Graduação em Informática (PPGI). A CAPES pelo suporte financeiro no processo de execução desta pesquisa. Enfim, agradeço a todos que de alguma forma contribuíram para que eu conseguisse ultrapassar as dificuldades e alcançar meu objetivo. Meu muito obrigado!

A História tem demonstrado que os mais notáveis vencedores normalmente encontraram obstáculos dolorosos antes de triunfarem. Eles venceram porque se recusaram a se tornarem desencorajados por suas derrotas. (Bryan Forbes)

RESUMO

Um dos principais desafios no desenvolvimento de *software* é garantir a funcionalidade dos sistemas de *software*, especialmente em sistemas embarcados críticos, tais como aeronáutico ou hospitalar, onde diversas restrições (por exemplo, tempo de resposta e precisão dos dados) devem ser atendidas e mensuradas de acordo com os requisitos do usuário, caso contrário uma falha pode conduzir a situações catastróficas. Logo, técnicas de verificação e teste de *software* são itens indispensáveis para um desenvolvimento com qualidade, onde tais técnicas visam confirmar os requisitos do usuário, bem como os comportamentos pré-estabelecidos para o *software*.

No contexto de verificação de *software*, visando à qualidade geral do produto, a técnica de verificação formal *model checking* tem sido utilizada para descobrir erros sutis em projetos de sistemas de *software* atuais. Contudo, a utilização da técnica *model checking* apresenta alguns desafios, tais como, lidar com a explosão do espaço de estados do modelo, integração com outros ambientes de testes mais familiares aos projetistas e tratamento e análise de contra-exemplos para reprodução de erros. De modo a lidar com estes problemas, uma possível solução é explorar as características já providas pelos *model checkers*, por exemplo, a verificação de propriedades de segurança e geração de contra-exemplos. Explorando este conjunto de características, juntamente com a utilização da inferência de invariantes e um tipo especial de *model checking*, denominado de *Bounded Model Checking* (BMC), esta tese apresenta um conjunto de métodos para complementar e aprimorar a escalabilidade e acurácia da verificação efetuada por *Bounded Model Checkers*. Estes métodos utilizam técnicas de transformações de código para explorar as características de *Bounded Model Checkers*, a fim de analisar propriedades de segurança e demonstrar erros em códigos escritos na linguagem de programação C.

Os métodos apresentados nesta tese são: (1) A geração e verificação automática de casos de teste baseado em propriedades de segurança geradas por um *Bounded Model Checker* para testes de unidade; (2) Automatizar a coleta e manipulação das informações dos contra-exemplos, de modo a demonstrar a causa principal do erro identificado; e (3) Utilização de invariantes dinamicamente/estaticamente inferidas, a partir do programa analisado, para restringir a exploração dos conjuntos de estados durante a execução da verificação pelo BMC. Desta forma, ajudando no aprimoramento da verificação efetuada por um BMC, no que concerne em auxiliar a sua verificação e na precisão dos resultados, pela utilização de invariantes de programas. As abordagens propostas, quando utilizadas isoladamente, fornecem alternativas complementares a verificação e, interligadas, aprimoram a verificação de código. Os resultados experimentais dos métodos propostos demonstram ser eficientes sobre *benchmarks* públicos de programas em C, encontrando defeitos não anteriormente encontrados por outros métodos que são estado-da-arte.

ABSTRACT

One of the main challenges in software development is to ensure the safety of the software systems, especially in critical embedded systems, such as aircraft or healthcare, where several constraints (e.g., response time and data accuracy) must be met and measured in accordance with the user requirements, otherwise a failure can lead to catastrophic situations. Thus, software verification and testing techniques are essential items for the software development with quality, where such techniques aim to confirm the user requirements, as well as, the predetermined behaviors for the software.

In the software verification context, aiming the product quality, the formal verification technique called model checking has been used to find subtle errors in actual projects of the software systems. However, the use of the model checking technique presents some challenges such as dealing with the model's state explosion problem, integration with software testing environments more familiar to designers, and handling counter-examples to reproduce the identified errors. In order to deal with these problems, a possible solution is to explore the characteristics already provided by the model checkers, e.g., verification of the safety properties and generation of counter-examples. Exploring this set of characteristics, coupled with the use of program invariants inference and a special kind of model checking, called Bounded Model Checking (BMC), this thesis presents a set of methods to complement and enhance the scalability and accuracy of the verification performed by Bounded Model Checkers. These methods adopted code transformation techniques to explore the characteristics of Bounded Model Checkers to analyze the safety properties and demonstrate errors in programs written in the C programming language.

The methods presented in this thesis are: (1) The automatic generation and verification of the test cases based on safety properties generated by a Bounded Model Checker for unit tests; (2) Automating the collection and manipulation of the data from the counter-examples, to demonstrate the main cause of the identified error; and (3) Adopting program invariants dynamically/statically inferred from the analyzed program, to restrict the exploration of the states sets while performing the verification by the BMC. This way, helping to improve the verification performed by a BMC, related to assist in the verification and accuracy of results, by adoption of the program invariants. The proposed approaches when used separately, provide additional options to the verification, and interconnected, improving the code verification. The experimental results of the proposed methods show to be efficient over public available benchmarks of C programs, finding errors not previously found by other methods that are state-of-the-art.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviações e Acrônimos

AA	Árvore de Alcançabilidade
ACSL	ANSI/ISO C Specification Langage
AST	Abstract Syntax Tree
AsmL	Abstract State Machine Language
ASPIC	Accelerated Symbolic Polyhedral Invariant Computation
BDD	Binary Decision Diagram
BMC	Bounded Model Checking
CBMC	C Bounded Model Checker
CTL	Computational Tree Logic
CFG	Control Flow Graph
CUnit	C Unit Testing Framework
CV	Condição de Verificação
DBC	Design by Contract
DMA	Direct Memory Access
E − ACSL	Executable ANSI/ISO C Specification Language
EGT	Execution Generated Testing
ESC/Java	Extended Static Checker for Java
ESBMC	Efficient SMT-Based Bounded Model Checker
FORTES	FORmal unit TESt generation
GPL	GNU General Public License
KLOC	Thousands (Kilos) of Lines of Code
JML	Java Modeling Language
LLBMC	Low-Level Bounded Model Checker
LLVM	Low Level Virtual Machine
LOC	Lines of Code
LTL	Linear-time Temporal Logic
POR	Partial Order Reduction
PCTL	Probabilistic Computation Tree Logic
SAT	Boolean Satisfiability
SDL	Specification and Description Language
SMG	Symbolic Memory Graph
SMT	Satisfiability Modulo Theories

$SV-COMP$	Competition on Software Verification
SUT	System Under Test
$VV\&T$	Verificação, Validação e Teste
$WCET$	Worst-Case Execution Time

Capítulo 1

Introdução

A complexidade de sistemas computacionais cresce exponencialmente. Muitas companhias e organizações estão rotineiramente lidando com *software* que contêm milhares de linhas de código, escritos por diferentes pessoas, e que usam diferentes linguagens, ferramentas e estilos (Hoder *et al.*, 2011a). Adicionalmente, estes sistemas de *software*, devido ao curto espaço de tempo para liberação do produto ao mercado, precisam ser desenvolvidos rapidamente e atingir um alto nível de qualidade. Porém, os programadores cometem enganos. Por exemplo, quando um programador acidentalmente escreve um dado requisito do sistema de forma incorreta. Um exemplo seria a alteração de uma atribuição de $x \leq 10$ para $x < 10$, bastante tempo e esforço é gasto em encontrar e corrigir estes erros (Gupta e Rybalchenko, 2009). Como consequência destes fatores, os erros durante o desenvolvimento de *software* tornam-se mais comuns. Desta forma, é necessário que as aplicações sejam projetadas considerando os requisitos de previsibilidade e confiabilidade, principalmente em aplicações de sistemas críticos, onde diversas restrições (por exemplo, tempo de resposta e precisão dos dados) devem ser atendidas e mensuradas de acordo com os requisitos do usuário, caso contrário uma falha pode conduzir a situações catastróficas. Por exemplo, o erro de cálculo da dose de radiação no Instituto Nacional de Oncologia do Panamá que resultou na morte de 23 pacientes (Wong *et al.*, 2010).

A fim de se obter um alto nível de qualidade no desenvolvimento dos sistemas de *software*, a execução do *software* deve ser controlada, e da mesma forma deve-se buscar formas de garantir que as propriedades definidas sejam atingidas. Por exemplo, a partir do conhecimento prévio dos comportamentos de cada execução do *software* e se estes atendem aos requisitos do usuário. Um modo de obter estas garantias é utilizar lógica computacional (principalmente lógica temporal) para validar as especificações destes sistemas. Em outras palavras, para comprovar que os comportamentos estão de acordo com as respectivas especificações do sistema para um dado momento de sua execução ou em todos os momentos, deve-se analisar ou verificar estes sistemas utilizando um raciocínio automatizado e baseado em métodos formais. Tendo como base esta solução, diversas estratégias de verificação e de teste estão sendo pesquisadas e aplicadas para garantir a qualidade do *software* (Hoder *et al.*, 2010; Rocha *et al.*, 2010; Cordeiro *et al.*, 2012a; Merz *et al.*, 2012).

A verificação formal tem desempenhado um papel importante para assegurar a previsibilidade e a confiabilidade na concepção de aplicações críticas. Segundo Bensalem e Lakhnech (1999), *model checking* é uma técnica baseada em métodos formais que é bem conhecida para provar propriedades de programas reativos. Esta técnica gera uma busca exaustiva no espaço de estados do modelo para determinar se uma dada propriedade é válida ou não (Baier e Katoen, 2008). A principal razão para o sucesso da técnica *model checking* é que ela funciona completamente automática, ou seja, sem qualquer intervenção do usuário. Contudo, a técnica *model checking* ainda possui alguns desafios, como por exemplo, como lidar com a explosão do espaço de estados do modelo, integração com outros ambientes de testes mais familiares aos projetistas e tratamento e análise de contra-exemplos (Clarke, 2008).

No contexto de *model checking* existe um tipo especial denominado de *Bounded Model Checking* (BMC) que geralmente utiliza o método *Boolean Satisfiability* (SAT), o qual tem sido introduzido como uma técnica complementar para diagramas de decisão binária (do inglês *Binary Decision Diagrams*, BDD) para aliviar o problema da explosão de estados (Biere, 2009). Entretanto, os *Bounded Model Checkers* que utilizam esta técnica podem sofrer de falta de memória ou mesmo uma verificação incompleta. Uma das razões é devido ao fato de que eles manipulam estruturas complexas. Dois exemplos são: (i) os *loops* aninhados, onde se tem cada interação desdobrada, além da análise das estruturas contidas em suas possíveis intersecções; e (ii) múltiplos chaveamentos de contexto, que requerem uma análise de cada *interleaving*[1] gerado, bem como suas interações. Assim, resultando na verificação de uma grande quantidade de informações, ou seja, diversas possibilidades a serem analisadas.

Visando contribuir com a verificação e testes de programas, as invariantes de programas tem desempenhado um papel importante para a identificação de propriedades a serem analisadas/validadas. As invariantes de programas são propriedades que contêm o relacionamento entre as variáveis ou constantes em uma linha específica do programa (Fouladgar *et al.*, 2011). Em outras palavras, invariantes de programas são fórmulas ou regras que surgem a partir do código fonte de um programa e permanecem únicas e inalteradas durante a execução de um programa com diferentes parâmetros. Assim, a verificação de programas geralmente utiliza assertivas baseadas em invariantes, para analisar o conjunto de estados alcançáveis a fim de identificar propriedades que possam ser violadas (Cao e Zhu, 2010). As pesquisas bibliográficas apontam para diversos campos de pesquisa em aberto na geração de invariantes. Isto se deve ao fato de que a maioria das abordagens para a geração de invariantes possuírem limitações, como por exemplo, o suporte as estruturas da linguagem do programa a ser verificado e as diferentes classes de invariantes, tais como: invariantes globais e de *loops* (Gupta *et al.*, 2009).

O contexto deste trabalho está situado no uso de metodologias e técnicas de verificação formal, focando principalmente em *Bounded Model Checking* e Invariantes de Programas. Este trabalho está interessado especificamente na parte de: geração de casos de testes baseados em propriedades de segurança de programas (por exemplo, segurança de vetores, ponteiros e objetos dinâmicos), ou

[1] Um *interleaving* representa uma possível execução do programa onde todos os eventos são concorrentes (Cordeiro e Fischer, 2011)

seja, testes de propriedades do programa que garantam que o programa não entre em um estado de erro ou bloqueio; simplificação do modelo a ser verificado por *model checkers*; e na demonstração de erros pela utilização de contra-exemplo de *model checkers*. O escopo deste trabalho é restrito a análise e verificação de códigos na linguagem de programação C. A linguagem de programação C foi escolhida, pois de acordo com Nagarakatte *et al.* (2010), a linguagem de programação C/C++ é de fato o padrão para a implementação de diversos tipos de *software*, incluindo *software* crítico, sendo este tipo de *software* um dos principais focos deste trabalho. Isto porque estes necessitam de uma verificação e validação mais rigorosa, devido a suas aplicações em ambientes (por exemplo, hospitalar) onde uma falha pode resultar em situações catastróficas.

Neste trabalho é apresentado um conjunto de métodos para complementar e aprimorar a verificação efetuada por *Bounded Model Checkers*, onde estes métodos utilizam técnicas de transformações de código para analisar propriedades de segurança e demonstrar erros em códigos escritos na linguagem C. Este conjunto de métodos proposto é composto por três abordagens que visam: (1) A geração e verificação automática de casos de teste baseado em propriedades de segurança geradas pelo ESBMC (*Efficient SMT-Based Bounded Model Checker* (Cordeiro *et al.*, 2012a)) com o *framework* CUnit[2] (Rocha *et al.*, 2010, 2015a); (2) Automatizar a coleta e manipulação das informação dos contra-exemplos, de modo a demonstrar a causa principal do erro identificado (Rocha *et al.*, 2012); e (3) Utilização de invariantes dinamicamente/estaticamente inferidas a partir do programa analisado para restringir a exploração dos conjuntos de estados durante a execução da verificação de um BMC (neste caso o ESBMC). Desta forma, ajudando no aprimoramento da verificação efetuada por um BMC no que concerne em auxiliar a sua verificação dos resultados pela utilização de invariantes de programas (Rocha *et al.*, 2015b). As abordagens propostas quando utilizadas isoladamente fornecem alternativas complementares a verificação e interligadas aprimoram a verificação de código.

1.1 Definição do Problema

Segundo Clarke (2008), *model checking* tem diversas vantagens se comparadas à outras técnicas. Segue uma lista parcial de algumas dessas vantagens: (i) o usuário de um *model checker* não necessita construir provas de corretude; (ii) o processo de verificação é automático; (iii) *model checking* é rápido se comparado a outros métodos, como por exemplo, o *proof checker* (Berghofer, 2002) que pode requerer meses de trabalho do usuário em modo interativo; (iv) se uma especificação (propriedade) não é satisfeita, o *model checker* produzirá um contra-exemplo do caminho da execução, demonstrando o porque a especificação não está segura; e (v) lógicas temporais podem ser facilmente expressas.

Apesar de algumas vantagens, também se observa algumas desvantagens do *Model Checking* (Clarke, 2008): (i) o *model cheking* não comprova o erro identificado no programa, o que ajudaria no entendimento do mesmo; (ii) a escrita de especificações nem sempre é uma tarefa trivial; e (iii)

[2]Disponível em http://cunit.sourceforge.net/

o problema da explosão do espaço de estados, que é causado devido ao grande número dos estados de um sistema concorrente ou devido a uma estrutura de dados, podendo chegar a números exponenciais.

Observando as desvantagens do *model checker*, pode-se identificar campos de pesquisa em aberto, no que diz respeito a: prover abordagens complementares à verificação efetuada por *model checkers*; explorar as características dos *model checkers*, visando contribuir para sua completude; e a integração de outras técnicas formais para prover ao modelo a ser verificado por um *model checker* mecanismos que facilitem e aumentem a precisão de sua verificação.

O problema considerado neste trabalho é expresso na seguinte questão: Como complementar e aprimorar a verificação de propriedades de segurança e a demonstração de erros pelos *Bounded Model Checkers*, explorando suas características como identificação das propriedades de segurança e contra-exemplos, e utilizando invariantes de programas com aplicação na linguagem de programação C, de tal modo que as suas desvantagens possam ser contornadas e as propriedades verificadas possam ser validadas?

1.2 Motivação

Atualmente existem inúmeros sistemas de *software* aplicados a áreas muito distintas como: ambiental, aeroespacial, militares e médicas. Apesar dos diferentes graus de prejuízo que uma eventual falha em algum destes sistemas possam provocar, um alto grau de confiabilidade é exigido para quase todos eles. Sendo assim, um dos motivos desde trabalho está vinculado à necessidade de se garantir a corretude dos sistemas, que é acompanhada pelo aumento do grau de complexidade exigido para os novos sistemas, ao mesmo tempo em que, por pressão econômica, os prazos de entrega se mantêm os mesmos ou até menores. Tanto o aumento da complexidade quanto a diminuição no prazo de entrega tornam bem mais difícil à tarefa da verificação do sistema a ser entregue. Isto estimula a pesquisa por alternativas automáticas de verificação e correção de *software* (D'Silva *et al.*, 2008a).

A otimização da técnica de *model checking* na linguagem C para o desenvolvimento de *software* é um fator importante para garantir a validade do comportamento e propriedades acerca do *software*. O desenvolvimento e otimização desta técnica contribuirá para a melhoria de qualidade no desenvolvimento de *software*, já que normalmente existe um alto custo envolvido na preparação, execução e gerenciamento dos testes e verificação destes (Sherer, 1991). Desta forma, a motivação deste trabalho está em aprimorar e demonstrar métodos que:

1. Auxiliem na geração de casos de testes de forma automática, provendo um tempo menor de resposta na criação e execução de testes do software, principalmente no que concerne à verificação do gerenciamento de memória de um programa, que é uma tarefa importante para evitar um comportamento inesperado, por exemplo, a violação da segurança de um ponteiro que resulta em um endereço inválido pode gerar um resultado incorreto do programa e não

necessariamente uma parada inesperada, bem como a ocorrência de um *memory leak* que não produz imediatamente um sintoma facilmente visível;

2. Agilizem o processo de identificação e depuração de erros de um *model checker*. Isto porque, os contra-exemplos gerados pelos *model checkers* podem gerar uma grande quantidades de informações (por exemplo, rastreamentos de erros com mais de 3.000 linhas de código (Rocha *et al.*, 2012)) o que torna o processo de depuração inviável, além de possíveis atrasos no projeto;

3. Aprimorem o processo de verificação de um *model checker*, reduzindo o modelo do programa a ser verificado pela utilização de restrições para guiar a exploração dos estados, uma vez que devido as características do programa o número de estados do modelo pode crescer exponencial, por exemplo, em programas com estruturas não determinísticas em *loops* aninhados.

1.2.1 Exemplo

Visando exemplificar uma das motivações deste trabalho, será utilizado o item (3) mencionado anteriormente que trata do problema da explosão de estados sofridos pelos *model checkers*. Assim, será utilizado um programa extraído do *benchmark* do SV-COMP (Beyer, 2013) apresentado na Figura 1.1, onde a variável `a` é uma constante inteira e observe que as variáveis `i` e `sn` são declaradas com um tipo maior do que o tipo da variável `n` para evitar *overfow* aritmético. Matematicamente, o código do programa é representado com a implementação da soma dada pela seguinte equação:

$$S_n = \sum_{i=1}^{n} a = na, n \geq 1 \tag{1.1}$$

No programa da Figura 1.1 a propriedade (representado pela assertiva na linha 10) deve ser verdadeira para qualquer valor de `n` (isto é, para qualquer desdobramento do programa). As técnicas de BMC têm dificuldades em provar a correção deste programa simples, uma vez que o valor do limite superior do *loop*, representado por `n`, é escolhido de forma não determinística, ou seja, a variável `n` pode assumir qualquer valor a partir de um tamanho do tipo *unsigned int*, que varia entre os diferentes tipos de computadores. Devido a está condição, o *loop* pode ser desdobrado $2^n - 1$ vezes (no pior caso $2^{32} - 1$ vezes em um inteiro de 32 bits) que é, portanto, pouco prático. Basicamente, um *Bounded Model Checker* simbolicamente executa várias vezes o incremento da variável i e computa a variável sn $4,294,967,295$ vezes. Um modo de resolver o problema de desdobrar o *loop* $2^n - 1$ vezes é guiar a verificação efetuada por um BMC utilizando invariantes de programas como restrições que auxiliaram a exploração dos conjuntos de estados verificados, reduzindo assim o modelo a ser verificado por um BMC.

A hipótese levantada neste trabalho é que métodos baseados em transformações de código juntamente com a utilização de invariantes de programas provêem um suporte complementar a verificação efetuada pelos BMCs no que diz respeito a: (1) reduzir o tempo de criação e execução

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

Figura 1.1: Programa de soma do SV-COMP.

de casos de teste; (2) integração da técnica *model checking* com teste de unidade; (3) agilizar o processo de identificação e depuração de erros; (4) auxiliar a exploração dos conjuntos de estados no modelo a serem verificados por um *model checker*; e (5) maior precisão na verificação de propriedades efetuada por um *model checker*.

1.3 Objetivos

O objetivo principal deste trabalho é apresentar um conjunto de métodos para a verificação formal de programas escritos na linguagem de programação C que use Bounded Model Checkers e que a verificação seja aprimorada pela redução do espaço de estados de tal forma que novos erros possam ser identificados e comprovados, além da geração automática de casos de testes que consigam avaliar diversos tipos de propriedades de segurança, incluindo segurança de memória.

Os objetivos específicos são:

1. Propor e analisar um método de geração e verificação de casos de teste de unidade, que consiga melhorar a taxa de resultados corretos na verificação de programas, não antes obtidos pelos trabalhos do estado-da-arte, baseado nas propriedades identificadas por um *Bounded Model Checker*, incluindo segurança de memória, e que estes testes possam ser integrados com alguma ferramenta de teste unitário disponível na literatura.

2. Formular um método para inferir invariantes de programas na verificação efetuada por um *Bounded Model Checker* de tal forma a demonstrar que é possível melhorar a taxa de resultados corretos na verificação de programas, em relação às principais técnicas disponíveis na literatura;

3. Especificar um método para automatizar a reprodução do possível defeito identificado no código fonte pela análise dos contra-exemplos, sendo que o método deve ser capaz de reproduzir a execução do programa apontada no contra-exemplo, bem como, apresentar resultados satisfatórios e comparáveis aos principais trabalhos da literatura;

4. Validar a aplicação dos métodos sobre *benchmarks* públicos de programas em C, a fim de examinar a sua eficácia e aplicabilidade.

1.4 Metodologia Proposta

A solução proposta neste trabalho visa lidar com os principais desafios em *model checking*, ou seja, como lidar com a explosão do espaço de estados do modelo, integração com outros ambientes de testes mais familiares aos projetistas e tratamento e análise de contra-exemplos para reprodução de erros. Desta forma, neste trabalho foram desenvolvidos métodos e suas respectivas ferramentas que foram avaliados usando *benchmarks* públicos de programas escritos em linguagem de programação C. Estes métodos foram desenvolvidos utilizando técnicas e métodos como análise estática e dinâmica, expressões regulares, instrumentação de código, abstração de predicados, análise de restrições e tecnologias de compiladores.

A Figura 1.2 mostra uma visão das partes que compõe este trabalho, onde é proposto um conjunto de três métodos baseados em transformações de código para aprimorar a verificação efetuada por *Bounded Model Checkers*. Nesta seção somente é apresentado um resumo de cada componente dos métodos propostos, mais detalhes são apresentados nos próximos capítulos. Os métodos propostos consistem dos seguintes componentes:

- **Map2Check - Gerador e analisador de casos de teste**: Criar e analisar casos de teste com assertivas, baseados nas propriedades de segurança geradas automaticamente por um *Bounded Model Checker*, neste trabalho foi adotado o ESBMC (Cordeiro *et al.*, 2012a), onde este método é composto por:

    1. **Gerador de propriedades**. As propriedades/*claims* utilizadas para a criação dos casos testes são geradas automaticamente pelo ESBMC.
    2. **Tradutor de *claims***. O tradutor é composto de um *parser*, com uma gramática que foi desenvolvida neste trabalho para as *claims*. A partir da identificação das estruturas compostas nas *claims* são aplicadas regras de transformação que irão efetuar a tradução automática das funções do ESBMC, utilizando assertivas;
    3. **Rastreador de código**. O analisador é responsável por efetuar uma leitura do código, a fim de identificar as variáveis do código, com o foco em ponteiros e objetos dinâmicos, bem como suas respectivas operações e atribuições. Como resultado final o analisador mapeia as variáveis com suas respectivas operações e sua localização no código;
    4. **Verificador de memória**. O verificador de memória visa instrumentar o código com funções (utilizando os dados identificados pelo Rastreador de código) que irão monitorar os endereços de memória e endereços que apontam estas variáveis durante a execução do código;

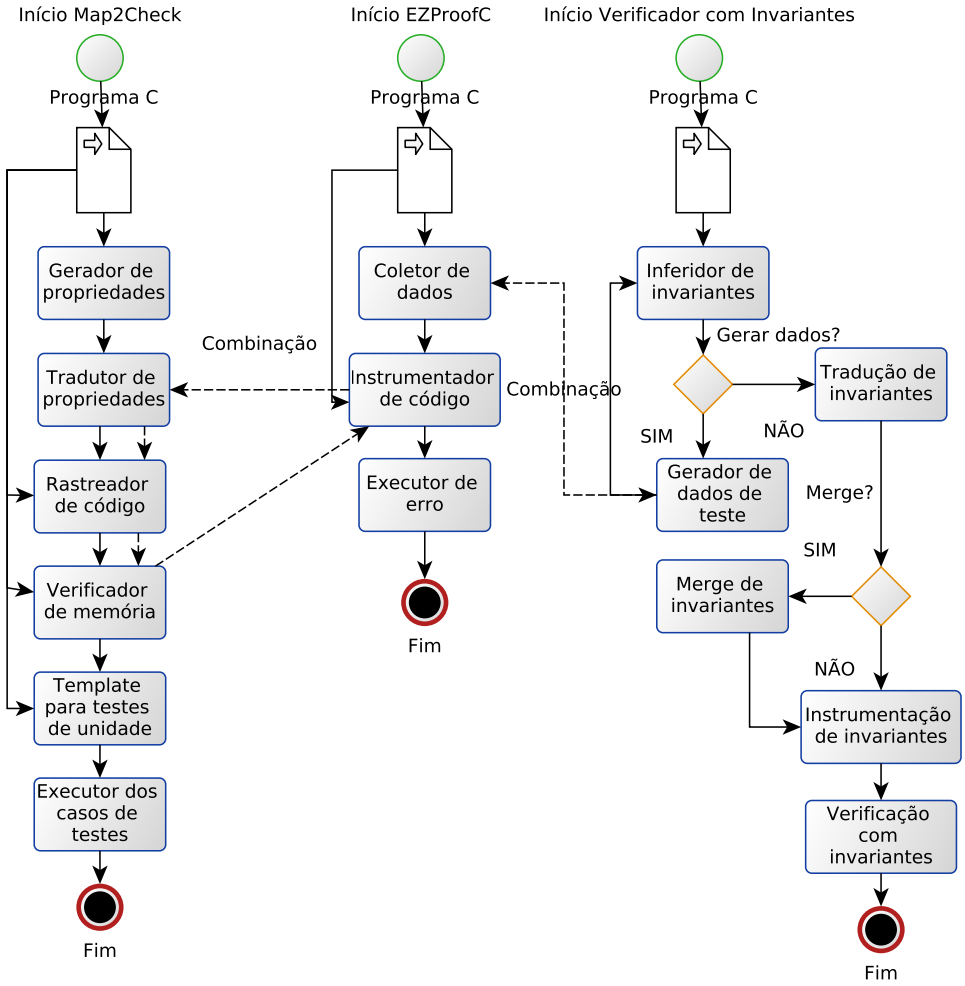


Figura 1.2: Visão geral dos componentes deste trabalho

5. ***Template* para testes de unidade**. Visa gerar uma entrada adaptada no programa analisado a partir de um *template* que contêm as estruturas de um *framework* de testes de unidade, estruturas como: *includes* específicos do *framework*, funções de configuração e execução dos testes de unidade;
6. **Executor dos casos de testes**. O código instrumentado com os casos de testes e então executado a fim de se identificar a violação das propriedades do programa.

- **EZProofC - Demonstrador de erros:** Responsável por explorar os contra-exemplos gerados pelo ESBMC para demonstração de erros pela geração de uma nova instância do código original instrumentado com as informações do contra-exemplo. Este componente possui conexão com o método Map2Check, onde esta conexão é para gerar uma dada assertiva a partir da propriedade identificada como violada no contra-exemplo gerado pelo ESBMC. Este método é composto pelos seguintes componentes:

1. **Coletor de dados.** Responsável por coletar os dados no contra-exemplo gerados pelo ESBMC, tais como: número da linha apontado, propriedades/*claims*, variáveis e seus respectivos valores.
2. **Instrumentador de código.** Criar uma nova instância do programa analisado, onde cada variável neste novo programa receba como atribuição os valores identificados no contra-exemplo. Adicionalmente, adiciona uma assertiva com a propriedade traduzida, para isso são utilizados os componentes do método Map2Check;
3. **Executor de erro.** Responsável por executar a nova instância do programa analisado contendo as atribuições de valores e uma assertiva com a propriedade identificada no contra-exemplo gerado pelo ESBMC.

- **Verificador com invariantes:** Responsável por inferir invariantes de programas que irão guiar (por meio de restrições) a exploração e verificação dos conjuntos de estados analisados por um *Bounded Model Checker*, neste caso foi adotado o ESBMC (Cordeiro *et al.*, 2012a), necessários para a identificação da violação de uma propriedade. O funcionamento deste componente consiste em:

  1. **Inferidor de invariantes**. Gera estaticamente/dinamicamente invariantes a partir do programa, incluindo pré- e pós-condições. As invariantes são geradas a partir de um conjunto de ferramentas no estado da arte: Daikon (Ernst *et al.*, 2007), PIPS (Ancourt *et al.*, 2010), ASPIC (Feautrier e Gonnord, 2010) e InvGen (Colón *et al.*, 2003);
  2. **Gerador de dados de teste.** Este componente possui duas abordagens para a geração de dados de teste, uma utilizando a ferramenta PathCrawler (Williams *et al.*, 2005a); e outra utilizando os dados coletados no contra-exemplo (reutilizando o componente do método EZProofC para a coleta dos dados no item 1), onde estes dados são correlacionados a fim de auxiliar a geração de invariantes com a entrada dos dados de rastreamentos do programa para ferramentas que executam análise dinâmica;
  3. **Tradução de invariantes**. As invariantes geradas no item 1 por cada ferramenta possui um padrão, linguagem e simbologia própria (em alguns casos não compatíveis com a linguagem de programação C) para as invariantes. Esta tradução consiste em identificar (por meio de expressões regulares) os elementos que compõem as invariantes geradas por cada ferramenta e então traduzir estes elementos pelos suportadas pela linguagem de programação C e pelo ESBMC.
  4. **Merge de invariantes**. Este componente efetua uma combinação das invariantes geradas, tendo como base as localizações em comum das invariantes e seus respectivos elementos;
  5. **Instrumentação de invariantes.** A partir das invariantes geradas é criada uma nova instância do programa analisado onde, nesta nova instância, as invariantes geradas são inseridas como assertivas que indicam ao ESBMC restrições que ele deve assumir durante a verificação do programa. Estas assertivas visam definir pré- e pós-condições, bem como reescrever estruturas de repetições (usando invariantes de *loop*);

6. **Verificação com invariantes.** Utilizando a nova instância do programa analisado com as invariantes (resultado da aplicação do componente no item 5) é então efetuado a verificação do programa utilizando o ESBMC.

1.5 Contribuições Propostas

A principal contribuição deste trabalho é a demonstração, implementação e avaliação de uma metodologia para a verificação e teste de programas, que complementa e aprimora a verificação efetuada por *Bounded Model Checkers*. As contribuições mais específicas são descritas a seguir:

1. **Map2Check - Geração de casos de teste**. O método proposto gera automaticamente casos de teste baseado em propriedades de segurança geradas por um *Bounded Model Checker* utilizando assertivas. Estas propriedades são relacionadas à segurança de ponteiros, gerenciamento de memória (por exemplo, alocação dinâmica de memória) e limites da indexação de *arrays*. Este trabalho também apresenta para este método, o desenvolvimento e implementação de um verificador de memória que utiliza um instrumentador e um rastreador de código que auxilia nesta verificação. As assertivas utilizadas podem ser simples assertivas da linguagem C, assim como assertivas providas por um *framework* de teste de unidade, neste caso o *framework* CUnit. Adicionalmente, é apresentado o desenvolvimento e implementação de um tradutor automático de propriedades, uma vez que algumas propriedades podem conter funções especificas do *Bounded Model Checker*, neste caso foi adotado o ESBMC (Cordeiro *et al.*, 2012a). Desta forma, esta contribuição auxilia na integração/utilização de técnicas formais por usuários não familiarizados, bem como complementa a verificação efetuada por *Bounded Model Checkers*;

2. **EZProofC - Coleta de dados de contra-exemplos para reprodução de erros.** O método proposto explora os contra-exemplos gerados pelo ESBMC, para a reprodução de erros pela geração de uma nova instância do código original instrumentado com as informações do contra-exemplo. Vale ressaltar que este método proposto utiliza partes do método Map2Check para a geração de casos de teste, tornando mais precisa a reprodução do erro identificado. Neste trabalho argumenta-se que esta contribuição visa prover um suporte complementar a reprodução de erros identificados pelo ESBMC, principalmente quando os contra-exemplos gerados podem ser longos ou difíceis de ser entendidos devido ao tamanho do programa ou pelos valores escolhidos pelo *solver* adotado pelo BMC.

3. **Verificação utilizando invariantes**. A verificação convencional tem como base todo o modelo a ser verificado. Neste método, entretanto, foca-se em restringir a exploração dos conjuntos de estados durante a verificação de um BMC (neste caso o ESBMC) pela utilização de invariantes de programas. Estas invariantes definem pré- e pós-condições para o programa analisado, bem como a pré-verificação de *loops* pela utilização de invariantes globais e de *loops*. Isto porque, em casos específicos as técnicas de BMC podem ter

dificuldades em provar a corretude de um programa, onde o número máximo de execuções de um dado *loop* é escolhido de forma não determinísticas ou um *loop* dependa de um valor de uma invariante de *loop* para determinar a sua execução ou não. Adicionalmente, este método possui uma abordagem para auxiliar na geração de invariantes que consiste em obter as atribuições de valores das variáveis em determinadas execuções do programa, por meio da análise de contra-exemplos (para isto são utilizadas partes do método EZProofC). Os valores coletados são então utilizados como dados testes no programa analisado para a geração das invariantes. A contribuição desta abordagem se deve ao fato que um dos desafios na geração de invariantes por meio de análise dinâmica é a criação de dados de teste para explorar as execuções do programa. O monitoramento destas execuções permite aos métodos de geração de invariantes coletarem os dados das variáveis, visando identificar as relações das variáveis e suas respectivas invariantes. Desta forma, esta contribuição foca em aprimorar a verificação efetuada por um BMC no que concerne em auxiliar a sua verificação e na e precisão dos resultados pela utilização de invariantes de programas.

4. **Levantamento de abordagens para inferência de invariantes de programa**. Este levantamento visa contribuir com uma revisão da literatura feita utilizando a técnica da revisão sistemática, a fim identificar e conhecer métodos para inferência de invariantes na verificação de código na linguagem de programação C existentes na literatura, extraindo suas principais características, e avaliando as vantagens e desvantagens de cada abordagem. Neste trabalho argumenta-se que esta contribuição tem impacto significativo para a identificação rápida e oportuna dos métodos com maior suporte (em relação a ferramental, estruturas da linguagem C e escalabilidade) para inferência de invariantes, bem como para a identificação dos seus respectivos pontos de melhorias.

1.6 Organização do Trabalho

A introdução deste trabalho apresentou o contexto, definição do problema, motivação, os objetivos, solução proposta e as contribuições desta pesquisa. Os capítulos restantes deste trabalho estão organizados da seguinte forma:

No Capítulo 2, **Conceitos e Definições**, é discutido os conceitos e definições abordados neste trabalho, tais como: Verificação e Teste de *Software*, Verificação Formal com *Model Checking*, Propriedades de Segurança, Invariantes de Programas, e Interpretação Abstrata de Programas.

No Capítulo 3, **Trabalhos Correlatos**, é analisado os principais trabalhos correlatos que focam em assuntos como: Geração de Casos de Teste, Verificação e Análise de Memória de Programas, Contra-exemplos de *Model Checkers* e Invariantes de Programas. Adicionalmente, é apresentado um levantamento bibliográfico, sobre invariantes de programas, feito utilizando a técnica da revisão sistemática.

No Capítulo 4, **Geração Automática de Casos de Teste para Gerenciamento de Memória de Programas em C usando *Bounded Model Checking***, é descrito o método Map2Check que cria e analisa casos de teste com assertivas, baseados nas propriedades de segurança geradas automaticamente por um *Bounded Model Checker*, bem como, é apresentado e analisado os resultados de um estudo empírico realizado com o objetivo de avaliar o método Map2Check quando aplicado à verificação dos *benchmarks* padrão ANSI-C e, além disso, uma comparação com as ferramentas: Valgrind's Memcheck (Nethercote e Seward, 2007), CBMC (Clarke *et al.*, 2004a), LLBMC (Merz *et al.*, 2012), CPAChecker (Beyer e Keremoglu, 2011), Predator (Dudka *et al.*, 2014), e ESBMC (Cordeiro *et al.*, 2012a).

No Capítulo 5, **Instanciação de Programas em C pela Utilização de Contra-Exemplos de BMC**, é descrito o método EZProofC que visa explorar os contra-exemplos gerados pelo *model checker* ESBMC, de tal forma que ele possa gerar um novo código instanciado com dados do contra-exemplo, a fim de reproduzir o erro identificado. Neste capítulo também é apresentado os resultados de um estudo experimental realizado com o objetivo de avaliar o método EZProofC quando aplicado à verificação de *benchmarks* padrão de programas ANSI-C. Adicionalmente, apresenta-se uma comparação com a ferramenta Frama-C (Canet *et al.*, 2009), usando os *plug-in* de análise de valores e `Jessie`.

No Capítulo 6, **Verificação de Programas usando *Bounded Model Checkers* com Invariantes de Programas**, é apresentado um método proposto para reduzir o número de estados no modelo a ser verificado por um *Bounded Model Checker*. A redução do modelo neste método consiste em adicionar restrições baseadas em invariantes de programas aos estados que são explorados pelos BMCs. Adicionalmente, descreve-se o um estudo empírico realizado com o objetivo de avaliar o método proposto para adicionar invariantes de programas à verificação executada por *Bounded Model Checkers*, neste caso o ESBMC. Adicionalmente, uma comparação com as ferramentas CPAChecker (Beyer e Keremoglu, 2011), CBMC (Clarke *et al.*, 2004a), e ESBMC (Cordeiro *et al.*, 2012a) sem as invariantes.

E por fim no Capítulo 7, **Conclusões e Trabalhos Futuros**, é exposto as considerações finais, os trabalhos futuros, e é apresentado a lista de publicações obtidas no desenvolvimento deste trabalho.

Capítulo 2

Conceitos e Definições

Este capítulo tem como objetivo apresentar os principais conceitos e definições abordados neste trabalho, tais como: Verificação e Teste de *Software*, *Model Checking*, Propriedades de Segurança, Invariantes de Programas, e Interpretação Abstrata de Programas.

2.1 Verificação e Teste de *Software*

O processo de desenvolvimento de *software* envolve uma série de atividades, por exemplo, no contexto de Garantia de Qualidade de Software são introduzidas, ao longo de todo o processo de desenvolvimento, atividades de VV&T - Verificação, Validação e Teste, com o objetivo de minimizar a ocorrência de erros e riscos associados (Delamaro *et al.*, 2007). Nas Seções 2.1.1 e 2.1.2 é definido teste e verificação formal de *software*, que são as atividades principais exploradas neste trabalho, no contexto de processo de desenvolvimento de *software* na área de verificação de *software*.

2.1.1 Teste de *Software*

Teste de *software* é um processo, ou uma série de processos, projetados para certificar que o código do programa para o computador faz o que foi projetado para fazer (Myers *et al.*, 2011), uma vez que o *software* deve ser previsível, consistente, e não apresentar comportamentos não esperados pelos usuários. Segundo Myers *et al.* (2011), quando você testa um programa, pretende-se acrescentar algum valor a ele. A agregação de valor por meio de testes significa elevar a qualidade ou a confiabilidade do programa. Aumentar a confiabilidade do programa significa encontrar e remover erros. Desta forma, teste de *software* é o processo de executar um programa com a intenção de encontrar erros.

Segundo Howden (1987), o teste pode ser classificado de duas maneiras: teste baseado em especificação (também conhecido como teste de caixa-preta) e teste baseado em programa (teste

de caixa-branca). De acordo com tal classificação, têm-se que os critérios da técnica funcional são baseados em especificação e tanto os critérios estruturais quanto baseados em erros são considerados critérios baseados em implementação.

O teste de produtos de *software* envolve basicamente quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados dos testes (Delamaro *et al.*, 2007). Essas atividades usualmente são desenvolvidas ao longo do próprio processo de desenvolvimento de *software*, e em geral, concretizam-se em três fases de teste (Pressman, 2001):

- O teste de unidade concentra esforços na menor unidade do projeto de *software*, ou seja, procura identificar erros de lógica e de implementação em cada módulo do *software*, separadamente. O teste de unidade faz uso massivo de técnicas de teste de caixa branca (baseado em código fonte do *software*), exercitando os caminhos específicos na estrutura de controle de um módulo para garantir uma cobertura completa e a detecção máxima de erros.
- O teste de integração é uma atividade sistemática aplicada durante a integração da estrutura do programa visando a descobrir erros associados às interfaces entre os módulos; o objetivo é, a partir dos módulos testados no nível de unidade, construir a estrutura de programa que foi determinada pelo projeto.
- O teste de sistema, realizado após a integração do sistema, visa a identificar erros de funções e características de desempenho que não estejam de acordo com a especificação. O teste de regressão é a atividade que ajuda a garantir que as mudanças (devido a testes ou por outras razões) não apresentam comportamento não intencional ou erros adicionais.

Segundo Myers *et al.* (2011), a limitação do teste de *software* está no fato que não é possível encontrar todos os erros. Em geral, é impraticável, muitas vezes impossível, encontrar todos os erros em um programa. Este problema fundamental, por sua vez, têm implicações em como serão planejados e executados os testes. Visando exemplificar algumas das limitações de teste será considerado o programa na Figura 2.1 apresentado em Bell (2005) que contém um método para calcular o produto de dois inteiros. Para testar o programa deve-se selecionar os valores e comparar com o resultado esperado. Assim, os valores de 21 e 568 são escolhidos.

```
1 public int product(int x, int y) {
2   int p;
3   p = x * y;
4   if (p == 42) p = 0;
5   return p;
6 }
```

Figura 2.1: Programa para calcular o produto de dois inteiros

O problema é que, por algum motivo - erro ou descuido - o programador decidiu incluir um `if`, o que leva a um valor incorreto em determinados casos. Os dados de teste que foi escolhido acima não revelaria esse erro. Assim, a escolha de dados de teste pode se tornar uma tarefa difícil e pode não garantir que um determinado erro seja identificado. Segundo Bell (2005), um modo seria

executar um teste exaustivo, onde seriam utilizados todos os possíveis valores. Contudo, para o método para multiplicar dois inteiros de 32 bits seria necessários 100 anos (assumindo o tempo de 1 milissegundo para a execução de uma instrução de multiplicar fornecida pelo *hardware* do computador). Assim, os testes exaustivos são quase sempre impraticáveis.

Segundo Myers *et al.* (2011), a consideração mais importante em testes de programa é o projeto e criação de casos de teste (um conjunto especifico de dados) eficazes. Dado que independentemente da fase, é o projeto e/ou a avaliação da qualidade de um determinado conjunto de casos de teste T utilizado para o teste de um produto P que pode determinar a qualidade do teste, pois em geral é impraticável utilizar todo o domínio de dados de entrada para avaliar os aspectos funcionais e operacionais de um produto em teste. O objetivo é utilizar casos de teste que tenham alta probabilidade de encontrar a maioria dos defeitos com um mínimo de tempo e esforço, por questões de produtividade.

2.1.2 Verificação Formal de *Software*

O uso de *software* tem sido feito em áreas muito distintas como: ambiental, aeroespacial, militares e médicas. Principalmente em infraestruturas industriais de missão crítica e de segurança crítica, uma vez que é, em princípio, a forma mais barata e mais eficaz para controlar sistemas complexos em tempo real. Logo, existe a necessidade de se garantir a corretude desses sistemas. Como uma alternativa para o teste, o que dificilmente é escalável a custos razoáveis e com uma cobertura satisfatória, a verificação formal automatizada emergiu, nesta última década, como um complemento útil promissor, com potencial para aplicações industriais interessantes (D'Silva *et al.*, 2008b; Cousot e Cousot, 2010).

Segundo Cousot e Cousot (2010), a ideia principal dos métodos formais é fazer provas automáticas em tempo de compilação para verificar propriedades do programa em tempo de execução. Três principais abordagens têm sido consideradas para verificação formal, sendo todas aproximações da semântica do programa (que define formalmente as possíveis execuções em todos os ambientes possíveis) formalizadas pela teoria da interpretação abstrata:

- Métodos dedutivos que produzem provas matemáticas formais de corretude usando provadores de teoremas ou assistentes de prova que precisam de interação humana para fornecer argumentos indutivos (que dificilmente são escaláveis para grandes programas que são modificados ao longo de grandes períodos de tempo) para a execução da prova;
- Verificação de Modelos (do inglês *Model Checking*) que explora exaustivamente modelos de execuções do programa, que podem ser sujeitos à explosão combinatória, para determinar a validade de propriedades do programa.
- A análise estática que engloba uma família de técnicas para calcular automaticamente informações sobre o comportamento de um programa sem executá-lo. Esta abordagem automatiza a abstração da execução do programa e sempre determina um resultado, mas que

este pode estar sujeito a falsos alarmes (são avisos que a especificação do programa pode não ser satisfeita, embora nenhuma execução efetiva do programa pode violar a especificação).

Na verificação formal de programas além da dificuldade de especificar quais as propriedades são de interesse na verificação em tempo de execução, todos os métodos formais se deparam com a indecidibilidade: a impossibilidade matemática para um computador, que é um dispositivo finito, provar a corretude de propriedades não triviais no comportamento (possibilidades infinitas ou extremamente grandes) de programas; e a complexidade: a impossibilidade de computadores para resolver questões decidíveis dentro de um período razoável de tempo com uma quantidade razoável de memória para grandes entradas de dados (Cousot e Cousot, 2010).

2.2 Estratégias de Análise de Programas

Os sistemas de *software* e *hardware* tornam-se cada vez mais complexos e, com isso, os programadores precisam de mecanismos para garantir a qualidade dos sistemas de *software*. Deste modo, a análise destes sistemas deve ser aprofundada ao ponto de abstrair e analisar as especificações requeridas para estes, de modo a identificar possíveis falhas.

Segundo Nethercote (2004), a análise de programas pode ser categorizada em quatro grupos, os dois primeiros são análise estática e análise dinâmica:

- (i) **Análise estática** que envolve a análise de código fonte de um programa ou código de máquina, sem executá-lo. Muitas ferramentas realizam análise estática, em particular compiladores; exemplos de análise estática utilizada por compiladores incluem a análise de corretude, tais como a verificação de tipos e análise para otimização. As ferramentas que realizam a análise estática somente necessitam ler o programa de modo a analisá-lo (Nethercote, 2004).
- (ii) **Análise dinâmica** que segundo Ball (1999), examina a execução do programa para derivar a segurança das propriedades para uma ou mais execuções, de tal modo que se pode detectar a violação das propriedades. Muitas ferramentas utilizam análise dinâmica, por exemplo, criadores de perfis de programas, *model checkers* e visualizadores de execução (Nethercote, 2004). As ferramentas que executam a análise dinâmica devem instrumentar[1] o programa com análises de código.

Estas duas abordagens são complementares. Segundo Ernst (2003), análise estática pode ser uma boa opção, se for considerado que ela executa todos os caminhos no programa, quando análise dinâmica somente considera um único caminho de execução. Todavia, a análise dinâmica é

[1] O verbo instrumentar é altamente utilizado na literatura para se referir a ação de adicionar código extra ao programa.

tipicamente mais precisa que a análise estática por ela trabalhar com valores reais em tempo de execução (Desoli *et al.*, 2002).

A análise dos programas também pode ser categorizada em outros dois grupos, de acordo com o tipo de código a ser analisado.

1. **Análise do código fonte**, segundo Nethercote (2004), envolve a análise de programas no nível de código fonte. Muitas ferramentas executam a análise de código fonte, os compiladores são um bom exemplo. Esta categoria de análise de código fonte do programa é realizada em representações que são derivados diretamente do código fonte, tal como gráficos de fluxo de controle. Análise do código fonte é geralmente feita em termos de estruturas da linguagem de programação, tais como, funções, declarações, expressões e variáveis.

2. **Análise binária**, segundo Nethercote (2004), envolve a análise de programas no nível de código de máquina, armazenado tanto como código objeto (*pre-linking*) ou código executável (*post-linking*). Normalmente a análise binária é feita em termos nas propriedades de código de máquina, tais como procedimentos, instruções, registros e em posições de memória.

Estas outras duas categorias também são complementares, a principal diferença entre elas é que a análise de código fonte é independente de plataforma (arquitetura e sistema operacional), mas dependente de uma linguagem específica, enquanto a análise binária é independente de linguagem, mas dependente de plataforma (Nethercote, 2004). Uma das principais vantagens da análise de código fonte é que ela tem acesso ao alto nível da informação, no qual pode ser mais poderoso, se comparado a análise binária que acessa ao baixo nível (tais como os resultados das posições de memória).

2.3 Verificação Formal com *Model Checking*

Model Checking é uma técnica baseada em métodos formais utilizadas para a análise e verificação de sistemas. Os métodos formais são linguagens matemáticas, onde por meios de técnicas e ferramentas é possível especificar e verificar tais sistemas (Clarke e Wing, 1996). *Model Checking* é uma técnica automática que gera uma busca exaustiva no espaço de estados do modelo, para determinar se uma dada propriedade é válida ou não (Baier e Katoen, 2008). O uso de métodos formais colabora significativamente para a compreensão do sistema, revelando inconsistências, ambiguidades e incompletudes que poderiam passar despercebidas (Clarke e Wing, 1996).

A verificação de um sistema é um aspecto fundamental para a aplicação de métodos formais. Esta verificação consiste em fazer um modelo do sistema que contenha suas propriedades mais relevantes (Fisteus, 2005). Segundo Clarke *et al.* (1999), aplicação de testes em um modelo consiste basicamente em:

- **Modelagem** é a primeira tarefa, converter um projeto em um formalismo aceito por uma ferramenta de verificação de modelos. Em muitos casos, isso é simplesmente uma tarefa de compilação, em outros casos, devido às limitações do tempo e da memória, a modelagem de um trabalho pode exigir a utilização da abstração de informações, para eliminar detalhes irrelevantes ou sem importância.

- **Especificação** é necessária para determinar as propriedades que o modelo deve satisfazer, geralmente é dado em algum formalismo lógico. Para *hardware* e sistemas de *software*, é comum usar a lógica temporal, que pode valer como o comportamento do sistema.

- **Verificação** efetua a verificação do sistema tendo como base as etapas anteriores. Em caso de o resultado da verificação ser negativo, muitas vezes é fornecido ao usuário o rastreamento do erro, este pode ser usado como um contra-exemplo para a propriedade testada e pode ajudar o projetista no rastreamento do erro. A análise do rastreamento do erro pode exigir uma modificação no sistema e a reaplicação do algoritmo de verificação de modelo.

O conceito de *model checking* engloba um conjunto de algoritmos eficientes que permitem verificar propriedades de modelos com um número finito de estados, que normalmente são expressos mediante lógica temporal. Como exemplo, a LTL (*Linear Temporal Logic*), PCTL (*Probabilistic Computation Tree Logic*) e a lógica CTL *(Computation Tree Logic)* (Baier e Katoen, 2008). Segundo Clarke *et al.* (1999), a lógica CTL baseia-se na lógica proposicional de ramificação do tempo, ou seja, uma lógica onde o tempo pode evoluir para mais do que um possível futuro, usando um modelo de tempo discreto. As fórmulas em CTL são compostas por: proposições atômicas, conectivos booleanos e operadores temporais. Os operadores temporais consistem em operadores de tempo futuro (Clarke *et al.*, 1999):

- **G** (*always* ou *globally*) especifica que uma propriedade é válida em todos os estados no caminho;

- **F** (*eventually* ou *in the future*) é usada para afirmar que a propriedade estará válida em algum estado no caminho;

- **X** (*next time*) exige que a propriedade esteja válida no segundo estado no caminho;

- **U** (*until*) assegura se existe um estado ao longo do caminho onde a segunda propriedade está válida e em cada estado anterior no caminho, a primeira propriedade é assegurada;

- **R** (*release*) requer que a segunda propriedade seja válida ao longo do caminho até incluir o primeiro estado onde a primeira propriedade é válida.

Os operadores de tempo são precedidos por quantificadores (**A** - todas as execuções, **E** - existe uma execução), conforme demonstrado na Figura 2.2, para os operadores **G**, **F** e **X** que são normalmente os mais utilizados. Em CTL, o tempo não é mencionado explicitamente. Os operadores temporais apenas permitem descrever propriedades em termos de “no próximo”, “eventualmente” ou “sempre”.

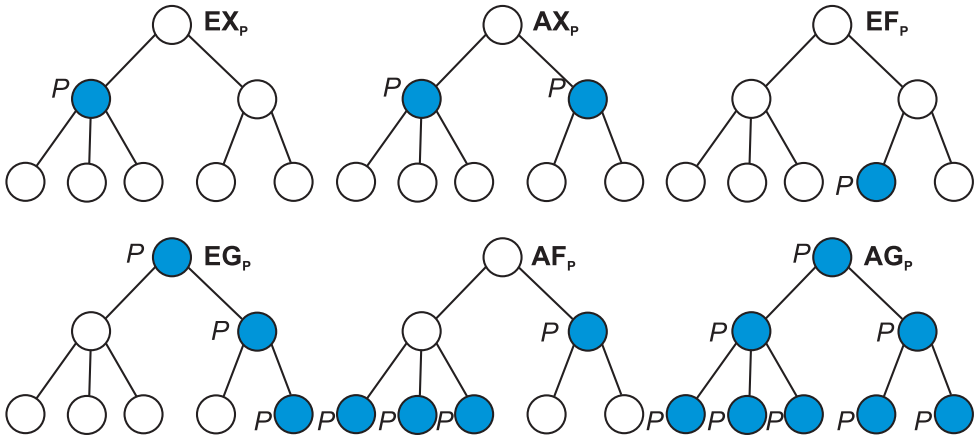


Figura 2.2: Operadores Temporais

2.3.1 *Bounded Model Checking* com ESBMC

A ideia básica do BMC é verificar (a negação de) uma dada propriedade em uma dada profundidade: dado um sistema de transições M, uma propriedade ϕ, e um limite (bound) k, o BMC desenrola o sistema k vezes e traduz o sistema em uma condição de verificação (CV) ψ tal que ψ é satisfeita se e somente se ϕ tem um contra-exemplo de profundidade menor ou igual a k. Os verificadores SAT padrões podem ser usados para verificar se ψ é satisfeito (Cordeiro *et al.*, 2012a).

A fim de lidar com o complexo crescimento dos sistemas, existem os SMT (*Satisfiability Modulo Theories*) *solvers* que podem ser usados para resolver as CVs geradas a partir das instâncias do BMC. Segundo De Moura e Bjørner (2008), o SMT decide a satisfabilidade de fórmula de primeira ordem usando a combinação de diferentes teorias de bases e assim generalizar a satisfabilidade proposicional provendo suporte a funções não interpretadas, aritmética, *bit-vectors*, tuplas, *arrays* e outras teorias decidíveis de primeira ordem.

Segundo Cordeiro *et al.* (2009), ESBMC é um *bounded model checker* para software embarcados em ANSI-C baseado em *SMT solvers*, no qual permite: (i) verificar *software* de uma única e múltiplas *threads* com variáveis compartilhadas e alocadas (Cordeiro e Fischer, 2011); (ii) verificação aritmética de *underflow* e *overflow*, segurança de ponteiros, limites de *arrays*, atomicidade, *deadlock*, *data race* e assertivas providas pelo usuário; (iii) verificar programas que fazem o uso de *bit-level*, ponteiros, *structs*, *unions* e aritmética de ponto fixo. ESBMC utiliza uma versão modificada do CBMC no *front-end* para analisar o código ANSI-C e para gerar as CVs através de execução simbólica. Vale dizer que o ESBMC tem apresentado resultados significativos na verificação de programas em C, como se pode observar pelas recentes participações do ESBMC na Competição Internacional de Verificação de Software (SV-COMP): em 2012 (Beyer, 2012; Cordeiro *et al.*, 2012b) obtendo o 1º lugar nas categorias *Concurrency* e *SystemC*, e o 3º lugar na classificação geral; em 2013 (Beyer, 2013; Morse *et al.*, 2013) obtendo o 2º lugar nas categorias *BitVectors* e *Loops*, e o 3º lugar na classificação geral; em 2014 (Beyer, 2014; Morse *et al.*, 2014) obtendo o 1º lugar na categoria *Sequentialized*; e em 2015 (Beyer, 2015) obtendo o 1º lugar nas categorias *BitVectors* e *Sequentialized*.

No ESBMC, o programa a ser analisado é modelado como um sistema de transição de estados $M = (S, R, S_0)$, o qual é extraído de um grafo de fluxo de controle (GFC). S representa o conjunto de estados, $R \subseteq S \times S$ representa o conjunto de transições (ou seja, os pares de estados que especificam como o sistema pode mudar de estado para estado) e $S_0 \subseteq S$ representa o conjunto de estados iniciais. Um estado $s \in S$ consiste do valor do contador do programa *pc* e os valores de todas as variáveis dos programas. Um estado inicial s_0 atribui a localização inicial do programa do GFC ao *pc*. ESBMC identifica cada transição $\gamma = (s_i, s_{i+1}) \in R$ entre dois estados s_i e s_{i+1} com uma fórmula lógica $\gamma = (s_i, s_{i+1})$ que captura as restrições sobre os valores correspondentes do contador do programa e das variáveis do programa (Cordeiro *et al.*, 2012a).

Segundo Cordeiro *et al.* (2012a), dado o sistema de transição M, uma propriedade de segurança ϕ, um limite de contexto C e um limite k, ESBMC constrói uma *árvore de alcançabilidade* (AA) que representa o desdobramento do programa para C, k e ϕ. ESBMC então deriva uma condição de verificação ψ_k^π para cada *interleaving* dado (ou caminho computado) $\pi = \{\nu_1, \ldots, \nu_k\}$ tal que ψ_k^π é satisfeito se e somente se ϕ tem um contra-exemplo de profundidade k que é exibido por π. ψ_k^π é dado pela seguinte fórmula lógica:

$$\psi_k^\pi = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{2.1}$$

onde a função I caracteriza o conjunto dos estados iniciais de M e $\gamma(s_j, s_{j+1})$ é a relação de transição de M entre as etapas de vezes de j e $j+1$. Desta forma, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ representa as execuções de M de tamanho i e ψ_k^π pode ser satisfeito se somente se para algum $i \leq k$ se existir um estado alcançável ao longo de π em i etapas no qual ϕ é violado. ψ_k^π é uma fórmula livre de quantificadores em um subconjunto decidível de lógica de primeira ordem, o qual é verificada a satisfabilidade por um solucionador SMT (*satisfiability modulo theories*). Se ψ_k^π é satisfeito, então ϕ é violado ao longo de π e o solucionador SMT prove uma atribuição satisfatória, do qual o ESBMC pode extrair os valores das variávies do programa para construir um contra-exemplo.

Um contra-exemplo é um rastreamento que demonstra que uma dada propriedade não esta segura no modelo (Baier e Katoen, 2008). O contra-exemplo permite ao usuário: (i) analisar a falha; (ii) entender a raiz do erro; e (iii) corrigir tanto a especificação ou modelo. Um contra-exemplo para uma propriedade ϕ é uma sequência de estados $s_0, s_1, \ldots, s_k$ com $s_0 \in S_0$, $s_k \in S$, e $\gamma(s_i, s_{i+1})$ para $0 \leq i < k$. Se ψ_k^π não é satisfeito, pode-se concluir que nenhum estado de erro é alcançável em k etapas ou menos ao longo de π. Finalmente, pode-se definir $\psi_k = \bigwedge_\pi \psi_k^\pi$ e utilizá-lo para verificar todos os caminhos.

2.3.1.1 Verificação de Propriedades LTL com ESBMC

Segundo Morse *et al.* (2015), muitos requisitos importantes sobre o comportamento do *software* podem ser expressos mais naturalmente como propriedades *liveness* em uma lógica temporal. Por exemplo: sempre que um dado botão é pressionado a carga eventualmente excede um nível mínimo,

neste caso, se as variáveis `pressed`, `charge`, e `min` de um programa representam o estado do botão, e o atual e mínimo nível da carga, respectivamente, este requisito pode ser modelo na seguinte fórmula LTL $G(\{pressed\} \rightarrow F\{charge{>}min\})$. Neste sentido, o ESBMC também provê suporte a verificação de propriedades expressas em *Linear Temporal Logic* (LTL). Este suporte consiste em converter a negada de uma fórmula em LTL (também denominada de *never claim* (Holzmann, 1997)) em um Autômato de Büchi (que é um autômato de estados finitos sobre palavras infinitas (Büchi, 1962)) que é composto com o programa em análise. Se o sistema composto admitir uma execução, então o programa analisado viola o requisito especificado.

As fórmulas LTL são definidas sobre proposições primitivas, operadores lógicos e operadores temporais da seguinte forma:

$$\begin{aligned} \varphi, \psi ::= \ & true \mid false \mid p \mid \neg\varphi \mid \varphi \vee \psi \\ & \mid X\varphi \mid F\varphi \mid G\varphi \mid \varphi U\psi \mid \varphi R\psi \end{aligned}$$

Sendo p uma expressão sobre as variáveis globais do programa em análise. Nas anotações LTL do ESBMC, estas expressões devem ser colocados entre colchetes e são tratados como valores verdadeiros de acordo com a semântica da linguagem C. Os operadores temporais são "no próximo estado" ou *next*(X), "em algum futuro estado" ou *eventually*(F), "em todos os futuros estados" ou *globally*(G), *until* (U), e *release* (R). $\varphi U\psi$ significa que φ deve estar continuamente assegurado até ψ está assegurado. ψ deve eventualmente se tonar verdadeiro. $\varphi R\psi$ significa que ψ deve está assegurado agora e continuar assegurado até φ se tornar verdadeiro também, ou para sempre, se φ nunca se tornar verdadeiro (Morse *et al.*, 2015).

2.3.1.2 Verificação de Propriedades usando K-indução no ESBMC

Uma técnica usualmente adotada para provar propriedades, para qualquer dada profundidade em um sistema, é a indução matemática. Uma variante denominada de k-indução tem sido aplicada com sucesso para garantir que os programas em C não contenham corridas de dados (Donaldson *et al.*, 2010), e em relação as restrições de tempo especificadas durante a fase de concepção de um sistema Eén e Sörensson (2003). Adicionalmente, a k-indução é uma técnica bem estabelecida na verificação de *hardware*, em que é mais fácil de ser aplicado, devido à relação de transição monolítica presente em projetos de *hardware* Eén e Sörensson (2003).

No ESBMC, o algoritmo de k-indução (na Figura 1) adota uma abordagem de profundidade iterativa e verifica, para cada k etapa até um valor máximo, três casos diferentes chamados de caso base, condição de avanço, e passo indutivo (Ramalho *et al.*, 2013; Rocha *et al.*, 2015b). O algoritmo de k-indução do ESBMC, na Figura 1, recebe como entrada P um programa em C juntamente com a propriedade de segurança ϕ. O algoritmo retorna *TRUE* (se não há nenhum caminho que viola a propriedade de segurança), *FALSE* (se existe um caminho que viola a propriedade de segurança), e *UNKNOWN* (se não foi possível computar uma resposta *TRUE* ou *FALSE*).

No caso base, objetiva-se encontrar um contra-exemplo de ϕ (uma propriedade) em até k interações do *loop*. A condição de avanço, onde o ESBMC verifica se os laços foram totalmente desenrolados e que a propriedade de segurança ϕ é mantida (é verdadeira) em todos os estados alcançáveis dentro de k desenrolamentos. O passo indutivo que o ESBMC verifica que, quando ϕ é válido para k desenrolamentos, ϕ também se mantém válido após o próximo desenrolamento do sistema (Rocha *et al.*, 2015b).

```
   Input: programa P' com invariantes e propriedades de segurança φ
   Output: TRUE, FALSE, ou UNKNOWN
 1 begin
 2   k = 1;
 3   while k <= iteracoes_maxima do
 4     if casoBase (P, φ, k) then
 5       apresenta o contra-exemplo s[0...k] ;
 6       return FALSE ;
 7     end
 8     else
 9       k=k+1 if condicaoAvanco (P, φ, k) then
10         return TRUE ;
11       end
12       else
13         if passoIndutivo (P, φ, k) then
14           return TRUE ;
15         end
16       end
17     end
18   end
19   return UNKNOWN;
20 end
```

Algoritmo 1: Algoritmo de k indução do ESBMC.

2.3.2 Propriedades de Segurança

Informalmente pode-se dizer que uma propriedade em tempo linear especifica o comportamento admissível (ou desejado) de um sistema (Baier e Katoen, 2008), por exemplo, a temperatura de um reator nuclear nunca deve exceder a 100^0C. Se um sistema falha ao satisfazer uma propriedade de segurança, então existe uma execução finita que revela esta fato. Assim a verificação da corretude do sistema com respeito às propriedades de segurança é um meio para validar o comportamento do sistema.

Uma propriedade de segurança (do inglês *safety property* (Baier e Katoen, 2008)) é definida segundo Clarke *et al.* (2003) como: *dado um sistema de transições* $ST = (S, S_0, E)$*, seja um conjunto* $B \subset S$ *que especifica um conjunto de maus estados tais que* $S_0 \cap B = \emptyset$*, pode-se dizer que* ST *é seguro com relação a B, denotada por* $ST \models \mathbf{AG}\neg B$ *se não existe um caminho no*

sistema de transição do estado inicial S_0 até o mau estado B. De outro modo é dito que ST não é seguro, denotado por $ST \not\models \boldsymbol{AG}\neg B$.

Neste trabalho, uma propriedade é definida com uma condição de verificação (CV), uma vez que se uma CV é válida então o programa satisfaz sua corretude. O ESBMC gera condições de verificação para as seguintes propriedades:

1. CVs para a checagem aritmética de *underflow* e *overflow* seguindo o padrão ANSI-C, ou seja, a codificação aritmética para um *overflow* do tipo *unsigned integer* (ou seja, *unsigned int, unsigned long int*) é verificada sobre a sua condição de contorno sobre a adição, subtração, multiplicação, divisão e operações de negação;

2. CVs para os limites da indexação dos *arrays*, ou seja, checa se a operação com os índices do *array* estão fora do limite definido;

3. CVs para segurança de ponteiros, ou seja, verifica se o ponteiro [2] faz referências a um objeto correto (representado por *SAME_OBJECT*) e também checa se o ponteiro é `NULL` ou um objeto inválido (representado por *INVALID_POINTER*);

4. CVs para alocação de memória dinâmica, o ESBMC verifica os argumentos para qualquer função *malloc, free* ou operação de *dereferencing* é um objeto dinâmico (representado por *IS_DYNAMIC_OBJECT*) e se o argumento para qualquer *free* ou operação de *dereferencing* é ainda um objeto válido (*VALID_OBJECT*).

2.4 Invariantes de Programas

Uma invariante de um programa é uma declaração das condições que devem ser verdadeiras para qualquer execução do programa que alcançar aquela localização (Chen *et al.*, 2011). Em outras palavras, invariantes podem ser definidas como uma relação proeminente entre as variáveis do programa. Segundo Fouladgar *et al.* (2011), invariantes em programas são fórmulas ou regras que surgiram a partir do código fonte de um programa e permanecem únicas e inalteradas em relação à fase de execução de um programa com diferentes parâmetros.

Formalmente, segundo Bi *et al.* (2010) pode-se definir invariantes a partir de um tipo de sistema de transição (Manna e Pnueli, 1995), onde este é o sistema de transição semi-algébrico (Chen *et al.*, 2007). Desta forma, dado que um sistema de transições é uma 5-tupla $\langle V, L, T, \ell_0, \Theta\rangle$, onde: V é um conjunto de variáveis; L é um conjunto de localizações; T é um conjunto de transições. Um estado s é uma tripla $\langle \ell_1, \ell_2, P_\tau\rangle$, onde ℓ_1 e ℓ_2 são respectivamente as pré-condições e pós-condições da localização das transições. A relação P_τ é uma assertiva de primeira ordem (ver Seção **??**) sobre $V \cup V'$, onde V denota o estado atual da variável e V' denota o próximo estado da variável; $\ell_0 \in L$ é

[2] Nota-se que a linguagem ANSI-C oferece dois operadores diferenciados o $*p$ e $p[i]$, onde p representa um ponteiro (ou *array*) e i representa um índice inteiro, ambos são atendidos.

o estado inicial da localização; Θ é uma assertiva de primeira ordem sobre V denotando a condição inicial.

Um sistema de transição semi-algébrico é um sistema de transição $\langle V, L, T, \ell_0, \Theta \rangle$, onde para cada transição $\tau \in T$ é uma quadrupla $\langle \ell_1, \ell_2, P_\tau, \Theta_\tau \rangle$, onde ℓ_1, ℓ_2 e P_τ possuem o mesmo conceito definido acima, Θ_τ é o guarda da transição. Somente se Θ_τ assegurar a transição que a transição pode ser efetuada. A transição P_τ é uma assertiva algébrica (Sankaranarayanan *et al.*, 2004) sobre $V \cup V^{'}$, e a condição inicial Θ e o guarda da transição Θ_τ é uma assertiva algébrica sobre V o qual contém inequações e desigualdades polinomiais e a condição inicial Θ é uma assertiva algébrica sobre V.

Finalmente segundo Bi *et al.* (2010), pode-se definir formalmente as invariantes da seguinte forma, seja $\Psi = \langle V, L, T, \ell_0, \Theta \rangle$ um sistema de transição semi-algébrico. Uma invariante em uma localização $\ell \in L$ é definida como uma assertiva φ sobre V , tal que φ é mantida sobre todos os estados que podem ser alcançados na localização ℓ. Uma invariante é uma assertiva φ o qual é mantida em todas as localizações do sistema de transição semi-algébrico.

2.4.1 *Templates* de Invariantes

Segundo Beyer *et al.* (2007b), um *template* de invariante é uma assertiva parametrizada fixada sobre variáveis de programas. Um *template* identifica os parâmetros desconhecidos, e restringem a "dimensão" das invariantes, por exemplo, o número de conjunções e o número de aplicações de funções. Isto significa que a forma do *template* determina a forma das invariantes resultantes. Formalmente, sendo que c varia sobre o conjunto de constantes inteiros, υ sobre o conjunto de variáveis de programa, $\int$ sobre um conjunto fixo de símbolos de funções não interpretadas e α sobre um conjunto fixo de parâmetros de *template*. A gramatica a seguir define o conjunto de restrições de *templates*:

Termos $t ::= \upsilon \mid \int(e_1, \ldots, e_n)$
Expressões $e ::= c \mid c \times t \mid \alpha \times t \mid e_1 + e_2 \mid e_1 - e_2$
Restrições $i ::= e \leq c \mid e_1 = e_2$
Templates $\xi ::= i \mid i \wedge \xi$

Um *template* de invariantes é uma conjunção finita de desigualdades. Uma invariante é expressa por um *template* de invariantes se existe uma avaliação dos parâmetros do *template* que produz a invariante (Beyer *et al.*, 2007b).

2.5 Interpretação Abstrata de Programas

A interpretação abstrata de programas consiste no uso de denotações para descrever computações em outro universo de objetos abstratos. Desta forma, os resultados da execução abstrata provêm

algumas informações sobre a computação atual. 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 $-1515 * 17 \Rightarrow -(+) * (+) \Rightarrow (-) * (+) \Rightarrow (-)$ 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 (Cousot e Cousot, 1977).

Segundo Cousot (2001), um entendimento mais restrito de interpretação abstrata é vê-la como uma teoria de aproximação do comportamento dinâmico de sistemas discretos (onde 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 (Cassandras e Lafortune, 2008), 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 $S[\![p]\!] \in \mathcal{D}$ em um domínio semântico $\mathcal{D}$ para cada programa p de Ł. O domínio semântico $\mathcal{D}$ pode ser sistemas de transição (Manna e Pnueli, 1995), traços, relações, funções de alta ordem e assim por diante. $\mathcal{D}$ é geralmente definido, em termos de composição, por indução sobre estruturas de objetos em tempo de execução (computações, dados e outros). S é geralmente definido, em termos de composição, por indução sobre estruturas sintáticas de programas, por exemplo, 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. Com base em Cousot e Cousot (2010), 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).

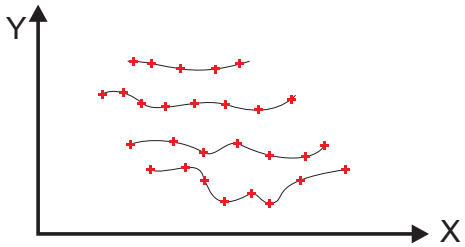


Figura 2.3: Conjunto de rastreamentos

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 $\{(x_i, y_i) : i \in \Delta\}$, que formaliza o método de prova de Floyd/Naur e a lógica de Hoare (Cousot, 2002).

- **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 (Cousot e Cousot, 1979). Assim invariantes locais são expressas com a seguinte forma $x \geq 0,\ y \geq 0$.

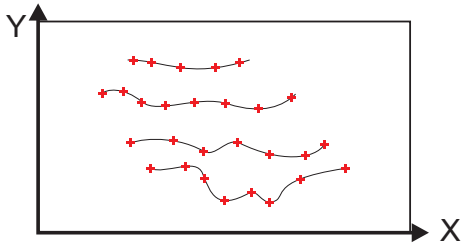


Figura 2.4: Abstração simples sobre o conjunto de rastreamentos

- **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 (Cousot e Cousot, 1977).

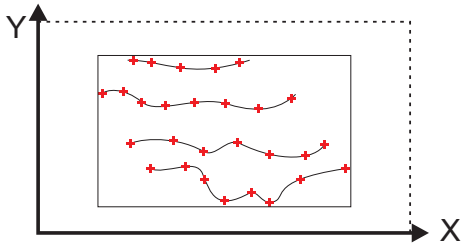


Figura 2.5: Abstração de Intervalos sobre o conjunto de rastreamentos

- **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 (Miné, 2006).

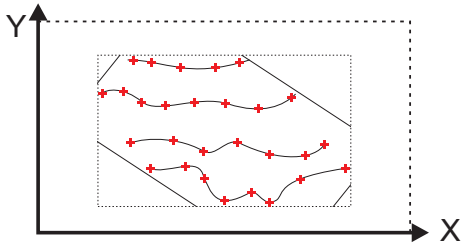


Figura 2.6: Abstração de Octógonos sobre o conjunto de rastreamentos

- **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 (Cousot e Halbwachs, 1978).

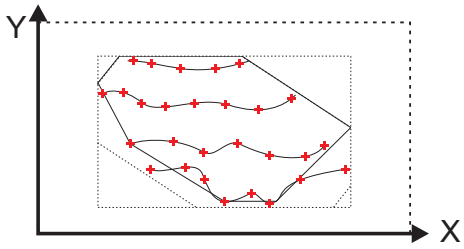


Figura 2.7: Abstração de Poliedros Convexos sobre o conjunto de rastreamentos

- **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 (Feret, 2005b).

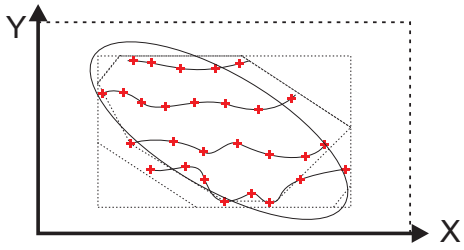


Figura 2.8: Abstração de Elipsoides sobre o conjunto de rastreamentos

- **Abstração Exponencial**: Esta é outro tipo de abstração não linear, onde as invariantes tem a forma $a^x \leq y$ (Feret, 2005a).

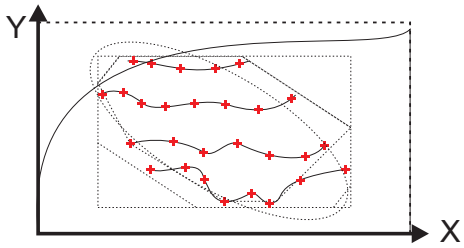


Figura 2.9: Abstração Exponencial sobre o conjunto de rastreamentos

2.6 Resumo

Este capítulo descreveu os principais conceitos necessários para o entendimento deste trabalho. Na Seção 2.1 foi definido teste e verificação formal de *software* que são atividades de verificação e validação, no processo de desenvolvimento, visando a garantia da qualidade. Na Seção 2.2 foi contextualizado as principais estratégias de análise de programas que são a análise dinâmica e estática, também é ressaltado que a análise dos programas também podem ser categorizada de acordo com o tipo de código a ser analisado, ou seja, análise de código fonte e binário. Na Seção 2.3 foi introduzida verificação formal utilizando *model checking* e abordou-se o uso de métodos

formais como uma estratégia para compreensão do sistema, identificação de inconsistências, ambiguidades e incompletude que poderiam passar despercebidos. Adicionalmente, na Seção 2.3.1 foi definido *bounded model checking* e apresentado sua aplicação com o *model checker* ESBMC, e por fim na Seção 2.3.2 foi explicado o que são propriedades de seguranças e contextualizado suas aplicações no *model checker* ESBMC. Na Seção 2.4 foi definido invariantes de programas como uma propriedade em um local do programa que é uma declaração das condições que devem ser verdadeiras, para qualquer execução do programa que alcançar aquela localização, bem como formalizado a definição de invariantes de programas a partir de um sistema de transição semi-algébrico. Adicionalmente, na Seção 2.4.1 foi apresentado o conceito de *Templates* de Invariantes que consiste em uma assertiva parametrizada fixada sobre variáveis de programas. E por fim na Seção 2.5 foi introduzido o conceito de interpretação abstrata de programas que consiste no uso de denotações para descrever computações em outro universo de objetos abstratos, e então é classificada interpretação abstrata nos seguintes tipos: Abstração Simples, Abstração de Intervalos, Abstração de Octógonos, Abstração de Poliedros Convexos, Abstração de Elipsoides e Abstração Exponencial.

Capítulo 3

Trabalhos Correlatos

Neste capítulo serão apresentados e discutidos os principais trabalhos existentes na literatura que se relacionam com os tópicos que formam a base para o desenvolvimento deste trabalho, com ênfase em: Geração de Casos de Teste, Verificação e Análise de Memória de Programas, Contra-exemplos de *Model Checkers* e Invariantes de Programas.

3.1 Geração de Casos de Teste

Na literatura existem diversos métodos e ferramentas para a geração de casos de teste. O GAteL (Marre e Arnould, 2000) é um método que tem como base um SUT (*System Under Test*) ou uma completa especificação do SUT, uma descrição do ambiente, um objetivo do teste, quais tipos de propriedades verificar (*safety* ou *liveness*), a fim de gerar os casos de teste. Todos os elementos devem ser fornecidos tendo como base a linguagem Lustre (Caspi *et al.*, 1987). Outros métodos e ferramentas que contribuem neste ambiente de geração de casos de teste são o AsmL (Barnett *et al.*, 2003), TorX (Tretmans e Brinksma, 2003), TestComposer e AutoLink (Schmitt *et al.*, 2000) que são fortemente ligadas a um domínio ou uma linguagem específica.

Segundo Barnett *et al.* (2003), o AsmL (*Abstract State Machine Language*) é uma linguagem de modelagem executável que está plenamente integrado ao *framework* `.NET` e outras ferramentas de desenvolvimento da Microsoft. O AsmL suporta meta-modelagem baseada em reflexão computacional, que permite uma exploração sistemática não-determinística no modelo. O TorX integra a geração automática de teste, execução de teste e análise de prova. TorX é baseado na teoria de teste **ioco**, que tem suas origens na teoria dos testes e recusa de equivalência para sistemas de transição (Tretmans e Brinksma, 2003). Segundo Schmitt *et al.* (2000), o Autolink e TestComposer são totalmente integrados em seus correspondentes ambientes de desenvolvimento. O TestComposer é construído em cima do simulador ObjectGeode (Kerbrat *et al.*, 1999); Autolink

é parte do validador Tau[1]. O simulador, assim como o validador é usado para localizar erros e inconsistências dinâmicas nas especificações SDL (*Specification and Description Language*).

Este trabalho visa complementar com este conjunto de soluções já existentes, tendo como domínio específico a linguagem de programação C no qual, na maioria das vezes, não é explorada ou contemplada. Deste modo a metodologia proposta utiliza o próprio código como base para a geração dos casos de teste, ou seja, sem a necessidade de uma modelagem extra.

No trabalho de Cadar e Engler (2005) é apresentado uma técnica que usa o código para gerar automaticamente os seus próprios casos de teste em tempo de execução pela utilização da combinação de execução simbólica e concreta (ou seja, regular). A sua principal contribuição é a análise de observação no código, para automaticamente gerar os seus próprios e potenciais casos de teste. Instanciando e executando o código com entradas concretas, construídas manualmente, esta técnica executa a sua própria entrada simbólica. Esta abordagem diz quais são os valores válidos (ou intervalos de valores) para a entrada no código, e assim gerar os casos de teste pela solução dessas restrições para valores concretos. Estes testes são chamados de *Execution Generated Testing* (EGT).

O trabalho aqui apresentado, assim como o trabalho de Cadar e Engler (2005), visa explorar o código detectando possíveis propriedades a serem testadas, utilizando técnicas e métodos de análise e verificação de código. Todavia, fazendo uma análise de comparação com o trabalho de Cadar e Engler (2005), neste trabalho não se visa a utilização de entradas simbólicas, pois em Cadar e Engler (2005) não é levado em conta os fatores de limitação do solucionador de satisfabilidade e memória utilizada. Neste trabalho se utiliza *Bounded Model Checkers* para identificar as propriedades e a partir destas propriedades então gerar os casos de testes.

3.2 Verificação e Análise de Memória de Programas

Abordagens relacionadas à verificação e análise de defeitos para a segurança da utilização de memória de programas tem sido um tema abordado por diferentes trabalhos ((Clause e Orso, 2010; Clarke *et al.*, 2004a; Cordeiro *et al.*, 2012a; Merz *et al.*, 2012; Dhurjati *et al.*, 2003; Maebe *et al.*, 2004; Nethercote e Seward, 2007; Beyer e Keremoglu, 2011; Dudka *et al.*, 2014)), isto porque tais defeitos são difíceis de detectar, uma vez que estes defeitos possuem características próprias da execução do programa e há dificuldade para criar um modelo que represente corretamente o gerenciamento dos endereços de memória.

O trabalho de Dhurjati *et al.* (2003) apresenta uma abordagem baseada em análise estática de segurança de memória para programas embarcados através de técnicas de compiladores e restrições semânticas sobre programas. As restrições semânticas são definidas, em termos de linguagem independente de baixo nível, e a verificação de segurança de compilador é implementada inteiramente em uma infraestrutura de compilador de linguagem independente

[1] http://www.itm.mu-luebeck.de/

chamada de LLVM (*Low Level Virtual Machine*) (Lattner, 2002). Vale ressaltar que a abordagem assume determinadas restrições sobre o programa analisado, por exemplo, que ele seja *single-threaded*. Este trabalho proposto também visa explorar técnicas de compiladores para este tipo de verificação, contudo a abordagem difere pelo fato de atualmente se utilizar análise dinâmica e se poder analisar programas *multi-threaded*, isto devido ao fato do ESBMC prover este suporte. Vale ressaltar que o ESBMC é uma ferramenta que está no estado da arte em verificação de código *multi-threaded* (Cordeiro *et al.*, 2012a; Beyer, 2012, 2013; Morse *et al.*, 2015).

No trabalho de Maebe *et al.* (2004) é descrito uma abordagem baseada em instrumentação dinâmica, que ocorre em nível de código de máquina e técnicas de coletor de lixo, que permite informar ao usuário onde a última referência para um bloco de memória foi perdido e onde ele estava perdido (identificação de vazamento de memória), sem a necessidade de recompilação ou *relinking*. Em relação a este trabalho proposto, também se utiliza a estratégia de rastrear todos os ponteiros que apontam para blocos de memória, contudo também são rastreados os outros tipos de variáveis, pois isto permite ao método proposto identificar, por exemplo, se o programa faz referência a um endereço inválido. Outro item que se tem em comum com a abordagem de Maebe *et al.* (2004) é que ao final do programa é apresentado ao usuário uma lista com os blocos de memória não liberados, porém difere pela utilização da instrumentação de código fonte.

A ferramenta Valgrind/Memcheck usa um *framework* de instrumentação binária dinâmica (Seward e Nethercote, 2005; Nethercote e Seward, 2007) para verificar endereços inválidos e vazamentos de memória em programas baseados em sua execução. Comparando Nethercote e Seward (2007) com o Map2Check, se tem em comum que a execução do código de instrumentação adicional é intercalada com a execução normal do programa; assim, preserva o comportamento normal do programa. No entanto, o Map2Check adota o ESBMC para gerar as *claims* (ou seja, as condições de verificação) por análise estática. Além disso, o Map2Check fornece uma biblioteca de funções que poderiam ajudar os desenvolvedores a estender os testes gerados pelo Map2Check no código fonte analisado. Por exemplo, utilizando funções para validar ponteiros fornecidos pela biblioteca em outros pontos do programa.

FSHELL é um ambiente de teste de caixa-branca de programas em C que suporta tanto o uso interativo e exploratório em uma linguagem de *script* (Holzer *et al.*, 2008). FSHELL usa uma linguagem de consulta adaptados para a análise do programa e despacha consultas sobre os caminhos dos programas para as ferramentas de análise de programas. FSHELL adota CBMC como *model checker*. No mesmo sentido, o Map2Check adota um BMC (neste caso o ESBMC) para gerar casos de teste. No entanto, a verificação dos casos de teste não é realizada por um BMC. Diferentemente, é adotado uma análise no rastreamento do fluxo de memória dos endereços de memória.

O trabalho de Clause e Orso (2010) apresenta uma técnica, denominada de LEAKPOINT, que não só detecta vazamento de memória, mas também aponta aos desenvolvedores os locais onde os erros subjacentes podem ser corrigidos. Esta técnica rastreia ponteiros para áreas dinamicamente alocadas de memória e, para cada área de memória, registra diversas informações relevantes. Estas

informações são usadas para identificar os locais em uma execução onde os vazamentos de memória ocorrem. A técnica apresentada é baseada em propagação dinâmica (ou fluxo de informação dinâmica) (Clause *et al.*, 2007) e utiliza o *framework* Valgrind como instrumentador binário. Comparando a técnica LEAKPOINT com o Map2Check, argumenta-se que a principal vantagem é a detecção de outros defeitos relacionados à memória de programa, por exemplo, a identificação de endereços inválidos e *free* inválidos. Além do fato de se utilizar a instrumentação de código fonte, que possibilita controlar exatamente onde se perde a última referência a um bloco de memória no caso de falhas de memória física (Seward e Nethercote, 2005).

CPAchecker é um *framework* para verificação de *software* baseado em configurações de análise de programas (do inglês *Configurable Program Analysis* (CPA)) (Beyer e Keremoglu, 2011; Löwe *et al.*, 2014). Assim, análises auxiliares, tais como monitoramento do contador de programa, modelagem das chamadas da pilha, e manter o controle de ponteiros de função, são implementadas como CPAs independentes. O mesmo ocorre para as análises principais, tais como, por exemplo, a análise do valor explícito e a análise com base na abstração de predicados, o qual também estão disponíveis como CPA dissociados dentro CPAchecker. Para verificar as propriedades de segurança de memória, o CPAchecker adota uma análise limitada que consiste em gráficos de memória concretas em combinação com uma instância de análise de valor explícito.

Em Clarke *et al.* (2004a) é apresentado o CBMC que implementa uma tradução precisa de *bit* a partir de um programa em C dado como entrada, anotado com assertivas e com loops desenrolados (via execução simbólica) até uma dada profundidade em uma formula. Se a formula é satisfeita, então existe uma execução no programa que viola uma dada assertiva. No CBMC propriedades de segurança de memória são reduzidas a assertivas instrumentadas no programa para identificar operações de ponteiros inválidos ou vazamento de memória. Esta instrumentação ocorre via uma análise estática executada pelo CBMC que visa a identificação de ponteiros e suas relações nas funções.

LLBMC é uma ferramenta para detectar erros em tempo de execução em programas em C e C++ (Merz *et al.*, 2012). LLBMC é basedo em *Bounded Model Checking* e SMT *solver*. De acordo com Merz *et al.* (2012), o diferencial do LLBMC entre os BMCs existentes é que o LLBMC opera sobre uma representação intermediaria de compilador e não diretamente sobre o código fonte do programa. No LLBMC a memória é modelada como um *array* de *bytes* sequenciais que difere de outras abordagens. De acordo com Merz *et al.* (2012), normalmente, a modelagem de memória no nível de *byte* causa uma penalidade de desempenho nos BMCs, mas o LLBMC utiliza técnicas de simplificação que, de acordo com avaliações empíricas, esta simplificação compensa a falta de desempenho.

Em Dudka *et al.* (2014) é apresentado o Predator que é um analisador de formas que utiliza o domínio abstrato de *symbolic memory graphs* (SMGs) visando suportar varias formas de manipulação de memória em baixo nível (por exemplo, aritmética de ponteiros e operações com blocos de memória) utilizada em otimizações de código C. Predator requer que todas as funções externas (por exemplo, *malloc* e *free*) sejam modeladas em relação a segurança de memória de

modo a excluir quaisquer efeitos colaterais em relação a execução de sua análise. A execução do Predator consiste em receber como entrada um código C e então executá-lo simbolicamente usando um conjunto de algoritmos para validar as propriedades relacionadas à memória.

O método proposto, assim como os métodos apresentados em Beyer e Keremoglu (2011); Clarke *et al.* (2004a); Merz *et al.* (2012); Dudka *et al.* (2014), visa a análise de propriedades de segurança de memória. Contudo, é diferente de outros verificadores no sentido de não utilizar técnicas de forma completa, como execução simbólica com solucionador de satisfabilidade. Dado que em alguns casos a aplicação da forma completa destas técnicas pode resultar em longos períodos de verificação. Assim, o método proposto visa complementar a verificação efetuada por estes verificadores, ou seja, utilizar as técnicas implementadas pelos verificadores como ponto de partida para a análise e, também, para ampliar a análise dos programas, implementado técnicas de teste, como testes de unidade.

3.3 Contra-exemplos de *Model Checkers*

Existem diferentes aspectos para entender ou analisar um contra-exemplo. Segundo Beer *et al.* (2009), nos últimos anos, o processo de encontrar um erro na fonte de origem tem atraído uma atenção significativa. Muitos trabalhos têm abordado este problema tais como, Copty *et al.* (2001), Jin *et al.* (2004), Riacs *et al.* (2002), Dong *et al.* (2003), Ball *et al.* (2003), Groce (2004), Groce e Kroening (2005), Chechik e Gurfinkel (2005), Shen *et al.* (2005), C.Wang *et al.* (2006), Griesmayer *et al.* (2007), Staber e Bloem (2007), Suelflow *et al.* (2008), abordando a questão de encontrar a causa central da falha no modelo, e propondo meios automáticos de extrair mais informações sobre o modelo, facilitando o processo de depuração.

Os contra-exemplos gerados pelos *model checkers* geralmente possuem um tamanho significativo, o que torna o processo de análise do mesmo uma tarefa muitas vezes complexa. O trabalho de Groce e Kroening (2005) apresenta uma técnica usada para minimizar o tamanho do contra-exemplo gerados por BMC, mas concentra-se na minimização semântica, isto é, no sistema de tipos da linguagem ANSI C. Basicamente, essa abordagem minimiza os valores das variáveis do contra-exemplo. Com relação ao EZProofC visa-se contribuir, assim como em Groce e Kroening (2005), no processo de tratamento de contra-exemplo ao usuário final, neste caso já apresentando os dados abstraídos do mesmo.

No Trabalho de Beer *et al.* (2009), é abordado o problema da análise de contra-exemplo de um determinado erro que ele representa. Usando a noção de causalidade, introduzido por Halpern e Pearl (2000), objetiva explicar o erro ocorrido, definindo um conjunto de causas para a falha da especificação. Estas causas são marcadas como "ponto vermelhos" e apresentados ao usuário como uma explicação visual da falha. Tendo em vista o trabalho de Beer *et al.* (2009), também visa-se contribuir com o entendimento do erro identificado. Contundo, no EZProofC pela reprodução do erro, ou seja, é apresentado ao usuário uma instância executável do contra-exemplo que poderá auxiliar o usuário a identificar soluções para evitar o erro.

Da mesma forma que em (Groce e Kroening, 2005; Beer *et al.*, 2009), visa-se também contribuir com a análise de contra-exemplos gerados na verificação de código pelos BMC, de modo que se o BMC gerar um falso contra-exemplo, se possa validar este através da execução do código com os valores nele apresentados.

3.4 Invariantes de Programas

As invariantes de programas têm sido adotadas como uma estratégia para aumentar a eficiência e precisão na verificação de *software*, principalmente quando aplicadas a técnica *model checking* (Nimmer e Ernst, 2002; Shacham *et al.*, 2005; Pasareanu *et al.*, 2005; Gupta e Rybalchenko, 2009; Donaldson *et al.*, 2010; Yeolekar *et al.*, 2013; Beyer *et al.*, 2015c).

Em Gupta e Rybalchenko (2009) é apresentado o projeto e implementação de InvGen, uma ferramenta automática para gerar invariantes a partir de aritméticas lineares para programas imperativos. InvGen utiliza uma abordagem baseada em restrições para gerar invariantes que combina técnicas de análise estática e dinâmica para resolver eficientemente as restrições. O InvGen transforma um programa de entrada, seja na linguagem C ou como uma relação de transição, em diversas expressões aritméticas lineares. InvGen utiliza um *template* baseado em técnicas para computar invariantes, o qual é dado como entrada para a ferramenta. Como resultado, a ferramenta também retorna uma invariante que prova a não alcançabilidade da localização de um erro ou falha. Vale ressaltar que InvGen utiliza uma coleção de heurísticas para melhorar a escalabilidade da geração de invariantes, e na análise dinâmica a ferramenta pode ser executada usando execução simbólica ou concreta. No método proposto, assim como em Gupta e Rybalchenko (2009) adota-se as invariantes de programas na verificação para gerar os conjuntos de estados alcançáveis que podem ter uma localização de um erro do programa. Contundo, no método proposto, adota-se InvGen apenas para gerar as invariantes de programas e não para provar a não alcançabilidade da localização de um erro.

No trabalho de Donaldson *et al.* (2010) é descrito uma ferramenta de verificação denominada de Scratch para detectar *data races* durante o acesso direto a memória (*Direct Memory Access* (DMA)) em processadores CELL BE da IBM (IBM, 2013). Este abordagem é adota para verificar programas em C usando a técnica de *k* indução que é implementado em Scratch e é executado em duas etapas, o caso base e o passo indutivo. A ferramenta é capaz de provar a ausência de *data races*, mas é restrita à verificação de uma a classe específica de problemas para um tipo particular de *hardware*. Diferentemente, o método proposto neste trabalho adota o ESBMC visando classes genéricas de problemas e não sendo dependente de um tipo específico de *hardware*. Os passos do algoritmo de *k* indução de Donaldson *et al.* (2010) são semelhantes ao adotado pelo ESBMC (Rocha *et al.*, 2015b), mas requer anotações no código para introduzir invariantes de *loops* que difere do método proposto onde as invariantes são geradas de forma automática.

No trabalho de Yeolekar *et al.* (2013) é utilizado a ferramenta Daikon (Ernst *et al.*, 2007) para inferir invariantes a partir do rastreamento de execuções do programa analisado, onde as

invariantes geradas são utilizadas como pós-condições para reduzir o espaço de estados computados para os fragmentos de códigos analisado por Daikon. Dado um programa como entrada, a execução deste método consiste em: gerar casos de teste (baseado em assertivas anotadas no programa) que atendam um determinado critério de cobertura do programa; verificar estes casos de teste com o CBMC, onde os dados do contra-exemplo serão utilizados como dados de rastreamento para a geração de invariantes com Daikon (foi criada uma versão que provê suporte ao CBMC); o corpo das funções do programa são substituídas pelas invariantes geradas como assumes; e finalmente este código com as invariantes é então verificado com o CBMC.

Comparando o trabalho aqui apresentado com o de Yeolekar *et al.* (2013), também se utiliza as invariantes como uma estratégia para melhorar a eficiência da verificação de *model checkers*. Contundo, o método proposto provê suporte a 4 ferramentas (PIPS, InvGen, ASPIC e Daikon) para a geração de invariantes e não somente ao Daikon. Vale ressaltar que foi entrado em contato com os autores de Yeolekar *et al.* (2013) a fim de obter a versão criada do Daikon com suporte a geração das invariantes no formato do CBMC. Contudo, a resposta foi que devido a nova versão do Daikon conter códigos proprietários, ela não poderia ser disponibilizada. Assim, neste trabalho criou-se uma versão modificada da ferramenta Daikon que está disponível em https://bitbucket.org/herberthb12/daikon-acsl. Esta versão modificada do Daikon provê suporte a geração das invariantes no formato ACSL (neste caso podendo ser utilizada a ferramenta Frama-C que provê suporte a análise de programas com anotações neste formato) e com a aplicação do método proposto no formato de expressões suportadas pela linguagem C (ver Seção 6.3.1). No trabalho de Yeolekar *et al.* (2013) também não foi identificado o suporte a invariantes geradas pelo Daikon do tipo $forall$ que são suportadas pela versão modificada (gerada neste trabalho) por meio da utilização do *plug-in* E-ACSL do Frama-C.

Em Yeolekar *et al.* (2013) é utilizado um método para a geração de casos de teste visando melhorar as invariantes geradas por Daikon. Este método gera assertivas que atendam a um determinado critério de cobertura e então identifica valores de testes que satisfaçam estas assertivas. No método proposto apresentado neste trabalho é apresentado duas abordagens (ver Seção 6.2.5 e 6.2.6) sendo uma baseado na ferramenta PathCrawler e outra baseada também em assertivas. Contudo, a baseada em assertivas difere de Yeolekar *et al.* (2013), pois ao invés de se gerar assertivas que atendam a um determinado critério de cobertura (onde esta pode ser uma tarefa difícil), o método adiciona assertivas em cada ponto de decisão do programa tentando assim obter a maior cobertura possível via análise de um BMC. No trabalho de Yeolekar *et al.* (2013) é efetuado a substituição do corpo das funções pelas invariantes geradas. No método proposto não ocorre a remoção de código e as invariantes são apenas adicionadas ao código do programa. Isto porque em parte acredita-se que a não remoção do código possa gerar menos efeitos colaterais para a análise do programa (conforme se observou em testes preliminares), e também evitando a geração de falsos contra-exemplos, como mencionado em Yeolekar *et al.* (2013).

No trabalho de Beyer *et al.* (2015c) é apresentado um método (implementado na ferramenta CPAChecker) que utiliza BMC com k indução combinado com invariantes de programas para detectar a violação de propriedades. As invariantes são utilizadas para fortalecer a hipótese

indutiva. A execução deste método consiste na execução de dois algoritmos concorrentemente. Um algoritmo é responsável pela geração de invariantes de programa, começando com invariantes imprecisas que são continuamente refinadas. O outro algoritmo é responsável por encontrar contra-exemplos com o BMC e construir provas de segurança com k indução, para o qual periodicamente coleta e utiliza as invariantes que o algoritmo anterior tem gerado. Para a geração de invariantes é utilizado um domínio abstrato com base em expressões sobre intervalos. De acordo com Beyer *et al.* (2015c), não é uma exigência do método, uma vez que este funciona com qualquer tipo de domínio.

Da mesma forma que o trabalho de Beyer *et al.* (2015c), segui-se a estratégia de utilização de invariantes com a verificação de programas usando BMC. Contudo, difere pelo fato que se utiliza mais de um algoritmo para a geração de invariantes que são utilizados por 4 diferentes ferramentas (PIPS, InvGen, ASPIC e Daikon) que adotam abordagens e domínios abstratos diferentes (por exemplo, poliédrico por PIPS e ASPIC; e por intervalos e octogonais por InvGen). Adicionalmente, o método proposto provê abordagens para a geração de dados de teste que podem ser utilizados para aprimorar as invariantes geradas pelas ferramentas adotadas (ver Seção 6.2.5 e 6.2.6). Neste método se adota o ESBMC como um BMC.

3.5 Revisão Sistemática sobre Invariantes de Programas

Esta seção tem como objetivo apresentar um levantamento bibliográfico baseado na técnica de revisão sistemática com o objetivo de identificar e conhecer métodos para inferência de invariantes em códigos na linguagem de programação C existentes na literatura, extraindo suas principais características, e avaliando as vantagens e desvantagens de cada abordagem. Um dos principais objetivos desta revisão sistemática é selecionar métodos para serem aplicados neste trabalho.

A técnica de revisão sistemática surgiu na medicina e foi trazida para a engenharia de *software*, quando Kitchenham *et al.* (2004) introduziu o conceito de Engenharia de *Software* Baseada em Evidência (ESBE), com o objetivo de fornecer meios pelos quais as melhores evidências atuais da pesquisa podem ser integradas com experiência prática e valores humanos no processo decisório relativo ao desenvolvimento e manutenção de *software*. Uma revisão sistemática "é um meio de identificar, avaliar e interpretar toda pesquisa disponível e relevante sobre uma questão de pesquisa, um tópico ou um fenômeno de interesse, e tem por objetivo apresentar uma avaliação justa de um tópico de pesquisa, usando uma metodologia confiável, rigorosa e auditável" (Kitchenham, 2004).

A revisão sistemática requer um esforço considerável quando comparada a uma revisão de literatura informal. Enquanto que a revisão de literatura informal é conduzida de forma *ad-hoc*, sem planejamento e critérios de seleção estabelecidos a *priori*, a revisão sistemática segue um protocolo formal para conduzir uma pesquisa sobre um determinado tema, com uma sequência bem definida de passos metodológicos (Mafra e Travassos, 2006). A aplicação de revisão sistemática da literatura requer que seja seguido um conjunto bem definido e sequencial de passos, segundo um protocolo de pesquisa desenvolvido apropriadamente. Este protocolo é construído

considerando um tema específico que representa o elemento central da investigação. Os passos da pesquisa, as estratégias definidas para coletar as evidências e o foco das questões de pesquisa são definidas explicitamente, de tal forma que outros pesquisadores sejam capazes de reproduzir o mesmo protocolo de pesquisa e, também, de julgar a adequação dos padrões adotados no estudo (Biolchini *et al.*, 2005).

O protocolo e o processo da condução da revisão sistemática efetuada neste trabalho são descritos no Apêndice A e em Rocha *et al.* (2013). Desta forma, nesta seção concentra-se em apresentar os principais resultados obtidos com a aplicação da técnica da revisão sistemática. Contundo, visando o melhor entendimento dos resultados obtidos a seguir serão pontuados alguns itens do planejamento para a aplicação da revisão sistemática, estes são: as questões de pesquisa investigadas e o ambiente utilizado. Este levantamento bibliográfico visa responder as seguintes perguntas:

- **Q1**: Quais são os métodos de inferência de invariantes para a verificação de código na linguagem de programação C?
  - **Q1.1**: Foi desenvolvido e está disponível alguma ferramenta para a aplicação do método?
  - **Q1.2**: O método proposto foi gerado a partir da integração com outros?
  - **Q1.3**: Qual é a estratégia (estática, dinâmica ou hibrida) de execução do método?
  - **Q1.4**: Quais foram os resultados positivos ou negativos da validação/experimentação do método?
    - **Q1.4.1**: Foi utilizado algum *benchmark* de programas em C para experimentação do método e este *benchmark* esta disponível?
  - **Q1.5**: Quais as limitações do método proposto?
  - **Q1.6**: Qual é a escalabilidade em termos de linhas de código do método proposto?

O ambiente utilizado para a execução da revisão sistemática foi o de bibliotecas digitais. Logo, optou-se pela biblioteca digital Scopus, acessível em http://www.scopus.com. A biblioteca Scopus permite a consulta e o acesso a diferentes fontes digitais via Web, através de expressões de busca pré-estabelecidas. Segundo a editora Elsevier (2013c), a Scopus é uma das maiores bases de dados de resumos e citações da literatura de pesquisa revisões em pares com mais de 20,500 títulos de mais de 5,000 editoras internacionais. Dentre estas editoras pode-se citar: Springer (Springer, 2013); IEEE Xplore Digital Library (IEEE, 2013); ACM Digital Library (ACM, 2013); ScienceDirect/Elsevier (Elsevier, 2013a); Wiley Online Library (Sons, 2013); British Computer Society (Society, 2013) dentre outras. Ainda segundo a editora Elsevier (2013c), a Scopus tem aproximadamente 2 milhões de novas gravações adicionadas a cada ano com atualizações diárias.

Com base nos resultados, ou seja, nas publicações retornadas pela execução de uma expressão de busca pré-estabelecida, uma serie de critérios de inclusão e exclusão de três filtros que foram

aplicados, a fim de selecionar as publicações consideradas relevantes (de acordo com o protocolo definido - ver Apêndice A) neste levantamento. Os três filtros consistem basicamente de: 1º Filtro, a seleção das publicações por uma análise preliminar do título, resumo/*abstract* e as palavras chaves das publicações; 2º Filtro, uma leitura na íntegra das publicações selecionadas no 1º filtro, visando identificar publicações que abordam inferência de invariantes em verificação de programas; e 3º Filtro, o objetivo é identificar quais publicações, mencionam programas na linguagem de programação C e que estejam relacionados com invariantes e verificação de código. Os critérios de inclusão e exclusão para cada um dos filtros são descritos no Apêndice A.

3.5.1 Análise e Publicação dos Resultados

A Tabela 3.1 detalha os resultados gerais de publicações identificadas pela máquina de busca, bem como o número de publicações aceitas em cada um dos filtros executados, de acordo com a expressão de busca executada na biblioteca Scopus.

Máquina de Busca	**Nº Total de Publicações**	**Publicações Selecionadas Após o Primeiro Filtro**	**Publicações Selecionadas Após o Segundo Filtro**	**Publicações Selecionadas Após o Terceiro Filtro**
Scopus	233	73	70	32

Tabela 3.1: Resultados gerais de publicações identificadas pelas maquinas de busca.

O Gráfico 3.1 apresenta uma visão geral do número total de publicações por ano retornadas pela biblioteca. Com base nesta informação pode-se observar que por volta de 2003 houve um crescimento no número de publicações referentes ao assunto. Acredita-se que este crescimento, em parte se deve ao fato do surgimento de métodos dinâmicos para inferência de invariantes, como será apresentado nas próximas seções. Adicionalmente, este número de publicações demonstra que existe um crescimento no interesse na comunidade científica na busca de soluções e melhorias para a inferência de invariantes de programas, bem como, também se pode notar que a partir de 2010 houve uma diminuição nas publicações. Acredita-se que esta diminuição se deve ao fato de que alguns problemas em abertos ainda não terem sido resolvidos e principalmente por parte das publicações do ano de 2013 ainda não estarem disponíveis em suas respectivas bibliotecas.

A partir das 233 publicações identificou-se que: 16 publicações eram repetidas entre as bibliotecas, onde estas foram eliminadas, restando um total de 217 publicações a serem analisadas no 1º filtro. As informações sobre as publicações como, editora, título da publicação, ano de publicação, e outras estão disponível em Rocha *et al.* (2013). Aplicando os critérios de exclusão do 1º filtro o número de publicações selecionadas foi de 73; na aplicação do 2º filtro foi identificado que 3 publicações não estavam disponíveis para *download* não atendendo os critérios definidos no protocolo da revisão sistemática, resultando assim em um total de 70 publicações à serem analisadas. As 70 publicações foram selecionadas/aprovadas, resultando em uma taxa de inclusão de 100% das publicações analisadas; No 3º filtro as 70 publicações (providas do 2º filtro) foram analisadas e, aplicando os critérios de exclusão, somente 32 publicações foram

Figura 3.1: Número de publicações por ano.

selecionadas. As informações coletadas das publicações (ver Seção 3.4) selecionadas no 3º filtro estão disponíveis em Rocha *et al.* (2013).

Com base nos fatos mencionados acima, vale ressaltar que o número total de publicações analisadas foi de 217, onde estas publicações abordavam 17 diferentes editoras, tais como: IEEE, Springer e ACM. O Gráfico 3.2, detalha o número de publicações por editoras.

3.5.2 Discussão sobre os Resultados

A fase de extração de informações, executada durante a aplicação do 3º filtro, permitiu coletar todas as informações necessárias para responder as questões de pesquisa propostas, como será apresentado mais detalhadamente a seguir. Analisando as publicações selecionadas, identificou-se que todas apresentaram resultados positivos na avaliação das abordagens propostas, mesmo considerando as respectivas limitações, como por exemplo, o não suporte a estruturas especifica de código em C. Desta forma, nestas próximas seções, apresenta-se os dados coletados para responder a cada questão de pesquisa proposta.

3.5.2.1 Q1.1: Foi desenvolvido e está disponível alguma ferramenta para a aplicação do método?

A questão de pesquisa **Q1.1**, descrita na Seção 3.4, busca verificar o apoio ferramental oferecido pelas publicações com as suas respectivas abordagens propostas. Desta forma, com base nas

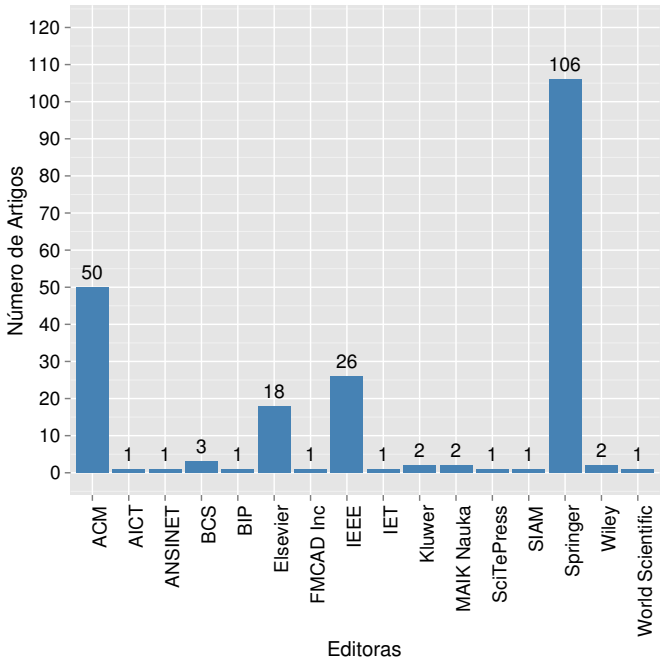


Figura 3.2: Número de publicações por editora.

informações coletadas, o gráfico na Figura 3.3 apresenta o percentual de publicações que oferecem este apoio ferramental. Logo, pode-se identificar que boa parte (40.6%) das abordagens fornece apoio ferramental.

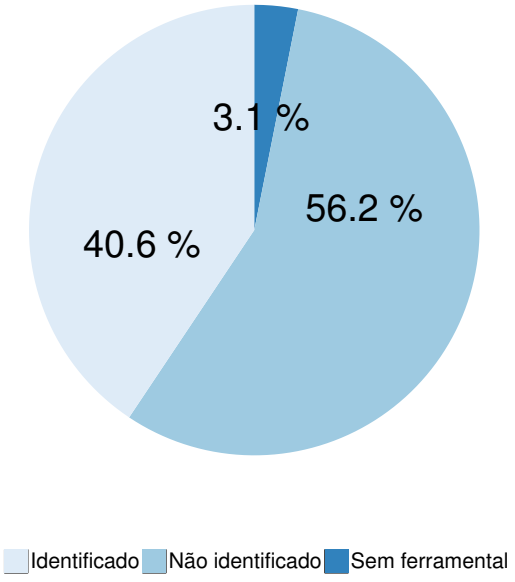


Figura 3.3: Disponibilidade de Apoio Ferramental.

3.5.2.2 Q1.2: O método proposto foi gerado a partir da integração com outros?

Objetivando classificar as técnicas adotadas para inferir invariantes de programa pelos trabalhos analisados nesta pesquisa, foram identificadas as principais técnicas adotadas em comum entre as publicações. A Tabela 3.2 apresenta as publicações classificadas pelas técnicas adotadas, onde a primeira coluna da tabela tem o identificador das publicações relativas às 32 publicações selecionadas a partir do terceiro filtro (ID), e as colunas a seguir são as técnicas obtidas das 32 publicações. O nome das técnicas foi definido, como a seguir:

- **Análise de predicado** (abrev. Predicados). Considera-se todas as técnicas relacionadas a identificar, gerar ou analisar predicados para apoiar a geração de invariantes de programa. Por exemplo, a técnica de abstração de predicado que é uma forma especial de interpretação abstrata (Cousot e Cousot, 1977), onde um determinado conjunto de predicados constrói o domínio abstrato (Flanagan e Qadeer, 2002). De acordo com Lahiri *et al.* (2003), usualmente o algoritmo de abstração de predicado gera uma abstração de estados finitos a partir de um sistema de com muitos ou infinitos estados. O modelo finito pode então ser usado no lugar do sistema original quando se usa *model checking* ou a para derivar invariantes.

- ***Provers***. Técnicas relacionadas a provar ou resolver problemas (por exemplo, a satisfabilidade de restrições ou equações não lineares) em algum tipo de forma genérica. Por exemplo, satisfabilidade Booleana (SAT), ou teorias de módulos de satisfabilidade (SMT) que têm sido amplamente adotados para determinar automaticamente se uma dada fórmula lógica proposicional é satisfativel, tanto na verificação de *software* quanto de *hardware* (De Moura e Bjørner, 2008; Nguyen *et al.*, 2012).

- **Interpretação Abstrata** (abrev. Int. Abstrata). A técnica consiste na utilização de denotações para descrever computações em outro universo de objetos abstratos. Desta forma, os resultados de uma execução abstrata foram derivados a partir de algumas informações sobre a atual execução (Cousot e Cousot, 1977).

- **Lógica de Mill** (abrev. Log. Mill). De acordo com Labed Jilani *et al.* (2012), são modeladas funções para capturar a semântica do programa. Neste sentido, se têm os conceitos de Invariante de Relação, e Funções de Invariantes como uma ferramenta para analisar *while loops*. Esses conceitos de computação (ou aproximação) de funções de *loop* geram assertivas com invariantes, computando mais pré-condições fracas para uma determinada pós-condição mais fortes e pós-condições para uma determinada pré-condição (Ghardallou, 2012).

- ***Templates***. Considera-se técnicas que adota *templates* de invariantes (Srivastava *et al.*, 2009) ou padrões para reconhecer a tipos específicos (por exemplo, relação linear) de invariantes de programa (Ernst *et al.*, 2001). Um *template* identifica os parâmetros desconhecidos, e restringir a dimensão da invariante, por exemplo, o número de conjunções e o número de aplicações de funções. Isto significa que a forma do *template* determina as invariantes geradas (Beyer *et al.*, 2007b).

- **Restrições**. Técnicas que geram restrições do programa analisados para derivar invariantes de programa. Normalmente, são técnicas que adotam uma representação paramétrica a partir de um mapa de invariantes. Então, as condições de segurança são codificadas como restrições nos parâmetros determinados, assim, gerando invariantes indutivas do programa (Gupta *et al.*, 2009).

- **Técnicas de Compiladores** (abrev. Compilador). Técnicas de análise de fluxo de dados (por exemplo, atribuições e chamadas de função), para gerar representação intermediária, e instrumentação de programas.

- **Análise Concreta** (abrev. Concreta). Baseado na execução ou computação de entradas específicas do programa. Normalmente, a execução concreta do programa é adotada para capturar entradas e saídas alvo para inferir invariantes de programa (Ernst *et al.*, 2007; Sagdeo *et al.*, 2011).

- **Análise Simbólica** (abrev. Simbólica). Relaciona a execução simbólica do programa. De acordo com King (1976), execução simbólica é uma extensão natural de execução normal, fornecendo as computações normais como um caso especial. Definições computacionais para os operadores básicos da linguagem são estendidos para aceitar entradas simbólicas ($\{\alpha_1, \alpha_2, \alpha_3, \dots\}$) e produzir fórmulas simbólicas como saída ($\{\alpha_1 \geq 0 \wedge \alpha_1 + 2 \times \alpha_2 \geq 0 \wedge \neg(\alpha_3 \geq 0)\}$).

- **Algoritmo de Aprendizagem** (abrev. Aprendizagem). Técnicas baseadas em aprendizado de máquina para relatar propriedades que são verdadeiras a partir de um treinamento que é executado usando dados arbitrários.

- **Outras**. Técnicas tais como: regressão linear (Sagdeo *et al.*, 2011); prova colorida (Hoder *et al.*, 2011b); algoritmos genéticos (Parsa *et al.*, 2011); e outras.

A Figura 3.4 mostra a tendência da adoção das técnicas pelas 32 publicações analisadas. Na figura, identifica-se que as técnicas mais utilizadas por estas publicações são técnicas de compiladores, análise de predicados, *provers* e *templates*. A seguir analisa-se as técnicas (ver Tabela 3.2) adotadas para a geração das invariantes de programa.

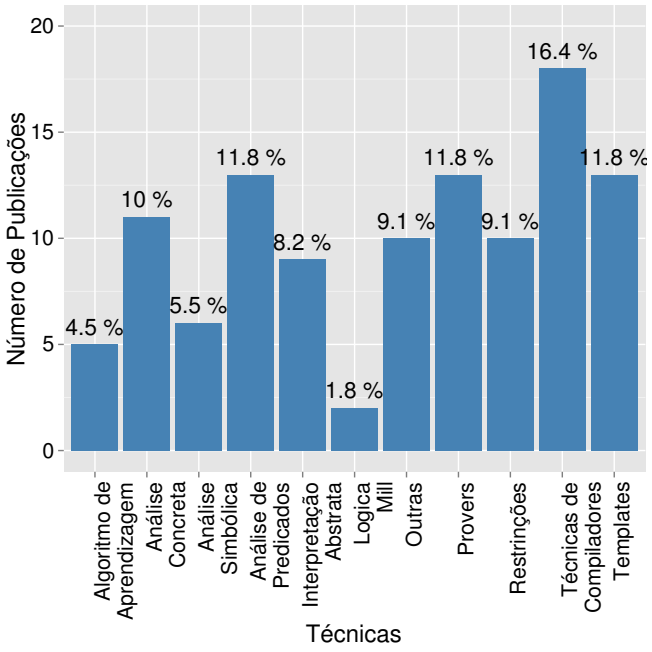


Figura 3.4: A tendência da adoção das técnicas por publicações.

Artigos	Técnicas										
	Predicados	**Int. Abstrata**	**Log. Mill**	***Templates***	**Restrições**	**Compilador**	**Provers**	**Concreta**	**Simbólica**	**Aprendizagem**	**Outras**
Lo_1997	X			X		X			X		
Ernst_1999				X		X		X		X	
Ernst_2001				X		X		X		X	
Ernst_2001b				X		X		X		X	
Perkins_2004						X					
Rodriguez_2005											X
Jain_2006	X	X						X			
Ernst_2007				X		X		X		X	
Beyer_2007	X	X		X	X						
Henzinger_2008					X		X		X		X
Zaks_2008						X	X				
Srivastava_2009_b	X			X	X	X	X				
Shikun_2009				X	X		X				
Gupta_2009		X			X		X	X	X		
Kahlon_2009		X			X	X					X
Gupta_2009b		X		X	X			X	X		
Srivastava_2009	X			X	X	X	X				
Gulwani_2009	X				X		X				
Kong_2010	X			X			X			X	
Ancourt_2010		X									X
Feautrier_2010		X				X					
Cohen_2010	X						X				
Hoder_2011_b	X					X	X		X		X
Sagdeo_2011	X					X		X			X
Wei_2011						X		X			
Sharma_2011	X				X		X				
Parsa_2011						X		X			X
Hoder_2011	X					X	X		X		X
Donaldson_2011		X				X					X
Labed_2012	X		X	X							
Nguyen_2012		X				X	X	X			X
Ghardallou_2012			X	X							

Tabela 3.2: Classificação dos artigos por técnicas.

Análise de Predicados. Analisando os métodos baseadas na análise de predicado, pode-se observar que a análise de predicado fornece um apoio significativo na análise do programa para inferir propriedades sobre o comportamento do programa, ou seja, para identificar as possibilidades dos fluxos de execução do programa. Por exemplo, a abstração de predicado é um método amplamente reconhecido (Jain *et al.*, 2006; Beyer *et al.*, 2007c; Gulwani *et al.*, 2009; Srivastava e Gulwani, 2009; Kong *et al.*, 2010) para a abstração sistemática de programas. Assim, abstraindo os dados do programa para manter o rastreamento de certos predicados e suas respectivas execuções do programa (Clarke *et al.*, 2004b). De acordo com Gulwani *et al.* (2009), no caso da utilização de domínios de abstração de predicado, uma das principais vantagens é que se pode representar disjunções ao contrário de outros domínios abstratos como domínio de poliedros. No entanto, esta expressividade vem com desvantagens: em primeiro lugar, o estado abstrato pode ter tamanho exponencial no número de predicados; e segundo, o domínio abstrato tem altura exponencial.

Interpretação Abstratata. De acordo com Cousot e Cousot (2010), interpretação abstrata é uma teoria da aproximação da semântica de linguagens de programação cuja aplicação principal é a análise estática A teoria da interpretação abstrata assegura a coerência entre as abstrações (as propriedades) e oferece técnicas de aproximação eficazes. Escolhendo abstrações que são coesas o suficiente para ser efetivamente computáveis e precisas o suficiente para evitar falsos alarmes, pode-se determinar a indecidibilidade e complexidade na análise e verificação de modelos e programas. Além disso, a interpretação abstrata não se aplica apenas a sistemas computadorizados, mas também para os sistemas onde o comportamento discreto ou complexo contínuo pode ser formalmente descritos como uma função de tempo (por exemplo, em: processamento de imagem, computação quântica, e etc.).

Lógica de Mill. A lógica de Mill é adotada para caracterizar laços por meio de uma função que define o seu espaço de estados. De acordo com Mraihi *et al.* (2012), invariantes de funções e as invariantes de relações são ferramentas de análise alternativas que são distintas das assertivas de invariantes (Hoare, 1969), mas são relacionadas a elas. As assertivas de invariantes buscam uma relação indutiva e as invariantes de funções buscam uma relação simétrica. As invariantes de relações não impõem quaisquer outras condições além da reflexividade e transitividade.

Templates. Analisando as publicações, nota-se que a adoção de *templates* é importante para fornecer um suporte para guiar a inferência de invariantes de programas. De acordo com Lo *et al.* (1997), a vantagem do uso de *template* é que todas as invariantes geradas são válidas. Usualmente, a aplicação consiste no casamento do *Template*, que é identificado pelas relações de recorrência envolvendo as variáveis de indução do *loop*. Contundo, de acordo com Sharma *et al.* (2011), as técnicas baseadas em *templates* sofrem de duas desvantagens: Elas não são totalmente automáticas, uma vez que exigem que o utilizador especifique o formato da invariante desejada; e em casos específicos, as técnicas baseadas em *templates* requerem resolver restrições não lineares, assim a sua aplicabilidade é limitada pela falta de algoritmos eficientes para a resolução de tais restrições.

Restrições. A geração automática de invariantes através da resolução de restrições tem sido extensivamente estudada como um meio clássico na verificação do programa. De acordo com Gupta *et al.* (2009), na prática as ferramentas existentes não escala nem mesmo programas de tamanho moderado. Isto é porque as restrições que precisam ser resolvidas, mesmo para pequenos programas, já são demasiado difícil (por exemplo, usando restrições não lineares) para os solucionadores de restrições.

Técnicas de Compiladores. A ideia básica da adoção de tecnologias de compilador, na verificação do programa, é analisar o comportamento de um programa para derivar suas propriedades. De acordo com Labed Jilani *et al.* (2012), a análise de programas é mais viável, devido aos recentes avanços em: sistemas de álgebra computacional; provadores de teoremas; e tecnologias de compiladores que fornecem um impulso tecnológico para realizar esta análise automaticamente. No que diz respeito a inferir invariantes de programas, usualmente existem duas maneiras de detecção invariantes que são chamadas de: estática e dinâmica. Na forma estática, as invariantes são detectadas por meio da adoção de técnicas de compiladores (por exemplo, a extração de gráficos de fluxo de dados do código fonte do programa) (Parsa *et al.*, 2011). A forma dinâmica consiste na observação da execução do programa, como por exemplo, a estratégia de análise de rastreamento utilizado por Daikon (Ernst *et al.*, 2007).

Provers. Analisando as publicações, nota-se que os solucionadores (SAT e SMT) tem um papel significativo para raciocinar sobre restrições geradas a partir do programa analisado para apoiar a geração de invariáveis. De acordo com Gupta *et al.* (2009), as abordagens baseadas em restrições adotam uma representação paramétrica a partir de um mapa de invariantes. Então, as condições de segurança são codificados como restrições nos parâmetros determinados, assim, gerando invariantes indutivas do programa. Vale a pena notar que, no contexto da verificação formal, os SMT *solvers* estão a cada dia ganhando mais importância com provas robustas. Eles permitem uma linguagem mais expressiva do que a lógica proposicional, apoiando um conjunto de procedimentos de decisão (para a aritmética, vetores de bit e *arrays*). Adicionalmente, os SMT *solvers* são mais rápidos que então provadores genéricos de teoremas de primeira ordem sobre fórmulas sem quantificadores (Bruttomesso *et al.*, 2010).

Análise Concreta. Na execução concreta, os métodos propostos costumam coletar um conjunto finito de estados alcançáveis usando técnicas guiadas por teste (Gupta e Rybalchenko, 2009). Neste contexto, as técnicas de análise dinâmica podem ser realizadas utilizando execução concreta ou simbólica, descobrindo as invariantes a partir do rastreamento de execução do programa. De acordo com Ernst (2001), a análise estática (que examina o texto do programa e análise as possíveis execuções e estados de execução) em relação à análise dinâmica, tem pontos fortes e fracos complementares: é menos preciso e eficiente; e a acurácia da análise de ponteiros permanece além do estado da arte. Em particular, a detecção de invariantes dinâmica não é considerada adequada porque os conjuntos de testes adotados geralmente não podem caracterizar plenamente todas as execuções (Ernst, 2001). No entanto, na prática, as suítes de teste padrões executam adequadamente (Ernst *et al.*, 2001).

Execução Simbólica. As técnicas de análise dinâmica podem ser aplicadas utilizando execução simbólica para descobrir invariantes. A vantagem é que uma execução simbólica pode representar uma grande, geralmente infinita, classe de execuções normais (King, 1976). No entanto, o raciocínio simbólico sobre grandes programas é limitado por ser impreciso Godefroid (2012). A escalabilidade das abordagens existentes para geração invariante é severamente limitada, devido aos altos custos de computação das técnicas de raciocínio simbólicos subjacentes Gupta e Rybalchenko (2009).

Algoritmos de Aprendizagem. Estes geralmente são adotados como uma caixa preta, onde somente é necessário projetar um classificador que guie o algoritmo de aprendizado para a geração de invariantes (Kong *et al.*, 2010). Entretanto, de acordo com Ernst *et al.* (2007), a análise dinâmica ou técnica de aprendizado de máquina tem uma possibilidade de *overfitting* (super generalização) por propriedades que são verdadeiras na execução dos treinamentos, mas não são verdadeiras no geral. Por exemplo, se houver um número inadequado de observações de uma variável particular, padrões observados sobre ele pode ser coincidência.

Outras Técnicas. Analisando estas publicações, observa-se que diferentes tipos de técnicas têm sido aplicados para refinar e apoiar a inferência de invariantes, tais como: bases de Gröbner (Cox *et al.*, 2010); redução de ordem parcial (POR) (Kahlon *et al.*, 2009); formulas de Presburger (Kelly *et al.*, 1996); derivação colorida (McMillan, 2008); algoritmos genéticos (Parsa *et al.*, 2011); regressão linear (Lawson e Hanson, 1995); k indução (Donaldson *et al.*, 2011); e solucionadores de equação (Nguyen *et al.*, 2012). Por exemplo, a k indução têm sido extensivamente estudada (Eén e Sörensson, 2003; Hagen e Tinelli, 2008; Donaldson *et al.*, 2011). A principal fraqueza da k indução é a sua sensibilidade para laços internos que podem conter insuficientes assertivas fortes para construir uma invariante indutiva que implica na correção do programa (Donaldson *et al.*, 2011). Desta forma, na maioria dos casos, existe uma necessidade de combinar outras técnicas encontradas nas publicações selecionadas neste trabalho para suportar a geração de invariantes (Nguyen *et al.*, 2012).

3.5.2.3 Q1.3: Qual é a estratégia (estática, dinâmica ou hibrida) de execução do método?

Com base nos dados coletados, classificou-se o modo como são executadas (ou aplicadas) as abordagens propostas de modo: Estático, Dinâmico ou Híbrido (caso a abordagem utilize as duas formas). A Figura 3.5 apresenta o resultado da classificação dos modos de execução por publicações. Acredita-se que o modo estático esteja presente na maioria das publicações, em parte, porque os primeiros métodos para a geração de invariantes utilizavam a análise estática (Jain *et al.*, 2006).

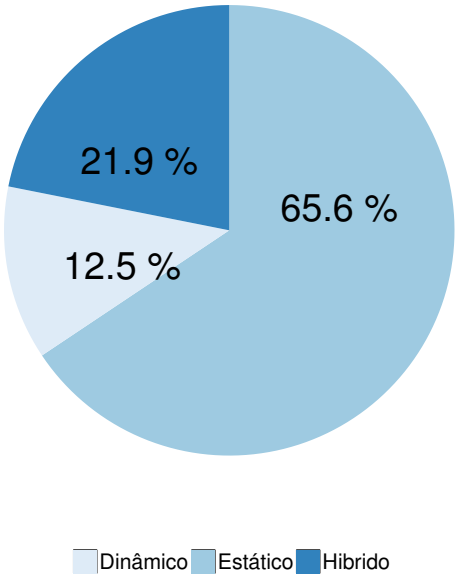


Figura 3.5: Modo de execução das abordagens propostas nas publicações.

3.5.2.4 Q1.4, Q1.4.1, Q1.6: Respostas baseada em Comparações das Publicações

Visando obter uma visão geral das publicações analisadas para responder as questões de pesquisa, efetuou-se uma comparação, no que diz respeito a completude de cada abordagem, das 32 abordagens selecionadas no 3º filtro. As métricas utilizadas foram extraídas com base nas questões de pesquisa definidas neste trabalho (detalhes ver Seção 3.4). Vale ressaltar que o principal objetivo desta comparação é medir a cobertura das abordagens diante das métricas propostas e não sua eficácia ou desempenho, ou seja, identificar as abordagens que atendam/satisfaçam o maior número de métricas.

A Tabela 3.3 apresenta as métricas definidas, sendo que cada coluna da tabela significa: (i) o identificador da métrica (ID); (ii) o nome da métrica (Nome); (iii) as opções definidas para cada métrica (Opções); e (iv) o peso/pontuação para a cobertura definida de cada opção das métricas analisadas (Pontuação de Cobertura), onde a pontuação é definida da seguinte forma: + representa 5 pontos, sendo que a pontuação máxima para cada categoria é de 15 pontos e zero quando não há pontuação.

As métricas apresentadas na Tabela 3.3 objetivam identificar: **ID=1** se a abordagem proposta possui um apoio ferramental e se este está disponível; **ID=2** o tipo do *benchmark* utilizado na avaliação prática da abordagem, ou seja, foi um *benchmark* desenvolvido pelo próprio autor da abordagem, é provido da literatura, da indústria ou uma combinação entre estes tipos de *benchmarks*; **ID=3** se mais de uma fonte de *benchmark* foi utilizada, ou seja, a abordagem foi avaliada em mais de um contexto (estruturas e formas de implementação); **ID=4** se a abordagem proposta foi comparada com outras na prática, ou seja, foi efetuada uma avaliação experimental

analisado itens tais como tempo, números ou classes de invariantes geradas; **ID=5** a escalabilidade da abordagem proposta em termos de LOC, neste caso a escalabilidade identificada e medida da abordagem, teve como base os experimentos apresentados (códigos utilizados) na publicação da abordagem; **ID=6** se a abordagem provê suporte a estruturas de código como ponteiros, *loops* aninhados e *arrays*, adicionalmente se este suporte é parcial (somente a algumas estruturas) ou total (caso nenhuma restrição tenha sido identificada); **ID=7** se a abordagem possui alguma restrição na sua forma de aplicação, ou seja, na forma como o método efetua a análise dos programas existe alguma restrição; **ID=8** se a abordagem fornece suporte a programas que utilizam concorrência, adicionalmente se este suporte é parcial (somente a algumas propriedades) ou total; e por último **ID=9** se provê suporte a ponto flutuante, sendo este parcial ou total.

Com base nas métricas apresentadas acima, analisou-se as publicações e suas respectivas abordagens. A Tabela 3.4 e 3.5 detalha os resultados da medição (pontuação) da cobertura de cada métrica para as abordagens, onde a tabela esta divida em basicamente quatro partes: (1) na primeira coluna o identificador das publicações, referentes as 32 tabelas das publicações listadas acima (ID). Este é composto de quatro partes, por exemplo, para o ID = 02_D_ernst:1999 temos que: (i) `02` é sua numeração em ordem crescente; (ii) `D` identifica qual é forma da aplicação da abordagem (D - Dinâmico, S - Estático e H - Híbrido caso utilize as duas formas); (iii) `ernest` o nome do autor da publicação; e (iv) `1999` o ano da publicação; (2) na segunda coluna, a métrica com ID=1 na Tabela 3.3 (Ferramental); (3) da terceira à sexta coluna, são as métricas relacionadas a avaliação experimental (ID=2 à ID=5 na Tabela 3.3) das abordagens (Avaliação Experimental); (4) da sétima à décima coluna, são as métricas relacionadas as limitações e suporte (ID=6 à ID=9 na Tabela 3.3) providos pelas abordagens (Limitações/Suporte); e por último (5) na décima primeira coluna é apresentado o total de pontos obtidos para cada publicação analisada. É importante ressaltar que as 10 primeiras abordagens com maior pontuação de cobertura sobre as métricas propostas estão listadas no início da tabela em negrito. Vale ressaltar que em caso de empate, as métricas foram utilizadas como critério de desempate na seguinte ordem dos IDs da Tabela 3.3: 1, 8, 9, 6, 5, 2, 3, 4, 7.

Métricas			
ID	**Nome**	**Opções**	**Pontuação de Cobertura**
1	Ferramental	Não possui/Não identificado	**0**
		Possui, mas não foi identificada sua disponibilidade	**+**
		Possui e está disponível	**++**
2	Tipo do Benchmark utilizado (abrev. `T Bench`)	Não foi aplicada avaliação prática	**0**
		Benchmarks do próprio artigo	**+**
		Benchmarks da literatura	**++**
		Benchmarks industriais	**+++**
		Combinação entre os tipos de benchmarks	**+++**
3	Mais de uma fonte de Benchmark (abrev. `> Bench`)	Sim	**++**
		Não	**0**
4	Comparação com outras abordagens (abrev. `Compara a Abordagem`)	Sim	**++**
		Não	**0**
5	Escalabilidade Apresentada em Termos de LOC (abrev. `Escalabilidade LOC`)	Sem experimentação prática/Não Identificado	**0**
		< 100 LOC	**+**
		≥ 100 e ≤ 999 LOC	**++**
		≥ 1 KLOC	**+++**
6	Suporta estruturas, como: arrays, ponteiros e loops aninhados (abrev. `Suporte a Estrutura C`)	Nenhuma das estruturas	**0**
		Parcial	**+**
		Total/Não identificado restrição	**++**
7	Restrições na forma da aplicação ou análise (abrev. `Restri. de Aplicação`)	Não identificado	**++**
		Com restrições	**+**
8	Suporta concorrência	Não/Não identificado	**0**
		Parcial	**+**
		Sim	**++**
9	Suporte a Ponto Flutuante	Não/Não identificado	**0**
		Parcial	**+**
		Sim	**++**

Tabela 3.3: Métricas para comparação de completude das abordagens selecionadas.

ID	Ferramental	Avaliação Experimental				Limitações/Suporte				
		T Bench	> Bench	Compara a Abordagem	Escalabilidade LOC	Suporte a Estrutura C	Restri. de Aplicação	Suporta concorrência	Suporte a ponto flutuante	Pontos
23_S_donaldson:2011	++	+++	0	0	+++	++	++	++	++	80
19_S_cohen:2010	++	+++	++	0	+++	++	++	++	0	80
15_S_kahlon:2009	+	+++	0	0	+++	++	++	++	++	75
05_D_perkins:2004	++	+++	++	0	+++	++	++	0	0	70
27_S_wei:2011	+	+++	0	0	+++	++	+	++	++	70
03_D_ernst:2001b	++	+++	++	0	++	++	++	0	0	65
04_D_ernst:2001	++	+++	++	0	++	++	++	0	0	65
20_S_feautrier:2010	++	++	++	++	++	+	++	0	0	65
31_D_nguyen:2012	+	+++	++	0	++	++	+	0	++	65
09_D_ernst:2007	++	+++	0	0	+++	++	++	0	0	60
21_S_ancourt:2010	++	++	++	0	+++	++	+	0	0	60
16_H_gupta:2009	++	++	0	++	++	++	++	0	0	60
02_D_ernst:1999	+	+++	++	0	++	++	++	0	0	60
28_H_sagdeo:2011	+	+++	0	++	++	++	++	0	0	60
26_S_sharma:2011	+	++	++	++	+	++	++	0	0	60
14_H_gupta:2009b	++	+++	0	0	++	++	++	0	0	55

Tabela 3.4: Comparação de completude das abordagens.

ID	Ferramental	Avaliação Experimental				Limitações/Suporte				
		T Bench	> Bench	Compara a Abordagem	Escalabilidade LOC	Suporte a Estrutura C	Restri. de Aplicação	Suporta concorrência	Suporte a ponto flutuante	Pontos
22_S_kong:2010	++	+++	++	0	+	++	+	0	0	55
10_S_zaks:2008	+	++	++	0	+++	+	++	0	0	55
24_S_hoder:2011	++	+++	++	0	+	+	+	0	0	50
07_S_jain:2006	+	+++	0	0	0	++	++	++	0	50
13_S_srivastava:2009b	+	++	++	0	+	++	++	0	0	50
18_S_srivastava:2009	+	++	++	0	+	++	++	0	0	50
29_S_hoder:2011b	+	+++	++	0	+	+	++	0	0	50
30_S_ghardallou:2012	+	++	0	0	+	++	+	0	++	45
32_S_labed:2012	+	+	0	0	+	++	+	0	++	40
12_S_gulwani:2009	+	++	0	0	+	++	++	0	0	40
08_S_beyer:2007	+	+	0	0	+	++	++	0	0	35
25_H_parsa:2011	+	+	0	0	+	++	++	0	0	35
06_S_rodriguez:2005	+	++	++	0	0	+	+	0	0	35
11_D_henzinger:2008	++	+	0	0	+	0	++	0	0	30
01_S_lo:1997	+	+	0	0	+	+	+	0	+	30
17_S_shikun:2009	0	+	0	0	+	++	++	0	0	30

Tabela 3.5: Comparação de completude das abordagens.

Nas Tabelas 3.4 e 3.5, identifica-se que a maior pontuação alcançada foi de 80 pontos de um total de 100 pontos. Portanto, observou-se que as 32 publicações não atenderam a todas as métricas propostas simultaneamente. Isso em parte é explicado devido ao fato que 87,5% do método proposto nas publicações não terem suporte a concorrência e 78,12% não terem suporte a programas de ponto flutuante. O maior número de métricas atendidas pelas publicações foi 7 de um total de 9. De outra forma, apenas $21,87\%$ das publicações atenderam 7 métricas. Estas publicações foram as seguintes: 1_S_donaldson:2011; 2_S_cohen:2010; 3_S_kahlon:2009; 5_S_wei:2011; 8_S_feautrier:2010; 9_D_nguyen:2012; e 16_S_sharma:2011.

As métricas relacionadas com a avaliação experimental mostraram que os métodos propostos pelas 32 publicações ainda estão abertos para melhorias e experimentações mais aprofundadas. Isto baseado na escalabilidade baseada em LOC's nas tabelas (veja a coluna "Escalabilidade LOC" nas Tabelas 3.4 e 3.5), apenas 25% têm escalabilidade para maior que 1 KLOC. O tipo do benchmark (na coluna T Bench da tabela) utilizado nos experimentos, 46,87% das publicações adotaram *benchmarks* industriais ou uma combinação de diferentes tipos de *benchmarks*. A análise de comparação (na coluna Compara a Abordagem) realizada com outros métodos nas publicações é de apenas 15,62%. Neste trabalho argumenta-se que esses dados são importantes quando se considera: a aplicação destes métodos em grandes programas que podem gerar resultados inesperados (ou seja, falsos positivos), isso por causa da falta de experimentação dos métodos; e para escolher um método para inferir invariantes de programa, por exemplo, visando a escalabilidade necessária para aplicar o método em sistemas grandes e complexos como o Kernel do Linux [2] que tem programas com milhões de LOC.

Analisando as Tabelas 3.4 e 3.5, também se identificou que $78,12\%$ das 32 publicações apresentaram suporte as estruturas de programas em C (na coluna Suporte a Estrutura C das tabelas), tais como: *arrays*, ponteiros e loops aninhados. No que diz respeito a restrições (na coluna Restri. de Aplicação das tabelas) na execução ou análise realizada pelos métodos propostos, $71,87\%$ nas publicações não têm restrições. Estes dados são significativos quando se leva em consideração a automação e a completude dos métodos propostos, por exemplo, para aplicação dos métodos propostos em programas industriais.

O número total de métodos propostos que suportam concorrência (na coluna Suporta concorrência das Tabelas 3.4 e 3.5) é de 12.5% e para ponto flutuante (na coluna Suporte a ponto flutuante) é de 18.75%. Essas métricas apresentam a menor taxa de cobertura nas tabelas. Portanto, argumenta-se que esses métodos propostos ainda estão abertos para melhorias, uma vez que várias aplicações (como, o Kernel do Linux) adotam programas concorrentes e de ponto flutuante. Atualmente, diversas ferramentas de verificação (Clarke *et al.*, 2005; Musuvathi e Qadeer, 2007; Cordeiro *et al.*, 2012a) já provêm suporte a concorrência e pontos flutuante. Neste trabalho se defende que a combinação dos métodos propostos para inferir invariantes de programa e ferramentas de verificação podem melhorar a verificação e validação de programas, aumentando a precisão e reduzindo o tempo da análise dos programas.

[2]Disponivel em https://www.kernel.org/

3.5.2.5 Q1.5: Quais as limitações do método proposto?

A questão de pesquisa **Q1.5**, detalhada na Seção 3.4, busca identificar quais são as principais limitações dos métodos propostos. Visando responder esta pergunta de forma mais detalhada do que apresentada anteriormente, classificou-se as limitações dos métodos em quatro categorias: **(A)** Propriedades que não são cobertas na verificação/inferência de invariantes. Por exemplo, *overflow* de inteiros ou *loops* aninhados; **(B)** O fluxo da abordagem proposta (incluindo transformações intermediarias) não está completamente coberto. Por exemplo, em uma transformação para um grafo de fluxo de controle algumas informações são ignoradas (por exemplo, fusões de entrada de arcos); **(C)** Estruturas da linguagem C que não são suportadas pelo método apresentado. Por exemplo, ponteiros ou *arrays*; e **(D)** Na estratégia de execução do método, ou seja, como o método efetua a análise do programa. Por exemplo, a verificação é feita analisando um procedimento por vez.

O gráfico na Figura 3.6 apresenta o percentual de publicações com base na classificação das limitações citadas anteriormente. Desta forma, identificou-se que 45.8% das limitações dos métodos apresentados nas publicações são referentes à forma de aplicação do método (categoria D), ou seja, como o método efetua a análise do programa, onde estas limitações estão ligadas diretamente com o que será analisado (ex. descartar valores de entradas) e como o programa será analisado (ex. apenas um procedimento por vez), onde estas limitações podem ter impacto direto no resultado dos métodos. Outro dado importante de ser ressaltado é que 29.2% das limitações se referem ao não suporte das estruturas da linguagem C (categoria C). Acredita-se que este seja um identificador que aponta que mesmo com o número crescente de novas abordagens para a inferência de invariantes, a maioria das abordagens ainda não fornece suporte total as estruturas da linguagem de programação C.

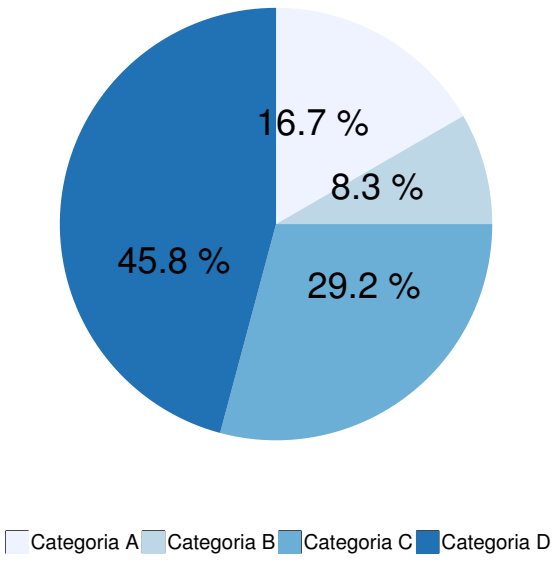


Figura 3.6: Classificação das limitações dos métodos.

3.5.3 Limitações desta revisão sistemática

Um dos grandes problemas com a aplicação da revisão sistemática da literatura está em encontrar todas as publicações relevantes (Kitchenham *et al.*, 2009). Com o objetivo de aliviar este problema. Adotou-se uma busca automatizada usando uma biblioteca digital para encontrar as publicações. Analisando a execução desta revisão sistemática, no entanto, foram identificadas algumas limitações relacionadas com as publicações identificadas, e os dados analisados, tais como:

- **LIM1**. Adotou-se somente a biblioteca digital Scopus (Elsevier, 2013b).
- **LIM2**. Um único pesquisador selecionou as publicações, embora outro pesquisador tenha verificado as publicações incluídas e excluídas.
- **LIM3**. Um único pesquisador extraiu os dados e outro pesquisador verificou a extração dos dados.

Em relação à primeira limitação (**LIM1**) isso implica que pode-se ter perdido alguma publicação relevante. Adotou-se a biblioteca Scopus que, no entanto, é uma das maiores bases de dados de publicações, incluindo editoras internacionais amplamente reconhecidos (como Springer (Springer, 2013), IEEE Xplore Digital Library (IEEE, 2013), e ACM Digital Library (ACM, 2013)), bem como, outras revisões sistemáticas (Brereton *et al.*, 2007; Barmi *et al.*, 2011; Unterkalmsteiner *et al.*, 2012; Magdaleno *et al.*, 2012; Kitchenham e Brereton, 2013; Verner *et al.*, 2014) adotaram e recomendam a biblioteca Scopus. Visando reduzir qualquer viés introduzido pela utilização de uma única fonte de indexação digital, primeiramente realizou-se uma busca manual nas principais fontes (como site de conferências e em websites de autores); abordando autores individuais para determinar se eles tinham publicado qualquer publicação relevante (esses dados identificados foram nomeados de lista de controle). Assim, adotou-se a lista de controle para validar as publicações retornadas pela biblioteca Scopus, ou seja, a lista de controle nos ajuda a verificar a execução da *string* de busca na biblioteca digital.

Como trabalho futuro, pretende-se ampliar esta pesquisa, incluindo outras bibliotecas digitais, com o objetivo de reunir novas publicações não indexadas pela biblioteca Scopus. Além disso, a *string* de busca foi projetada para encontrar o número máximo de métodos para inferir invariantes de programa em programas em C, mas é possível que se tenha perdido alguma publicação que usou uma terminologia diferente para descrever seus métodos. Nos resultados da pesquisa também se omitiu, relatórios técnicos ou teses de pós-graduação. Uma vez que se partiu da seguinte suposição de que estudos de qualidade da literatura estarão presentes em *jornals* ou conferências. Vale a pena notar que o escopo da pesquisa foi definido sobre bibliotecas digitais através de suas respectivas máquinas de busca.

A segunda limitação (**LIM2**), dado o interesse em identificar os métodos para inferir invariantes em programas em C, pode-se de forma errônea ter sido incluído publicações com métodos que

adotam apenas em um subconjunto específico da linguagem de programação C, mas se preferiu considerar essas publicações do que excluir publicações relevantes. Em relação à terceira limitação (**LIM3**), mesmo que apenas uma pessoa tenha extraído os dados de todas as publicações e isto possa potencialmente ter introduzido viés. Nesta pesquisa, os outros autores revisaram todos os dados reunidos com o objetivo de aumentar a confiabilidade dos dados gerados.

3.6 Resumo

Neste capítulo foi descrito e analisado os principais trabalhos relacionados aos métodos e técnicas para geração de casos de teste, verificação e análise de memória de programas, contra-exemplos de *model checkers*, e invariantes de programas. Na Seção 3.1, foram abordados trabalhos relacionados à geração de casos de teste com diferentes estratégias de aplicação, por exemplo, utilizando especificações de teste, linguagem de modelagem, e entradas simbólicas em diferentes domínios de utilização. Na Seção 3.2, foram apresentados trabalhos relacionados a verificação e análise de defeitos no gerenciamento de memória de programas. Na Seção 3.3, foram descritos trabalhos que focam nos aspectos para entender e/ou analisar contra-exemplo gerados por *model checkers*. Na Seção 3.4, foram analisados trabalhos que fazem utilização de invariantes de programas na verificação de programas em C. E por fim na Seção 3.5, foi apresentado um levantamento bibliográfico, sobre invariantes de programas, feito utilizando a técnica da revisão sistemática. Neste levantamento, foram identificados métodos para inferência de invariantes de programas na verificação de código na linguagem de programação C existentes na literatura, extraindo suas principais características, e avaliando as vantagens e desvantagens de cada abordagem.

Capítulo 4

Geração Automática de Casos de Teste para Gerenciamento de Memória de Programas em C usando *Bounded Model Checking*

Este capítulo descreve o método **Map2Check** para geração automática de casos de teste para gerenciamento de memória de programas em C, e um estudo empírico realizado com o objetivo de avaliar o método proposto, quando aplicado à verificação dos *benchmarks* padrão ANSI-C, bem como, é apresentado um comparação com as ferramentas Valgrind's Memcheck (Nethercote e Seward, 2007), CBMC (Clarke *et al.*, 2004a), LLBMC (Merz *et al.*, 2012), CPAChecker (Beyer e Keremoglu, 2011), Predator (Dudka *et al.*, 2014), e ESBMC (Cordeiro *et al.*, 2012a).

4.1 Método Map2Check

Map2Check é uma melhoria do método FORTES (*FORmal unit TESt generation*) que foi originalmente proposto no trabalho de mestrado de Rocha *et al.* (2010); Rocha (2011). FORTES explora as propriedades de segurança gerados por BMCs para criar casos de teste. Entretanto, FORTES não gera casos de teste para gerenciamento de memória. Map2Check está disponível para download em https://github.com/hbgit/Map2Check sobre a licença GPL.

A Figura 4.1 apresenta uma visão geral do método Map2Check, onde as caixas e setas com linhas tracejadas representam respectivamente os novos componentes e possibilidades de execução. O método proposto consiste nas seguintes etapas:

- *(I)* Identificação das propriedades de segurança;

- *(II)* Coleta de informações das propriedades de segurança;
- *(III)* Tradução das propriedades de segurança;
- *(IV)* Rastreamento de memória;
- *(V)* Instrumentação do código com assertivas;
- *(VI)* Implementação dos testes; e
- *(VII)* Execução dos testes.

Visando explicar as principais etapas do método proposto, utilizou-se o programa 960521 – 1_false-valid-free.c (ver Figura 4.2) da 3ª edição da Competição de Verificação de Software (SV-COMP 2014); este programa pertence a categoria *Memory Safety* (Beyer, 2014). Este programa foi utilizado como um exemplo de execução, uma vez que 55.6% das ferramentas na categoria *Memory Safety* não foram capazes de identificar o erro.

4.2 Etapa I: Identificação das propriedades de segurança

Map2Check adota o ESBMC para efetuar a identificação das propriedades de segurança. O ESBMC recebe como entrada o programa C que será analisado e usa a opção `--show-claims`, que mostra as propriedades de segurança que o ESBMC identifica como aquelas que podem vir a ser violadas. No contexto de *bounded model checking*, uma `claim` é o mesmo que uma propriedade. Exemplos de propriedades de segurança são: *buffer overflow*, divisão por zero, *overflow* aritmético e assim por diante.

A necessidade da verificação destas *claims* tem como principal argumento que elas podem ser violadas em alguma execução dos caminhos possíveis do programa. É interessante notar que as *claims* geradas automaticamente não necessariamente correspondem a erros (violação das propriedades), mas elas são apenas faltas em potencial. Se uma dessas *claims* corresponde a um erro, este erro deve ser determinado por uma análise mais aprofundada. No método Map2Check, esta análise consiste da tradução desta propriedade em um caso de teste e da respectiva execução deste. Cada propriedade de segurança (ou *claim*) gerada pelo ESBMC contém as seguintes informações: a propriedade identificada, localização e a assertiva a ser checada.

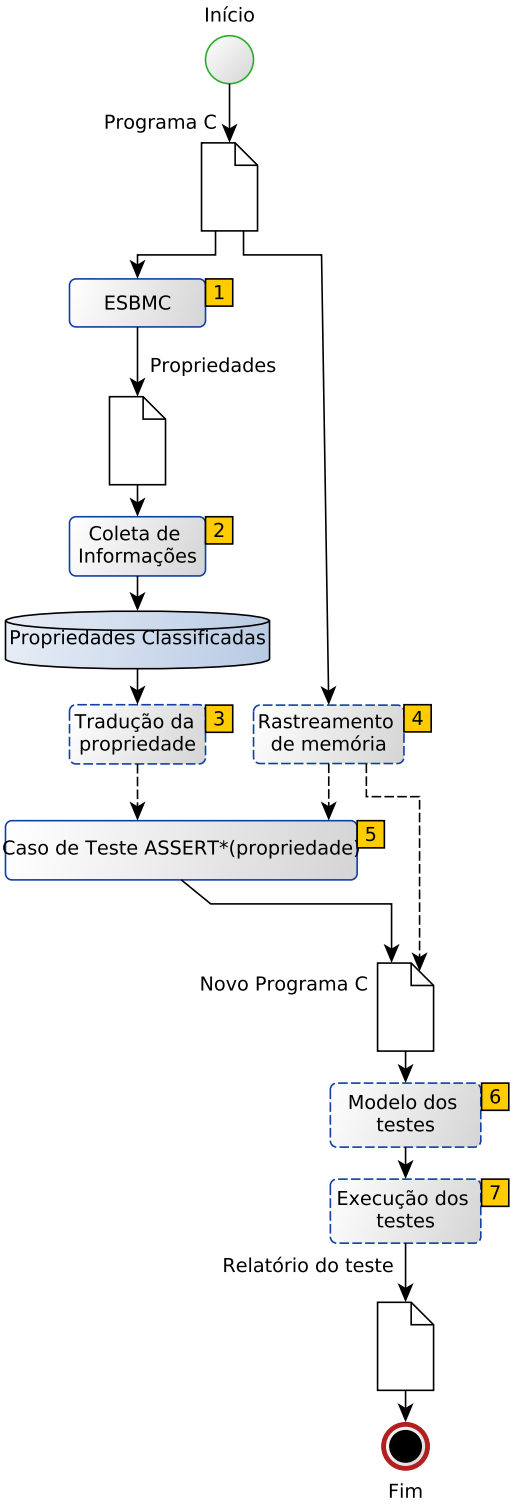


Figura 4.1: Fluxo da estrutura do método Map2Check.

```
1  #include <stdlib.h>
2
3  int *a, *b;
4  int n;
5
6  #define BLOCK_SIZE 128
7
8  void foo ()
9  {
10   int i;
11   for (i = 0; i < n; i++)
12     a[i] = -1;
13   for (i = 0; i < BLOCK_SIZE - 1; i++)
14     b[i] = -1;
15 }
16
17 int main ()
18 {
19   n = BLOCK_SIZE;
20   a = malloc (n * sizeof(*a));
21   b = malloc (n * sizeof(*b));
22   *b++ = 0;
23   foo ();
24   if (b[-1])
25   { /* invalid free (b was iterated) */
26   free(a); free(b); }
27   else
28   { free(a); free(b); } /* ditto */
29
30   return 0;
31 }
```

Figura 4.2: Programa 960521 − 1_false-valid-free.c do SVCOMP'14

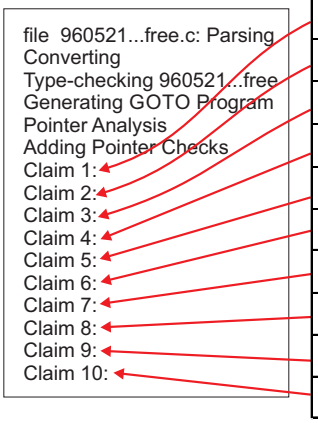
A Figura 4.3 apresenta um exemplo de uma *claim* gerada automaticamente pelo ESBMC. Neste exemplo, a *claim* 1 apresenta um potencial *lower bound* do objeto dinâmico "a" na linha 12 da função `foo`. Todas as *claims* identificadas pelo ESBMC no código C analisado são armazenadas para tratamento posterior nas próximas etapas do método.

```
$ esbmc --64 --no-library --show-claims
  960521-1_false-valid-free.c
file 960521-1_false-valid-free.c: Parsing
Converting
Type-checking 960521-1_false-valid-free
Generating GOTO Program
Pointer Analysis
Adding Pointer Checks
Claim 1:
  file 960521-1_false-valid-free.c line 12 function foo
  dereference failure: dynamic object lower bound
  !(POINTER_OFFSET(a) + i < 0) || !(IS_DYNAMIC_OBJECT(a))
```

Figura 4.3: Identificação das propriedades de segurança.

4.3 Etapa II: Coleta de Informações das Propriedades de Segurança

A segunda etapa do Map2Check faz um tratamento no resultado da Etapa I para simplesmente coletar e deixar mais acessível informações importantes e necessárias para as etapas seguintes. As informações a serem coletadas são: (i) a identificação da *claim* (ID = 7;); (ii) os comentários sobre a *claim* (*dereference failure: dynamic object upper bound*); (iii) o número da linha no código onde a *claim* ocorreu (Linha = 26); e (iv) a propriedade identificada pela *claim* (!(POINTER_OFFSET((void *)b) < 0) || !(IS_DYNAMIC_OBJECT(b))). O método utiliza expressões regulares para coletar as informações das *claims*. O resultado da aplicação da segunda etapa no código exemplo da Figura 4.2 pode ser visto na Figura 4.4, onde se apresenta as informações sobre as claims (propriedades) encontradas.



ID	Comentários	Linha	Propriedade
Claim 1	dereference failure: dynamic object lower bound	12	!(POINTER_OFFSET(a) + i < 0) || !(IS_DYNAMIC_OBJECT(a))
Claim 2	dereference failure: dynamic object upper bound	12	!(POINTER_OFFSET(a) + i >= DYNAMIC_SIZE(a)) || !(IS_DYNAMIC_OBJECT(a))
Claim 3	dereference failure: dynamic object lower bound	14	!(POINTER_OFFSET(b) + i < 0) || !(IS_DYNAMIC_OBJECT(b))
Claim 4	dereference failure: dynamic object upper bound	14	!(POINTER_OFFSET(b) + i >= DYNAMIC_SIZE(b)) || !(IS_DYNAMIC_OBJECT(b))
Claim 5	dereference failure: dynamic object lower bound	24	!(-1 + POINTER_OFFSET(b) < 0) || !(IS_DYNAMIC_OBJECT(b))
Claim 6	dereference failure: dynamic object upper bound	24	!(-1 + POINTER_OFFSET(b) >= DYNAMIC_SIZE(b)) || !(IS_DYNAMIC_OBJECT(b))
Claim 7	dereference failure: dynamic object lower bound	26	!(POINTER_OFFSET((void *)b) < 0) || !(IS_DYNAMIC_OBJECT(b))
Claim 8	dereference failure: dynamic object upper bound	26	!(POINTER_OFFSET((void *)b) >= DYNAMIC_SIZE(b)) || !(IS_DYNAMIC_OBJECT(b))
Claim 9	dereference failure: dynamic object lower bound	28	!(POINTER_OFFSET((void *)b) < 0) || !(IS_DYNAMIC_OBJECT(b))
Claim 10	dereference failure: dynamic object upper bound	28	!(POINTER_OFFSET((void *)b) >= DYNAMIC_SIZE(b)) || !(IS_DYNAMIC_OBJECT(b))

Figura 4.4: Aplicação da segunda etapa.

4.4 Etapa III: Tradução das propriedades de segurança

Esta etapa tem como objetivo traduzir as *claims* geradas pelo ESBMC para assertivas no programa C analisado; estas *claims* possuem funções especificas que somente são executadas pelo ESBMC, por exemplo, `INVALID-POINTER`. Esta função verifica se um ponteiro é `NULL` ou um objeto inválido. Assim, o Map2Check traduz as *claims* geradas pelo ESBMC em assertivas para efetuar a verificação sem a intervenção do ESBMC. Nas *claims* geradas pelo ESBMC, identificou-se as seguintes funções: `POINTER_OFFSET`, `INVALID-POINTER`, `SAME-OBJECT`, `POINTER_OBJECT`, `DYNAMIC_SIZE`, `IS_DYNAMIC_OBJECT`, e outros elementos (por exemplo, variáveis temporárias) usados somente pelo ESBMC. O tradutor traduz cada função do ESBMC usando um *parser* de uma gramática (desenvolvida neste trabalho, ver Figura 4.5) para as *claims*.

As regras da gramática apresentadas na Figura 4.5 visam atender os principais padrões da formação da estrutura das *claims*. Na Figura 4.5 as regras significam: de 1 a 4 a identificação dos principais *tokens* e composição de variáveis temporárias das *claims*; a regra 5 (*mathFuncClaims*) os padrões para as expressões matemáticas entre as variáveis; a regra 6 (*allFuncClaims*) contêm as regras

⟨*startNegOp*⟩ = '!' (1)

⟨*lParen*⟩ = '(' (2)

⟨*rParen*⟩ = ')' (3)

⟨*varTmpEsbmc*⟩ = 'tmp$' , [0-9]+ (4)

⟨*mathFuncClaims*⟩ = ⟨*expression2Calc*⟩ | ⟨*exprUseArrayVar*⟩ (5)

⟨*allFuncClaims*⟩ = ⟨*funPointOffset*⟩ | ⟨*funInvalidPoint*⟩ | ⟨*funSameObject*⟩ (6)
| ⟨*funPointObject*⟩ | ⟨*funDynamicSize*⟩ | ⟨*funOIsDynamic*⟩ | ⟨*funNIsDynamic*⟩ | ⟨*funValidObject*⟩

⟨*opBaseFuncClaim*⟩ = (⟨*startNegOp*⟩ | ⟨*Empty*⟩) , ⟨*varTmpEsbmc*⟩ | ⟨*startNegOp*⟩ , ⟨*funOverflow*⟩ (7)
| ⟨*startNegOp*⟩ , ⟨*lParen*⟩ , ⟨*allFuncClaims*⟩ , ⟨*rParen*⟩ | ⟨*startNegOp*⟩ , ⟨*complCalOpClaim*⟩
| ⟨*startNegOp*⟩ , ⟨*mathFuncClaims*⟩ | ⟨*lParen*⟩ , ⟨*mathFuncClaims*⟩ , ⟨*rParen*⟩
| ⟨*mathFuncClaims*⟩

Figura 4.5: Pequeno trecho da gramática para as *claims*

de composição de cada função das *claims*, por exemplo, a função `IS_DYNAMIC` responsável por identificar se um dado objeto é dinâmico; e por último a regra 7 (*opBaseFuncClaim*) que contêm as estruturas básicas das *claims* com a chamada das suas respectivas funções, por exemplo, a *claim* $!(INVALID - POINTER(i + lin))$.

A identificação de cada *claim* e seus respectivos componentes são passados como entrada para o tradutor que aplica as regras apropriadas de transformação a cada *claim*. As regras de transformação visam converter as *claims* em funções que podem ser executadas pelo programa em C que está sendo analisado, sem a intervenção do ESBMC. Visando a execução das funções geradas pelo tradutor, Map2Check provê uma biblioteca para programas em C que suporta a execução das funções geradas pelo tradutor.

As regras de transformação são classificadas nas seguintes formas: (1) **Forma Simples** que tem o seguinte padrão $A \ \ expr \ \ B$, onde A e B são simples valores de variáveis e $expr$ é um operador matemático; (2) **Forma Simples de Negação** que tem o seguinte padrão $!(A \ expr \ B)$, onde este padrão se difere de (1) por negar a expressão; e finalmente (3) **Forma Complexa** que tem o seguinte padrão: $\gamma \parallel \alpha$, $!(\gamma \parallel \alpha)$, $!(\gamma \rightarrow \alpha)$, $!((\gamma \parallel \alpha) \ expr \ (\gamma \parallel \alpha))$, $!(\gamma \subset \alpha \parallel \alpha \subset \gamma)$, onde α e γ são funções das *claims* do ESBMC.

As regras de transformação são aplicadas às *claims* do ESBMC como seguem: na **Forma Simples** e **Forma Simples de Negação**, a transformação é executada pela manipulação somente do retorno de funções e variáveis temporárias do ESBMC, e os outros itens na *claim* não são modificados; na **Forma Complexa**, antes de aplicar as regras de transformação, o método precisa identificar se a *claim* tem mais de uma função (α e γ) ou uma estrutura condicional; nestes casos, cada função é manipulada individualmente e então todas as *claims* traduzidas são escritas. A seguir são apresentadas as regras de transformação:

1. **RETURN**. É efetuada uma leitura do código, utilizando como base o número da linha onde a *claim* foi identificada, visando identificar os argumentos retornados pela função `return`;

   Exemplo: $return\_value\_isalnum\$9 == 0$ é traduzido para $isalnum(src[*i - 1]) == 0$

2. **POINTER_OFFSET**. A função é suprimida restando somente as variáveis e suas respectivas referências, ou seja, os índices, assim focando nos valores de *offset* das variáveis;

   Exemplo: $(POINTER\_OFFSET(c :: replace :: locate :: pat) + offset)$ é traduzido para $^*(pat + offset)$

3. **INVALID-POINTER**. A função é substituída pela função do Map2Check `IS_VALID_POINTER_MF` que possui os seguintes parâmetros: (i) uma lista contendo todos os endereços de memória do programa, (ii) o endereço de memória do ponteiro e (iii) o endereço de memória que esta apontado. Vale ressaltar que neste caso os argumentos desta função são obtidos com base na variável indicada pela *claim* que contém esta função, com a exceção do (i) que é criado pelo Map2Check (ver Seção 4.5);

   Exemplo: $INVALID - POINTER(i + pat)$ é traduzido para
   $IS\_VALID\_POINTER\_MF(LIST\_LOG, (void*)\&(i + pat), (void*)(intptr\_t)(i + pat))$

4. **SAME-OBJECT**. Esta função sempre recebe dois objetos como parâmetros e é simplesmente traduzida pelo símbolo de comparação (==) entre os objetos.

   Exemplo:
   $SAME - OBJECT(c :: replace :: locate :: pat, \&c :: replace :: main :: 1 :: pat[0])$
   é traduzido para $(pat == pat[0])$

5. **POINTER_OBJECT**. A função é substituída pela função do Map2Check `IS_VALID_POINTER_MF` conforme apresentado no Item 3;

6. **DYNAMIC_SIZE**. A função é substituída pelo valor indicado para o tamanho de memória alocado, ou seja, o tamanho definido na declaração da variável que compõem esta função, onde este tamanho foi definido utilizando as funções *malloc*, *calloc* ou *realloc*. O tamanho da alocação de memória é obtido pela leitura e aplicação de expressões regulares no código dentro do escopo analisado, sendo esta leitura do ponto atual da função para os anteriores, visando à primeira identificação;

   Exemplo: $DYNAMIC\_SIZE(x)$ é traduzido para $(r * sizeof(double))$

7. **IS_DYNAMIC_OBJECT**. A função é substituída pela função do Map2Check `IS_VALID_DYN_OBJ_MF` e o seu argumento é utilizando como argumento para esta nova função do Map2Check que recebe como parâmetro os argumentos utilizando pela função `IS_VALID_POINTER_MF` descritos no Item 3.

4.5 Etapa IV: Rastreamento de Memória

O rastreamento de memória executado pelo Map2Check visa estender a verificação das *claims* relacionadas à segurança de ponteiros. Consequentemente, a identificação de ponteiros e objetos inválidos permite analisar desalocação inválida de memória e vazamentos de memória (do inglês *memory leak*). A execução do rastreamento de memória consiste em duas fases: (1) identificação e rastreamento das variáveis no código do programa analisado, bem como as operações e atribuições, e (2) instrumentar o código fonte do programa analisado com funções específicas que irão monitorar os endereços de memória e os endereços apontados por estas variáveis de acordo com a execução do programa.

O Algoritmo 2 mostra como a identificação e rastreamento das variáveis são executados. A complexidade do tempo de execução deste algoritmo é $O(n^2)$, onde n é o número de nós em uma árvore de sintaxe abstrata (do inglês *Abstract Syntax Tree* - AST) do programa analisado. Note que no algoritmo, e em sua explicação, usa-se os seguintes termos: **Objeto**, o que significa que a variável analisada é um ponteiro ou variável alocada dinamicamente; **Variável Simples**, que são variáveis que não são ponteiros; e **Mapeamento**, o que significa que uma dada variável do programa analisado está sendo identificada e as suas características e operações (declaração e atribuições) estão sendo coletadas e salvas.

O Algoritmo 2 recebe como entrada uma AST, que é gerada a partir do programa em C analisado. Map2Check utiliza Pycparser[1] que efetua a transformação de um código em C para uma AST. O algoritmo percorre cada nó da AST e, para cada nó, identifica o escopo local que é divido nas funções do programa (linha 4) e no escopo global (linha 8) que são as instruções fora das funções definidas no programa.

Na Linha 8 é iniciado o mapeamento das variáveis globais do programa. Este mapeamento identifica se o nó da AST refere-se a uma declaração de uma variável, o qual é identificado pelo tipo `Decl` na AST. A Linha 8 executa a chamada da função `getDataFromVar` que recebe dois parâmetros como argumento, `nó` e `habilitaBuscaGlobal`. O parâmetro `nó` é o nó atual da AST que contém a declaração de uma variável a ser mapeada e `habilitaBuscaGlobal` é um valor Booleano (0 para falso e 1 para verdadeiro). Neste caso particular, **1** significa que uma busca é executada em todas as funções do programa para monitorar as atribuições da variável identificadas no `nó`. Do mesmo modo, a Linha 4 do Algoritmo 2 identifica se o nó atual da AST refere-se a uma função do programa, para então realizar o mapeamento do nó da AST que refere-se a uma declaração de uma variável, mas apenas naquela função analisada.

A função `getDataFromVar` do algoritmo (linha 10) consiste na identificação se a variável mapeada é um ponteiro e, em seguida, coleta os dados desta variável. Se a variável não é um ponteiro (variável simples), a função apenas executa o mapeamento da variável. O mapeamento é realizado pela coleta e listagem dos dados fornecidos pela função `getNodeData` (na linha 21). A função `getNodeData` recebe como entrada o nó que está sendo analisado e obtém os seguintes dados: (i)

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

```
   Input: Árvore de Sintaxe Abstrata (AST)
   Output: A identificação das variáveis (Mapa)
 1 begin
 2   compound_func = Não especificado
 3   foreach nó IN AST do
 4     if tipo(nó) == FuncDef then
 5       compound_func = obtém a subárvore do nó
 6       foreach subNo ONDE compound_func == Decl do
         getDataFromVar(subNo, 0) ;
 7     end
 8     else if tipo(node) == Decl then getDataFromVar(nó, 1) ;
 9   end
10   Function getDataFromVar(nó, habilitaBuscaGlobal)
11     if tipo(nó) é um ponteiro then
12       if nó tem uma atribuição then
13         Mapeamento dos dados a partir de getNodeData (nó)
14       end
15       if habilitaBuscaGlobal then
16         searchVarAssigInAllFunctions (nó)
17       else
18         searchAssigIn (compound_func, nó)
19       end
20     else
21       Mapeamento dos dados a partir de getNodeData (node)
22     end
23   end
24 end
```

Algoritmo 2: Coleta das variáveis para o rastreamento de memória

o número da linha no código fonte em que a variável está localizada; (ii) o nome da variável; (iii) o nome do escopo/função em que a variável está localizada; e (iv) se o objeto é dinâmico.

Na linha 11 se a variável é um ponteiro, então é executado o mapeamento do objeto (usando a função `getNodeData`) somente se a declaração identificada no nó analisado também inclui uma atribuição. De outra forma, o mapeamento é realizado somente após a primeira atribuição. Assim, o método evita o mapeamento de ponteiros não inicializados, que podem conter lixo de memória. Adicionalmente é feita uma pesquisa para rastrear as atribuições da variável ponteiro (operações, alocação e desalocação de memória) de acordo com seu escopo (linha 15). Se o objeto está no escopo global, então é feita uma pesquisa em todas as funções do programa (linha 16); caso contrário, a pesquisa é realizada apenas no escopo onde o objeto está localizado (linha 18).

A segunda fase é instrumentar o código fonte com funções específicas do Map2Check que irão monitorar os endereços de memória e os endereços apontados pelas variáveis de acordo com a execução do programa. Para cada linha identificada no mapeamento (na fase anterior) para o programa analisado, o método proposto insere, após a linha identificada, a função `mark_map_MF`, onde esta função recebe como entrada os dados mapeados para aquela linha. A função

`mark_map_MF` gerencia uma lista (denominada de `LIST_LOG`) de variáveis que contêm: o endereço da memória; o endereço da memória que aponta; o identificador do seu escopo; um identificador quando é dinâmico; um identificador quando é executado a função *free*; e o número da linha do código fonte. A lista `LIST_LOG` possui o rastreamento dos endereços de memória já executadas para aquele ponto atual do programa. No Map2Check, foi desenvolvido um biblioteca em C que contém as funções específicas, que permitam a execução da função `mark_map_MF`, bem como as funções mencionadas anteriormente.

A verificação das propriedades analisadas é realizada através da aplicação e execução das funções da biblioteca do Map2Check, como mostrado na lista a seguir. As funções nos itens 3 e 4 são geradas como casos de teste pelo Map2Check e não são fornecidos a partir das *claims* do ESBMC. Da mesma forma, o Map2Check também fornece casos de teste para operações de união para verificar se ocorre a reescrita de endereço de memória dinâmico.

1. **IS_VALID_DYN_OBJ_MF**. Esta função identifica se um objeto dinâmico é válido. Neste caso, o método efetua uma busca na lista `LIST_LOG` pelo o endereço de memória apontado pela variável que está sendo rastreada. Se o endereço de memória for encontrado, o método adota estas verificações: (1) o método efetua uma busca na lista para identificar se o endereço de memória apontado foi previamente rastreado; e (2) o método busca na lista pelo o atributo que identifica se a variável é ainda um objeto dinâmico, isto é, se a variável aponta para um endereço válido.

2. **IS_VALID_POINTER_MF**. Esta função busca na lista `LIST_LOG` apenas pelo endereço de memória apontado pela variável analisada, de modo a identificar se a variável está apontando para um endereço válido. Se a memória aponta para um objeto dinâmico, então ele verifica se é um objeto válido usando a função `IS_VALID_DYN_OBJ_MF`.

3. **INVALID_FREE**. Esta função identifica se um dado objeto dinâmico pode ser liberado/desalocado da memória adequadamente, por exemplo, usando a função *free* da linguagem de programação C. A biblioteca efetua a chamada da função `IS_VALID_DYN_OBJ_MF` para identificar se o objeto dinâmico é válido.

4. **CHECK_MEMORY_LEAK**. A função identifica se no fim da execução do programa alguma memória alocada não foi liberada. Esta função efetua uma busca na lista `LIST_LOG` pelos endereços de memória que ainda são dinâmicos, verificando na lista se o atributo que identifica se um endereço é dinâmico é válido. Como resultado, se for identificado em um dado ponto do programa que existe algum objeto dinâmico, a função identifica este resultado como um *memory leak*.

A Tabela 4.1 apresenta um exemplo da execução do rastreamento de memória do programa analisado (ver Figura 4.2). Analisando a execução do rastreamento do método proposto, identificou-se que o programa analisado possui um inválido *free* na linha 28. Isto é causado porque na linha 22, a variável `b` foi iterada, como apresentado no `ID` = 4 (da Tabela 4.1) que aponta para o

endereço 0xb44034. Assim, o inválido *free* apresentado na linha 28 é apresentado no `ID` = 260 da tabela, uma vez que o endereço de memória que o ponteiro aponta não é uma requisição para um bloco de endereço de memória válido do *heap*, como apresentado na linha da Tabela 4.1 onde o `ID` = 4 e o campo `É dinâmico` = 0.

ID	Endereço de Memória	Aponta para	Escopo	É dinâmico	É um *Free*	Número da Linha
260	0x601050	0xb44034	global	0	1	28
259	0x601060	0xb44010	global	0	1	28
...	...	...	...	...	...	...
133	0xb44034	(nil)	global	0	0	14
...	...	...	...	...	...	...
6	0xb44010	(nil)	global	0	0	12
5	0x7fff39f18a2c	(nil)	foo	0	0	10
4	0x601050	0xb44034	global	0	0	22
3	0x601050	0xb44030	global	1	0	21
2	0x601060	0xb44010	global	1	0	20
1	0x601058	(nil)	global	0	0	4

Tabela 4.1: O resultado da execução do rastreamento de memória no programa analisado.

4.6 Etapa V: Instrumentação do código com assertivas

Esta etapa tem como objetivo criar casos de teste, baseado em assertivas, que são incluídos no código fonte com a sua respectiva propriedade segurança/*claim* gerada pelo ESBMC e também pelo Map2Check. Essa etapa adiciona uma assertiva que contém a propriedade de segurança identificada na Etapa II (ver Seção 4.3) e na Etapa IV (ver Seção 4.5). Esta assertiva pode ser um simples `assert` da linguagem C ou uma assertiva de um *framework* de testes unitários, neste trabalho foi adotado o *framework* de testes CUnit.

Esta etapa identifica a linha de código fonte a partir de cada propriedade identificada, a fim de acrescentar uma assertiva em uma linha anterior, que é identificado pela propriedade no código fonte que está sendo verificado. Por exemplo, no programa da Figura 4.2 para a linha 28 é adicionada a seguinte assertiva: `ASSERT(INVALID_FREE(LIST_LOG, (void *)(intptr_t)(b), 28))`.

4.7 Etapa VI: Implementação dos testes

Esta etapa aplica um modelo para a execução dos testes ao programa analisado. O método proposto tem dois modelos. O primeiro usa somente assertivas do C, onde é inserido um `include` para a biblioteca do Map2Check na nova instância do código fonte do programa analisado. O segundo é o modelo para CUnit, onde é aplicado um *template* fornecido pelo método ao o programa analisado, que tem os seguintes itens: (i) *includes* para o CUnit, biblioteca do Map2Check, e do programa analisado; (ii) funções de configuração para o CUnit; (iii) funções que têm os casos de teste que

serão testados; e (iv) uma nova função `main` que será executada pelo CUnit. A Figura 4.6 mostra os itens do *template* e seus respectivos códigos.

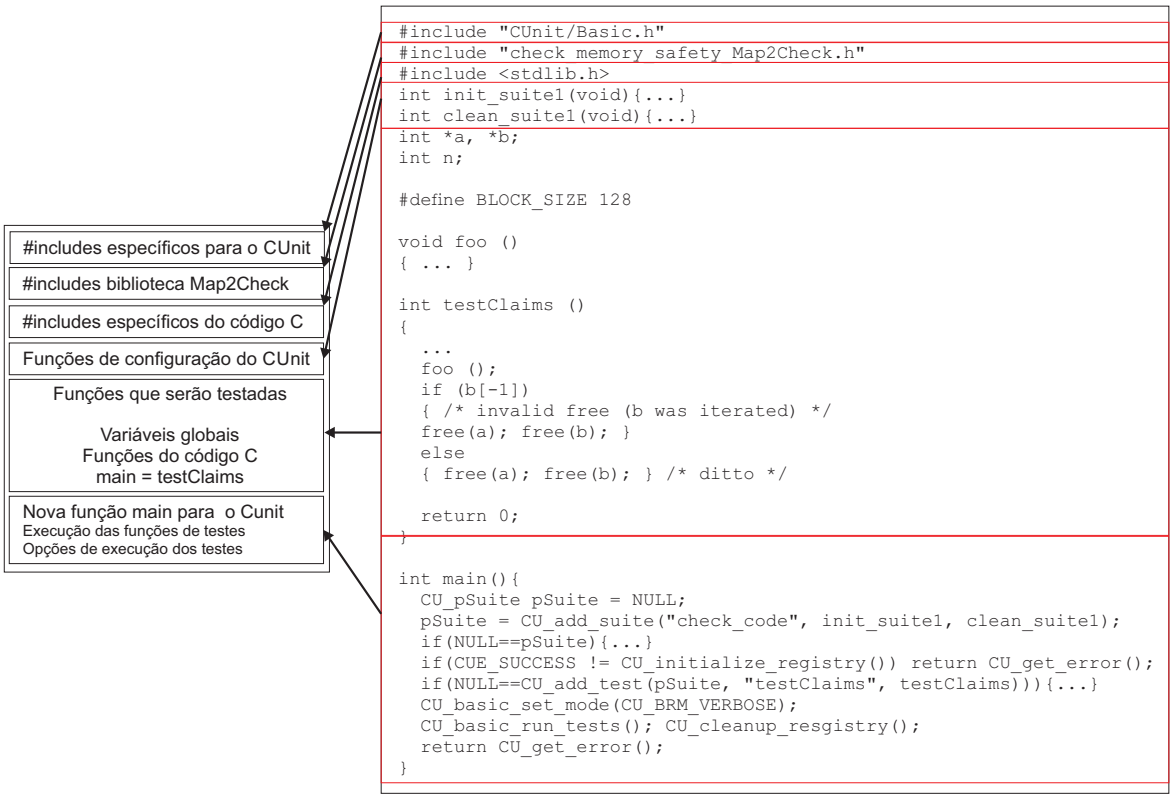


Figura 4.6: `Template` para execução dos testes com CUnit.

A biblioteca do CUnit e do Map2Check são coletadas diretamente do *template* provido pelo método. Os *includes* do programa analisado são copiados diretamente do código fonte. As funções de configuração do CUnit são usadas a partir do *template*. O método proposto renomea a função *main* para *testClaims*, isto porque a nova função *main* e seu respectivo conteúdo são obtidos diretamente do *template* do método. Esta nova função *main* contém as chamadas de execução das funções de configuração do CUnit, bem como, chamada para função *testClaims* (antiga função *main*). O resultado da aplicação deste *template* é uma nova instância do programa analisado que está pronto para ser testado e executado pelo *framework* CUnit.

4.8 Etapa VII: Execução dos testes

Nesta última etapa, o Map2Check oferece duas opções: (1) executar os casos de teste usando assertivas da linguagem de programação C (adicionadas na etapa anterior) ou (2) executar os casos de teste usando assertivas do *framework* de teste de unidade, neste caso o CUnit. Objetivando exemplificar o resultado desta etapa, aqui se adotou a segunda opção. Como resultado, o CUnit executa os testes no novo programa que tem os casos de teste gerados a partir das propriedades de segurança do ESBMC e Map2Check, validando, assim, cada assertiva. Basicamente, os casos de teste são analisados durante a execução da nova instância do programa analisado, onde cada caso

de teste gerado pelo método proposto pode resultar em sucesso ou falha, ou seja, violar ou não a propriedade analisada. Cada falha no teste é reportada pelo *framework* no final da execução do novo programa. A Figura 4.7 apresenta o resultado da execução desta última etapa.

```
VIOLATED PROPERTY
  Type    : Invalid FREE
  Location: In the line {28}
  Last Use: In the line {22}

FAILED
    1. mf_960521-1_false-valid-free.c:108
INVALID_FREE(LIST_LOG, (void *)(intptr_t)b,28)

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      0      1        0
             asserts    516    516    515      1      n/a

Elapsed time =    1.880 seconds
```

Figura 4.7: Resultado da execução com o Map2Check.

Vale a pena notar que os casos de teste são analisados durante a execução do programa, assim é possível melhorar a cobertura da execução do teste do programa adotando diferentes entradas de teste para o programa. Por exemplo, adotando a ferramenta PathCrawler (Williams *et al.*, 2005b) que gera automaticamente entradas de teste para as funções escritas em ANSI C. A análise do PathCrawler é baseado na análise dinâmica e usa a lógica restrições para resolver (parcialmente) predicados nos caminho de execução do programa e identificar entradas de teste para as funções.

4.9 Resultados Experimentais com Map2Check

Esta seção descreve o planeamento, concepção, execução e análise dos resultados de um estudo empírico realizado com o objetivo de avaliar o método proposto para criação e verificação de casos de teste para o gerenciamento de memória, quando aplicado à verificação dos *benchmarks* padrão ANSI-C e, além disso, uma comparação com as ferramentas: Valgrind's Memcheck (Nethercote e Seward, 2007), CBMC (Clarke *et al.*, 2004a), LLBMC (Merz *et al.*, 2012), CPAChecker (Beyer e Keremoglu, 2011), Predator (Dudka *et al.*, 2014), e ESBMC (Cordeiro *et al.*, 2012a).

Os experimentos foram conduzidos em um computador Intel Core i7-2670QM CPU, 2.20GHz, 32GB RAM com Linux OS. O método proposto está implementado em uma ferramenta denominada de Map2Check usando o *model checker* ESBMC.

4.9.1 Planejamento e Projeto dos Experimentos

Esta avaliação empírica tem como objetivo analisar a capacidade do método Map2Check gerar casos de teste relacionados ao gerenciamento de memória e para verificá-los. Assim, investiga-se as seguintes questões de pesquisa:

QP1: Os casos de teste gerados pelo método Map2Check são suficientes para identificar um dado defeito no programa analisado?

QP2: Qual é a capacidade do Map2Check para verificar os casos de teste?

QP3: Qual é a capacidade do Map2Check em detectar defeitos de gerenciamento de memória em comparação com as ferramentas existentes na literatura?

Visando responder as três questões de pesquisa, considerou-se 61 programas em ANSI-C da categoria *Memory Safety* do *benchmark* do SV-COMP 2014 (Beyer, 2014). Neste caso, considerou-se apenas os programas relacionados com a segurança da memória de programas. Nesta categoria, as propriedades a serem verificadas são: (i) **p_valid-free** - Todas as desalocações de memória são válidas; (ii) **p_valid-deref** - Todas as referências a ponteiros são válidas; e (iii) **p_valid-memtrack** - Toda a memória alocada é rastreada, ou seja, aponta para um endereço válido ou foi liberada corretamente.

Nos *benchmarks* do SV-COMP, alguns programas adotam funções específicas para expressar informações especiais. Por exemplo, na categoria *Memory Safety* tem a função `__VERIFIER_nondet_int()` que modela valores inteiros não determinísticos. No Map2Check, implementou-se uma função para simular os valores inteiros não determinísticos; a execução da função retorna um número aleatório (0 ou 1) a partir de um *array* de acordo com a seguinte distribuição definida: 30% para 0 e 65% para 1. Alguém poderia argumentar que esta abordagem depende de sorte para ter uma cobertura correta da execução do programa para validar as assertivas geradas. Isso pode ser verdade, mas adotou-se esta simulação de não determinismo já que em testes preliminares, esta simulação foi suficiente para detectar 70% das propriedades violadas.

Realizou-se a avaliação utilizando as seguintes ferramentas: (1) Aplicação do método Map2Check (ver Seção 4); Vale a pena notar que, neste caso, o ESBMC não é utilizado para verificar o programa, mas apenas para gerar as propriedades; (2) A execução da ferramenta Valgrind/MemCheck com as seguintes opções: `-leak-check = yes -undef-value-errors = yes`; (3) Os resultados da aplicação das ferramentas: CBMC, LLBMC, CPAChecker, Predator e ESBMC foram obtidas literalmente do SV-COMP 2014 (Beyer, 2014), porque as opções adotadas para executar as ferramentas desta experiência são as mesmas e o *hardware*[2] usado é similar. Vale a pena notar que é necessário compilar o programa a ser executado com Valgrind/Memcheck; portanto, adotou-se a função não determinística implementada na biblioteca do Map2Check.

Para análise dos programas usando o Map2Check e Valgrind/MemCheck, cada um dos programas do *benchmark* é executado 3 vezes, por causa do modelo não-determinístico nos programas. É importante notar que a partir dessas 3 executções, é considerado a execução classificada como `FAILED` (se houver), ou seja, uma execução que a ferramenta identificou uma violação de propriedade.

[2]Computador Intel i7-2600 CPU, 3.4GHz, 16GB RAM com GNU/Linux

4.9.2 Execução e Análise de Resultados do Experimento

Depois de executar as ferramentas sobre os *benchmarks*, obteve-se os resultados apresentados na Tabela 4.2, onde cada linha desta tabela significa: (1) nome da ferramenta (Ferramenta); (2) número total dos programas que satisfazem a especificação identificada pela ferramenta (Resultados Corretos); (3) número total dos programas que a ferramenta identificou um erro para um programa que atende a especificação, ou seja, alarme falso ou análise incompleta (Falsos Negativos); (4) número total dos programas que a ferramenta não identifica o erro, ou seja, erro não encontrado ou análise fraca (Falsos Positivos); (5) número total dos programas em que a ferramenta não conseguiu efetuar verificação devido à falta de recursos, falha da ferramenta (*crash*), ou a ferramenta excedeu o tempo de verificação de 15 min (Unknown e TO); (6) o tempo de execução em minutos da verificação para todos os programas na categoria (Tempo).

Ferramenta	CPAChecker	Map2Check	Valgrind	CBMC	Predator	LLBMC	ESBMC
Resultados Corretos	59	58	57	46	43	31	7
Falsos Negativos	0	0	0	8	0	0	0
Falsos Positivos	0	0	0	2	12	0	36
Unknown e TO	2	3	4	5	6	30	18
Tempo	23.33min	190.98min	151.57min	200min	76.66min	416.66min	139.06min

Tabela 4.2: Resultados da avaliação das ferramentas usando os *benchmarks* do SVCOMP'14.

Visando responder a questão de pesquisa **QP3** (ver Seção 4.9.1), a Tabela 4.2 demonstra que o Map2Check tem identificado 95.08% dos resultados corretos, enquanto o CPAChecker identificou 95.72%, Valgrind 93.44%, e as outras ferramentas somente conseguiram detectar menos de 76% dos resultados corretos. Note que Map2Check não gerou falsos positivos e falsos negativos. Map2Check gerou somente 3 resultados Unknown e TO. Acredita-se que isto é, em parte, por causa da execução concreta do programa. Com o objetivo de melhorar o tempo de verificação, no futuro pretende-se adotar uma verificação estática baseada em domínios abstratos (Ströder *et al.*, 2014).

Em relação as questões de pesquisa **QP1** e **QP2**, pode-se inferir que Map2Check gerou e verificou casos de teste com sucesso. Levando-se em conta a **QP1**, Map2Check foi capaz de gerar casos de teste corretos no sentido de: identificar um dado defeito no programa analisado e não gerar assertivas incorretas nos casos de teste que poderiam resultar em um falso alarme na execução do teste. Em resposta à **QP2**, também se identificou que a execução das funções instrumentados funcionaram corretamente, uma vez que as funções instrumentados apoiaram a execução dos casos de teste sem resultados incorretos.

Os resultados apresentados na Tabela 4.2 mostram que o Map2Check pode ser adotado como uma técnica complementar para a verificação realizada por ferramentas que utilizem BMC. Map2Check pode fornecer suporte para a análise dos programas, principalmente quando a ferramenta BMC não pode, geralmente por causa de um longo período de verificação (Time Out); ou quando existem falsos negativos ou falsos positivos. Portanto, se comparar os resultados do ESBMC (que foi o

BMC adotado pelo método proposto) ao Map2Check, ESBMC identificou 7 resultados corretos enquanto que o Map2Check identificou 58 onde, neste caso, Map2Check pode ser visto como um complemento para o ESBMC. Da mesma forma, o ESBMC gerou 18 resultados `Unknown e TO`, mas o Map2Check foi capaz de determinar com sucesso o resultado para 15 destes 18 programas.

Analisando o consumo de memória pelas ferramentas em cada programa do *benchmark* do SV-COMP 2014, identificou-se que o Map2Check é a 2ª ferramenta que consome menos memória (total de 3.680 MB) na verificação de programas, e a 1ª é a ferramenta Predator (total de 1.600 MB), conforme mostrado na Figura 4.8. Analisando esta figura, identificou-se que a partir do 32º programa (a linha vertical na Figura 4.8), houve um aumento do consumo de memória para mais de 50 MB a partir de 5 de 7 ferramentas analisadas. No entanto, o Map2Check em 95% dos programas consumiu cerca de 50 MB. Assim, o Map2Check não teve variação considerável em relação ao consumo de memória que é diferente de outras ferramentas, como por exemplo, o LLBMC que consumiu mais de 10.000 MB para programas específicos do *benchmark*.

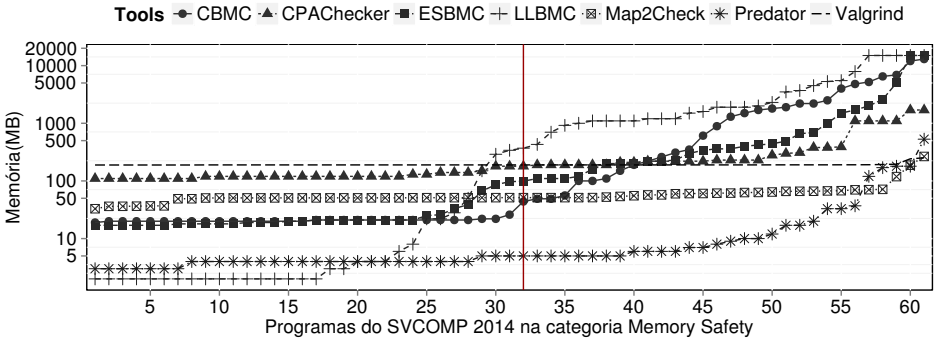


Figura 4.8: Memória consumida pelas ferramentas para cada programa.

Vale a pena notar que Map2Check consome menos memória do que ESBMC, devido ao fato que Map2Check adotou o ESBMC apenas para gerar as *claims*, que consome cerca de 20 MB. Desta forma, Map2Check em 53% dos programas tem consumido menos memória, com exceção da ferramenta Predator. No entanto, Map2Check identificou 25% mais resultados corretos que o Predator. Acredita-se que o consumo de memória do Map2Check pode ser melhorado devido ao fato de que somente para a geração de casos de teste foi utilizado 78, 98% da memória total. Por conseguinte, otimizar a tradução das *claims* teria um impacto significativo na redução do consumo de memória. Desta forma, se abordará este ponto em estudos futuros.

Observa-se que o tempo de execução da verificação do Map2Check foi 54, 16% mais rápido do que o LLBMC e 4, 5% do que o CBMC, como mostra a Tabela 4.2 e na Figura 4.9. É interessante notar que o tempo para gerar as *claims* é de cerca de 1s que está incluído no a Tabela 4.2 e Figura 4.9. Analisando o tempo de verificação do Map2Check, observa-se que, embora o tempo tenha sido maior do que as outras ferramentas, Map2Check só não identificou mais resultados corretos e gerou menos `Unknown e TO` do que a ferramenta CPAChecker. Acredita-se que o tempo total de verificação do Map2Check, em parte, pode ser explicado pela execução concreta dos programas com a simulação do não determinismo.

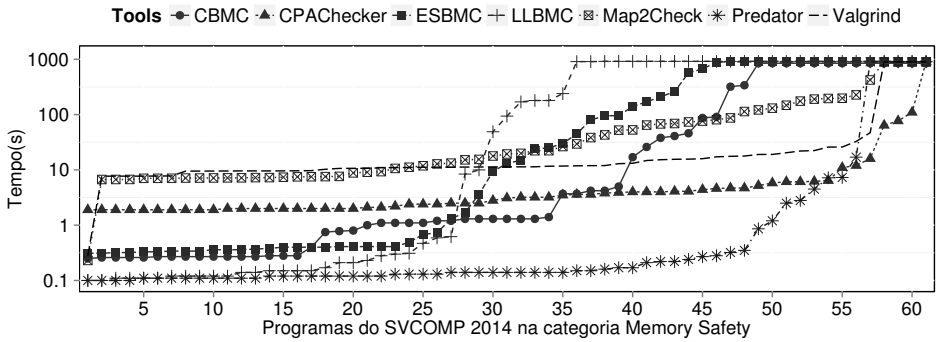


Figura 4.9: Tempo consumido pelas ferramentas na análise dos programas.

Pode-se argumentar que a execução concreta deve ser muito mais rápido do que a execução simbólica realizada pelas ferramentas adotadas neste experimento. Em parte, isso poderia ser explicado pela estratégia adotada para desenrolar laços nos programas e sua respectiva condição de parada, onde nos programas do *benchmark* se usa a função `__VERIFIER_nondet_int()` (que modela valores inteiros não determinísticos) em estruturas de laços. No Map2Check a implementação desta função retorna um número aleatório (0 ou 1) a partir de um *array* de acordo com a seguinte distribuição definida: 30% para 0 e 65% para 1. Assim, um BMC poderia completar a verificação mais rapidamente de um programa do que o Map2Check que depende de uma função aleatória para determinar a condição de parada de um laço.

Visando analisar a avaliação do Map2Check no contexto do SV-COMP 2014 (Beyer, 2014) na categoria *Memory Safety*, precisa-se levar em consideração as regras adotadas no SV-COMP 2014. Por exemplo, os resultados que podem ser definidos com pontos negativos, por exemplo, um `TRUE` incorreto recebe -8 pontos. Para mais detalhes, ver D. Beyer (Beyer, 2014). Neste caso, o Map2Check poderia alcançar o 1º lugar do SV-COMP 2014 na categoria *Memory Safety* com uma pontuação de 95 pontos, onde, na verdade, no SV-COMP 2014 o primeiro lugar foi do CPAChecker com a mesma pontuação de 95 pontos; o 2ª lugar foi do LLBMC com uma pontuação de 38 pontos; e o 3º lugar foi do Predator com uma pontuação de 14 pontos.

Recentemente, a ferramenta Map2Check participou do SV-COMP 2015 (Beyer, 2015), na categoria *Memory Safety*. As principais diferenças entre as duas competições foram: (i) o total de programas no SV-COMP 2014 foi de 61, e no SV-COMP 2015 foi de 205; e (ii) a pontuação foi atualizada com o objetivo de penalizar resultados incorretos. Map2Check ganhou em 6º de 9 ferramentas. Map2Check superou ferramentas como Forester (Holik *et al.*, 2015), Seahorn (Kahsai *et al.*, 2015) e o CBMC (Clarke *et al.*, 2004a). Analisando os resultados do Map2Check no SV-COMP 2015, identificou-se que o Map2Check foi a 4ª ferramenta que consumiu menos tempo (total de 8.400s) e memória (total de 70 GB) na verificação de programas. Map2Check gerou 0 falsos positivos e 15 falsos negativos. Estes resultados incorretos produzidos pela ferramenta na competição são devido a erros na implementação. Desde a apresentação da ferramenta, corrigiu-se alguns erros, e melhorou consideravelmente a sua execução. Tendo em consideração apenas os resultados corretos (os programas que satisfaz a especificação identificada pela ferramenta), o Map2Check iria ganhar

o segundo lugar, onde o número total de programas corretos foi 165 de 205; o tempo total da verificação foi de 2.100s; e o consumo de memória foi de 9.100 MB.

Estes resultados, embora de caráter preliminar, sugerem fortemente que o método proposto pode ser eficaz na geração e verificação de casos de teste de gerenciamento de memória para programas em C. Vale ressaltar que Map2Check na verificação efetuada reporta o rastreamento do programa que guia os desenvolvedores para os locais onde estão os erros de gerenciamento de memória causados pela violação de uma propriedade e, em seguida, os desenvolvedores podem corrigir os erros. Portanto, neste trabalho se argumenta que o Map2Check integra teste e verificação. O teste baseia-se na análise dinâmica e verificação de assertivas. As assertivas contêm um conjunto de especificações para a validação de blocos de memória. Esta verificação é semelhante à realizada por Delahaye *et al.* (2013), onde as condições de Pré e Pós condições baseadas na especificação formal do programa são convertidos em código C executável.

4.10 Resumo

Neste Capítulo detalhou-se e exemplificou-se as etapas de execução do método Map2Check que é baseado no método FORTES (*FORmal unit TESt generation*) que visa a geração automática de casos de teste (baseados em assertivas) em programas em C usando as propriedades de segurança geradas por um *Bounded Model Checker*, objetivando uma verificação complementar à efetuada pelo BMC (Rocha *et al.*, 2010; Rocha, 2011). Na Seção 4.9, detalhou-se o planeamento, concepção, execução e análise dos resultados de um estudo empírico realizado com o objetivo de avaliar o método Map2Check para criação e verificação de casos de teste para o gerenciamento de memória. Adicionalmente, apresentou-se uma comparação do método Map2Check (que foi implementado em uma ferramenta de *software*) com as ferramentas: Valgrind's Memcheck (Nethercote e Seward, 2007), CBMC (Clarke *et al.*, 2004a), LLBMC (Merz *et al.*, 2012), CPAChecker (Beyer e Keremoglu, 2011), Predator (Dudka *et al.*, 2014), e ESBMC (Cordeiro *et al.*, 2012a). O resultado dos experimentos demonstram que o método Map2Check detectou os mesmos defeitos que as ferramentas no estado da arte em análise de memória, além de determinar mais resultados corretos que 5 das 6 ferramentas comparadas, incluindo o ESBMC. O Map2Check não gerou falsos positivos e negativos. Em uma comparação do Map2Check com o resultado das ferramentas apresentados no *Competition on Software Verification* 2014 (SV-COMP), na categoria *Memory Safety*, o resultado foi que o Map2Check teve a mesma pontuação que o 1º lugar. O Map2Check ganharia o 2º lugar em relação ao menor consumo de memória na verificação dos programas.

Capítulo 5

Instanciação de Programas em C pela Utilização de Contra-Exemplos de BMC

Este Capítulo descreve as principais etapas do método denominado de EZProofC para automatizar a reprodução do possível defeito identificado no código fonte pela análise dos contra-exemplos. É também apresentado um estudo experimental realizado com o objetivo de avaliar o método EZProofC quando aplicado à verificação de *benchmarks* padrão de programas ANSI-C. Adicionalmente, é apresentado uma comparação do método proposto com a ferramenta Frama-C (Canet *et al.*, 2009).

5.1 Método EZProofC

O **EZProofC** [1] visa explorar os contra-exemplos gerados pelo *model checker* ESBMC, de tal forma que ele possa gerar um novo código instanciado com dados do contra-exemplo, a fim de reproduzir o erro identificado. É importante ressaltar aqui que se pode adotar qualquer ferramenta BMC. O método proposto consiste das seguintes etapas:

- (I) Pré-processamento de código;
- (II) Verificação formal com ESBMC;
- (III) Instanciação de código; e
- (IV) Execução de código e confirmação de defeitos.

O método EZProofC foi originalmente proposto no trabalho de mestrado de Rocha *et al.* (2010). Entretanto, nesta tese o método foi ajustado e melhorado, principalmente no que concerne a execução das etapas do método, na formalização do algoritmo para coleta e instanciação dos dados

[1] Disponível em https://sites.google.com/site/ezproofc/

do contra-exemplo (etapa iii do método), e a integração com o método Map2Check para a tradução automática das propriedades de segurança (ver Seção 4.4) e o rastreamento de memória (ver Seção 4.5). A Figura 5.1 apresenta uma visão geral do método EZProofC. Vale ressaltar que os blocos com linha tracejadas na Figura 5.1 foram as melhorias implementadas neste trabalho.

Visando explicar as principais etapas do método proposto, utilizou-se o código *tTflag_arr_two_loops_bad.c* do *benchmark* Verisec [2] do programa Sendmail [3] que é um servidor padrão de e-mail (SMTP) Unix. Este código tem 64 linhas de código que visam um *parse* de uma *string* de dígitos em dois inteiros assinados.

5.2 Etapa I: Pré-processamento de Código

Na primeira etapa, o código analisado é pré-processado usando a ferramenta *UNCRUSTIFY*[4] que irá processar o código, de modo a definir um padrão de formatação, envolvendo itens como: identação, delimitadores de bloco, um comando por linha, definição de estruturas e outros aspectos de formatação, como apresentado no código exemplo da Figura 5.2. Esta etapa de pré-processamento permite uma melhor definição das estruturas contidas no código, facilitando a sua manipulação para a aplicação das próximas etapas. É importante notar que a Figura 5.2 apresenta somente um fragmento do código original.

5.3 Etapa II: Verificação Formal com ESBMC

Na segunda etapa, o EZProofC utiliza o ESBMC para verificar as propriedades que são violadas no código analisado. O ESBMC divide a verificação em dois níveis: No primeiro nível, o ESBMC determina quais propriedades podem ser violadas pela utilização de uma análise estática preliminar (utilizando interpretação abstrata), a fim de determinar localizações no programa que potencialmente contém um erro. Estas localizações são propriedades que são denominadas de *claims*. Devido à imprecisão da análise estática, existe a necessidade de ir para o segundo nível, isto é, o ESBMC precisa confirmar que essas *claims* são de fato erros genuínos usando uma técnica de verificação mais completa e precisa. Vale ressaltar que durante a verificação, o ESBMC adota a técnica *program slicing* (Tip, 1995) que visa simplificar determinadas expressões a serem analisadas.

[2] http://se.cs.toronto.edu/index.php/Verisec_Suite
[3] http://www.sendmail.org
[4] http://uncrustify.sourceforge.net

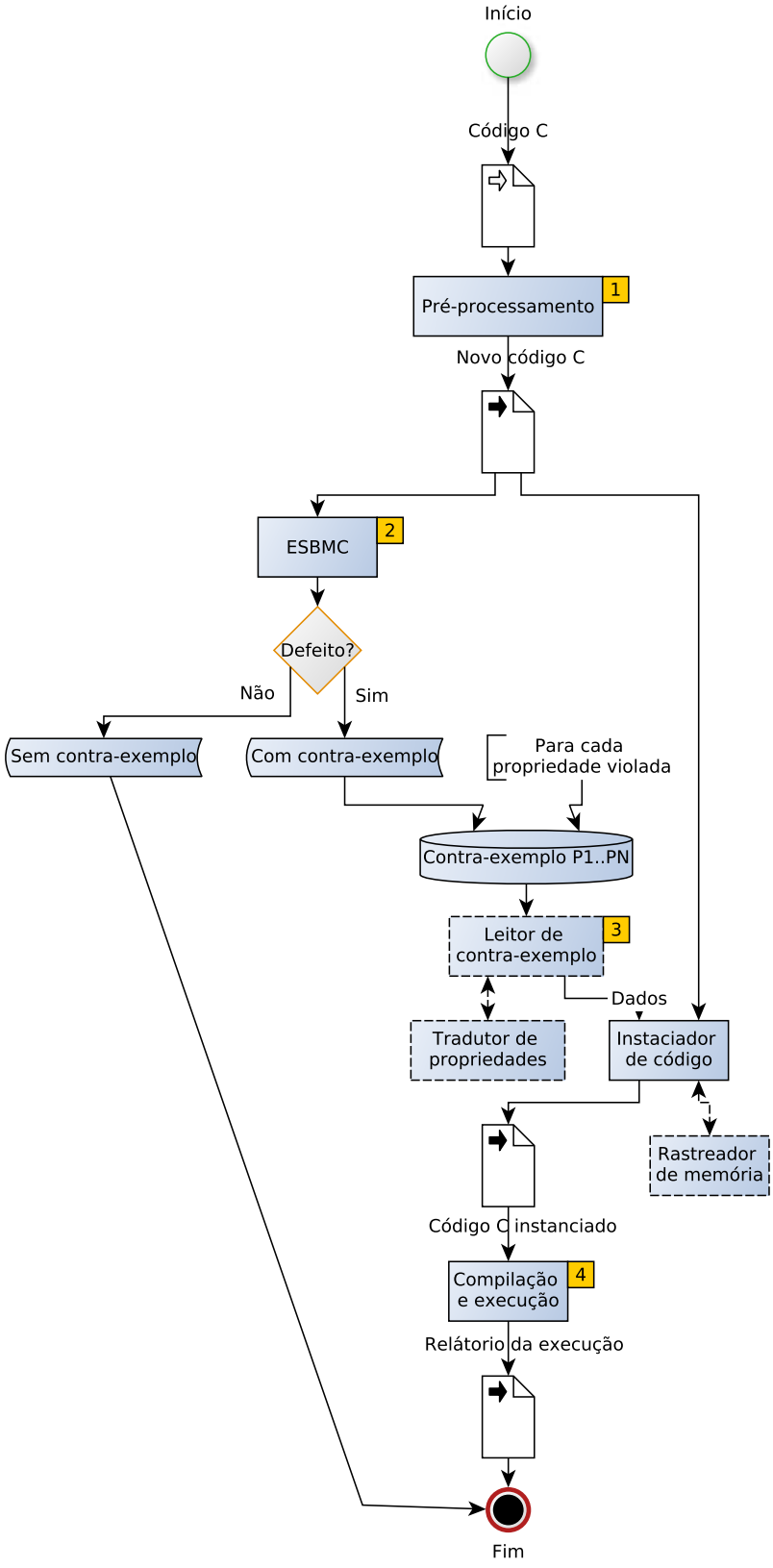


Figura 5.1: Estrutura do fluxo do método proposto.

```
1  #define INSIZE 14
2  int main (void){
3  unsigned char in[INSIZE+1];
4  unsigned char c;
5  int i, j;
6  int idx_in = 0;
7  ...
8  /*accumulate last(int) from in (char[])*/
9  c = in[idx_in];
10 if (c == '-')
11 {
12  i=0;
13  idx_in++;
14  c = in[idx_in];
15  while (('0' <= c) && (c <= '9'))
16  {
17    j = c - '0';
18    i = i * 10 + j;
19    idx_in++;
20    c = in[idx_in];
21   }
22 }
23 }
```

Figura 5.2: Fragmento do código C já pré-processado.

O resultado de verificação pode ser classificado em dois modos: o código foi verificado e não há nenhum contra-exemplo, isto é, a propriedade foi verificada mas nenhum erro foi encontrado até um dado *bound k*; ou o código foi verificado e possui um contra-exemplo, ou seja, uma violação da propriedade foi encontrada, como mostrado na Figura 5.3, que apresenta a violação da propriedade "`idx_in<15`" identificada no fragmento do código apresentado na Figura 5.2 (linha 20). Visando explicar claramente cada umas das próximas etapas do método, decidiu-se analisar somente uma *claim* especificada para a linha 20 da Figura 5.2.

A Figura 5.3 apresenta os estados da execução do programa com as variáveis e seus respectivos valores que reproduzem a violação da propriedade. A propriedade "`idx_in<15`" é violada devido ao fato de que o índice do *array* `in`, a variável `idx_in` excede o limite superior do *array* `in` como foi definido na linha 3 (`in[INSIZE+1]`) na Figura 5.2, onde `INSIZE` é definido com o valor 14. Como o *loop* na linha 15 não controla o valor da variável `idx_in`, no estado 98 (apresentado no contra-exemplo, Figura 5.3) esta variável recebe um valor maior que o limite superior do *array* `in`, assim causando a violação da *claim* relacionado ao `UPPER BOUND`.

```
Counterexample:

(......)
State 55 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 9 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::c=45 (00101101)

State 58 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 13 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::idx_in=9 (00000000000000000000000000001001)

State 59 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 14 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::c=48 (00110000)

State 96 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 17 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::j=3 (00000000000000000000000000000011)

State 97 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 18 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::i=33 (00000000000000000000000000100001)

State 98 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 19 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::idx_in=15 (00000000000000000000000000001111)

State 93 file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 20 function main thread 0
------------------------------------------------------------
  pre_tTflag_arr_two_loops_bad::main::1::c=51 (00110011)

Violated property:
  file ccode.pre/pre_tTflag_arr_two_loops_bad.c line 20 function main
  array `in' upper bound
  idx_in < 15

VERIFICATION FAILED
```

Figura 5.3: Contra-exemplo.

5.4 Etapa III: Instanciação de Código

A terceira etapa é dividida em duas fases: a análise de contra-exemplos produzidos na etapa 2 (Seção 5.3) e na geração de um novo código C instanciado [5] com os valores das variáveis apresentadas no contra-exemplo. O Algoritmo 3 detalha o modo de execução de ambas as fases. A complexidade do tempo de execução do algoritmo é $O(n + m)$, onde n é o tamanho do código C analisado e m é tamanho do contra-exemplo. As entradas do Algoritmo 3 são o código analisado (`Code`) e o seu respectivo contra-exemplo (`CE_Out`). Inicialmente, o contra-exemplo (`CE_Out`) é analisado para coletar diversos dados, tais como: (1) as variáveis envolvidas na violação da propriedade; (2) o número da linha onde os valores são atribuídos às variáveis; e (3) o valor de cada variável. Estas informações são obtidas pela função `GetValuesCEER` (linha 1 do Algoritmo 3) pela aplicação de expressões regulares no arquivo do contra-exemplo. Esta função retorna um conjunto que contêm dados sobre as variáveis encontradas no contra-exemplo (por exemplo, `Var{vline = 9,var = c,vvalue = 45}`), a propriedade violada (`P`) e o número da linha onde a propriedade foi violada (`line_p`).

[5] Neste caso instanciar significa criar um novo código modificado baseado no código original

```
   Input: Code, CE_Out
   Output: Novo código instanciado
   // Primeira fase
 1 Var,P,line_p ← GetValuesCEER(CE_Out);
 2 SCE ← {Var{vline,var,vvalue},P,line_p};
 3 size ← GetTotalLineCE(SCE[Var[]]);
 4 Lines, tline ← GetValuesCode(Code);
 5 SCode ← {Lines{ }, tline};
 6 UPCASE ← {Conjunto de casos específicos para coleta de dados do contra-exemplo};
 7 i, k ← 1;
   // Segunda fase
 8 while i ≤ SCode[tline] do
 9     if i == SCE[Var[vline[k]]] AND k ≤ size then
10         if SCE[P] OR SCE[Var[vvalue[k]]] ∈ UPCASE then
11             New_Line ← StartTrigger(SCE[P], SCE[Var[vvalue[k]]]);
12             WriteLineCode(New_Line); k ← k + 1;
13         end
14         else
15             New_Line ← "SCE[Var[var[k]]] = SCE[Var[vvalue[k]]]";
16             WriteLineCode(New_Line); k ← k + 1;
17         end
18     end
19     else
20         WriteLineCode(SCode[Lines[i]]); i ← i + 1;
21     end
22 end
```

Algoritmo 3: Algoritmo Counterexample2NewCode

Após a extração dos dados do contra-exemplo, o algoritmo percorre as linhas o código C analisado (linha 8), bem como o contra-exemplo. Se o número da linha da variável identificado no contra-exemplo é igual ao número de linha do código analisado, o algoritmo pode gerar uma nova linha de código, onde a variável identificada recebe o valor obtido do contra-exemplo. Por exemplo, os seguintes valores coletados do contra-exemplo `line = 9, var = c` e `value = 45` resulta na variável `New Line` (linha 15 do algoritmo) em receber o texto "c = 45", a qual irá gerar a nova linha do código. É importante ressaltar que a instanciação das variáveis no novo código é executada estritamente de acordo com a sequencia identificada no contra-exemplo. Por exemplo, se a mesma variável no contra-exemplo é mencionado várias vezes na mesma linha (por exemplo, em *loops*), apenas o último valor encontrado na contra-exemplo será atribuído a variável no código instanciado.

Visando melhorar a coleta de dados do contra-exemplo, o método proposto pode exigir uma abordagem distinta em alguns casos específicos, quando é aplicada à etapa de verificação do método EZProofC (ver Seção 5.3) ou acionado pela análise de contra-exemplo. A linha 10 do Algoritmo 3 verifica se a propriedade ou uma variável no contra-exemplo está em um conjunto dos casos específicos já predefinidos (linha 6 variável `UPCASE`). Assim, se houver algum caso específico no contra-exemplo que tenha sido identificado, a abordagem adequada é aplicada através da adoção da função `StartTrigger` na linha 11, como é apresentado seguir. Vale ressaltar que com relação ao tratamento da propriedade identificada no contra-exemplo o método EZProofC utiliza o Tradutor de *Claims* (ver Seção 4.4) e do Rastreamento de Memória (ver Seção 4.6)

anteriormente mencionado no método FORTES, a fim de auxiliar no processo de reprodução do erro identificado.

(i) Quando a violação de uma propriedade é identificada e não há informações suficientes sobre o contra-exemplo, é necessário utilizar na etapa de verificação com ESBMC, particularmente em código pequenos, a opção `--no-slice` o qual não remove equações não utilizadas do programa para gerar o contra-exemplo. Outra forma de diversificar os valores das variáveis, e assim, o resultado em contra-exemplo, consiste em aplicar valores não-determinísticos a elas (por exemplo, `a[0]=nondet_int()`);

(ii) Em alguns casos específicos, a violação da propriedade `UPPER BOUND` pode gerar um contra-exemplo sem os dados sobre o limite superior do *array*. Neste caso, em primeiro lugar, o método identifica o nome do *array* e por meio de uma análise do código, pode identificar o limite superior do *array*. Este procedimento é realizado por dois elementos: o primeiro é a função `NUM_OF(arr)` para obter o tamanho do *array*; e o segundo elemento é uma assertiva que irá conter o resultado da função `NUM_OF(arr)` e o índice do *array* que foi identificado no contra-exemplo, deste modo a estrutura da assertiva é dada da seguinte forma `assert((N)<=NUM_OF(arr)-1)`, onde `N` é o valor do índice que será adotado para validar o limite (*bound*) do *array*;

(iii) Considerando as violações de alocação dinâmica de memória, o método proposto analisa: (1) se o ponteiro aponta para um objeto inválido; (2) se o objeto é considerado um objeto dinâmico; (3) se o argumento na chamada da função *free* é ainda um ponteiro válido. O objetivo desta análise é obter uma declaração correta sobre a propriedade identificada.

A segunda fase desta terceira etapa do método EZProofC tem como objetivo gerar um novo código instanciado. O método então gera uma cópia do código original que está sendo analisado (na linha 20 do Algoritmo 3), e substitui as atribuições das variáveis utilizando os valores específicos identificados na primeira fase (na linha 12 ou 16 do algoritmo). Nos casos de propriedades como `UPPER BOUND` ou `LOWER BOUND`, o método proposto inclui as assertivas no código instanciado para reproduzir o erro, tal como mencionado anteriormente na linha 11 sobre os gatilhos na análise do contra-exemplo. Tais assertivas contêm as propriedades identificas no contra-exemplo. O resultado final desta etapa é um código C instanciado com os valores das variáveis que são extraídos do contra-exemplo, como demonstrado na Figura 5.4. É importante notar que no contra-exemplo (ver Figura 5.3), a propriedade violada foi um `UPPER BOUND`, e seus dados foram "`idx_in<15`". Neste caso, na linha 20 da Figura 5.4, o método proposto inclui uma assertiva visando reproduzir o erro identificado anteriormente.

Em particular, neste exemplo, é óbvio que a assertiva irá falhar. Isso ocorre porque a instrução anterior atribui exatamente um valor que contradiz a assertiva. No entanto, é importante observar que esta atribuição provém diretamente do contra-exemplo, o que implica que existe uma situação em que esta atribuição ocorre em um dos possíveis caminhos de execução do programa.

```
1  #define INSIZE 14
2  int main (void){
3  unsigned char in[INSIZE+1];
4  unsigned char c;
5  int i, j;
6  int idx_in = 0;
7  ...
8  /*accumulate last(int) from in (char[])*/
9  c =45 ; //<- by EZProofC
10 if (c == '-')
11 {
12  i =0 ;
13  idx_in = 9 ; //<- by EZProofC
14  c =48 ; //<- by EZProofC
15  while (('0' <= c) && (c <= '9'))
16  {
17    j =3 ; //<- by EZProofC
18    i =33 ; //<- by EZProofC
19    idx_in = 15 ; //<- by EZProofC
20    assert(idx_in <15); //<- by EZProofC
21    c =51 ; //<- by EZProofC
22  }
23 }
24 }
```

Figura 5.4: Código C instanciado com o contra-exemplo.

5.5 Etapa IV: Execução de Código e Confirmação de Erros

Na terceira etapa do método proposto é gerado um programa instanciado para cada propriedade violada. Nesta quarta etapa, cada código instanciado é compilado e executado. O resultado da execução demonstra o erro, `Line:20:main: Assertion & 'idx_in<15' failed. Aborted`, que foi apontado pelo contra-exemplo.

5.6 Resultados Experimentais com EZProofC

Esta seção descreve o planejamento, projeto, execução e análise dos resultados de um estudo experimental realizado com o objetivo de avaliar o método proposto quando aplicado à verificação de *benchmarks* padrão de programas ANSI-C e, além disso, uma comparação com a ferramenta Frama-C (Canet *et al.*, 2009; Kirchner, 2012) version Boron-20100401. Frama-C é um conjunto de ferramentas dedicadas à análise de programas escritos em C. Frama-C faz com que seja possível observar conjuntos de valores possíveis para as variáveis do programa em cada ponto de execução. Frama-C também permite verificar no código fonte se uma dada especificação formal fornecida é satisfeita. As especificações podem ser escritas em uma linguagem específica, neste caso, nas especificações de linguagem ANSI/ISO (ACSL). Os experimentos foram conduzidos em um computador Intel Core 2 Duo CPU, 2Ghz, 3GB RAM com Linux OS. O método proposto foi desenvolvido utilizando a versão 1.18 do ESBMC.

5.6.1 Planejamento e Projeto dos Experimentos

O objetivo desta avaliação empírica é analisar o impacto do método proposto com a finalidade de reproduzir os erros identificados pelo *model checker* ESBMC nos programas analisados. A fim de avaliar o método proposto, considerou-se 211 programas ANSI-C a partir de seis diferentes *benchmarks* selecionados, com o objetivo de avaliar a capacidade e o desempenho dos métodos e técnicas para a identificação e reprodução dos erros. Vale notar que essas suítes de *benchmark* de programas ANSI-C representam implementações reais.

Os *benchmarks* públicos utilizados foram: (i) EUREKA (Eureka, 2012) que contém programas que permitem avaliar a escalabilidade de *model checkers* na verificação de problemas complexos. É interessante observar que alguns programas representam mais do que uma execução, com diferentes dados de entrada. Por exemplo, o programa `bubble_sort1_13.c` representa 13 instâncias (de 1 até 13) do programa `bubble_sort1.c`. O programa `prim4_8.c` representa 5 instâncias (de 4 até 8) do programa `prim.c`; (ii) SNU (SNU, 2012) que contém programas em C utilizados para análise de pior caso de tempo de execução, onde tais programas são, em sua maioria, algoritmos de análise numérica e DSP (*Digital Signal Processing*); (iii) WCET (MRTC, 2012) que, do mesmo modo que o SNU, contém programas utilizados para análise do pior caso de tempo de execução; (iv) NEC (NEC Laboratories America, Inc., 2012) que contém programas em C que permitem facilmente verificar a detecção de erros, uma vez que este *benchmark* oferece programas ANSI-C identificados com e sem erros; (v) Siemens/SIR (SIR Project, 2012) que é um suíte de programas para um analisador léxico e correspondência de padrão; e (vi) alguns programas ANSI-C obtidos do tutorial do CBMC (*C Bounded Model Checker*) (Daniel Kroening, 2012).

Durante esta avaliação empírica, cada programa do *benchmark* foi executado com três métodos: (1) A aplicação do método EZProofC (ver Seção 5), ou seja, o pré-processamento de código, identificação de *claims*, verificação, análise de contra-exemplos, e instanciação código; (2) A aplicação da ferramenta Frama-C com a opção `-val`, o que significa que o *plug-in* de análise de valores é chamado de tal forma que ele calcula automaticamente domínios de variação para as variáveis do programa. Este *plug-in* é utilizado para inferir ausência/presença de erros de execução, e (3) A aplicação da ferramenta Frama-C com a opção `Jessie`, que é um *plug-in* que permite a verificação dedutiva de programas em C anotado com ACSL (Baudin *et al.*, 2009). As condições de verificação (CV) são verificadas pelo provador de teoremas `Z3` (Research, 2012), que é o provador padrão usado pelo ESBMC. Desta forma, a ferramenta Frama-C foi executado da seguinte forma: `frama-c -jessie -jessie-atp=z3 <file.c>`, onde `<file.c>` é o código C que será verificado.

5.6.2 Execução do Experimento e Análise de Resultados

Depois de executar os *benchmarks*, foram obtidos os resultados apresentados na Tabela 5.1, onde cada coluna desta tabela significa: (1) a identificação do programa (ID); (2) o nome do programa C e, adicionalmente, em certos casos, o intervalo das instâncias, por exemplo, file1_13.c, o que

significa que existem 13 instâncias, de 1 a 13. Na Tabela 5.1, os programas de 1 até 16 são do *benchmark* EUREKA, de 17 até 19 são do tutorial do CBMC, o programa 20 do *benchmark* da NEC, do 21 ao 22 do *benchmark* do SNU, o programa 23 do *benchmark* WCET, e o programa 24 do *benchmark* SIR; (3) o número de linhas do código do programa (#L); (4) a quantidade de *warnings* (#W) e o tempo de execução (TW) do Frama-C com o *plug-in* de análise de valores; (5) o número total de propriedades (ou `claims`) que podem ser violadas (#P), o tempo de execução da identificação propriedades gasto pelo ESBMC e EZProofC (TC), o tempo de execução da verificação de todas as propriedades pelo ESBMC (TV), número total de propriedades que tenham sido violadas e reproduzidas usando o método EZProofC (#V), e o número de linhas dos contra-exemplos (CE); e (6) o número de propriedades encontradas em comum (*Same Claims & Warnings*) entre o EZProofC (*claims*) e o Frama-C (*warnings*).

É importante notar que, para os programas com várias instâncias, o número de propriedades violadas apresentados é o maior valor entre as instâncias. Adicionalmente, no caso dos programas com mais que uma instância, o número de linhas nos contra-exemplos (#CE) e propriedades encontradas em comum (*Same Claims & Warnings*) é, respectivamente, o maior contra-exemplo encontrado e o maior número de propriedades encontradas em comum. Os resultados da aplicação do método proposto, bem como a ferramenta de EZProofC estão disponíveis em https://sites.google.com/site/ezproofc/.

ID	**Programa**	**#L**	**Frama-C**		**EZProofC/ESBMC**					***Same Claims & Warnings***
			#W	**TW**	**#P**	**TC**	**TV**	**#V**	**CE**	
1	bf5_20.c	49	6	<1s	33	<1s	<60s	0	-	0
2	bubble_sort1_13.c	51	2	<1s	25	<1s	<15s	0	-	0
3	fibonacci1_13.c	25	1	<1s	1	<1s	<1s	0	-	0
4	init_sel_sort1_13.c	54	2	<1s	25	<1s	<15s	0	-	0
5	minmax1_13.c	19	6	<1s	9	<1s	<3s	0	-	0
6	minmax_unsafe1_13.c	19	6	<1s	9	<1s	<4s	1	16	0
7	n_k_gray_codes1_13.c	45	36	<1s	22	<1s	<120s	0	-	11
8	no_init_bubble_sort_safe1_13.c	25	2	<1s	14	<1s	<7s	1	32	1
9	no_init_sel_sort1_13.c	41	5	<1s	25	<1s	<15s	12	144	3
10	no_init_sel_sort_safe1_13.c	28	5	<1s	14	<1s	<7s	1	32	3
11	no_init_sel_sort_unsafe1_13.c	28	5	<1s	14	<1s	<7s	1	32	3
12	prim4_8.c	79	12	<1s	30	<1s	<60s	0	-	3
13	selection_sort1_13.c	54	2	<1s	25	<1s	<15s	0	-	0
14	strcmp1_13.c	15	4	<1s	6	<1s	≈14400s	3	80	0
15	sum1_13.c	21	1	<1s	1	<1s	<1s	1	48	0
16	sum_array1_13.c	11	1	<1s	7	<1s	<3s	1	8	0
17	assert_unsafy.c	15	4	<1s	1	<1s	<1s	1	24	0
18	bound_array.c	16	2	<1s	10	<1s	<10s	1	30	1
19	division_by_zero.c	32	3	<1s	1	<1s	<1s	1	24	1
20	ex26.c	29	4	<1s	8	<1s	≈420s	2	1236	1
21	crc_det.c	125	1	<1s	15	<1s	≈840s	0	-	1
22	select_det.c	122	3	<1s	39	<1s	≈14400s	3	40	1
23	cnt_nondet.c	139	0	<1s	16	<1s	<1s	0	-	0
24	Siemens_print_tokens2.c	508	90	<1s	51	<1s	≈18000s	1	3344	34

Tabela 5.1: Detalhes relacionados a execução do *benchmarks*

Analisando a Tabela 5.1 pode-se observar que o método é EZProofC é escalável para qualquer tamanho de código e contra-exemplo, uma vez que a complexidade do algoritmo do método proposto é $O(n + m)$. O tempo de execução do EZProofC é, portanto, linear, mesmo quando se

consideram diferentes tamanhos de código, como se pode ver no tempo de execução dos experimentos.

Alguém poderia argumentar que os *benchmarks* selecionados podem não representar bem todos os cenários possíveis para a aplicação do método proposto, principalmente quando se leva em conta o tamanho dos programas em termos de LOCs. No entanto, como um exemplo, considere a experiência com o programa da Tabela 5.1 com ID igual a 20, que tem apenas 29 LOC, mas foi aquele que produziu alguns dos maiores contra-exemplos, neste caso 1236 linhas. Note ainda que, estes contra-exemplos, tem um rastreamento que mostra todas as variáveis, bem como as atribuições incluídas em uma execução específica (por exemplo, incluindo *loops*) que irá resultar na violação da propriedade que foi identificada pelo ESBMC, ou seja, o desenrolar de uma execução específica do programa. A desvantagem da ferramenta EZProofC é que ela conta com a escalabilidade do *model checker* adotado, uma vez que depende apenas para gerar os contra-exemplos. Além disso, o método proposto é capaz de ser escalável para grandes tamanhos de contra-exemplos, neste caso, entre 8 a 3344 linhas.

É importante salientar que os resultados sobre *warnings* (na coluna #W) da ferramenta Frama-C com o *plug-in* de análise de valores são muito eficientes, proporcionando ao usuário um bom suporte para explorar o código que foi analisado. No entanto, tais *warnings* não foram apenas relacionados com propriedades de segurança, mas envolveu uma análise da estrutura do código (por exemplo, retorno de funções). Isso explica, em parte, por que o número de propriedades entre o EZProofC (*claims*) e o Frama-C (*warnings*) (coluna *Same Claims & Warnings* da Tabela 5.1) são diferentes.

A ferramenta Frama-C também permite o uso de outros *plug-ins*, por exemplo, o *plug-in* `Jessie`, que se destina a efetuar a verificação dedutiva de programas C. O programa em C não necessita de ser completo ou anotado com especificações para ser analisado com o *plug-in* Jessie (INRIA, 2010). No entanto, no experimento realizado, o *plug-in* Jessie não encontrou qualquer violação de propriedade, ou seja, nenhum erro foi encontrado, embora Frama-C tenha apontado vários *warnings*. O *plug-in* Jessie também permite provar que as funções satisfazem as suas especificações que podem ser expressas em ACSL. Entende-se que a verificação do Frama-C poderia ser melhorada escrevendo essas especificações no código C analisado. No entanto, a inclusão de tais especificações pode ser difícil e propensa a erros, especialmente para código legado. Portanto, se comparar o uso de Frama-C/Jessie e o EZProofC, argumenta-se que a grande vantagem do EZProofC está em não necessitar de tais especificações auxiliares. EZProofC é um método completamente automático que não necessita da escrita de especificações, e nem pré-condições e pós-condições. Além disso, no caso de as Frama-C, o usuário tem que atuar explicitamente para reproduzir o erro, usando os valores calculados.

Nestes experimentos tem uma situação que deve ser pontuada sobre a aplicação do método EZProofC. O programa 24 na Tabela 5.1 é considerado ser uma versão de ouro (ou seja, a suposta versão correta do programa). Levando-se em conta que este código é grande, e exige uma significativa quantidade de memória, a verificação foi feita por funções. Particularmente, verificou-se a função `get_token`. O erro identificado neste código é a violação de um `UPPER`

`BOUND` do *array* `buffer`, que é declarado com o limite superior igual a 80. No entanto, com base na reprodução do erro, percebe-se que o índice desse *array*, a variável `i`, ultrapassou o limite superior, causando a violação da propriedade $i < 81$, da mesma forma que foi identificado no trabalho de Cordeiro *et al.* (2012a).

Analisando os resultados obtidos nestes experimentos, identificou-se que a manipulação do contra-exemplo, nem sempre é uma tarefa trivial. Durante as experiências, obteve-se contra-exemplos relativamente grandes (por exemplo, cerca de 3344 linhas). No entanto, a aplicação do método proposto diminui substancialmente a complexidade desta tarefa, ou seja, o EZProofC resolve o problema em menos de 1s (sem contar com o tempo de verificação do *model checker*), para manipular uma grande quantidade de dados, variáveis e seus valores. É importante enfatizar a necessidade de verificação de cada propriedade (*claim*) identificada no código analisado. Isso ocorre porque essas propriedades não correspondem necessariamente a erros, mas estas são apenas potenciais falhas. Esta é a razão pela qual o número de propriedades identificadas na Tabela 5.1 é maior ou igual ao número de erros reproduzidos.

5.7 Resumo

Este Capítulo descreveu as etapas do método EZProofC que explora e coletas os dados (variáveis e seus valores) em contra-exemplos gerados pelo *model checker* ESBMC, de tal forma que ele possa gerar um novo código instanciado com dados do contra-exemplo, a fim de reproduzir o erro identificado. Na Seção 5.6, descreveu-se o planejamento, projeto, execução e análise dos resultados de um estudo experimental realizado com o objetivo de avaliar o método EZProofC quando aplicado à verificação de *benchmarks* padrão de programas ANSI-C. Os resultados do experimento demonstram que o EZProofC foi capaz de reproduzir 100% dos erros identificados pelo ESBMC, além de ser escalável para grandes tamanhos de contra-exemplo (por exemplo, contra-exemplo com 3.344 linhas de código). Adicionalmente, apresentou-se uma comparação com a ferramenta Frama-C (Canet *et al.*, 2009), usando os *plug-in* de análise de valores e `Jessie`. Os resultados desta comparação demonstraram que a grande vantagem do EZProofC está em não necessitar da escrita de especificações auxiliares para a identificação de erros, uma vez que estas foram necessárias pelo Frama-C.

Capítulo 6

Verificação de Programas usando *Bounded Model Checkers* com Invariantes de Programas

Este Capítulo descreve as principais etapas do método proposto para inferir invariantes de programas na verificação efetuada por um *Bounded Model Checker*, e um estudo empírico realizado com o objetivo de analisar o método proposto, quanto ao impacto do uso de invariantes na verificação de programas. Adicionalmente, é apresentado um comparação com as ferramentas CPAChecker (Beyer e Keremoglu, 2011), CBMC (Clarke *et al.*, 2004a), e ESBMC (Cordeiro *et al.*, 2012a) sem o uso de invariantes.

6.1 Verificador com Invariantes de Programas

Este método proposto para inferir invariantes de programas na verificação de um BMC visa reduzir o número de estados no modelo a ser verificado. A redução do modelo neste método consiste em adicionar restrições baseadas em invariantes de programas aos estados que são explorados pelos BMCs. As restrições são pré- e pós-condições para as funções, bem como a pré-verificação de *loops* pela utilização de invariantes globais e de *loops*. O método proposto consiste das seguintes etapas:

(i) Geração de Invariantes;

(ii) Tradução e Instrumentação de Invariantes;

(iii) Mesclagem das Invariantes;

(iv) Verificação Guiada por Invariantes de Programas usando o ESBMC.

A Figura 6.1 apresenta uma visão geral do método proposto. Vale ressaltar que as caixas com linhas sólidas representam componentes que são reutilizando sem efetuar qualquer modificação. De modo a explicar as principais etapas do método proposto utilizou-se o código demonstrado na Figura 6.2. Este programa pertence ao *benchmark* da ferramenta InvGen (Gupta e Rybalchenko, 2015).

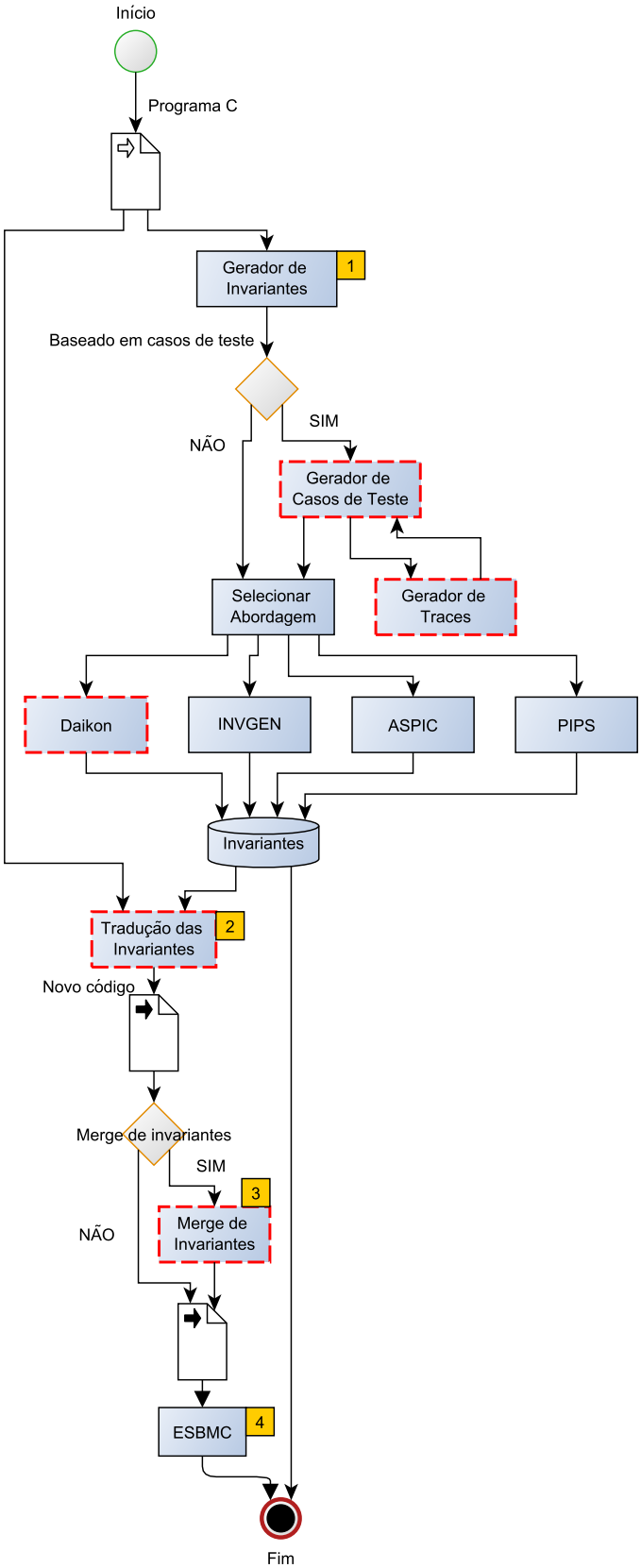


Figura 6.1: Estrutura do fluxo do método proposto para inferência de invariantes.

```
1  #include <assert.h>
2
3  int main() {
4    int n0, n1,n2;
5    int i = 0; int k = 0;
6
7    while( i < n0 ) {
8      i++; k++;
9    }
10   i = 0;
11   while( i < n1 ) {
12     i++; k++;
13   }
14
15   i = 0;
16   while( i < n2 ) {
17     i++; k++;
18   }
19
20   i = 0;
21   while( i < n2 ) {
22     i++; k--;
23   }
24
25   i = 0;
26   while( i < n1 ) {
27     i++; k--;
28   }
29   i = 0;
30   while( i < n0 ) {
31     assert(k > 0);
32     i++; k--;
33   }
34   return 0;
35 }
```

Figura 6.2: Programa seq-len.c do benchmark InvGen.

6.2 Etapa I: Geração de Invariantes

O código analisado é dado como entrada ao gerador de invariantes que é responsável por inferir/derivar invariantes a partir do programa. Tais invariantes são relacionadas à pré- e pós-condições do programa, especificamente invariantes globais, de funções e *loops*. O gerador de invariantes é um conjunto de abordagens composto por Daikon (Perkins e Ernst, 2004; Ernst *et al.*, 2007), ASPIC (Feautrier e Gonnord, 2010), PIPS (Ancourt *et al.*, 2010), e InvGen (Colón *et al.*, 2003). Nesta etapa o usuário do método pode optar em utilizar todas as abordagens combinadas ou somente uma delas, visto que determinados programas podem tender mais a uma dada abordagem. Estas abordagens foram selecionadas com base no resultado do levantamento bibliográfico (ver Seção 3.5.1) efetuado neste trabalho. Vale ressaltar que cada invariante gerado pela respectiva ferramenta é armazenado em uma base de dados para utilização nas próximas etapas, com as seguintes informações: o número da linha respectiva a invariante, comentários da ferramenta sobre a invariante, e a invariante. O relatório gerado pela ferramenta também é armazenado. A geração das invariantes do programa é executada por cada uma das abordagens como apresentado a seguir.

6.2.1 Gerando Invariantes com Daikon

Daikon é um detector de invariantes para programas escritos em C, C++, Java e Perl. Daikon executa a detecção de invariantes durante a execução do programa, observando os valores que são computados pelo programa, e então relata as propriedades que são verdadeiras sobre as execuções observadas. Exemplos de invariantes detectadas são, constantes ($x = a$), diferente de zero ($x \neq 0$), intervalos ($a \leq x \leq b$), relações lineares ($y = ax + b$), ordenação ($x \leq y$), e outras.

Visando permitir a execução e inferência das invariantes pelo Daikon para programas em C, o método proposto adota a ferramenta Kvasir [1] como *front-end* para C. O Kvasir executa programas em C e C++ e cria arquivos (.dtrace) com os dados de rastreamento das variáveis e seus valores através da análise do execução do programa binário, em tempo de execução, os quais serão passados como entrada para o Daikon. Para utilizar o Kvasir, deve-se compilar o programa analisado utilizando o formatado de depuração `DWARF-2` [2] (por exemplo, no compilador `gcc` usando a opção `-gdwarf-2`) e sem otimizações (`gcc` usando a opção `-O0`). Desta forma, o método proposto efetua a compilação do programa analisado e então executa o Kvasir da seguinte forma:

Compilação, que é efetuada com o seguinte comando:

```
gcc -gdwarf-2 -O0 <file.c> -o <file.exe>
```

Sendo `<file.c>` o arquivo do programa em C a ser analisado e `<file.exe>` o nome do binário a ser gerado pela compilação.

Geração do `ppt-file`, que é uma lista de todos os pontos (funções) do programa (respectivamente todas as variáveis) no programa para um arquivo especificado, utilizando o Kvasir com o seguinte comando:

```
kvasir-dtrace -dump-ppt-file=FunctionNamesTest.ppts ./file.exe
```

O `FunctionNamesTest.ppts` é o nome do arquivo que conterá a lista de todos os pontos do programa e `file.exe` é nome no binário do programa que está sendo analisado.

Geração de rastreamento com o Kvasir, que consiste na monitoração da execução do programa analisado para criar um rastreamento das variáveis e seus valores durante a execução, o Kvasir é utilizado com o seguinte comando:

```
kvasir-dtrace --ppt-list-file=FunctionNamesTest.ppts --with-dyncomp
--decls-file=daikon-output/test.decls
--dtrace-file=daikon-output/test-all.dtrace ./file.exe
```

A opção `--ppt-list-file` define que o rastreamento seja feito em pontos específicos do programa analisado a partir de um dado arquivo. A opção `--with-dyncomp` permite determinar quais variáveis têm o mesmo tipo abstrato. Esta informação pode melhorar o desempenho de Daikon e permitir que ele produza um conjunto mais relevante de

[1] http://plse.cs.washington.edu/daikon/download/doc/daikon.html#Kvasir
[2] http://www.dwarfstd.org/

invariantes. A opção `--decls-file` define o nome do arquivo que terá a listagem do nome das funções e variáveis (chamado de declarações) e `--dtrace-file` define o nome do arquivo que conterá o rastreamento da execução do programa.

Geração das invariantes com Daikon, que consiste em analisar os arquivos de rastreamento (`daikon-output/test.decls` e `daikon-output/test-all.dtrace`) para inferir as invariantes do programa. Os invariantes são impressas para uma saída padrão, e uma representação binária das invariantes são escritas em `test-all.inv.gz`. As invariantes geradas no arquivo são então armazenadas para as próximas etapas. Esta execução é feita com o seguinte comando:

```
java daikon.Daikon daikon-output/test.decls
daikon-output/test-all.dtrace
```

6.2.2 Gerando Invariantes com ASPIC

ASPIC é uma ferramenta de geração automática de invariantes baseada no domínio abstrato poliédrico (ver Seção 2.5) para programas. ASPIC implementa uma análise da relação linear em um autômato de contador numérico, sendo o seu principal diferencial a utilização da técnica "*abstract acceleration*" (Gonnord e Halbwachs, 2006) que contribui para a precisão na geração de invariantes.

A geração das invariantes com ASPIC consiste em receber como entrada uma representação textual de um autômato numérico interpretado e como resultado ASPIC gera um mapeamento de todos os pontos de controle para invariantes numéricas afins. Em Feautrier e Gonnord (2010) é demonstrada a utilização de ASPIC com c2fsm que é um pré-processador que automatiza a construção do autômato de entrada para o ASPIC a partir de um programa em C. Desta forma, no método proposto o ASPIC é executando da seguinte sequência:

Preprocessamento do programa analisado. Este passo é efetuado, pois o c2fsm que é utilizado pelo método proposto não executa um pré-processamento do programa antes de efetuar o *parse*. Logo, no método é utilizado o gcc como um preprocessador do programa.

```
gcc -E <file.c> > <file_e.c>
```

Geração do autômato como entrada para o ASPIC usando c2fsm com o seguinte comando:

```
c2fsm <file.c> -fst
```

A opção `-fst` indica ao c2fsm que a saída gerada será um autômato no formato ASPIC. O nome do arquivo de saída (.fst) é derivado do nome da primeira declaração da função do programa analisado (`file.c`). Vale ressaltar que o c2fsm possui algumas limitações com relação a estruturas e operações de programas em C, como não suportar: `switch`, `enums`, `unions`, expressões condicionais ($x?y:z$), e operadores como: $/=$, $\%=$, $>>=$, $<<=$.

Geração das invariantes usando o ASPIC, que consiste em analisar o arquivo (.fst) com o autômato gerado a partir do programa analisado e então inferir a invariantes. Esta operação é feita com o seguinte comando:

```
aspic <file.c> -cinv
```

A opção `-cinv` define que as invariantes geradas estejam no formato de expressões da linguagem de programação C.

6.2.3 Gerando Invariantes com PIPS

PIPS utiliza um mecanismo de abstração que adota restrições, como poliedros, para especificar pré- e pós-condições, bem como transformadores de estado. Este mecanismo visa uma análise modular e para limitar os tempos de análise, onde a estratégia utilizada permite estimar/aproximar o comportamento de *loops*. O diferencial da abordagem PIPS é que são computados transformadores de estados, ao invés de predicados de estados (Ancourt *et al.*, 2010).

PIPS executa sua análise em duas etapas: (1) cada instrução do programa está associada a um transformador, o que representa sua função de transferência subjacente. Este é um procedimento executado de baixo para cima, a partir de instruções elementares, então trabalhando em instruções compostas e até definições de funções; (2) invariantes poliédricas são propagadas junto com instruções, utilizando transformadores previamente calculados.

No método proposto, PIPS recebe o programa analisado como entrada e, em seguida, ele gera invariantes que são dadas como comentários em torno de instruções no código C do programa analisado. Visando a execução do PIPS, o método proposto utiliza um *script* na linguagem tpips [3] que é provida pelo projeto PIPS. Neste *script* são definidos os comandos para análise e transformações do programa analisado com o foco na geração das invariantes (ver Figura 6.3). A execução do *script* é feita da seguinte forma:

```
tpips script_inv.pips
```

6.2.4 Gerando Invariantes com InvGen

O funcionamento de InvGen consiste em tomar um programa como entrada sobre expressões aritméticas lineares e, a partir de um *template* de invariantes, são computados as invariantes do programa (Colón *et al.*, 2003). InvGen utiliza uma abordagem baseada em restrições para gerar invariantes combinando técnicas de análise estática e dinâmica (execução simbólica ou concreta) para resolver eficientemente as restrições.

O programa analisado é passado para os analisadores dinâmicos e estáticos. Os resultados de cada análise, em conjunto com o programa e com os *templates* são passados para o gerado de restrições.

[3] http://pips4u.org/doc/line-interface.html

```
# ---- configurações do projeto para o programa
delete precod
create precod \
    confuse.c

# ---- configurações de formatação

setproperty PRETTYPRINT_ALL_DECLARATIONS TRUE
setproperty PRETTYPRINT_C_CODE TRUE

# ---- gera arquivos ausentes e rotinas

setproperty PREPROCESSOR_MISSING_FILE_HANDLING "generate"

# ---- transformações de loops
capply LOOP_TILING[%ALL]
capply FULL_UNROLL[%ALL]
capply PARTIAL_EVAL[%ALL]
capply SCALARIZATION[%ALL]

# ---- eliminação de código morto

apply SUPPRESS_DEAD_CODE[%ALL]

# ---- elimina declaração inútil após clonagem e código morto

apply CLEAN_DECLARATIONS[%ALL]

# ---- obtenção de informações interprocedimentais

activate TRANSFORMERS_INTER_FULL
activate PRECONDITIONS_INTER_FULL
capply   PARTIAL_EVAL[%ALL]
activate PRINT_CODE_PRECONDITIONS

# ---- impressão do código com invariantes
display PRINTED_FILE[%ALL]


close
quit
```

Figura 6.3: Script tpips para a geração de invariantes.

As restrições são resolvidas por um solucionador de restrição (*solver*). Se o *solver* encontrar uma solução então InvGen retorna as invariantes. InvGen provê um *front-end* [4] para programas em C que traduz o programa em um programa na linguagem de entrada para o InvGen. O *front-end* também contém um interpretador abstrato (InterProc (Colón *et al.*, 2003)) que utiliza os domínios abstratos de intervalo, octagonal e poliédrico.

O *front-end* do InvGen exibe como resultado do programa analisado (`input.c`) a relação de transição do programa (`output.pl`) que é anotada com os resultados computados pela InterProc. A execução do *front-end* é dada pelo seguinte comando:

```
frontend -main <functname> -domain <1..4> -o <output.pl> <input.c>
```

As opções definidas para o *front-end* são: `-main` define a função principal e `-domain` que possui 4 opções de domínios abstratos. O InvGen então recebe como entrada o arquivo `output.pl` para inferir as invariantes. Em seguida, o InvGen minimiza a relação de transições do programa e coleta

[4] http://www.tcs.tifr.res.in/ agupta/invgen/frontend.html

informações da execução do programa (via execução simbólica e concreta) para reduzir a complexidade na solução das restrições. No método proposto, InvGen é executado da seguinte forma:

```
invgen -timeout 900 <output.pl> -dump_inv
```

Um tempo limite de 900 segundos é definido para a execução do InvGen, bem como, é utilizado a opção -dump_inv para escrever as invariantes geradas em um arquivo. Vale ressaltar que InvGen possui atualmente algumas limitações para processar as funções no código do programa analisado, tais como: não conter qualquer chamadas de funções, *array* e ponteiros. Como recomendado na documentação de InvGen, o método proposto adota uma variável (NONDET) para modelar escolhas não determinísticas.

As invariantes geradas são armazenadas em uma base de dados que pode ser analisada pelo usuário, caso o usuário julgue que as invariantes inferidas não possuem uma qualidade significativa. O método proposto fornece duas abordagens que visam melhorar a geração das invariantes pela utilização de casos de teste gerados automaticamente. As abordagens são: (1) Inception que gera casos de teste baseado em contra-exemplos gerados por BMCs (ver Seção 6.2.5); e (2) PathCrawler que utiliza execução simbólica para a geração de valores como casos de teste (ver Seção 6.2.6). O objetivo destes métodos é gerar valores de entradas para as variáveis do programa a fim de obter a maior cobertura de execução do programa analisado. Desta forma, os valores de entradas gerados pelos métodos são adicionados em uma nova instância do programa analisado que auxilia na geração de invariantes. De modo a explicar as principais etapas do método Inception e PathCrawler, utilizou-se o código demonstrado na Figura 6.2. Vale ressaltar que a geração de casos de teste foi essencial para a geração de invariantes deste programa nos experimentos realizados neste trabalho (ver Seção 6.6).

6.2.5 Geração de dados de teste usando Inception

O método Inception foi desenvolvido neste trabalho visando a automação para a geração de dados como casos de testes. A aplicação do método consiste em: (1) instrumentar o código analisado com `assert(0)` em pontos de controle do programa; (2) efetuar a verificação com o BMC para cada `assert(0)`; (3) coletar os valores gerados em cada contra-exemplo; (4) criar uma nova instância do código analisado, para cada contra-exemplo gerado; e (5) instrumentar nesta nova instância do programa os valores das variáveis de acordo com valores apresentados no contra-exemplo.

(1) Instrumentação de *asserts*

A instrumentação de `assert(0)` visa explorar os caminhos de execução do programa, com base na verificação executada pelo ESBMC, no código do programa analisando. Isto pelo fato de o BMC interpretar o `assert(0)` como uma localização de falha no programa. Desta forma, o método Inception instrumenta no código um `assert(0)` para cada ponto de decisão no programa, a fim de utilizar o BMC para gerar contra-exemplos para cada caminho de execução do programa. Vale

ressaltar que para variáveis não inicializadas no programa o método Inception atribui uma chamada a uma função que será utilizada pelo BMC para gerar valores não determinísticos. Por exemplo, para inteiros é usando a função `__VERIFIER_nondet_int()` no ESBMC.

O Algoritmo 4 demonstra como são inseridos os `assert(0)` no programa analisado. O Algoritmo 4 recebe como entrada o programa analisado no formato de um Grafo de Fluxo de Controle $G = (N, E)$, onde os nós em N representam declarações e as arestas em E representam a transferência do controle entre declarações. Assim, para cada nó $n \in N$ que possua duas aresta (tomada de decisão) $e \in E$ é inserido em cada nó a partir de n um `assert(0)` no fim do bloco básico, ou seja, no fim das declarações do nó. Sendo que no nó final $exit \in N$ o Algoritmo 4 também insere um `assert(0)`, visando assim a análise dos valores das declarações deste último ponto de execução do programa.

```
   Input: CFG do programa (G)
   Output: Programa instrumentado com assert(0)
 1 begin
 2     foreach node n IN G do
 3         if |e| a partir de n é igual a 2 then
 4             Inseri assert(0) no fim do nó a direita a partir de n ;
 5             Inseri assert(0) no fim do nó a esquerda a partir de n ;
 6         end
 7         else if n é igual exit, onde exit ∈ N em G then
 8             Inseri assert(0) no fim do nó ;
 9         end
10     end
11 end
```

Algoritmo 4: Algoritmo do Inception para inserção de *assert(0)*

A Figura 6.4 apresenta o resultado da aplicação do Algoritmo 4. Neste novo programa tem-se uma localização de falha (`assert(0)`) para cada resultado de uma tomada de decisão no fluxo de execução do programa. Desta forma, o método Inception visa obter a maior cobertura de execução do programa baseado em múltiplas execuções do programa.

(2) Verificação guiada por `assert(0)`

Nesta etapa do método Inception é efetuada a verificação de cada `assert(0)` pelo ESBMC. Esta verificação ocorre em duas fases: a identificação dos `assert(0)` no novo programa e a verificação de cada `assert(0)`. Inception efetua a identificação dos *assert(0)* executando ESBMC que recebe como entrada o programa C que será analisado e usa a opção `-show-claims`, que mostra as propriedades de segurança que o ESBMC identifica como aquelas que podem vir a ser violadas. Neste momento, Inception seleciona das *claims* listadas pelo ESBMC somente aquelas identificadas como *assertion* no resultado do ESBMC. A Figura 6.5 apresenta uma parte do resultado da identificação das *claims* pelo ESBMC sobre o novo programa analisado.

```
1  #include <assert.h>
2
3  int main() {
4    int n0 = __VERIFIER_nondet_int();
5    int n1 = __VERIFIER_nondet_int();
6    int n2 = __VERIFIER_nondet_int();
7    int i = 0; int k = 0;
8
9    while( i < n0 ) {
10     i++; k++;
11     assert(0); //INCEPTION
12   }
13   assert(0); //INCEPTION
14   i = 0;
15   while( i < n1 ) {
16     i++; k++;
17     assert(0); //INCEPTION
18   }
19   assert(0); //INCEPTION
20   i = 0;
21   while( i < n2 ) {
22     i++; k++;
23     assert(0); //INCEPTION
24   }
25   assert(0); //INCEPTION
26   i = 0;
27   while( i < n2 ) {
28     i++; k--;
29     assert(0); //INCEPTION
30   }
31   assert(0); //INCEPTION
32   i = 0;
33   while( i < n1 ) {
34     i++; k--;
35     assert(0); //INCEPTION
36   }
37   assert(0); //INCEPTION
38   i = 0;
39   while( i < n0 ) {
40     //assert(k > 0);
41     i++; k--;
42     assert(0); //INCEPTION
43   }
44   assert(0); //INCEPTION
45   return 0;
46 }
```

Figura 6.4: Código C com asserts para geração de testes.

A verificação de cada `assert(0)` é efetuada com base no número da *claim* identificada na fase 1. Para a verificação de cada *claim* é definido um tempo máximo de 180 segundos. Os contra-exemplos da verificação de cada *claim* são armazenados para utilização das próximas etapas do método Inception. A Figura 6.6 apresenta como o programa é explorado na verificação de cada *claim* gerada pela inserção do `assert(0)`, assim explorando diferentes tomadas de decisão como é apresentado execução da *claim* 5 e 12 na figura. O objetivo da verificação destes caminhos de execução (guiado pelas *claims*) do programa é gerar valores para as variáveis envolvidas nestes caminhos, ou seja, gerar um conjunto de valores (dados de entrada para teste) que possibilite a execução destes caminhos de execução.

```
herbert@nbh /media/$ esbmc --show-claims seq_len_asserts.c
file seq_len_asserts.c: Parsing
Converting
Type-checking seq_len_asserts
Generating GOTO Program
GOTO program creation time: 0.097s
GOTO program processing time: 0.001s
Claim 1:
  file seq_len_asserts.c line 11 function main
  assertion
  FALSE
```

```
1. #include <assert.h>
2.
3. int main() {
4.   int n0 = __VERIFIER_nondet_int();
5.   int n1 = __VERIFIER_nondet_int();
6.   int n2 = __VERIFIER_nondet_int();
7.   int i = 0; int k = 0;
8.
9.   while( i < n0 ) {
10.    i++; k++;
11.    assert(0); //INCEPTION
12.  }
...
```

Figura 6.5: *Claim* com `assert(0)`.

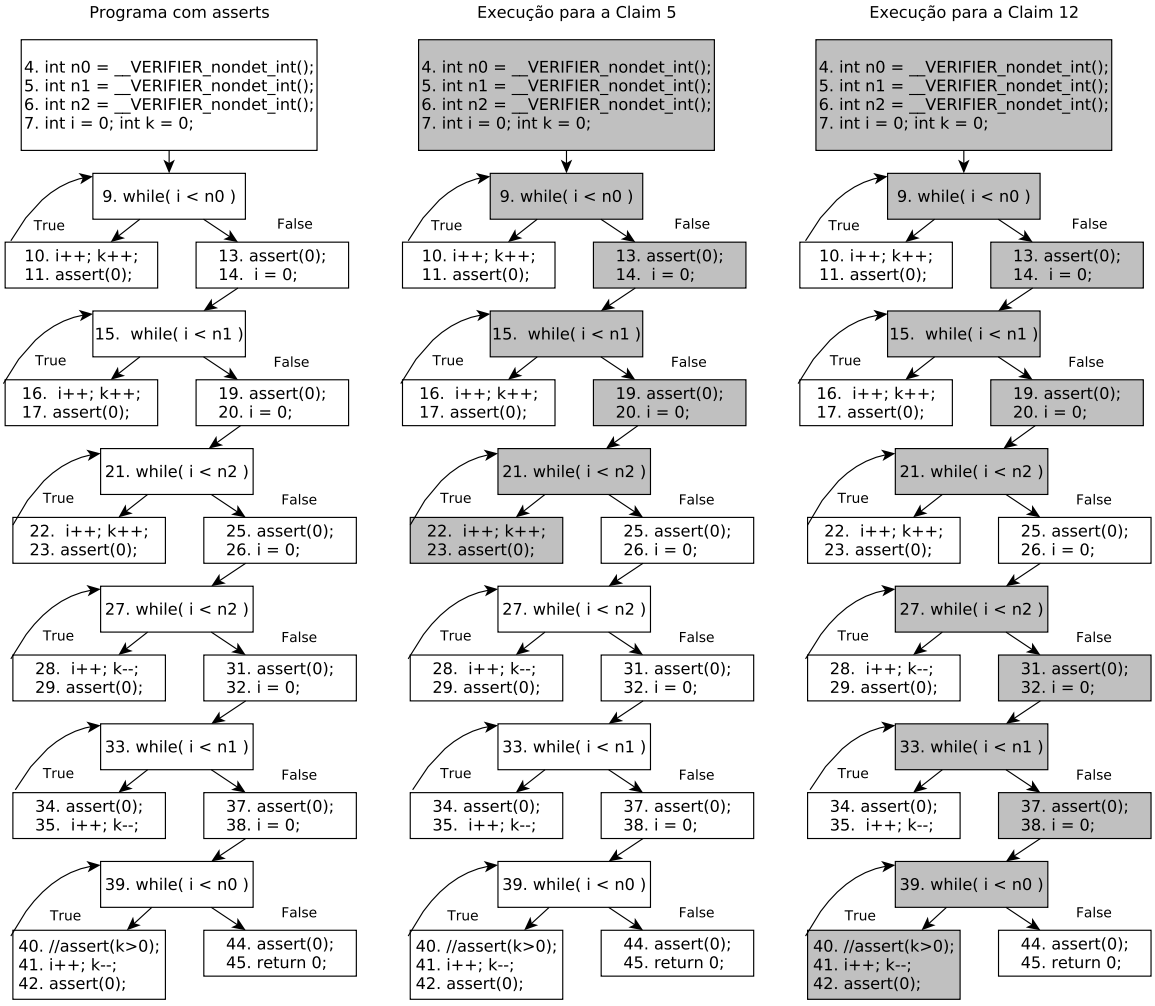


Figura 6.6: Programa no formato CFG na verificação dos `assert(0)`.

Geração de casos de teste baseado em contra-exemplos

Os contra-exemplos gerados a partir de cada *claim* verificada pelo ESBMC são analisados para gerar um novo código C instanciado. Neste novo código os valores das variáveis terão os valores identificados no contra-exemplo. Logo, os casos de testes gerados serão instâncias do programa analisado com valores dos contra-exemplos gerados por cada *claim*. O número de casos de testes

gerados é igual ao número de contra-exemplos gerados pelas *claims* verificadas pelo ESBMC. Desta forma, a cobertura total dos casos de teste referentes à execução do programa é medida com a soma da execução das instancias do programa gerado. Por exemplo, se de 12 *claims* for possível verificar/gerar contra-exemplos para apenas 6, logo tem-se 50% de cobertura .

A geração de cada novo código C instanciado a partir dos contra-exemplos de cada *claim* é executado utilizando o Algoritmo 3 (ver Seção 5.4) do método EZProofC. A diferença neste caso é que o código C passado como entrada é o código instrumentado com os `assert(0)` (ver Figura 6.4), onde durante a leitura do código no algoritmo os `assert(0)` são comentados. Adicionalmente os `asserts` no código original, anteriormente comentados, são descomentados. Assim, permitindo a execução original do programa com os novos valores de entrada (gerados no contra-exemplo) no resultado final do algoritmo.

O Algoritmo 3 consiste em: extrair os dados do contra-exemplo utilizando expressões regulares e então lê o código C analisado para gerar um novo código instanciado. Assim, se o número da linha da variável identificado no contra-exemplo é igual ao número de linha do código analisado, o algoritmo pode gerar uma nova linha de código, onde a variável identificada recebe o valor obtido do contra-exemplo. Por exemplo, os seguintes valores coletados do contra-exemplo na linha 4 para a variável $n0$ é -2147483647, logo o algoritmo irá gerar a seguinte atribuição $n0 = -2147483647$. É importante ressaltar que a instanciação das variáveis no novo código é executado estritamente de acordo com a sequência identificada no contra-exemplo. Por exemplo, se a mesma variável no contra-exemplo é mencionada várias vezes na mesma linha (por exemplo, em *loops*), apenas o último valor encontrado na contra-exemplo será atribuído a variável no código instanciado. A Figura 6.7 apresenta o resultado do método Inception para um caso de teste referente a *claim* 7.

Finalizado a geração dos códigos pelo método Inception, cada nova instância de código referente as *claims*/`asserts` é repassado novamente a Etapa I para a geração das invariantes (ver Seção 6.2). Assim, para cada nova instância de código será gerada suas respectivas invariantes que o método proposto irá inserir estas invariantes em uma cópia do código original (sem os valores de entradas gerados). Desta forma, serão geradas n cópias do programa com as invariantes relacionadas a cada *claim*/`assert`. Cada um destes códigos será analisado pelo método proposto, ou seja, ocorrerá a tradução das invariantes no código, bem como, a verificação de cada código. Vale ressaltar que quando utilizado o método Inception, a verificação será medida de acordo com os resultados de cada *claim*, ou seja, se de 12 *claims* teve 9 com resultado de `SUCCESSFUL` (sem propriedades violadas) o programa analisado está correto em 75% de sua cobertura. Visando exemplificar o resultado da aplicação do método proposto a Tabela 6.1 apresenta uma das invariantes geradas anteriormente (a partir do programa na Figura 6.2) pela ferramenta InvGen e em comparação a nova invariante gerada a partir de cada um dos novos programas referentes a cada *claim* e seus respectivos valores de entrada com a ferramenta InvGen.

```
1  #include <assert.h>
2
3  int main() {
4    int n0 = -1073741824;
5    int n1 = 1;
6    int n2 = 1;
7    int i = 0; int k = 0;
8
9    while( i < n0 ) {
10     i++; k++;
11     //assert(0); //INCEPTION
12   }
13   //assert(0); //INCEPTION
14   i = 0;
15   while( i < n1 ) {
16     i++; k++;
17     //assert(0); //INCEPTION
18   }
19   //assert(0); //INCEPTION
20   i = 0;
21   while( i < n2 ) {
22     i++; k++;
23     //assert(0); //INCEPTION
24   }
25   //assert(0); //INCEPTION
26   i = 0;
27   while( i < n2 ) {
28     i++; k--;
29     //assert(0); //INCEPTION
30   }
31   //assert(0); //INCEPTION
32   i = 0;
33   while( i < n1 ) {
34     i++; k--;
35     //assert(0); //INCEPTION
36   }
37   //assert(0); //INCEPTION
38   i = 0;
39   while( i < n0 ) {
40     assert(k > 0);
41     i++; k--;
42    // assert(0); //INCEPTION
43   }
44   //assert(0); //INCEPTION
45   return 0;
46 }
```

Figura 6.7: Caso de teste referente a *claim* 7.

6.2.6 Geração de dados de teste usando PathCrawler

A ferramenta PathCrawler gera entradas de teste que satisfaçam os critério de cobertura de todos os caminhos de uma dada função (escrita em ANSI-C) sob teste, com um limite definido pelo usuário sobre o número de iterações de *loop* nos caminhos cobertos. PathCrawler baseia-se numa combinação de instrumentação de código e solução de restrições (Williams *et al.*, 2005a).

No método proposto cada função do programa dado como entrada é analisado pelo PathCrawler. Adicionalmente, para cada função identificada no programa analisado, o método proposto cria uma cópia do código analisado, onde cada variável não inicializada na função é transformada em argumentos de entrada da respectiva função. Esta etapa adicional visa contribuir na exploração dos

Invariante do InvGen	**Código com a *Claim***	**Invariante da *Claim* com InvGen**
$n2 + n0 - k - i =< 0, n0 - k =< 0, n1+n0-k =< 0, n2+n1+n0-k-i =< 0$	1	$n2 = 0, n1 = 0, n0 = 1, k = 1, i = 0$
	2	$n2 = 0, n1 = 0, n0 = 1, k = 1, i = 0$
	3	$1 =< 0$
	4	$n2 = 0, n1 = 1, n0 = 1, k = 2, i = 0$
	5	$1 =< 0$
	6	$n2 = 1, n1 = 1, n0 = 1, k = 3 - i, i >= 0, i =< 1$
	7	$n2 = 1, n1 = 1, n0 = -1073741824, i >= 0, k = 2 - i, i =< 1$
	8	$n2 = 1, n1 = 1, n0 = 1, k = 3 - i, i >= 0, i =< 1$
	9	$n2 = 1, n1 = 1, n0 = 1, i >= 0, k = 3 - i, i =< 1$
	10	$n2 = 1, n1 = 1, n0 = 1, k = 3 - i, i >= 0, i =< 1$
	11	$n2 = 1, n1 = 1, n0 = 1, i >= 0, k = 3 - i, i =< 1$
	12	$n2 = 1, n1 = 1, n0 = 1, k = 3 - i, i >= 0, i =< 1$

Tabela 6.1: Invariantes geradas a partir de Inception

possíveis caminhos de execução do programa analisado. Assim, para cada função analisada pelo PathCrawler são gerados os valores de entrada para os argumentos de cada função.

A ferramenta PathCrawler é disponível via web [5] e também como um *plug-in* da ferramenta de análise de programas Frama-C [6]. O método proposto adota uso do PathCrawler com a ferramenta Frama-C. Desta forma, cada função do programa é analisado pelo PathCrawler com a execução do seguinte comando:

```
frama-c -pc -main <nome da função> <file>.c
```

A opções repassadas ao Frama-C são `-pc` que habilita a análise do programa com o PathCrawler e `-main` que define a função atual a ser analisada. Os valores de entrada geradas pelo PathCrawler são armazenados no diretório `testcases`. Vale ressaltar que por padrão o PathCrawler irá explorar todos os caminhos possíveis de execução da função. Logo, isto poderá gerar uma quantidade grande de possibilidades que resultará em um longo período de execução, bem como, a possibilidade do solucionador de restrições não identificar uma solução. Desta forma, o usuário do método proposto pode efetuar algumas customizações (adição ou alteração de parâmetros) referentes à execução do PathCrawler, a fim de otimizar sua execução. Por exemplo, restringidos os valores de domínios das variáveis. No exemplo apresentado, o valor das variáveis (de entrada da função) foi delimitado no seguinte intervalo $0 \leq var \leq 500$, onde *var* é cada variável de entrada.

Finalizado a execução da geração dos valores de teste pelo PathCrawler, o método proposto cria uma nova instância do programa original analisado. Nesta nova instância, cada variável (os

[5] http://pathcrawler-online.com/
[6] http://frama-c.com/pathcrawler.html

argumentos da função) anteriormente analisada pelo PathCrawler é inicializada no topo da função com o seu respectivo valor gerado pelo PathCrawler. Desta forma, o novo programa gerado, quando executado, irá atingir a cobertura máxima de código (possibilidades de execução) identificada pelo PathCrawler. Assim, possibilitando as ferramentas de inferência de invariantes gerarem novos dados (caminhos de execução) no programa. A Figura 6.8 apresenta um trecho código do programa analisado resultante da execução do método e a Tabela 6.2 o resultado de algumas novas invariantes geradas a partir deste novo programa para as ferramentas InvGen e ASPIC.

```
1  int main() {
2
3    int n0 = 455; // a patir do PathCrawler
4    int n1 = 387; // a patir do PathCrawler
5    int n2 = 59; // a patir do PathCrawler
6
7    int i = 0;
8    int k = 0;
9
10   while( i < n0 ) {
11     i++;
12     k++;
13   }
14
15   ...
16 }
```

Figura 6.8: Trecho do novo código gerado a partir do PathCrawler.

Ferramentas	Inv. Antigas	Inv. Novas
ASPIC	$i + k + 2 > n0$ && $i >= 0$ && $i + 2 > n0$ && $k + 2 > 0$	$i + k > 59$ && $i > 59$ && $n0 == 455$ && $n1 == 387$ && $n2 == 59$
InvGen	$n2 + n0 - k - i =< 0, n0 - k =< 0, n1 + n0 - k =< 0, n2 + n1 + n0 - k - i =< 0$	$n2 = 59, n1 = 387, n0 = 455, k = 901 - i, i >= 0, i =< 59$

Tabela 6.2: Invariantes a partir de PathCrawler

6.3 Etapa II: Tradução e Instrumentação de Invariantes

Nesta etapa são efetuadas as traduções das invariantes geradas e armazenadas na primeira etapa (ver Seção 6.2). A tradução das invariantes consiste basicamente em: (i) identificar as estruturas de cada invariante gerada pela sua respectiva ferramenta, onde esta identificação é feita utilizando expressões regulares responsáveis por interpretar as estruturas providas por cada ferramenta; e (ii) tradução das estruturas e elementos das invariantes geradas por estruturas suportadas pela linguagem de programação C e pelo ESBMC. Vale ressaltar que esta etapa de tradução é importante visto que cada ferramenta usada para a geração de invariantes utilizada pelo método proposto possui um padrão, linguagem e simbologia própria para a geração das invariantes. Desta forma, nesta seção será abordado como é efetuado a tradução das invariantes geradas por cada ferramenta adotada neste método proposto. Vale ressaltar que o método proposto é flexível a adição de novas ferramentas/abordagens para a geração de invariantes, deste que nesta etapa do método seja definida a sua forma de utilização em programa em C.

6.3.1 Tradução das invariantes do Daikon

As invariantes geradas por Daikon (ver Seção 6.2) tem como base os dados de rastreamento gerados para o programa que está sendo analisado, conforme apresentado na Seção 6.2.1. Finalizado a identificação das invariantes nos dados de rastreamento, o Daikon pode exibir as invariantes em diversos formatos [7] (usando a opção `--format`) como apresentado abaixo. Contudo, estes formatos não são diretamente suportados pela linguagem C, ou seja, não é possível utilizar as invariantes geradas em um programa C.

Daikon. Formato padrão do Daikon consiste no uso de estruturas em Java, lógica matemática, e algumas extensões adicionais do Daikon.

DBC. Formato de projetos por contrato usado pela ferramenta Jtest da Parasoft [8].

ESC/Java. Suportado pela ferramenta ESC/Java (*Extended Static Checker for Java*) que é uma ferramenta para encontrar erros em programas Java usando anotações inseridas no código fonte.

Java. Escreve a saída como expressões em Java. Contundo, neste formato existem algumas exceções como as invariantes geradas na saída dos métodos que podem utilizar a *tag* `old` e as invariantes relacionadas ao retorno de valores de métodos que podem usar a *tag result*. Assim estas expressões contendo estas *tags* não serão códigos válidos em Java.

JML. Formato *Java Modeling Language* [9].

Simplify. Formato esperado pelo provador de teoremas automatizado Simplify.

CSharpContract. Formato para código `C#` baseado em contratos [10].

Neste trabalho foi criada uma versão alterada da ferramenta Daikon [11] com suporte a impressão das invariantes no formato ACSL (*ANSI/ISO C Specification Language*) (Baudin *et al.*, 2009). ACSL é uma linguagem formal para expressar propriedades comportamentais de programas em C. Esta linguagem pode especificar propriedades funcionais através de anotações no código.

A criação desta nova versão do Daikon tem como base o formato do ESC/Java já suportado pelo Daikon. O formato do ESC/Java tem como base o JML, sendo que o formato do ESC/Java gerado pelo Daikon está mais próximo das expressões da linguagem Java. Podería-se ter escolhido o formato JML, contudo em alguns casos para este formato são utilizados métodos em Java que são interpretados pelo Daikon (`daikon.tools.runtimechecker`). Isto porque, para programas

[7] http://plse.cs.washington.edu/daikon/download/doc/daikon.html#Invariant-syntax
[8] http://www.parasoft.com/
[9] http://www.eecs.ucf.edu/ leavens/JML/
[10] http://research.microsoft.com/en-us/projects/contracts/
[11] https://bitbucket.org/herberthb12/daikon-acsl

em Java, Daikon pode anotar o programa analisado com as invariantes geradas e então efetuar a verificação [12] destas invariantes. Segue um exemplo de invariantes no formato JML e ESC/Java:

JML: $\backslash old(daikon.Quant.getElement\_int(arr, \backslash new(return))) == 2$

ESC/Java: $\backslash old(arr[\backslash new(return)]) == 2$

A linguagem das anotações em ACSL tem um formato similar ao utilizado pelas anotações originais (fora do Daikon) do JML. Contudo, ACSL tem aspectos que se diferenciam do JML, tais como (Baudin *et al.*, 2009): ACSL não suporta estruturas apoiadas por JML como herança, exceções, e obter o tamanho de um bloco de memória alocada; Em JML se houver uma precondição sobre o comprimento de um *array* '`a`', que poderia facilmente ser expressado utilizando a expressão em Java '`a.length`' que retorna o comprimento do *array* '`a`'. Em ACSL, o programador tem que adicionar um argumento extra para o tamanho do *array*, ou adicionar o limite inferior do tamanho do *array*. Por exemplo, adotando $//@\backslash valid(a + 0..(n - 1));$ e em programas C pode-se definir um rótulo para o salto de uma instrução `goto`. Assim em ASCL a expressão $\backslash at(a[i], A)$ significa que $a[i]$ é avaliado no rótulo A. A Tabela 6.3 apresenta algumas diferenças em relação à sintaxe de JML e ACSL.

JML	**ACSL**
modifiable,assignable	assigns
measured_by	decreases
loop_invariant	loop invariant
decreases	loop variant
$(\backslash forall\tau\ x\ ;\ P;\ Q)$	$(\backslash forall\tau\ x\ ;\ P\ ==>\ Q)$
$(\backslash exist\tau\ x\ ;\ P;\ Q)$	$(\backslash existst\tau\ x\ ;\ P\ \&\&\ Q)$
$\backslash max\tau\ x\ ;\ a\ <=\ x\ <=\ b\ ;\ f$	$\backslash max(a, b, \backslash lambda\tau\ x\ ;\ f)$

Tabela 6.3: Invariantes a partir de PathCrawler

A criação da nova versão do Daikon com suporte ao ACSL consistiu em escrever os códigos relacionados à impressão das invariantes no formato do ACSL tendo como base o código provido no Daikon (nos métodos format_using) para o formato ESC/Java, mas aplicando as devidas diferenças mencionadas anteriormente. Vale ressaltar que umas das funcionalidades que se adicionou no Daikon é a identificação dos tamanhos de *arrays* que é feita por meio da leitura do código e expressões regulares. As invariantes geradas por Daikon para o programa da Figura 6.2 são basicamente: $return == 0$ e $argv[]elements == "./seq - len.exe''$. Assim, visando exemplificar o resultado da geração das invariantes no formato ACSL, utilizou-se o programa na Figura 6.9. Na Tabela 6.4 são apresentadas algumas das invariantes geradas do programa na Figura 6.9 com a nova versão do Daikon nos formatos ACSL e ESC/Java para comparação.

[12] http://plse.cs.washington.edu/daikon/download/doc/daikon.html#Runtime_002dcheck-instrumenter

```
1  #define MAX 10
2  int buffer[MAX];
3  int range_updated=MAX;
4
5  void handle_buffer();
6  int check_range(int range, int vet[]);
7
8  foo(){
9      char var_1 = 'a';
10     double var_2 = 1.2;
11     int var_3 = 12;
12     int * var_ptr_int = &var_3;
13 }
14
15 int check_range(int range, int vet[]){
16   int i;
17   int var_a = 2;
18   for(i=0;i<MAX;i++){
19     vet[i]=1*var_a;
20     var_a = i * var_a + range;
21   }
22
23   if(range < MAX){
24     range_updated=range;
25     handle_buffer();
26     foo();
27     return 1;
28   }else{
29     return 0;
30   }
31 }
32
33 void handle_buffer(){
34   int i;
35   for(i=0; i<=range_updated;i++){
36     buffer[i]=i; // BUG?
37   }
38 }
39
40 int main(){
41   range_updated = MAX;
42   check_range(8,buffer);
43   return 1;
44 }
```

Figura 6.9: Programa exemplo_mot.c.

ESC/Java	**ACSL**
$i == buffer.length$	$i == 10 - 1$
$\backslash old(i) == buffer[\backslash old(i)]$	$\backslash at(i, Pre) == buffer[\backslash at(i, Pre)]$
$buffer.length - 1 == buffer[i - 1]$	$10 - 1 == buffer[i - 1]$
$\backslash old(buffer[\backslash new(i) - 1]) == buffer[\backslash old(i)]$	$\backslash at(buffer[\backslash at(i, Here) - 1], Pre) == buffer[\backslash at(i, Pre)]$
$!(\backslash forall\ int\ i;\ (0 <= i\ \&\&\ i <= buffer.length - 1)\ ==>\ (buffer[i]! = \backslash old(i)))$	$!(\backslash forall\ int\ ACSLii;\ (0 <= ACSLii\ \&\&\ ACSLii <= 10 - 1)\ ==>\ (buffer[ACSLii]! = \backslash at(i, Pre)))$

Tabela 6.4: Invariantes geradas com a nova versão do Daikon

Objetivando a tradução das invariantes geradas por Daikon (já no formato ACSL), o método proposto então insere as invariantes geradas como anotações ACSL em uma nova cópia do código

do programa analisado. A inserção das invariantes consiste em identificar as funções de onde as invariantes foram inferidas no relatório gerado pelo Daikon e quais invariantes são referentes a entrada (pré-condições) da função (representado no Daikon por `:::ENTER`) e a saída (pós-condições em `:::EXIT`). As invariantes referentes a entrada da sua respectiva função são inseridas após o cabeçalho da função como anotações usando a `tag requires` e as invariantes referentes à saída são inseridas no fim da função usando a `tag assert`. A Figura 6.10 apresenta um trecho do código da Figura 6.9 anotado com as invariantes geradas por Daikon.

```
1  ...
2  handle_buffer(){
3
4    /*@ requires i == buffer[i];
5      @ requires buffer == \null;
6      @ requires (\forall int i; (0 <= i && i <= 9) ==> (buffer[i] == 0));
7    */
8
9    int i;
10   for(i=0; i<=range_updated;i++){
11     buffer[i]=i; // BUG?
12   }
13
14   //@ assert i == 9;
15   //@ assert \at(i,Pre) == buffer[\at(i,Pre)];
16   //@ assert \at(buffer[\at(i,Here)-1],Pre) == buffer[\at(i,Pre)];
17 }
18 ...
```

Figura 6.10: Programa exemplo_mot.c anotado com as invariantes em ACSL.

Finalizado a inserção das invariantes no programa que está sendo analisado, o método proposto tem como objetivo converter estas anotações com invariantes em expressões na linguagem C para então efetuar a verificação com um BMC. A transformação das anotações com invariantes é feita pelo método proposto utilizando um *plug-in* da ferramenta Frama-C (Correnson *et al.*, 2009) denominado de E-ACSL (*Executable ANSI/ISO C Specification Language*) [13] (Delahaye *et al.*, 2013). O *plug-in* E-ACSL traduz automaticamente um programa C anotado em outro programa executável que falha em tempo de execução se uma dada anotação é violada. Vale ressaltar que neste caso as anotações são transformadas em código C no programa analisado. A ideia principal do E-ACSL é ser usado como um verificador de assertivas em tempo de execução, onde o código traduzido contém a função `e_acsl_assert` com os argumentos referentes às anotações. Contudo, neste método proposto utilizou-se o E-ACSL apenas para a geração do código, ou seja, traduzir as anotações em código C, conforme é apresentado na Figura 6.11.

Finalizado a geração do código a partir das anotações, o método proposto então efetua uma leitura do código para identificar a função `e_acsl_assert` que contém as expressões matemáticas relacionadas às invariantes geradas. Esta identificação objetiva a geração de um código final que será então verificado por um BMC, neste caso o ESBMC. A geração do código para o BMC consiste em transformar as funções `e_acsl_assert` em assumes, por exemplo, usando a função `__ESBMC_assume` para o ESBMC. Adicionalmente, o método também identifica *if*s que

[13] http://frama-c.com/eacsl.html

```
1  ...
2  int handle_buffer(void)
3  {
4    ...
5    int i;
6    /*@ requires i ≡ buffer[i];
7        requires (int *)buffer ≡ \null;
8        requires ∀ int i; 0 ≤ i ∧ i ≤ 9 ⟹ buffer[i] ≡ 0;
9    */
10   {
11     int __e_acsl_forall;
12     int __e_acsl_i;
13     e_acsl_assert(i < 10,(char *)"RTE",(char *)"handle_buffer",
14                   (char *)"index_bound: i < 10",27);
15     e_acsl_assert(0 <= i,(char *)"RTE",(char *)"handle_buffer",
16                   (char *)"index_bound: 0 <= i",27);
17     e_acsl_assert(i == buffer[i],(char *)"Precondition",
18                   (char *)"handle_buffer",(char *)"i == buffer[i]",27);
19     e_acsl_assert(buffer == (void *)0,(char *)"Precondition",
20                   (char *)"handle_buffer",(char *)"(int *)buffer == \\null",
21                   28);
22     __e_acsl_forall = 1;
23     __e_acsl_i = 0;
24     while (1) {
25       if (__e_acsl_i <= 9) ; else break;
26       e_acsl_assert(__e_acsl_i < 10,(char *)"RTE",(char *)"handle_buffer",
27                     (char *)"index_bound: __e_acsl_i < 10",29);
28       e_acsl_assert(0 <= __e_acsl_i,(char *)"RTE",(char *)"handle_buffer",
29                     (char *)"index_bound: 0 <= __e_acsl_i",29);
30       if (buffer[__e_acsl_i] == 0) ;
31       else {
32         __e_acsl_forall = 0;
33         goto e_acsl_end_loop1;
34       }
35       __e_acsl_i ++;
36     }
37     e_acsl_end_loop1: ;
38     e_acsl_assert(__e_acsl_forall,(char *)"Precondition",
39                   (char *)"handle_buffer",
40                   (char *)"\\forall int i; 0 <= i && i <= 9 ==> buffer[i] == 0",
41                   29);
42     ...
43   /*@ assert i ≡ 9; */
44   e_acsl_assert(i == 9,(char *)"Assertion",(char *)"handle_buffer",
45                 (char *)"i == 9",46);
46   /*@ assert \at(i,Pre) ≡ buffer[\at(i,Pre)]; */
47   e_acsl_assert(__e_acsl_at_4 < 10,(char *)"RTE",(char *)"handle_buffer",
48                 (char *)"index_bound: __e_acsl_at_4 < 10",0);
49   e_acsl_assert(0 <= __e_acsl_at_4 ,(char *)"RTE",(char *)"handle_buffer",
50                 (char *)"index_bound: 0 <= __e_acsl_at_4",0);
51   e_acsl_assert(__e_acsl_at_3 == buffer[__e_acsl_at_4],(char *)"Assertion",
52                 (char *)"handle_buffer",
53                 (char *)"\\at(i,Pre) == buffer[\\at(i,Pre)]",47);
54 ...
```

Figura 6.11: Programa exemplo_mot.c gerado pelo E-ACSL.

contenham variáveis com o sufixo `__e_acsl_`, pois estes *if*s fazem parte de um código auxiliar gerado pelo E-ACSL para anotações, por exemplo, com o uso de *tags* como o `\forall` que é transformado em um código com uma estrutura de repetição e condicionais (ver exemplo nas linhas 24 e 25 na Figura 6.11). Na linha 17 do código na Figura 6.11 se tem a seguinte função:

```
e_acsl_assert(i == buffer[i],(char *)“Precondition”,
(char *)“handle_buffer”,(char *)“i == buffer[i]”,27);
```

A estrutura da função `e_acsl_assert` consiste em: no primeiro argumento da função é uma assertiva/expressão da invariante (`i == buffer[i]`); o segundo argumento é o tipo da assertiva identificado pelo E-ACSL (`(char *)"Precondition"`); o terceiro é o nome da função onde foi identificado a anotação (`(char *)"handle_buffer"`); o quarto é o texto referente a anotação anteriormente identificada (`(char *)"i == buffer[i]"`); e por último o quinto é o número da linha onde foi identificada a anotação com a invariante. Assim, para a geração do assume para o BMC. O método proposto gera o novo código para o BMC, onde para cada função `e_acsl_assert` identificada o método substitui esta função pela função `__ESBMC_assume` que receberá como argumento o primeiro argumento da função `e_acsl_assert` que contém a assertiva referente a invariante gerada pelo Daikon. Vale ressaltar que também será adicionado uma função `__ESBMC_assume` que terá como argumento a condição de cada `if` identificado pelo método. A Figura 6.12 apresenta o resultado desta tradução.

```
1  ...
2  int handle_buffer(void)
3  {
4    ...
5    int i;
6    /*@ requires i ≡ buffer[i];
7        requires (int *)buffer ≡ \null;
8        requires ∀ int i; 0 ≤ i ∧ i ≤ 9 ⟹ buffer[i] ≡ 0;
9    */
10   {
11     int __e_acsl_forall;
12     int __e_acsl_i;
13
14     __ESBMC_assume(i < 10);
15     __ESBMC_assume(0 <= i);
16     __ESBMC_assume(i == buffer[i]);
17     __ESBMC_assume(buffer == (void *)0);
18
19     __e_acsl_forall = 1;
20     __e_acsl_i = 0;
21     while (1) {
22       __ESBMC_assume(__e_acsl_i <= 9);
23       if (__e_acsl_i <= 9) ; else break;
24
25       __ESBMC_assume(__e_acsl_i < 10);
26       __ESBMC_assume(0 <= __e_acsl_i);
27
28       if (buffer[__e_acsl_i] == 0) ;
29       else {
30         __e_acsl_forall = 0;
31         goto e_acsl_end_loop1;
32       }
33       __e_acsl_i ++;
34     }
35     e_acsl_end_loop1: ;
36     __ESBMC_assume(__e_acsl_forall);
37     ...
38   /*@ assert i ≡ 9; */
39   __ESBMC_assume(i == 9);
40   /*@ assert \at(i,Pre) ≡ buffer[\at(i,Pre)]; */
41   __ESBMC_assume(__e_acsl_at_4 < 10);
42   __ESBMC_assume(0 <= __e_acsl_at_4);
43   __ESBMC_assume(__e_acsl_at_3 == buffer[__e_acsl_at_4]);
44 ...
```

Figura 6.12: Programa exemplo_mot.c traduzido de E-ACSL para o ESBMC.

6.3.2 Tradução das invariantes do ASPIC

As invariantes geradas por ASPIC (ver Seção 6.2) estão no formato suportado pela linguagem de programação C. Contudo, elas estão armazenadas em um relatório no formato de texto. Logo, a tradução destas invariantes consiste em: selecionar do relatório com as invariantes, geradas por ASPIC, trechos específicos contendo as palavras `stop`, `wh`, e `ERROR`; e coletar as invariantes selecionadas dos trechos de texto do relatório para aplicar um formato que será utilizado pelo ESBMC, na verificação do programa analisado.

No relatório gerado por ASPIC são apresentados diversas palavras (tais como: `stop`, `wh`, `ERROR`, `init_5` e outras) que delimitam invariantes para pontos específicos do programa (ver um trecho na Figura 6.13 para o programa na Figura 6.2). No método proposto, optou-se por utilizar apenas as invariantes apontadas pelas palavras `stop`, `wh`, e `ERROR`, uma vez que estas indicam respectivamente: o fim da execução do programa; pontos relacionados a condições de repetição (*loops*) e rótulos de localização de propriedades a serem verificadas (por exemplo, com o uso de `assert` do C representado com `if` e `goto ERROR`). Vale ressaltar que não se optou em utilizar todas as palavras que apontam uma dada invariante, pois no relatório do ASPIC nem todas as palavras contém um rastreamento direto (*backtrack*) ao código do programa analisado. A seleção destes trechos de texto com as invariantes é feito via expressões regulares.

```
1
2  ***
3  Aspic by Laure Gonnord, version 3.3
4  Binary compiled on jeudi 12 septembre 2013
5  Input file = main.fst
6  ***
7  ...
8  *** Results :
9     * Invariants =
10 stop -----> {0}
11
12 start -----> {1>=0}
13
14 init_2 -----> {1>=0}
15
16 lbl_5 -----> {i>=0 && i<n0 && i==k}
17
18 wh -----> {i>=0 && i<n0 && i+k+2>n0}
19
20 ...
21
22 ERROR  -----> {i+k+2>n0 && i>=0 && i+2>n0 && k+2>0}
23    * Remember that we have no bad region
24    * Acceleration has been applied around 1 location(s)
25    * Stats =  { time=0.129661; iterations=25; descendings=0; }
26
27  Finished !
```

Figura 6.13: Trecho do relatório com as invariantes gerado por ASPIC.

Efetuado a seleção das invariantes que serão utilizadas na verificação do programa analisado (ver Seção 6.5), o método coleta as invariantes em cada linha obtida através da extração de cada expressão/predicado gerada na invariante que está entre as chaves no texto, e então adicionada esta invariante a função `__ESBMC_assume` suportada pelo ESBMC. O método proposto pode então

adicionar as invariantes traduzidas em uma nova instância do código analisado. Caso o usuário do método opte em fazer a mesclagem das invariantes geradas por outras ferramentas, está inclusão só ocorrerá após a tradução de todas as invariantes geradas pelas ferramentas.

A instrumentação das invariantes do código analisado consiste em gerar uma nova cópia do código analisado, conforme é apresentado na Figura 6.14. Nesta nova instância, as invariantes obtidas do relatório do ASPIC, com as palavras/delimitadores `stop` e `ERROR`, são inseridas no início da função após a declaração das variáveis. E as invariantes obtidas em `wh` são inseridas após o cabeçalho do laço (*loop*) na função. Caso exista mais de um laço estes são identificados no relatório do ASPIC. Durante a inserção das invariantes, o método identifica quantos laços o programa possui para selecionar as invariantes geradas, uma para cada laço, ou seja, das invariantes em `wh`, onde `wh` estará presente mais de uma vez no relatório do ASPIC.

```
1  int main() {
2    int n0, n1,n2;
3    int i = 0; int k = 0;
4
5    __ESBMC_assume(i+k+2>n0 && i>=0 && i+2>n0 && k+2>0); // ERROR
6    __ESBMC_assume(i>=0 && i<n0 && i+k+2>n0); //wh
7    while( i < n0 ) {
8      i++;
9      k++;
10   }
11
12   ...
13
14 }
```

Figura 6.14: Trecho do novo código gerado a partir das invariantes do ASPIC.

6.3.3 Tradução das invariantes do PIPS

As invariantes geradas por PIPS (como apresentado na Figura 6.15) já possuem certa similaridade com a sintaxe da linguagem C. Contudo, em alguns casos as invariantes geradas apresentam expressões matemáticas em um formato (exemplo, $2j < 5t$) que não são aceitas pela sintaxe da linguagem C. Logo, neste caso a expressão $2j < 5t$ deveria ser $2 * j < 5 * t$. Outro item identificado nas invariantes é o sufixo *#init* que é utilizado para distinguir valores antigos de novos valores (Maisonneuve *et al.*, 2014). Assim, a fim de se utilizar as invariantes geradas pelo PIPS, é necessário se efetuar a tradução das estruturas apresentadas, bem como a criação de mecanismo que auxiliam na validação das expressões nas invariantes.

```
1
2  //  P() {}
3
4  int main(int argc, char *argv[])
5  {
6
7  //  P() {}
8
9      int n0, n1, n2;
10
11 //  P(n0,n1,n2) {}
12
13     int i = 0;
14
15 //  P(i,n0,n1,n2) {i==0}
16
17     int k = 0;
18
19 //  P(i,k,n0,n1,n2) {i==0, k==0}
20
21 ...
22
23 //  P(i,k,n0,n1,n2) {i==0, 0<=k, n0<=k, n0+n1<=k, n0+n1+n2<=k,
24 //     n0+n2<=k, n1<=k, n1+n2<=k, n2<=k}
25
26     while (i<n2) {
27
28 //  P(i,k,n0,n1,n2) {0<=i, n0+n1+n2<=i+k, n0+n2<=i+k, n1+n2<=i+k,
29 //     n2<=i+k, i+1<=n2}
30
31         i++;
32
33 //  P(i,k,n0,n1,n2) {1<=i, n0+n1+n2+1<=i+k, n0+n2+1<=i+k,
34 //     n1+n2+1<=i+k, n2+1<=i+k, i<=n2}
35
36         k--;
37     }
38
39     ...
40
41 }
```

Figura 6.15: Trecho do resultado com as invarantes gerado por PIPS.

O método proposto para efetuar a tradução das invariantes utiliza o Algoritmo 5 que recebe como entrada o código gerado por PIPS (`PIPSCode`), que contêm as invariantes em forma de comentários, e gera como saída um novo código (`NovoCodeInv`) com as invariantes no formato suportado pela sintaxe da linguagem C. A complexidade do tempo de execução do algoritmo é $O(n^2)$, onde n é o tamanho do código C com as invariantes geradas pelo PIPS. O Algoritmo 5 é divido em três partes: (1) para a identificação da estrutura `#init` no código com as invariantes do PIPS; (2) geração de código auxiliar para suporte na tradução da estrutura `#init` nas invariantes; e por último (3) a tradução das expressões matemáticas contidas nas invariantes, que consiste na transformação do formato das invariantes do PIPS para o formato da sintaxe da linguagem C.

Na Linha 3 do Algoritmo 5 é executado a primeira parte da tradução das invariantes que consiste em percorrer cada linha do código do programa com as invariantes e identificar se um dado comentário consiste em uma invariante gerada por PIPS (linha 4). Casos seja uma invariante, é identificado se ela contém a estrutura `#int` e então é armazenada a localização da invariante, bem como, o tipo e nome da variável que é o prefixo da estrutura `#int` (linha 6).

```
   Input: PIPSCode - Código C com as invariantes do PIPS
   Output: NovoCodeInv - Novo código com as invariantes suportadas pela
           linguagem C
   // dicionário para identificação dos #init
1  dict_variniteloc ← { }
   // lista para o novo código gerado na tradução
2  NovoCodeInv ← { }
   // Parte 1 - identificação dos #init nas invariantes
3  foreach linha do PIPSCode do
4      if é um comentário PIPS com este formato // P(w,x) {w == 0,
       x#init > 10} then
5          if no comentário tiver o padrão ([a-zA-Z0-9_]+)#init then
6              dict_variniteloc[linha] ← a variavél com sufixo #init
7          end
8      end
9  end
   // Parte 2 - geração de código para os #init nas invariantes
10 foreach linha do PIPSCode do
11     NovoCodeInv ← linha
12     if for o início de uma função then
13         if alguma linha desta função ∈ dict_variniteloc then
14             foreach variavél em dict_variniteloc do
15                 NovoCodeInv ← Declarar uma variavél com este formato tipo
                   var_init = var;
16             end
17         end
18     end
19 end
   // Parte 3 - correção no formato das invariantes
20 foreach linha do NovoCodeInv do
       // lista para com a invariantes em novo formato
21     listinvpips ← { }
22     NovoCodeInv ← linha
23     if é um comentário PIPS com este formato // P(w,x) {w == 0,
       x#init > 10} then
24         foreach expressão em {w == 0, x#init > 10} do
25             listinvpips ← Reformular a expressão de acordo com sintax do C e
               substituir #init por _init
26         end
27         NovoCodeInv ← __ESBMC_assume(concatenar a invariantes em
           listinvpips com &&)
28     end
29 end
```

Algoritmo 5: Algoritmo de tradução das invariantes do PIPS

Finalizando a identificação das estruturas `#int` nas invariantes, a segunda parte do Algoritmo 5 é executada na linha 10 que consiste em percorrer novamente cada linha do código do programa com as invariantes (`PIPSCode`) e identificar o início de cada função no código. Para cada função identificada, é verificado se naquela função foi identificado alguma estrutura `#int` (linha 13). Caso tenha sido identificado, para cada variável que tinha o sufixo `#int`, é gerado uma nova linha de código, no início da função, com a declaração de uma variável auxiliar que conterá o valor antigo da variável, ou seja, o seu valor no início da função. A nova variável criada tem o seguinte formato `var_init`, onde `var` é o nome da variável. Assim, durante a execução desta etapa do algoritmo é gerado uma nova instância do código (`NovoCodeInv`).

Na última parte do algoritmo (linha 20), cada linha da nova instância do código (`NovoCodeInv`), gerada anteriormente, é percorrida para transformar as expressões das invariantes no comentário do PIPS em expressões suportadas pela linguagem C. Esta transformação consiste na aplicação de expressões regulares (linha 25), a fim de: adicionar operadores (exemplo, de $2j$ para $2 * j$) e substituir a estrutura `#int` apra `_int`. Para cada comentário/invariante analisado é gerada uma nova linha de código com o novo formato, onde nesta linha cada expressão analisada é concatenada com o operador `&&` e adicionada a função `__ESBMC_assume`, conforme é apresentado na Figura 6.16.

```
1  ...
2      i = 0;
3
4  //   P(i,k,n0,n1,n2) {i==0, 0<=k, n0<=k, n0+n1<=k, n1<=k}
5
6  __ESBMC_assume( i==0 && 0<=k && n0<=k && n0+n1<=k && n1<=k );
7
8      while (i<n2) {
9
10 //   P(i,k,n0,n1,n2) {0<=i, i<=k, i+n0<=k, i+n0+n1<=k, i+n1<=k,
11 //       i+1<=n2}
12
13 __ESBMC_assume( 0<=i && i<=k && i+n0<=k && i+n0+n1<=k && i+n1<=k && i+1<=n2 );
14         i++;
15
16 //   P(i,k,n0,n1,n2) {1<=i, i<=k+1, i+n0<=k+1, i+n0+n1<=k+1, i+n1<=k+1,
17 //       i<=n2}
18
19 __ESBMC_assume( 1<=i && i<=k+1 && i+n0<=k+1 && i+n0+n1<=k+1 && i+n1<=k+1 && i<=n2 );
20         k++;
21     }
22 ...
```

Figura 6.16: Trecho do novo código gerado a partir das invariantes do PIPS.

6.3.4 Tradução das invariantes do InvGen

As informações das invariantes geradas por InvGen contêm o número da linha (apresentado como um intervalo) no código analisado de onde a invariante foi inferida, e a invariante com suas respectivas expressões, conforme é apresentado na Figura 6.17. As invariantes geradas por InvGen já estão no formato suportado pela linguagem de programação C. Contudo, as expressões contidas em cada invariante estão concatenadas com vírgulas. Assim, a tradução executada para as invariantes consiste em substituir as vírgulas pelo operador `&&` e então adicionar a invariante traduzida na função `__ESBMC_assume`.

```
1 #pc(main-37-38): [n0-k-i=<0]
2 #pc(main-30-33): [n0-k=<0,n1+n0-k-i=<0]
3 #pc(main-23-27): [n2+n0-k-i=<0,n0-k=<0,n1+n0-k=<0,n2+n1+n0-k-i=<0]
4 #pc(main-16-21): [n0-k+i=<0,n1+n0-k+i=<0,i>=0]
5 #pc(main-9-15): [n0-k+i=<0,i>=0]
6 #pc(main-1-10): [k=i]
```

Figura 6.17: Trecho do relatório com as invariantes gerado por InvGen.

A instrumentação das invariantes do código analisado consiste em gerar uma nova cópia do código analisado onde, nesta nova instância, após cada linha identificada (o fim do intervalo, por exemplo, em `main-23-27` é o `27`) no relatório do InvGen é inserido a invariante traduzida. Desta forma, gerando um novo código com as invariantes geradas, conforme é apresentado na Figura 6.18. Caso o usuário do método opte em mesclar as invariantes geradas por outras ferramentas, a inserção das invariantes ocorrerá como no caso do ASPIC, somente após a tradução de todas as invariantes geradas pelas outras ferramentas. No caso do InvGen, durante a mesclagem dos invariantes, será analisado se para uma dada linha identificada pelo InvGen também foi identificada pelo PIPS, a fim de compilar as invariantes.

```
1
2   i = 0;//
3   __ESBMC_assume(n0-k+i<=0 && i>=0);
4   while( i < n1 ) {
5     i++;
6     k++;
7   }
8
9   i = 0;//
10  __ESBMC_assume(n0-k+i<=0 && n1+n0-k+i<=0 && i>=0);
11  while( i < n2 ) {
12    i++;
13    k++;
14  }
15
16  i = 0; //
17  __ESBMC_assume(n2+n0-k-i<=0 && n0-k<=0 && n1+n0-k<=0 && n2+n1+n0-k-i<=0);
18  while( i < n2 ) {
19    i++;
20    k--;
21  }
```

Figura 6.18: Trecho do novo código gerado a partir das invariantes do InvGen.

6.4 Etapa III: Mesclagem de Invariantes

Nesta etapa é efetuado a mesclagem das invariantes geradas e armazenadas na segunda etapa (ver Seção 6.3). Logo, finalizado a tradução das invariantes, caso o usuário do método opte por efetuar a mesclagem das invariantes geradas, o método proposto efetua uma análise sobre as invariantes traduzidas visando identificar diferenças entre as invariantes geradas para efetuar a concatenação das invariantes. Desta forma, a mesclagem das invariantes consiste em: identificar diferenças entre as invariantes geradas e adicionadas em cada linha do programa analisado; e a geração do código analisado com as invariantes concatenando os códigos gerados por cada ferramenta e suas respectivas invariantes (ver Seção 6.3).

As informações coletadas na primeira e segunda etapa (ver Seção 6.2 e 6.3) são utilizadas para identificar se as invariantes são iguais. A identificação é feita comparando cada linha de código com invariante gerada por uma ferramenta com as invariantes geradas pela outras ferramentas. Vale ressaltar que o código com invariantes selecionado para iniciar a comparação é que aquele que possui o maior número de invariantes geradas. Na comparação os seguintes itens coletados

da geração das invariantes são utilizados para auxiliar a identificação de invariantes diferentes: (i) os comentários das ferramentas para a invariante analisada utilizando expressões regulares; (ii) o número de elementos (variáveis e operadores) na invariante, para a representação de operadores matemáticos como por exemplo o <=, => e $+ - * /$; e (iii) a identificação das variáveis que compõem a invariante, ou seja, a variável existe ou não na invariante.

Efetuada a identificação e comparação das invariantes é gerado um relatório apontando às linhas do código referente às invariantes. Este relatório contém: o número da linha para o qual a invariante foi inserida; se a invariante é igual ou diferente de uma gerada por outra ferramenta; e a invariante copiada do código analisado e gerado na Etapa II. Tomando como base o código que possui o maior número de invariantes adicionadas, o método proposto gera uma nova instância/cópia deste código concatenado as invariantes identificadas como diferentes (no relatório gerado na comparação) para uma determinada linha do código do programa. A concatenação das invariantes consiste em unir as expressões na invariante com o operador && dentro da função `__ESBMC_assume`.

6.5 Etapa IV: Verificação Guiada por Invariantes de Programas com ESBMC

Nesta última etapa, o método proposto repassa o código gerado com as invariantes instrumentadas ao ESBMC para verificar as propriedades de segurança do programa analisado. Neste caso, a verificação será guiada pelas invariantes inferidas do programa que durante a execução da verificação irão definir restrições aos caminhos a serem explorados no modelo do BMC. Assim, simplificando o modelo a ser verificado pelo BMC.

O ESBMC provê suporte para a execução da verificação em dois modos. A primeira é a execução simples do BMC, que consiste em verificar (a negação de) uma dada propriedade em uma dada profundidade do modelo (mais detalhes ver Seção 2.3.1). A segunda é a verificação usando k indução (ver Seção 2.3.1.2) que consiste em usar uma abordagem de profundidade iterativa, e verifica para cada k etapa até um valor máximo, três casos diferentes chamados de caso base, condição de avanço, e passo indutivo (Rocha *et al.*, 2015b). Deste modo, o usuário pode optar por um destes modos de verificação a fim de aperfeiçoar o resultado de sua verificação com base em um conhecimento prévio do programa analisado. Adicionalmente, identificou-se (durante a execução de testes preliminares) que, para que a abordagem da k indução seja mais eficiente, existe a necessidade de uma reverificação/refinamento do resultado encontrado pelo ESBMC.

A reverificação do programa analisado com as invariantes, implementado neste método proposto, consiste em modificar o algoritmo de k-indução do ESBMC, na Seção 2.3.1.2. Nesta nova versão, apresentada no Algoritmo 6, são adicionadas as variáveis `ultimo_resultado` (que armazena o último resultado identificado na verificação de uma dada etapa da k indução) e `force_casobase` que é um identificador para a execução da reverificação na etapa do caso base. A principal diferença na execução do Algoritmo 6 consiste em identificar se na etapa de condição de avanço (linha 17)

e no passo indutivo (linha 22) o resultado da verificação foi `TRUE`, ou seja, não houve violação de propriedades. Dessa forma, deve ser efetuado uma nova verificação do programa com um *bound* maior (linha 7, neste caso optou-se por um incremento de 5 no *bound* atual) na etapa do caso base. Caso seja identificada alguma propriedade violada o resultado será `FAILED`, caso contrário o último resultado armazenado na variável `ultimo_resultado` (neste caso `TRUE`) é retornado.

```
   Input: programa P' com invariantes e propriedades de segurança
          φ
   Output: TRUE, FALSE, ou UNKNOWN
 1 begin
 2     k = 1;
 3     ultimo_resultado = UNKNOWN ;
 4     force_casobase = 0 ;
 5     while k <= iteracoes_maxima do
 6         if force_casobase > 0 then
 7             k=k+5 ;
 8         end
 9         if casoBase (P', φ, k) then
10             apresenta o contra-exemplo s[0...k] ;
11             return FALSE ;
12         end
13         else
14             if force_casobase > 0 then
15                 return ultimo_resultado;
16             end
17             k=k+1 if condicaoAvanco (P', φ, k) then
18                 force_casobase = 1 ;
19                 ultimo_resultado = TRUE ;
20             end
21             else
22                 if passoIndutivo (P', φ, k) then
23                     force_casobase = 1 ;
24                     ultimo_resultado = TRUE ;
25                 end
26             end
27         end
28     end
29     return UNKNOWN;
30 end
```

Algoritmo 6: Algoritmo de k indução com reverificação no caso base.

6.6 Resultados Experimentais usando Invariantes na Verificação de Programas

Esta seção descreve o planeamento, concepção, execução e análise dos resultados de um estudo empírico realizado com o objetivo de avaliar o método proposto para adicionar invariantes de programas à verificação executada por *Bounded Model Checkers* (ver Seção 6), neste caso o ESBMC. O estudo foi conduzido aplicando o método proposto sobre *benchmarks* públicos de programas em C. Os experimentos foram conduzidos em um computador Intel Xeon CPU $E5-2670$ CPU, 2.60GHz, 115GB RAM com Linux 3.13.0 – 35-generic x86_64.

6.6.1 Planejamento e Projeto dos Experimentos

Esta avaliação empírica tem como objetivo analisar a capacidade do método proposto, sobre *benchmarks* públicos de programas em C, para contribuir com a verificação executada pelo ESBMC. Desta forma, nesta avaliação, investiga-se as seguintes questões de pesquisa:

QP1: As ferramentas para a geração de invariantes são capazes de suportar as diferentes estruturas da linguagem de programação C?

QP2: As abordagens propostas para geração de dados de teste contribuem para a geração de invariantes?

QP3: As invariantes geradas contribuem na verificação executada pelo ESBMC?

Visando responder as questões de pesquisa, a execução deste experimento foi dividida em duas partes: Na primeira parte, considerou-se 16 programas em ANSI-C dos seguintes *benchmarks* públicos: 6 programas do InvGen (Gupta e Rybalchenko, 2015), 7 programas da categoria *loops* do SV-COMP 2014 (Beyer, 2014) e 3 programas do projeto Compsys do ENS Lyon/LIP Lab (Feautrier, 2015). Assim, efetua-se uma análise conceitual dos principais pontos do método proposto que consiste na geração das invariantes e a verificação de programas com as invariantes; e na segunda parte, analisando o resultado da questão de pesquisa `QP1` e `QP3`, considerou-se: 142 programas em ANSI-C da categoria *loops* do SV-COMP 2015 (Beyer, 2015); e 34 programas em ANSI-C com o uso em sistemas embarcados, dos *benchmarks*: Powerstone (Scott *et al.*, 1998) que contém um conjunto de programas em C para sistemas embarcados, por exemplo, para controle de automóvel e aplicações de fax; SNU real-time (SNU, 2012) e o WCET (MRTC, 2012) com programas em C para análise de tempo da execução de pior caso. Assim, visa-se aprofundar a análise do impacto do uso de invariantes na verificação de programas com o ESBMC. Adicionalmente, para esta segunda parte do experimento, apresenta-se uma comparação com as ferramentas: CPAChecker (Beyer e Keremoglu, 2011), CBMC (Clarke *et al.*, 2004a), e ESBMC (Cordeiro *et al.*, 2012a) sem o uso de invariantes.

A execução da primeira parte dos experimentos consistiu em: (1) executar as ferramentas abordadas (PIPS, InvGen, ASPIC e Daikon) no método proposto para gerar as invariantes de cada programa dos *benchmarks* selecionados. Os comandos adotados na execução das ferramentas foram os apresentados na Seção 6.2; (2) instrumentar as invariantes geradas em (1) nos programas, conforme apresentado na Seção 6.3; (3) verificar os programas (com e sem as invariantes geradas) dos *benchmarks* usando o ESBMC (versão 1.24). Esta verificação foi efetuada com a abordagem simples (opção `--unwind` no ESBMC) e com a *k* indução (opção `--k-induction-parallel`); (4) gerar os dados de teste para os programas como apresentado na Seção 6.2.5 e 6.2.6, e então reexecutar a geração das invariantes com as ferramentas para novamente instrumentar as invariantes nos programas; e por fim (5) verificar os programas com as invariantes geradas em (4) usando o ESBMC (ver Seção 6.5).

Vale a pena notar que, para a geração das invariantes com a ferramenta Daikon, é necessário compilar e executar o programa a ser analisado. Portanto, adotou-se a função não determinística (`__VERIFIER_nondet_int()`) implementada na biblioteca do Map2Check (ver Seção 4.9.1) para simular estes valores na execução do programa. Cada um dos programas do *benchmark* foi executado 3 vezes, por causa do modelo não determinístico nos programas. É importante notar que Daikon permite a concatenação dos rastreamentos dos dados (usando a opção `--dtrace-append`) gerados de diferentes execuções do programa. Sendo assim, neste experimento efetuou-se 3 execuções do programa, a fim de possibilitar uma melhor cobertura de execução do programa, bem como, uma melhor análise do Daikon.

A execução da segunda parte dos experimentos consistiu em: (1) selecionar uma das ferramentas (aquela que apresente melhor suporte aos programas em C) para gerar as invariantes dos programas do *benchmark* do SV-COMP 2015 na categoria *loops*; (2) instrumentar as invariantes nos programas do *benchmark*, conforme apresentado na Seção 6.3; (3) executar a verificação dos programas com as invariantes usando o ESBMC (versão 1.25.2) com *k* indução; e por último (4) executar a verificação dos programas dos *benchmarks* com o CPAChecker (r15596), ESBMC v1.25.2 (sem utilização das invariantes) e o CBMC v5.0 para efetuar uma comparação dos resultados.

6.6.2 Execução e Análise de Resultados da Parte 1 do Experimento

Nesta primeira parte do experimento visa-se analisar a capacidade das ferramentas adotadas no método proposto para a geração de invariantes sobre os *benchmarks*. Assim, depois de executar as ferramentas sobre os *benchmarks*, obteve-se os resultados apresentados na Tabela 6.5, onde cada coluna desta tabela significa: (1) o identificador do programa (ID); (2) nome do programa (Programa); e (3) o nome das ferramentas divididas com as colunas com o *status* da geração das invariantes (*Status*), e o tempo para a geração das invariantes (Tempo). O *Status* da geração das invariantes é classificado como `OKAY` quando a geração foi efetuada com sucesso, `FAILED` quando a ferramenta não foi capaz de gerar as invariantes, e `TO` quando o tempo de execução da ferramenta ultrapassou 900 segundos. Na tabela também contêm os tempos de execução das ferramentas

auxiliares utilizadas para dar suporte à geração das invariantes. Por exemplo, no caso do ASPIC, o tempo de execução do c2fsm que é usado para converter um programa em C em um autômato.

Analisando os resultados da Tabela 6.5, verificou-se que a ferramenta PIPS foi capaz de gerar as invariantes para 93.75% dos programas; ASPIC para 87.5%; InvGen para 56.25%; e Daikon para 68.75%. Os programas identificados como `FAILED` no ASPIC estão relacionados ao fato de que o c2fsm utilizado para gerar a entrada para o ASPIC não foi capaz de manipular as estruturas dos programas, gerando resultados como *language not implemented*. No caso do InvGen, os `FAILED` estão também relacionados ao caso da ferramenta não prover suporte a determinadas estruturas dos programas, neste caso: chamada de funções, *arrays* e ponteiros. Os resultados `FAILED` do Daikon significam que a ferramenta foi executada corretamente (ou seja, suporta as estruturas dos programas), contudo não foi capaz de gerar as invariantes. No Daikon, houve o único caso (programa com ID = 4) que ultrapassou os 900 segundos definidos no experimento para a análise do programa.

Objetivando analisar a verificação executada pelo ESBMC sobre os programas dos *benchmarks* com e sem a utilização das invariantes, efetuou-se primeiramente a verificação dos programas dos *benchmarks* **sem** a utilização das invariantes, conforme apresentado na Tabela 6.6, onde cada coluna da tabela significa: (1) O identificador do programa (ID); (2) O Nome do programa (Programa); (3) Os resultados da verificação do ESBMC usando o modo BMC simples (opção `--unwind`); e (4) Os resultados da verificação do ESBMC usando *k* indução (opção `--k-induction-parallel`). As colunas (3) e (4) são subdivididas em `Tempo` com o tempo total da verificação e `Status` que pode resultar em: `TRUE` onde não houve a violação de propriedades no programa, `FAILED` onde é identificada a violação de uma propriedade (o programa têm um *Bug*), e `UNKNOWN` onde houve uma falha na execução da ferramenta (*crash*), ou a ferramenta excedeu o tempo de verificação de 900 segundos.

ID	Programa	Ferramentas											
		PIPS		ASPIC			InvGen			Daikon			
		Status	Tempo(s)	Status	Tempo(s) c2fsm	Tempo(s) aspic	Status	Tempo(s) frontend	Tempo(s)	Status	Tempo(s) gcc	Tempo(s) 3x ksavir	Tempo(s) daikon
1	invgen/confuse.c	OKAY	0.47	OKAY	0.76	0.18	OKAY	0.03	0.27	OKAY	0.08	6.35	1.17
2	invgen/disj_simple.c	OKAY	0.35	OKAY	0.36	0.10	FAILED	-	-	FAILED	0.08	2.81	1.09
3	invgen/nested9.c	OKAY	0.78	OKAY	0.51	0.18	OKAY	0.04	0.24	FAILED	0.07	2.74	1.05
4	invgen/seq-len.c	OKAY	1.10	OKAY	0.63	0.31	OKAY	0.08	0.28	TO	0.08	>900	1.16
5	invgen/svd.c	OKAY	2.65	FAILED	-	-	OKAY	0.48	3.76	FAILED	0.07	2.59	-
6	invgen/mergesort.c	OKAY	1.38	OKAY	0.68	0.98	OKAY	0.18	5.24	OKAY	0.16	7.71	3.71
7	sv_comp_loops/array_bug.c	OKAY	0.50	OKAY	0.45	0.10	FAILED	-	-	OKAY	0.06	7.59	3.13
8	sv_comp_loops/for_infinite_loop_2.c	OKAY	0.66	OKAY	0.50	0.12	OKAY	0.02	0.09	FAILED	0.27	540.00	1.11
9	sv_comp_loops/terminator_01_bug.c	OKAY	0.57	OKAY	0.43	0.10	FAILED	-	-	OKAY	0.12	180.00	1.15
10	sv_comp_loops/terminator_02_bug.c	OKAY	0.69	OKAY	0.51	0.14	FAILED	-	-	OKAY	0.06	2.82	1.13
11	sv_comp_loops/insertion_sort_bug.c	OKAY	0.72	OKAY	0.71	0.13	FAILED	-	-	OKAY	0.08	2.77	1.17
12	sv_comp_loops/verisec_sendmail__tTflag_...loop_bad.c	OKAY	0.66	FAILED	-	-	FAILED	-	-	OKAY	0.07	2.77	1.12
13	sv_comp_loops/trex01.c	OKAY	0.73	OKAY	0.56	0.16	OKAY	0.05	0.16	OKAY	0.08	4.30	1.15
14	compsys_team_rank/nestedLoop.c	OKAY	0.72	OKAY	0.46	0.12	OKAY	0.05	0.14	OKAY	0.08	360.00	3.35
15	compsys_team_rank/random2d.c	OKAY	0.77	OKAY	0.74	0.16	OKAY	0.07	0.08	OKAY	0.07	2.78	1.14
16	compsys_team_rank/realheapsort.c	FAILED	-	OKAY	0.74	0.20	FAILED	-	-	OKAY	0.12	2.77	1.19

Tabela 6.5: Geração das invariantes por ferramentas

ID	Programa	Verificação SEM invariantes			
		BMC simples		k indução	
		Status	Tempo	Status	Tempo
1	invgen/confuse.c	UNKNOWN	900.00	UNKNOWN	900.00
2	invgen/disj_simple.c	UNKNOWN	0.06	TRUE	0.06
3	invgen/nested9.c	FAILED	0.07	FAILED	0.06
4	invgen/seq_len.c	UNKNOWN	900.00	UNKNOWN	0.11
5	invgen/svd.c	UNKNOWN	900.00	UNKNOWN	900.00
6	invgen/mergesort.c	TRUE	0.02	TRUE	0.02
7	sv_comp/array_bug.c	FAILED	0.06	FAILED	0.06
8	sv_comp/for_infinite_loop_2.c	UNKNOWN	900.00	TRUE	0.02
9	sv_comp/terminator_01_bug.c	FAILED	0.06	FAILED	0.06
10	sv_comp/terminator_02_bug.c	FAILED	0.07	FAILED	0.06
11	sv_comp/insertion_sort_bug.c	FAILED	0.59	FAILED	0.75
12	sv_comp/verisec_sendmail__tTflag_arr_one_loop_bad.c	TIMEOUT	900.00	FAILED	0.55
13	sv_comp/trex01.c	TIMEOUT	900.00	TRUE	0.09
14	compsys_team_rank/random2d.c	TIMEOUT	900.00	TRUE	0.02
15	compsys_team_rank/nestedLoop.c	TIMEOUT	900.00	TRUE	0.05
16	compsys_team_rank/realheapsort.c	TIMEOUT	900.00	TRUE	0.05

Tabela 6.6: Verificação dos programas sem o uso de invariantes

Analisando os resultados obtidos na Tabela 6.6, identificou-se que os programas com ID = 1, 4 e 5 resultaram em `UNKNOWN` nos dois modos de verificação efetuado pelo ESBMC. Assim, selecionou-se estes programas para efetuar a verificação com o ESBMC usando as invariantes geradas pelas ferramentas PIPS, ASPIC e InvGen. Os resultados do Daikon não foram utilizados, pois para o programa com ID = 4 e 5 não foram geradas invariantes e para o programa com o ID = 1 a invariante gerada consistiu em *return* == 0, a qual jugou-se não ter um impacto significativo para a verificação do programa. As invariantes geradas por cada ferramenta foram traduzidas e instrumentadas de forma manual nos programas selecionados, conforme as etapas definidas na Seção 6.3. Finalizado a instrumentação das invariantes nos programas, foi efetuada a verificação dos programas usando o ESBMC. A Tabela 6.7 apresenta os resultados da verificação dos programas usando as invariantes. Vale ressaltar que, nesta verificação, usou-se a *k* indução do ESBMC, uma vez que nos resultados apresentados na Tabela 6.6 observou-se que a abordagem da *k* indução conseguiu verificar 7 programas a mais que a abordagem com o BMC simples.

ID	Programa	Verificação usando Invariantes					
		PIPS		InvGen		ASPIC	
		Status	Tempo (s)	Status	Tempo (s)	Status	Tempo (s)
1	Invgen/confuse.c	TRUE	3.405	UNKNOWN	900	UNKNOWN	900
4	invgen/seq-len.c	UNKNOWN	1.934	UNKNOWN	900	UNKNOWN	900
5	invgen/svd.c	UNKNOWN	900	UNKNOWN	900	Sem invariantes	

Tabela 6.7: Verificação dos programas com o uso de invariantes

Analisando os resultados da Tabela 6.7 verificou-se que a ferramenta PIPS foi a única que inferiu invariantes que contribuíram com a verificação do ESBMC no programa de ID = 1, ou seja, as invariantes geradas auxiliaram o ESBMC durante a verificação a fim de determinar um resultado. No ID = 4 houve um *crash* na execução da verificação do ESBMC. As invariantes geradas pelas ferramentas InvGen e ASPIC não contribuíram para determinar um resultado da verificação do

programa no tempo máximo de 900 segundos definidos para o experimento. Com base nos resultados que foram gerados da verificação com as invariantes, inferidas pelas ferramentas, observou-se que ainda há a necessidade de uma melhoria das invariantes geradas para a verificação. Assim, aplicou-se a abordagem proposta nas Seções 6.2.5 e 6.2.6 para a geração de dados de teste visando à inferência de novas invariantes. A Tabela 6.8 apresenta o resultado da execução das abordagens Inception e PathCrawler sobre os programas.

ID	**Programa**	**Geração de dados de teste**						
		Pathcrawler			**Inception**			
		Status	**Cobertura**	**Tempo(s)**	**Status**	**Cobertura**	**Claims Verificadas**	**Tempo(s)**
1	invgen/confuse.c	OKAY	8.33%	2	OKAY	83.33%	5 de 6	185
4	invgen/seq-len.c	OKAY	100.00%	3	OKAY	100.00%	12 de 12	12
5	invgen/svd.c	OKAY	94.44%	10	OKAY	100.00%	39 de 39	39

Tabela 6.8: Resultados da geração dos dados de teste

Na Tabela 6.8 observou-se que as abordagens foram capazes de explorar a máxima cobertura (no geral acima de 70%) de execução dos programas selecionados para a geração dos dados de teste. Contudo, para o programa com ID = 1 o PathCrawler obteve somente 8.33% da cobertura de execução e não foi capaz de determinar os dados/valores para teste. Logo, não gerando uma nova instância do programa, como apresentado na Seção 6.2.6. A partir das novas instâncias dos programas gerados por Inception e PathCrawler com os dados de teste. Adicionalmente, a geração de invariantes foi feita usando somente as ferramentas InvGen e ASPIC, uma vez que o resultado destas ferramentas para auxiliar a verificação dos programas (ver Tabela 6.7) não foi suficiente para pelo menos um programa. Assim, as novas invariantes geradas foram instrumentadas nas novas instâncias dos programas (conforme apresentado na Seção 6.3), e então efetuadas a verificação destes programas usando o ESBMC com *k* indução.

O resultado da verificação dos programas com as novas invariantes inferidas com os dados de testes gerados pelo PathCrawler é como segue: para o programa com o ID = 1 na Tabela 6.7 não houve dados gerados, logo não foi geradas novas invariantes e não foi efetuado sua verificação; no programa como o ID = 4 o resultado foi `UNKNOWN` tanto usando as invariantes geradas por InvGen e ASPIC, como anteriormente, devido a um *crash* na execução da verificação do ESBMC; e para o ID = 5 o resultado da verificação, combinando PathCrawler e InvGen, foi `TRUE` (sem propriedades violadas), onde este é um novo resultado que difere do apresentado na Tabela 6.7 com um `UNKNOWN`). Combinando o PathCrawler e ASPIC não foi efetuado a verificação deste programa, pois o ASPIC não gerou invariantes devido ao fato de não prover suporte as estruturas do programa. Logo, analisando os resultados da verificação com as invariantes geradas a partir do PathCrawler pode-se notar que houve uma melhoria, visto que com a aplicação desta abordagem (combinando PathCrawler e InvGen) foi-se capazes de obter o resultado do programa com o ID = 5.

Utilizando o método Inception para a geração das invariantes, o resultado da verificação para os programas na Tabela 6.7 consistiu na verificação das novas instâncias geradas por Inception com as invariantes inferidas por InvGen e ASPIC. Para o programa com ID = 1 na Tabela 6.7, Inception

gerou 6 novas instâncias do programa que combinado com as invariantes de InvGen resultou em: uma instância o resultado da verificação foi `TRUE`; para duas instâncias não foram geradas invariantes, logo não foram efetuadas as suas verificações; e para três instâncias o resultado da verificação foi `UNKNOWN`. Usando as invariantes geradas por ASPIC nas 6 novas instâncias do programa com ID = 1, o resultado da verificação foi `UNKNOWN`. Assim, para o programa com ID = 1 na Tabela 6.7, usando o método Inception mais as invariantes de InvGen pode-se considerar que este programa esta 16.66% livre de propriedades violadas, ou seja, o resultado da verificação obtido a partir de 1 de 6 instâncias do programa analisado.

Analisando o resultado da aplicação de Inception para o programa com o ID = 4 na Tabela 6.7, Inception gerou 12 novas instâncias do programa que combinado com as invariantes de InvGen e ASPIC resultou em: 9 instâncias com resultado da verificação em `TRUE` com as invariantes de InvGen e 12 em `TRUE` usando ASPIC; 1 instância com resultado da verificação em `UNKNOWN` usando as invariantes de InvGen; e para 2 instâncias não foi gerado invariantes com InvGen. Logo, analisando os resultados, pode-se dizer que usando as novas invariantes de InvGen que o programa com o ID = 4 está 75% livre de erros (propriedades violadas), ou seja, o resultado da verificação obtido a partir de 9 das 12 instâncias do programa analisado. Com base nas invariantes geradas por ASPIC pode-se afirmar que o programa com o ID = 4 está 100%, ou seja, foi possível verificar todas as instâncias geradas por Inception a partir do programa analisado. Assim, pode-se notar que a utilização do método Inception contribui para geração de resultados na verificação que difere do que apresentado anteriormente na Tabela 6.7 como `UNKNOWN`. Vale ressaltar que não foi apresentado os resultados para o programa com o ID = 5, pois o resultado da aplicação do Inception gerou 39 instâncias do programa e como o processo de geração e instrumentação das invariantes têm sido efetuado de forma manual, acredita-se que este processo manual poderia resultar em um longo período de tempo para aplicação das etapa do método e também na possibilidade sua aplicação de forma incorreta.

6.6.3 Execução e Análise de Resultados da Parte 2 do Experimento

Nesta segunda parte do experimento visa-se aprofundar a análise do impacto do uso de invariantes na verificação de programas com o ESBMC. Desta forma, adotou-se as seguintes configurações para a execução do método proposto (ver Seção 6): para a geração de invariantes foi utilizado à ferramenta PIPS, conforme apresentado na Seção 6.2.3, e para a verificação do programa, com as invariantes instrumentadas, o método da *k* indução no ESBMC. Vale ressaltar que estas configurações adotadas neste experimento têm como base os resultados obtidos na parte 1 desta avaliação (ver Seção 6.6.2). O método proposto com estas configurações foi implementado na ferramenta denominada de DepthK [14]. Adicionalmente, para esta segunda parte do experimento, apresentou-se uma comparação com as ferramentas:

- DepthK v1.0 com *k* indução e invariantes, as opções de execução utilizadas foram as definidas no *wrapper script* da ferramenta disponível no seu repositório;

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

- ESBMC v1.25.2 somente com *k* indução. As opções de execução utilizadas foram às definidas no *wrapper script* para o SV-COMP 2013 [15] como recomendado por Lucas Cordeiro (mantenedor);
- CBMC v5.0 com *k* indução, sendo executada com *script* provido (em Beyer *et al.* (2015b)) por Michael Tautschnig (mantenedor); e
- CPAChecker [16] com *k* indução e invariantes, na revisão 15596 do seu repositório SVN. As opções utilizadas foram as definidas em Beyer *et al.* (2015b). Contudo, visando uma melhor apresentação dos resultados, será abordando apenas os resultados das opções que apresentaram os melhores resultados como será apresentado a seguir. Onde estas opções estão definidas em Beyer *et al.* (2015b) com os seguintes títulos: CPAchecker cont.-ref. k-Induction (k-Ind InvGen) e CPAchecker no-inv k-Induction.

Nesta avaliação, como mencionado na Seção 6.6.1, considerou-se 142 programas em ANSI-C da categoria *loops* do SV-COMP 2015 (Beyer, 2015), os quais são organizados em: 49 programas contendo propriedades válidas (ou seja, a ferramenta de verificação deve ser capaz de provar a corretude do programa), e 93 programas contendo propriedades inválidas (isto é, a ferramenta de verificação deve ser capaz de falsificar a propriedade). Adicionalmente, considerou-se 34 programas em ANSI-C com o uso em sistemas embarcados, dos *benchamarks* Powerstone (Scott *et al.*, 1998), SNU real-time (SNU, 2012), e o WCET (MRTC, 2012). Na verificação de cada programa o tempo de execução na CPU foi limitado em 900 segundos, bem como o consumo de memória em 15 GB. Todos os benchmarks e os resultados completos desta avaliação estão disponíveis no repositório da ferramenta DepthK.

Depois de executar as ferramentas sobre os *benchmarks*, obteve-se os resultados apresentados na Tabela 6.9 para o *benchmark* do SV-COMP 2015; e na Tabela 6.10 para o *benchmark* dos programas de sistemas embarcados, onde cada linha destas tabelas significa: (1) nome da ferramenta (Ferramenta); (2) número total dos programas que satisfaz a especificação identificada pela ferramenta (Resultados Corretos); (3) número total dos programas que a ferramenta identificou um erro para um programa que atende a especificação, ou seja, alarme falso ou análise incompleta (Falsos Negativos); (4) número total dos programas que a ferramenta não identifica o erro, ou seja, não foi encontrado propriedade violada ou análise fraca (Falsos Positivos); (5) número total dos programas em que a ferramenta não conseguiu efetuar verificação devido à falta de recursos, falha da ferramenta (*crash*), ou a ferramenta excedeu o tempo de verificação de 15 min (Unknown e TO); (6) o tempo de execução em minutos da verificação para todos os programas na categoria (Tempo).

[15] http://sv-comp.sosy-lab.org/2013/
[16] https://svn.sosy-lab.org/software/cpachecker/trunk

Ferramenta	DepthK + k-induction	ESBMC + k-induction	CPAchecker no-inv k-Induction	CPAchecker cont.-ref. k-Induction (k-Ind InvGen)	CBMC + k-induction
Resultados Corretos	94	70	78	76	64
Falsos Negativos	1	0	0	1	3
Falsos Positivos	0	0	4	7	1
Unknown e TO	47	72	60	58	74
Tempo	190.38min	141.58min	742.58min	756.01min	1141.17min

Tabela 6.9: Resultados da avaliação sobre os *benchmarks* do SVCOMP'15 da categoria *loops*.

Ferramenta	DepthK + k-induction	ESBMC + k-induction	CPAchecker no-inv k-Induction	CPAchecker cont.-ref. k-Induction (k-Ind InvGen)	CBMC + k-induction
Resultados Corretos	17	18	27	27	15
Falsos Negativos	0	0	0	0	0
Falsos Positivos	0	0	0	0	0
Unknown e TO	17	16	7	7	19
Tempo	77.68min	54.18min	1.8min	1.95min	286.06min

Tabela 6.10: Resultados da avaliação sobre os programas dos *benchmarks* Powerstone, SNU, e WCET.

Os resultados experimentais foram avaliados da seguinte forma: verificou-se o resultado da verificação e do tempo apresentado para cada programa do *benchmark* pelas ferramentas. Adotou-se o mesmo esquema de pontuação que é adotado no SVCOMP [17]. Logo, para cada erro verdadeiro (propriedade violada) encontrado, 1 ponto é atribuído; para cada prova correta de segurança, 2 pontos são atribuídos. Uma pontuação de 6 pontos é subtraída para cada alarme errado (falsos positivos) informado pela ferramenta, e 12 pontos são subtraídos para cada prova errada de segurança (falso negativo). Este esquema de pontuação valoriza mais a prova de segurança do programa que encontrar contra-exemplos, e pune significativamente respostas erradas. A Figura 6.19 e Figura 6.20 apresentam os resultados comparativos (das ferramentas adotadas no experimento) entre as pontuações para o *benchmark* do SV-COMP e para o *benchmark* dos programas de sistemas embarcados respectivamente. Vale ressaltar que para os programas de sistemas embarcados, considerou-se que estes não tinham propriedades violadas. Isto porque nestes *benchmarks* não havia uma identificação se no programa ocorria uma violação de propriedade.

Analisando os resultados na Figura 6.19, nota-se que as melhores pontuações foram do DepthK combinado *k* indução e invariantes com 140 pontos, ESBMC usando *k* indução com 105 pontos, e CPAchecker no-inv *k-induction* com 101 pontos. Na Figura 6.20, identificou-se que as melhores pontuações foram do CPAchecker no-inv *k-induction* com 54 pontos, ESBMC usando *k* indução com 36 pontos, e DepthK combinado *k* indução e invariantes com 34 pontos. Desta forma, observou-se que o DepthK obteve um resultado inferior nos *benchmarks* de sistemas embarcados.

[17] http://sv-comp.sosy-lab.org/2015/rules.php

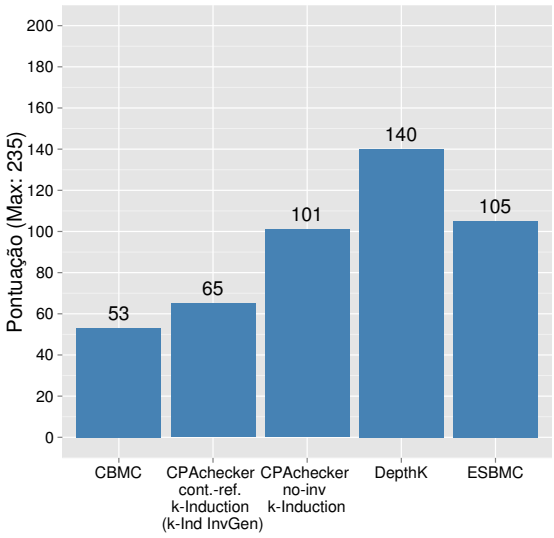


Figura 6.19: Pontuação no *benchmark* do SV-COMP

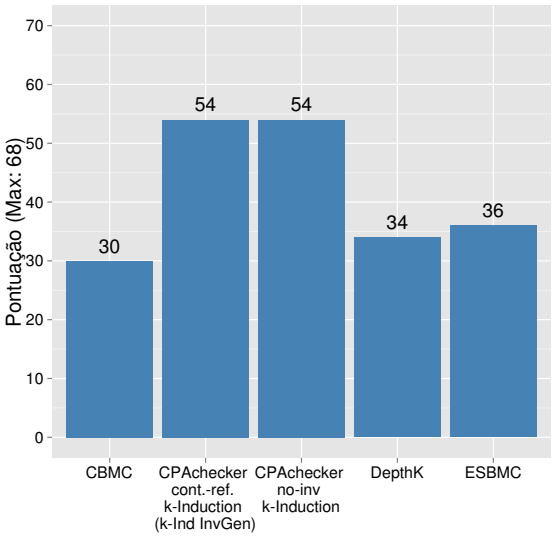


Figura 6.20: Pontuação no *benchmark* dos sistemas embarcados

Contundo, o resultado do DepthK ainda foi superior ao do CBMC e no *benchmark* do SV-COMP obteve a maior pontuação. Analisando os *logs* da execução do DepthK, identificou-se que em parte a baixa pontuação nos *benchmarks* de sistemas embarcados foi devido a 35.30% de resultados classificados como `Unknown`, ou seja, quando não foi possível determinar um resultado ou devido a uma falha de execução da ferramenta. Nos *logs* da execução do DepthK, identificou-se falhas relacionadas a geração de invariantes e na geração do código que é repassado o ESBMC. Vale ressaltar que o DepthK ainda está na sua primeira versão de desenvolvimento e, assim, os resultados são promissores.

Visando analisar o impacto da utilização das invariantes na utilização da verificação com *k* indução, classificou-se a distribuição do resultado do DepthK e ESBMC por cada etapa da verificação com *k* indução, ou seja, o caso base, a condição de avanço, e o passo indutivo. Adicionalmente, foram incluídas as verificações que resultam em `Unknown` (houve uma falha na execução da ferramenta - *crash*) e `timeout` (tempo de execução da ferramenta ultrapassou 900 segundos). Nesta análise, utiliza-se somente os resultados do DepthK e ESBMC, pois estes estão envolvidos com o método proposto e também porque não foi possível identificar as etapas da *k* indução nos *logs* padrões de execução das outras ferramentas. A Figura 6.21 apresenta a distribuição do resultado das etapas da verificação com *k* indução para o *benchmark* do SV-COMP, e a Figura 6.22 para o *benchmark* dos programas de sistemas embarcados.

Analisando a distribuição dos resultados na Figura 6.21, notou-se que o DepthK foi capaz de provar mais de 25.35% de propriedades durante a etapa do passo indutivo que o ESBMC, bem como, na Figura 6.22 para mais de 29.41% também no passo indutivo. Como resultado, observou-se que as invariantes ajudaram a provar que os laços foram suficientemente desenrolados e se a propriedade é válida, tendo em conta que no passo indutivo é verificado que sempre que P (o programa analisado) é válido para k desenrolamentos, ele também é válido após o próximo

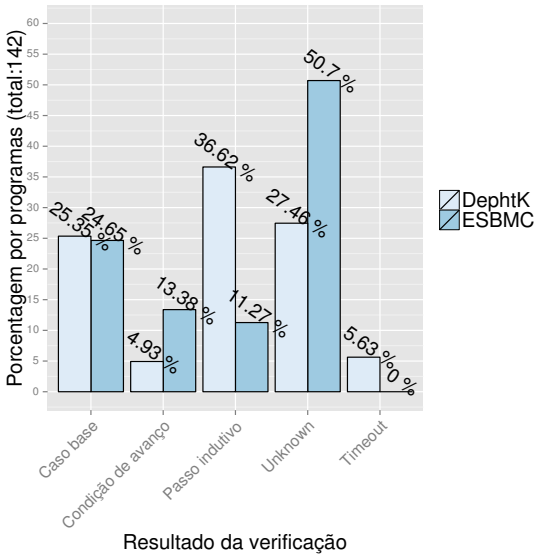


Figura 6.21: Resultado da verificação das etapas da *k* indução para os programas do SV-COMP

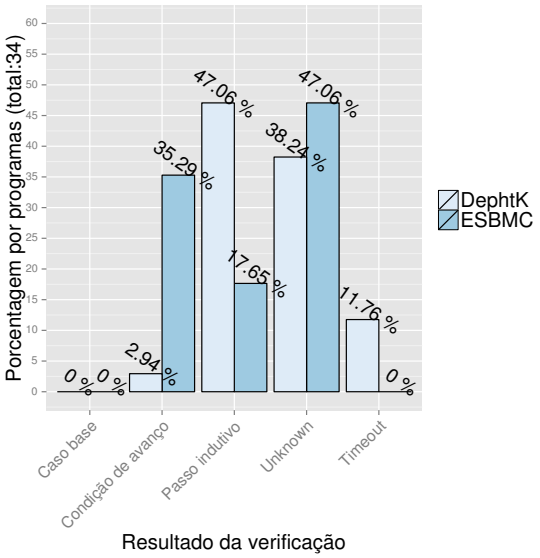


Figura 6.22: Resultado da verificação das etapas da *k* indução para os programas embarcados.

desenrolamento do sistema. Identificou-se também que o DepthK não encontrou uma solução para a verificação em 33.09% (somando `Unknown` e `Timeout`) dos programas na Figura 6.21 e 50% na Figura 6.22. Isto é explicado em parte pelas invariantes geradas a partir de PIPS que não são fortes o suficiente para a verificação com a *k* indução, quer devido aos transformadores ou devido às invariantes que não são convexas; e também devido a alguns erros na implementação. O ESBMC com *k* indução não encontrou solução para a verificação em 50.7% (somando `Unknown` e `Timeout`) dos programas na Figura 6.21, ou seja, 17.61% a mais que o DepthK; e na Figura 6.22 o ESBMC não encontrou solução para a verificação em 47.06%, logo somente 3.64% a menos que o DepthK. Assim, fornecendo evidências de que as invariantes podem melhorar os resultados da verificação.

Analisando o tempo de verificação das ferramentas na Tabela 6.9 para os programas do *benchmark* do SV-COMP, notou-se que o DepthK executou em menor tempo que as demais ferramentas, exceto pelo ESBMC, como pode-se observar também na Figura 6.23, com o tempo em relação a cada programa. Isto se explica pelo fato de ter um tempo adicional para a geração das invariantes. Na Tabela 6.10 identificou-se que o DepthK somente executou em menor tempo que o CBMC, como também pode-se identificar na Figura 6.24 com o tempo em relação a cada programa. Contudo, notou-se que o tempo de execução do DepthK é proporcional ao do ESBMC, uma vez que a diferença de tempo é uma constante de 23.5min, onde este tempo no DepthK está relacionado a geração de invariantes.

Acredita-se que o tempo de execução do DepthK pode ser significativamente melhorando pela correção de alguns erros de implementação. Isto porque grande parte dos resultados gerados como `Unknown` estão relacionados a falhas na execução da ferramenta; e por possiveis ajustes e adições nos comandos do *script* do PIPS (ver Seção 6.2.3) para geração das invariantes, uma vez que o PIPS

possui um conjunto amplo de comandos para transformação de código que podem ter impacto na geração das invariantes e os programas do *benchamarks* de sistemas embarcados podem conter estruturas que venham a requerer transformações especificas de código.

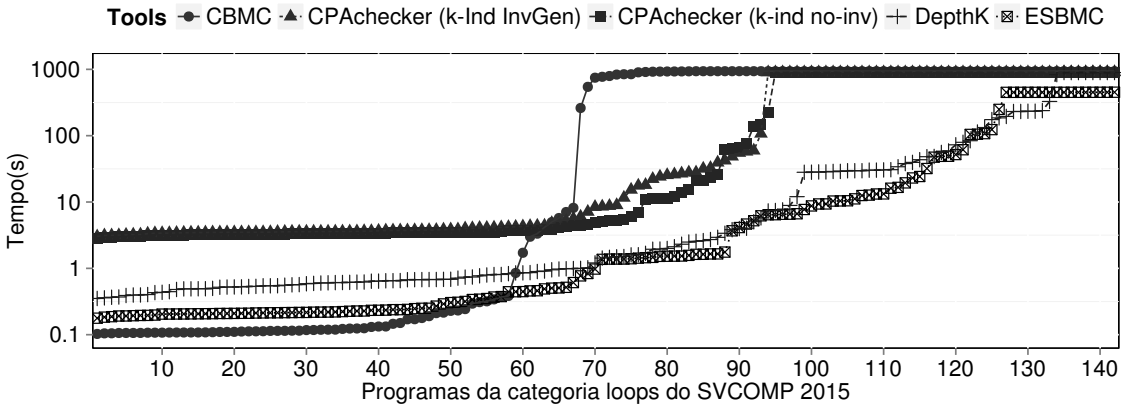


Figura 6.23: Tempo de verificação nos programas do SV-COMP 2015.

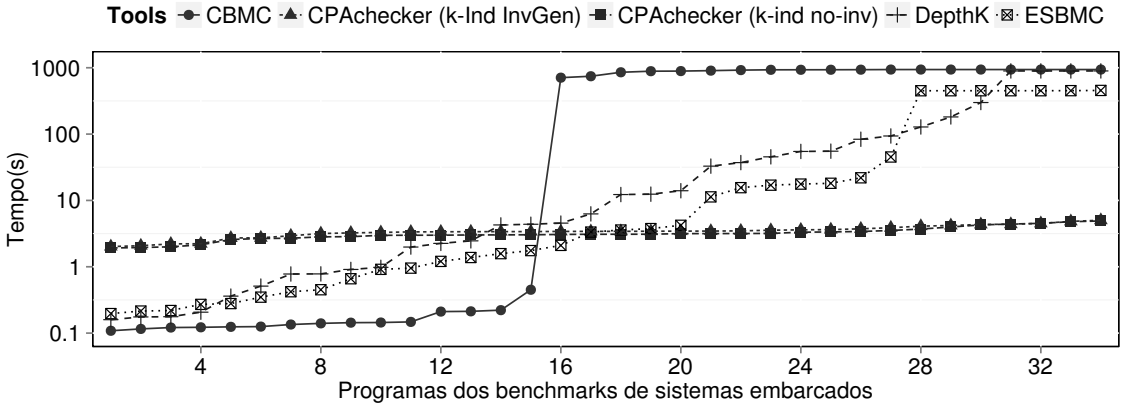


Figura 6.24: Tempo de verificação nos programas dos sistemas embarcados.

6.7 Resumo

Neste Capítulo foi apresentado o método para redução do modelo a ser verificado por um *Bounded Model Checker* pela utilização de invariantes de programas. As invariantes de programas utilizadas são geradas a partir de uma combinação e modificações propostas neste trabalho em ferramentas no estado da arte, estas ferramentas são: ASPIC (Feautrier e Gonnord, 2010), Daikon (Ernst *et al.*, 2007), PIPS (Ancourt *et al.*, 2010) e InvGen (Colón *et al.*, 2003). Na Seção 4.9, descreveu-se o planeamento, concepção, execução e análise dos resultados de um estudo empírico realizado com o objetivo de avaliar o método proposto para adicionar invariantes de programas à verificação executada por *Bounded Model Checkers* (ver Seção 6), neste caso o ESBMC. Visando analisar o método proposto, a execução deste experimento foi dividida em duas partes. Na

primeira parte (ver Seção 6.6.2), efetuou-se uma análise conceitual dos principais pontos do método proposto que consiste na geração das invariantes e a verificação de programas com as invariantes. O resultado demonstrou que as ferramentas (Daikon (Perkins e Ernst, 2004), ASPIC (Feautrier e Gonnord, 2010), PIPS (Ancourt *et al.*, 2010) e InvGen (Colón *et al.*, 2003)) utilizadas pelo método proposto foram capazes de manipular e gerar as invariantes, bem como o método foi capaz de traduzir as invariantes geradas e utilizá-las na verificação com o ESBMC. Na verificação observou-se o impacto da utilização das invariantes, uma vez que de 3 programas que o ESBMC não podia determinar um resultado sem as invariantes e com as invariantes pode determinar o resultado de 2 deles. Na segunda parte do experimento apresentado na Seção 6.6.3 objetivou-se aprofundar a análise do impacto do uso de invariantes na verificação de programas com o ESBMC. Desta forma, adotou-se as seguintes configurações (baseadas nos resultados da primeira parte do experimento) para a execução do método proposto, implementado na ferramenta DepthK: para a geração de invariantes foi utilizado à ferramenta PIPS; e para a verificação do programa, com as invariantes instrumentadas, o método da k indução do ESBMC. Adicionalmente, foi efetuado uma comparação com as ferramentas CPAChecker (Beyer e Keremoglu, 2011), CBMC (Clarke *et al.*, 2004a), e ESBMC (Cordeiro *et al.*, 2012a) sem as invariantes. Os resultados dos experimentos demonstraram ser promissores, uma vez que na avaliação com o programas da categoria *loops* do SV-COMP 2015, o método determinou 11.27% a mais resultados corretos que o CPAChecker (Beyer e Keremoglu, 2011) que obteve o segundo melhor resultado nesta análise. Entretanto, vale ressaltar que alguns pontos de melhorias foram identificados na ferramenta DepthK em relação a correção de falhas na execução da ferramenta, e possíveis ajustes de implementação para a geração das invariantes com PIPS. Isto porque, os resultados com os programas dos *benchmarks* de sistemas embarcados apresentaram resultados inferiores, onde o DepthK somente obteve melhor resultado que a ferramenta CBMC. Contudo, argumentados que o método proposto, em comparação com outros do estado da arte, apresentou resultados promissores que indicam a sua eficiência.

Capítulo 7

Conclusões e Trabalhos Futuros

Um dos principais desafios no desenvolvimento de *software* é garantir a segurança dos sistemas de *software*, especialmente em sistemas embarcados críticos, onde diversas restrições (por exemplo, tempo de resposta e precisão dos dados) devem ser atendidas e mensuradas de acordo com os requisitos do usuário, caso contrário uma falha pode conduzir a situações catastróficas. No contexto de verificação de *software*, visando à qualidade geral do produto, os *Bounded Model Checkers* (BMCs) estão sendo utilizados para descobrir erros sutis em projetos de sistemas de *software* atuais.

Neste trabalho apresentou-se um conjunto de métodos que visam complementar e aperfeiçoar a verificação efetuada pelos *Bounded Model Checkers*, e contribuir com a familiarização da técnica de verificação formal *model checking*. O problema abordado neste trabalho foca em explorar as principais características de BMC, propondo métodos que visam integrar o BMC com outras técnicas de teste, assim como melhorar a aplicação e precisão da verificação de BMC.

Objetivando demonstrar a usabilidade prática dos métodos propostos, realizou-se avaliações experimentais utilizando diferentes *benchmarks* públicos de programas escritos em C, tais como, da categoria *Memory Safety* do SV-COMP 2014 (Beyer, 2014); EUREKA (Eureka, 2012); SNU (SNU, 2012); WCET (MRTC, 2012); NEC (NEC Laboratories America, Inc., 2012); Siemens/SIR (SIR Project, 2012); InvGen (Gupta e Rybalchenko, 2015); da categoria *loops* do SV-COMP 2015 (Beyer, 2015); e Powerstone (Scott *et al.*, 1998).

Nas avaliações experimentais dos métodos propostos, também foi apresentado uma comparação do método Map2Check (ver Seção 4) com as ferramentas: Valgrind's Memcheck (Nethercote e Seward, 2007), CBMC (Clarke *et al.*, 2004a), LLBMC (Merz *et al.*, 2012), CPAChecker (Beyer e Keremoglu, 2011), Predator (Dudka *et al.*, 2014), e ESBMC (Cordeiro *et al.*, 2012a). Uma comparação do método EZProofC (ver Seção 5) com a ferramenta Frama-C (Canet *et al.*, 2009). E para o método de verificação com invariantes (ver Seção 6) uma comparação com as ferramentas CPAChecker (Beyer e Keremoglu, 2011), CBMC (Clarke *et al.*, 2004a), e ESBMC (Cordeiro *et al.*, 2012a) sem a utilização de invariantes. Os resultados obtidos nos experimentos (ver Seção **??**) demonstraram que a metodologia proposta é satisfatória e eficaz em relação à problemática abordada neste trabalho. Em uma visão geral dos resultados obtidos tem-se que:

- O método Map2Check, que gera e verifica automaticamente casos de teste em programas em C (ver Seção 4), detectou os mesmos defeitos que as ferramentas no estado da arte em análise de memória, além de determinar mais resultados corretos que 5 das 6 ferramentas comparadas, incluindo o ESBMC. Além disso, o Map2Check não gerou falsos positivos e negativos.

- O método EZProofC, que instancia programas em C pela utilização de contra-exemplos do BMC (ver Seção 5), foi capaz de reproduzir 100% dos erros identificados pelo ESBMC, além de ser escalável para grandes tamanhos de contra-exemplo (por exemplo, contra-exemplo com 3.344 linhas de código). O EZProofC em comparação com o Frama-C (Canet *et al.*, 2009) identificou/reproduziu os defeitos no programas sem a necessidade da escrita de especificações auxiliares.

- O método de verificação com invariantes (ver Seção 6) demonstrou que as ferramentas (Daikon (Perkins e Ernst, 2004), ASPIC (Feautrier e Gonnord, 2010), PIPS (Ancourt *et al.*, 2010) e InvGen (Colón *et al.*, 2003)), utilizadas pelo método proposto, foram capazes de manipular e gerar as invariantes, bem como o método foi capaz de traduzir as invariantes geradas e utilizá-las na verificação com o ESBMC apresentando resultados que antes não eram possíveis somente com o ESBMC. Adicionalmente, a ferramenta DepthK (utilizando as invariantes do PIPS e k indução) demonstrou resultado promissores, uma vez que apresentou 3.45% a mais de novos resultados corretos, dentre os 176 programas usados no experimento, quando comparado com a ferramenta CPAChecker, a qual apresentou os melhores resultados dentre todas as ferramentas. Além disso, o DepthK apresentou 13.07% a mais de novos resultados corretos quando comparado com o ESBMC sem invariantes.

7.1 Trabalhos Futuros

A aplicação de verificação e testes são técnicas indispensáveis para o desenvolvimento de *software* de alta qualidade. Neste sentido, a integração da verificação formal e testes de *software* têm sido adotados como uma solução amplamente reconhecida para melhorar a qualidade do *software*. Essa integração visa aliviar as fraquezas destas estratégias (Comar *et al.*, 2012; Groce e Joshi, 2008; Holzer *et al.*, 2008). Em teste de *software*, por exemplo, um esforço significativo é necessário para gerar casos de teste eficazes e, como resultado, erros sutis são difíceis de detectar por meio de testes e pode provocar um retrabalho no desenvolvimento do *software* após este está implantado. De acordo com Kebrt e Sery (2009), a adoção de técnicas como *model checking* no processo de desenvolvimento industrial ainda é muito lento. Isso é causado por duas razões principais: escalabilidade limitada para *softwares* com grande quantidade de linhas de código, e a falta de apoio ferramental na integração do processo de desenvolvimento.

Nos últimos anos, pode-se observar uma tendência para a aplicação de técnicas de verificação formal ao nível de implementação. A técnica *Bounded Model Checking* (BMC) está indo nessa direção, uma vez que tem sido aplicada em *softwares* escritos em ANSI-C/C++ (Clarke *et al.*,

2004a; Cordeiro *et al.*, 2012a; Merz *et al.*, 2012). Desta forma, este trabalho apresentou métodos que adotam a integração de métodos formais com a utilização de BMC e técnicas de teste, visando prover um apoio ao processo de desenvolvimento de *software*. Portanto, difundindo a aplicação de métodos formais e ajudando os desenvolvedores não muito familiarizados com este assunto a verificar programas complexos de *software*. Logo, como continuidade do desenvolvimento deste trabalho, propõem-se os seguintes pontos para os métodos propostos.

Map2Check - Geração e Verificação Automática de Casos de Teste em Programas em C usando BMC (ver Seção 4):

1. Estender o instrumentador de código para prover suporte à instrumentação binária, assim ampliando a capacidade do método para outras formas de representação do código, tais como, código de máquina ou uma representação intermediaria de baixo nível como LLVM (Lattner, 2015);

2. Adicionar verificação estática baseado em domínio abstrato (Ströder *et al.*, 2014), visando melhorar o tempo de execução por meio de uma análise preliminar dos dados de rastreamento do programa;

3. Incluir técnicas verificação como injeção de falhas (Arlat *et al.*, 1990) e testes mutantes (Jia e Harman, 2010) visando aprimorar a execução do fluxo do programa, assim permitindo a geração e execução de casos de teste com maior poder de exploração em uma ampla gama de propriedades;

4. Adicionar uma validação do relatório/contra-exemplo gerado pelo Map2Check para a violação de uma da propriedade, pela utilização de técnicas de análise de programas dirigidas a comprovação de erros, por exemplo, como a técnica de *witness validation* apresentado em Beyer *et al.* (2015a). Desta forma, eliminando a necessidade de uma inspeção manual pela geração de falsos alarmes.

EZProofC - Instanciação de Programas em C pela Utilização de Contra-Exemplos de BMC (ver Seção 5):

1. Estender o método proposto para se utilizar outros *model checkers* com os seus respectivos contra-exemplos, tais como o CPAChecker (Beyer e Keremoglu, 2011), Blast Beyer *et al.* (2007a), e Java PathFinder Havelund (1999);

2. Adicionar ao método uma abordagem (por exemplo, adotando testes mutantes (Jia e Harman, 2010)) para explorar novas possibilidades de execução da nova instância criada do programa a partir do contra-exemplo. Assim, gerando novos casos de testes para determinar a violação da propriedade identificada pelo BMC.

Verificação com invariantes de programas (ver Seção 6):

1. Estender os comandos no *script* do PIPS para transformações de código e geração de invariantes com foco em chamadas de funções e desenrolamento de *loops* que podem gerar decisões não determinísticas para a execução do programa analisado;

2. Efetuar uma análise estática do programa com as invariantes já instrumentadas no código para determinar de formar preliminar a violação de uma dada propriedade, como por exemplo, apresentado em Nimmer e Ernst (2001) para programas em Java;

7.2 Observações Finais

As aplicações de *software* no geral requerem certo grau de previsibilidade e confiança, mas principalmente os sistemas embarcados que são usados em uma ampla gama de aplicações sofisticadas, tais como em telefones celulares, aviônicos e médicos. A flexibilidade exigida em tais aplicações aumenta significativamente o um número de funções que são implementadas no *software*, em vez de no *hardware*. Logo, a verificação e garantia da correção deste *software* é crítica devido a considerações de segurança em vários domínios de aplicação.

No que concerne à verificação e teste de *software* escrito na linguagem de programação C, este trabalho propôs um conjunto de métodos para complementar e aprimorar a verificação efetuada por *Bounded Model Checkers* para *software* escrito em C, onde estes métodos utilizam técnicas de transformações de código para analisar propriedades de segurança e demonstrar erros em códigos escritos na linguagem C. As abordagens propostas quando utilizadas isoladamente fornecem alternativas complementares à verificação e, quando interligadas aprimoram a verificação de código. Desta forma, esta tese visa contribuir no processo de desenvolvimento de *software* confiável e previsível. Contudo, o desenvolvimento de *software* confiável é um problema complexo (Jhala e Majumdar, 2009) e os métodos de verificação e teste de *software* ainda estão em contínuo desenvolvimento, uma vez que o aprimoramento da aplicação destes métodos tem sido amplamente explorado pela comunidade científica, principalmente em relação à sua aplicação de forma automática. Por exemplo, verificadores de *software* ainda estão sob forte desenvolvimento, como observado recentemente por Griesmayer *et al.* (2010) e Beyer (2015). Portanto, o desenvolvimento de métodos para verificação e teste de *software* ainda é uma área de pesquisa fértil e que ainda há um campo fértil para ser explorado.

7.3 Lista de Publicações

Esta seção apresenta a lista de publicações obtidas no desenvolvimento desta pesquisa, bem como colaborações relacionadas ao tema deste trabalho.

Lista de publicações obtidas a partir deste trabalho:

1. Herbert Rocha, Raimundo Barreto, Lucas Cordeiro, e Arilo Dias Neto. *Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples*. **In 9th International Conference on Integrated Formal Methods (IFM)**, Lecture Notes in Computer Science, pages 128-142. Springer. 2012, Pisa - Itália.

2. Herbert Rocha e Raimundo Barreto. *Corretude e Segurança de Sistemas Embarcados Críticos Usando Bounded Model Checkers*. **In II Brazilian Conference on Critical Embedded Systems (CBSEC), Student Workshop**. 2012, Campinas - Brasil.

3. Herbert Rocha e Raimundo Barreto. *Correctness and Safety of the Critical Embedded Systems using Bounded Model Checkers with Simplified Verification Methods*. **In Pan-American Software Quality Institute workshop (PASQI), Doctoral Symposium**. 2013, Puntarenas - Costa Rica.

4. Herbert Rocha, Raimundo Barreto e Lucas Cordeiro. *Caracterização do Estado da Arte sobre Invariantes para Inferência de Propriedades na Verificação de Software*. **Relatório Técnico RT-GISE-005**, Universidade Federal do Amazonas, Setembro 2013, Amazonas - Brasil.

5. Herbert Rocha, Raimundo Barreto e Lucas Cordeiro. *Memory Management Test-Case Generation of C Programs Using Bounded Model Checking*. **In 13th edition of the International Conference on Software Engineering and Formal Methods (SEFM)**, Lecture Notes in Computer Science. Springer. 2015, York - United Kingdom.

Lista de publicações em colaboração com outros trabalhos:

1. Cunha E. G., Custodio, M. M., Rocha H. O., Barreto R. S. *Formal Verification of UML Sequence Diagrams in the Embedded Systems Context*. **In II Workshop de Sistemas Embarcados (WSE) - Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC)**, pp. 39-45. IEEE. 2011, Florianópolis - Brasil.

Apêndice A

Protocolo da Revisão Sistemática

Este apêndice apresenta as etapas do processo executado neste trabalho para a condução da revisão sistemática que envolveu três etapas baseadas em Mafra e Travassos (2006):

1. **Planejamento da Revisão**: os objetivos da pesquisa são listados e o protocolo da revisão é definido.
2. **Condução da Revisão**: nesta atividade, as fontes para a revisão sistemática são selecionadas, os estudos primários são identificados, selecionados e avaliados de acordo com os critérios de inclusão, exclusão, e de qualidade estabelecidos durante o protocolo da revisão.
3. **Análise dos Resultados**: os dados dos estudos são extraídos e sintetizados para análise e apresentação dos resultados.

As três etapas citadas anteriormente foram usadas juntamente com as diretrizes definidas por Biolchini *et al.* (2005), Mafra e Travassos (2006) e Kitchenham (2004). Entretanto, como o objetivo deste trabalho é realizar um estudo exploratório de caracterização de área pode-se dizer que esta revisão sistemática se caracteriza como uma quasi-sistemática (Travassos *et al.*, 2008), pois segue o mesmo processo da revisão sistemática e preserva o rigor e mesmo formalismo para as fases metodológicas de elaboração de protocolo e execução da revisão, mas sem a aplicação de uma meta-análise a princípio, que pode ser aplicada posteriormente.

A.1 Planejamento da Revisão Sistemática

Objetivo. O objetivo deste estudo foi esquematizado a partir da estrutura proposta pelo paradigma GQM (do inglês *Goal, Question, and Metric*) (Basili *et al.*, 1994):

Analisar	publicações científicas através de um estudo baseado em revisão sistemática.
Com o propósito de	identifica-las
Com relação as	vantagens e desvantagens da aplicação de métodos para a inferência de invariantes na verificação de código na linguagem de programação C
Do ponto de vista do	pesquisador
No contexto	acadêmico ou industrial para a verificação de código na linguagem de programação C.

Tabela A.1: Objetivo do estudo utilizando o paradigma GQM.

Formulação da Pergunta. Busca-se respostas para as seguintes perguntas:

- **Q1**: Quais são os métodos de inferência de invariantes para a verificação de código na linguagem de programação C?
  - **Q1.1**: Foi desenvolvido e está disponível alguma ferramenta para a aplicação do método?
  - **Q1.2**: O método proposto foi gerado a partir da integração com outros?
  - **Q1.3**: Qual é a estratégia (estática, dinâmica ou hibrida) de execução do método?
  - **Q1.4**: Quais foram os resultados positivos ou negativos da validação/experimentação do método?
    - **Q1.4.1**: Foi utilizado algum *benchmark* de programas em C para experimentação do método e este *benchmark* esta disponível?
  - **Q1.5**: Quais as limitações do método proposto?
  - **Q1.6**: Qual é a escalabilidade em termos de linhas de código do método proposto?

Escopo da Pesquisa. Para delinear o escopo da pesquisa foram estabelecidos critérios para garantir, de forma equilibrada, a viabilidade da execução (custo, esforço e tempo), acessibilidade aos dados e abrangência do estudo. A pesquisa dar-se-á a partir de bibliotecas digitais através das suas respectivas máquinas de busca e, quando os dados não estiverem disponíveis eletronicamente, através de consultas manuais.

Critérios de Seleção de Fontes. Para as bibliotecas digitais é desejado:

- Possuir máquina de busca que permita o uso de expressões lógicas ou mecanismo equivalente;
- Incluir em sua base publicações da área de exatas ou correlatas que possuam relação direta com o tema a ser pesquisado;
- As máquinas de busca deverão permitir a busca no texto completo das publicações.

Além disso, os mecanismos de busca utilizados devem garantir resultados únicos através da busca de um mesmo conjunto de palavras-chaves (*string* de busca). Quando isto não for possível, deve-se estudar e documentar uma forma de minimizar os potenciais efeitos colaterais desta limitação.

Métodos de Busca das Publicações. As fontes digitais foram acessadas via Web, através de expressões de busca pré-estabelecidas. A biblioteca digital consultada foi a Scopus, acessível em http://www.scopus.com. Segundo a editora Elsevier (2013c), a Scopus é uma das maiores bases de dados de resumos e citações da literatura de pesquisa revisões em pares com mais de 20,500 títulos de mais de 5,000 editoras internacionais. Dentre estas editoras pode-se citar: Springer (Springer, 2013); IEEE Xplore Digital Library (IEEE, 2013); ACM Digital Library (ACM, 2013); ScienceDirect/Elsevier (Elsevier, 2013a); Wiley Online Library (Sons, 2013); British Computer Society (Society, 2013) dentre outras. Ainda segundo a editora Elsevier (2013c), a Scopus tem aproximadamente 2 milhões de novas gravações adicionadas a cada ano com atualizações diárias.

String de Busca. A *string* de busca foi definida segundo o padrão PICO (do inglês *Population, Intervention, Comparison, Outcomes*) (Kitchenham e Charters, 2007), conforme a estrutura abaixo:

- População: Trabalhos publicados em conferências e periódicos que relacionam invariantes na verificação de código para as diferentes estruturas da linguagem de programação C;
- Intervenção: Inferência de invariantes relacionadas à verificação de código para as diferentes estruturas da linguagem de programação C;
- Comparação: Análise da completude das abordagens identificadas para a inferência de invariantes, no sentido de medir a cobertura das abordagens diante de métricas propostas e não sua eficácia ou desempenho, utilizando as questões de pesquisa (ver Seção A.1) como fonte para a extração de métricas;
- Resultados: A partir dos relatos de abordagens para inferência de invariantes, pretende-se verificar a cobertura que cada abordagem apresenta na manipulação das diferentes estruturas da linguagem de programação C para a inferência de invariantes.

Como este estudo representa um estudo de mapeamento/caracterização, a *string* de busca (para execução na biblioteca digital Scopus como mencionado anteriormente) foi definida de acordo com dois aspectos: População e Intervenção, como é apresentado na estrutura abaixo.

- População: publicações que fazem referências à inferência de invariantes (e sinônimos):
  - **Palavras-Chave**: “invariants tool” OR “program invariants” OR “invariants method” OR “inferring invariants” OR “invariants inference” OR “invariants detection” OR “loop invariants” OR “inferring loop invariants” OR “invariant generation” OR “invariant generator” OR “invariant assertions” OR “invariant detector” OR “invariant checker” OR “dynamic invariant inference” OR “static invariant inference” OR “simplifying loop invariant” OR “loop invariant generator” OR “automatic generation

of invariants" OR "invariant of functions" OR "invariant relations" OR "global invariants" OR "statically computed invariants" OR "module invariants"

- Intervenção: verificação de código (e sinônimos):
  - **Palavras-Chave**: "formal verification" OR "source code verification" OR "program verification" OR "local verification" OR "program analysis" OR "termination analysis" OR "symbol elimination" OR "predicate abstraction" OR "predicate" OR "abstract interpretation" OR "contract-based checkers" OR "proving verification conditions" OR "dynamic analysis" OR "static analysis" OR "identifying program properties" OR "theorem prover" OR "pre-conditions" OR "post-conditions" OR "pre- and post-conditions" OR "pre and post conditions"

A.1.1 Procedimentos de Seleção e Critérios

A estratégia de busca será aplicada por um pesquisador para identificar as publicações em potencial. A seleção das publicações dar-se-á em 4 etapas:

1. **Seleção e catalogação preliminar dos dados coletados**. A seleção preliminar das publicações será feita a partir da aplicação da expressão de busca às fontes selecionadas. Cada publicação será catalogada em um banco de dados criado especificamente para este fim e armazenada em um repositório para análise posterior;

2. **Seleção dos dados relevantes - [1º filtro]**. A seleção preliminar com o uso da expressão de busca não garante que todo o material coletado seja útil no contexto da pesquisa, pois a aplicação das expressões de busca é restrita ao aspecto sintático. Dessa forma, após a identificação das publicações através dos mecanismos de buscas, deve-se ler o título, os resumos/*abstracts* e as palavras-chave e analisá-los seguindo os critérios de inclusão e exclusão identificados a seguir. Neste momento, poder-se-ia classificar as publicações apenas quanto aos critérios de exclusão, entretanto, para facilitar a análise e reduzir o número de publicações das quais se possam ter dúvidas sobre sua aceitação, deve-se também classificá-las quanto aos critérios de inclusão. Devem ser excluídas as publicações contidas no conjunto preliminar que:
   - **CE1-01**: Não serão selecionadas publicações em que as palavras-chave da busca não apareçam no título, resumo e/ou texto da publicação (excluem-se os seguintes campos: as seções de agradecimentos, biografia dos autores, referências bibliográficas e anexas).
   - **CE1-02**: Não serão selecionadas publicações em que descrevam e/ou apresentam '*keynote speeches*', tutoriais, cursos e similares.
   - **CE1-03**: Não serão selecionadas publicações em que o contexto das palavras-chave utilizadas no artigo leve a crer que a publicação não cita uma abordagem para utilização de invariantes.

- **CE1-04**: Não serão selecionadas publicações em que o contexto das palavras-chave utilizadas no artigo leve a crer que a publicação não cita uma abordagem para verificação de código.
- **CE1-05**: Não serão selecionadas publicações que não cita uma abordagem para utilização de invariantes que esteja relacionada com verificação de código.

Podem ser incluídas apenas as publicações contidas no conjunto preliminar que:

- **CI1-01**: Podem ser selecionadas publicações em que o contexto das palavras-chave utilizadas no artigo leve a crer que a publicação cita uma abordagem para utilização de invariantes na verificação de código.
- **CI1-02**: Podem ser selecionadas publicações em que o contexto das palavras-chave utilizadas no artigo leve a crer que a publicação cita recomendações de melhoria na utilização de abordagens com invariantes na verificação de código.

3. **Seleção dos dados relevantes - [2º filtro]**. Apesar de limitar o universo de busca, o 1º filtro empregado não garante que todo o material coletado seja útil no contexto da pesquisa. Por isso, após a leitura na íntegra dos artigos selecionados no 1º filtro, deve-se verificar que as publicações respeitem os critérios abaixo. O objetivo deste 2º filtro é identificar artigos que relacionam invariantes e verificação de código.

   - **CS2 -INV -VER_COD**: Não devem ser selecionadas publicações que não contextualizem invariantes e não citam verificação de código.
   - **CS2 +INV -VER_COD**: Não devem ser selecionadas publicações que citam invariantes, mas não estejam contextualizadas em verificação de código.
   - **CS2 -INV +VER_COD**: Não devem ser selecionadas publicações que não citam invariantes, mas citam verificação de código.

   Dessa forma, todas as publicações devem respeitar o critério abaixo:

   - **CI2 +INV +VER_COD**: Só podem ser selecionadas publicações que contextualizem invariantes e citem verificação de código.

4. **Seleção dos dados relevantes - [3º filtro]**. No 3º filtro, o objetivo é identificar quais artigos mencionam programas na linguagem de programação C e que estejam relacionados com invariantes e verificação de código. Para isso, deve-se verificar que as publicações respeitem o critério abaixo:

   - **CS3 +INV +VER_COD -LANG_C**: Não devem ser selecionadas publicações que contextualizem invariantes, citem verificação de código e não mencionem (ou não provêm suporte direto [1]) a linguagem de programação C.

   Dessa forma, todas as publicações devem respeitar o critério abaixo:

[1] Suporte direto neste caso significa que o método proposto recebe como entrada um código na linguagem de programação C ou em um subconjunto da linguagem C.

- **CS3 +INV +VER_COD +LANG_C**: Só podem ser selecionadas publicações que contextualizem invariantes, citem verificação de código e que mencionem (ou provêm suporte direto) a linguagem de programação C.

A.1.2 Condução da Revisão Sistemática

As publicações recuperadas pelas máquinas de busca foram organizadas pelo gerenciador de referências bibliográficas Mendeley [2]. O Mendeley permitiu a indexação dos itens, ou seja, criou uma lista com os nomes e outras informações para pesquisas instantâneas. Ele ainda possui um rastreador automático de referências internas nos documentos, campos de pesquisa e filtros detalhados, e possibilita a anotação participativa e identificação de repetições.

O planejamento e a execução da revisão sistemática ocorreu no período de junho de 2012 a janeiro de 2013 e as publicações foram selecionadas de acordo com os critérios de inclusão e exclusão estabelecidos durante o protocolo da revisão. As referências das publicações que estavam inacessíveis pela Web foram excluídas, assim como as publicações repetidas.

A *string* de busca, relacionada na Seção A.1, foi executada na máquina de busca da biblioteca Scopus, como definido anteriormente. Contudo, vale ressaltar que a *string* de busca foi modificada algumas vezes (contendo um total de 7 versões), do início da aplicação da revisão sistemática até a sua última versão que é apresentada na Seção A.1. As modificações foram necessárias devido a dois fatores: o primeiro, um grande número de publicações retornadas pela máquina de busca, contabilizando em alguns momentos um total de mais de 3800 publicações; e o segundo, alguns artigos da lista de controle (lista de artigos já conhecidos e utilizados como base de referência) não estavam sendo retornados pelas máquinas de busca.

Com base nos resultados da máquina de busca, os filtros definidos (ver Seção A.1) foram aplicados nas publicações retornadas, desta forma foram selecionadas publicações que atendessem os critérios definidos para se efetuar a coleta de dados da publicação. A partir dos resultados da aplicação do 3º filtro, os seguintes dados foram extraídos:

- Dados da publicação: Título; Autor(es); Palavras-chave; Fonte de publicação; e Ano de publicação.
- Resumo da publicação: Uma breve descrição do estudo.
- Dados derivados das características de interesse declaradas nas questões de pesquisa:
  - Método(s) utilizados: métodos para inferência de invariantes relacionados à verificação de programas;
  - Ferramenta(s): caso tenha sido desenvolvido uma ferramenta para o método de inferência de invariantes;

[2] Ferramenta Mendeley Desktop versão 1.7.1. Mais informações no site: http://www.mendeley.com/download-mendeley-desktop/.

  - Impacto (positivo x negativo): indicação se o método para inferência de invariantes demonstrou resultados eficientes durante a sua experimentação;
  - Validação do método (sim ou não): se sim, qual tipo de estudo foi utilizado: *benchmarks*, códigos da indústria, estudo de caso e outros;
  - Limitações do método: por exemplo, quais estruturas de código o método utilizando invariantes não suporta;

- Dados para um melhor entendimento dos resultados:
  - Integração de métodos: se o método foi gerado a partir da integração com outro método;
  - Modo de aplicação do método: se o método para inferência de invariantes é executado de forma estática ou dinâmica;
  - Perspectivas futuras: questão de pesquisa sugerida para trabalhos futuros se houver alguma.

- Comentários adicionais do pesquisador.

Referências Bibliográficas

ACM. ACM Digital Library. Disponível em http://dl.acm.org/, 2013. Acessado em 18 de Março de 2013.

Ancourt C., Coelho F. e Irigoin F. A Modular Static Analysis Approach to Affine Loop Invariants Detection. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, páginas 3–16. Elsevier, 2010.

Arlat J., Aguera M., Amat L., Crouzet Y., Fabre J.-C., Laprie J.-C., Martins E. e Powell D. Fault Injection for Dependability Validation: A Methodology and Some Applications. In *IEEE Transactions on Software Engineering (TSE)*, páginas 166–182. IEEE, 1990.

Baier C. e Katoen J.-P. *Principles of Model Checking*. MIT Press, 2008.

Ball T. The Concept of Dynamic Analysis. In *7th European Software Engineering Conference (ESE)*, páginas 216–234. Springer-Verlag, 1999.

Ball T., Naik M. e Rajamani S. K. From symptom to cause: Localizing errors in counterexample traces. In *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, páginas 97–105. ACM, 2003.

Barmi Z. A., Ebrahimi A. H. e Feldt R. Alignment of Requirements Specification and Testing: A Systematic Mapping Study. In *4th International IEEE Conference on Software Testing, Verification and Validation (ICST)*, páginas 476–485, 2011.

Barnett M., Grieskamp W., Gurevich Y., Schulte W., Tillmann N. e Veanes M. Scenario-oriented Modeling in AsmL and its Instrumentation for Testing. In *Proceedings 2nd International Workshop on Scenarios and State Machines (SCESM)*. Springer, 2003.

Basili V. R., Caldiera G. e Rombach H. D. The experience factory. In *Encyclopedia of Software Engineering*, páginas 469–476. Wiley, 1994.

Baudin P., Filliâtre J. C., Marché C., Monate B., Moy Y. e Prevosto V. ACSL: ANSI/ISO C Specification Language. In *CEA LIST and INRIA*, 2009. Disponível em http://frama-c.cea.fr/acsl.html. Acessado em 13 de Janeiro de 2011.

Beer I., Ben-David S., Chockler H., Orni A. e Trefler R. Explaining Counterexamples Using Causality. In *Computer Aided Verification (CAV)*, páginas 94–108. Springer, 2009.

Bell D. *Software Engineering for Students*. Addison-Wesley, 2005. ISBN 9780321261274.

Bensalem S. e Lakhnech Y. Automatic Generation of Invariants. In *Formal Methods in System Design (FMSD)*, páginas 75–92. Kluwer Academic Publishers, 1999.

Berghofer S. Program Extraction in Simply-Typed Higher Order Logic. In *Types for Proofs and Programs, Second International Workshop (TYPES)*, páginas 21–38. Springer, 2002.

Beyer D. Competition on Software Verification (SV-COMP). In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS)*, páginas 504–524. Springer, 2012.

Beyer D. Second competition on Software Verification - (Summary of SV-COMP 2013). In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, páginas 594–609. Springer, 2013.

Beyer D. Status Report on Software Verification (Competition Summary SV-COMP 2014). In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, páginas 373–388,. Springer, 2014.

Beyer D. 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.

Beyer D., Dangl M., Dietsch D., Heizmann M. e Stahlbauer A. Verification Witnesses - Error-Witness Automata. Disponível em http://www.sosy-lab.org/ dbeyer/cpa-witnesses/, 2015a. Acessado em 20 de Abril de 2015.

Beyer D., Dangl M. e Wendler P. Boosting k-Induction with Continuously-Refined Invariants. Disponível em http://www.sosy-lab.org/~dbeyer/cpa-k-induction/, 2015b. Acessado em 15 de Maio de 2015.

Beyer D., Dangl M. e Wendler P. Combining k-Induction with Continuously-Refined Invariants. In *Computing Research Repository (CoRR)*, volume abs/1502.00096, 2015c. Disponível em http://arxiv.org/abs/1502.00096. Acessado em 20 de Abril de 2015.

Beyer D., Henzinger T. A., Jhala R. e Majumdar R. The software model checker Blast: Applications to software engineering. In *International Journal on Software Tools for Technology Transfer (STTT)*, páginas 505–525. Springer, 2007a.

Beyer D., Henzinger T. A., Majumdar R. e Rybalchenko A. Invariant synthesis for combined theories. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, páginas 378–394. Springer-Verlag, 2007b.

Beyer D., Henzinger T. A., Majumdar R. e Rybalchenko A. Path invariants. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, páginas 300–309. ACM, 2007c.

Beyer D. e Keremoglu M. E. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification (CAV)*, páginas 184–190. Springer-Verlag, 2011.

Bi Z., Shan M. e Tian X. Automatic generation of non-linear loop invariant. In *Journal of Computational Information Systems (JCIS)*, páginas 3335–3344. Binary Information Press, 2010.

Biere A. Bounded Model Checking. In *Handbook of Satisfiability*, páginas 457–481. IOS Press, 2009. ISBN 978-1-58603-929-5.

Biolchini J., Mian P. G. e Natali A. C. C. Systematic Review in Software Engineering. Technical Report RT-ES 679/05, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, May 2005.

Brereton P., Kitchenham B. A., Budgen D., Turner M. e Khalil M. Lessons from Applying the Systematic Literature Review Process Within the Software Engineering Domain. In *Journal of Systems and Software*, páginas 571–583. Elsevier, 2007.

Bruttomesso R., Pek E., Sharygina N. e Tsitovich A. The OpenSMT Solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, páginas 150–153. Springer-Verlag, 2010.

Büchi J. R. On a Decision Method in Restricted Second Order Arithmetic. In Nagel E., Suppes P. e Tarski A., editors, *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science (LMPS'60)*, páginas 1–11. Stanford University Press, 1962.

Cadar C. e Engler D. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of the 12th International Conference on Model Checking Software (SPIN)*, páginas 2–23. Springer-Verlag, 2005.

Canet G., Cuoq P. e Monate B. A Value Analysis for C Programs. In *Source Code Analysis and Manipulation (SCAM)*, páginas 123–124. IEEE, 2009.

Cao Y. e Zhu Q. Finding loop invariants based on Wu's characteristic set method. In *Information Technology Journal*, páginas 349–353. Asian Network for Scientific Information, 2010.

Caspi P., Pilaud D., Halbwachs N. e Plaice J. A. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, páginas 178–188. ACM, 1987.

Cassandras C. G. e Lafortune S. *Introduction to Discrete Event Systems*. SpringerLink Engineering. Springer, 2008.

Chechik M. e Gurfinkel A. A framework for counterexample generation and exploration. In *8th International Conference on Fundamental Approaches to Software Engineering (FASE), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, páginas 220–236. Springer, 2005.

Chen S., Li Z., Song X. e Li M. An iterative method for generating loop invariants. In *Proceedings of the 5th International Frontiers in Algorithmics Workshop (FAW), and 7th International Conference on Algorithmic Aspects in Information and Management (AAIM)*, páginas 264–274. Springer-Verlag, 2011.

Chen Y., Xia B., Yang L. e Zhan N. Generating polynomial invariants with DISCOVERER and QEPCAD. In *Formal Methods and Hybrid Real-Time Systems (FMHRTS)*, páginas 67–82. Springer-Verlag, 2007.

Clarke E., Grumberg O. e Peled D. *Model Checking*. MIT Press, 1999.

Clarke E., Fehnker A., Han Z., Krogh B., Stursberg O. e Theobald M. Verification of hybrid systems based on counterexample-guided abstraction refinement. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, páginas 192–207. Springer-Verlag, 2003.

Clarke E., Kroening D. e Lerda F. A Tool for Checking ANSI-C Programs. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, páginas 168–176. Springer, 2004a.

Clarke E., Kroening D., Sharygina N. e Yorav K. Predicate Abstraction of ANSI-C Programs Using SAT. In *Formal Methods in System Design (FMSD)*, páginas 105–127. Kluwer Academic Publishers, 2004b.

Clarke E., Kroening D., Sharygina N. e Yorav K. Satabs: Sat-based predicate abstraction for ansi-c. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, páginas 570–574. Springer-Verlag, 2005.

Clarke E. M. 25 years of model checking. chapter The Birth of Model Checking, páginas 1–26. Springer-Verlag, 2008. ISBN 978-3-540-69849-4.

Clarke E. M. e Wing J. M. Formal Methods: State of the Art and Future Directions. In *ACM Computing Surveys (CSUR)*, páginas 626–643. ACM, 1996.

Clause J., Li W. e Orso A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA)*, páginas 196–206. ACM, 2007.

Clause J. e Orso A. LEAKPOINT: pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, páginas 515–524. ACM, 2010.

Colón M., Sankaranarayanan S. e Sipma H. Linear invariant generation using non-linear constraint solving. In *15th International Conference Computer Aided Verification (CAV)*, páginas 420–432. Springer, 2003.

Comar C., Kanig J. e Moy Y. Integrating formal program verification with testing. In *Embedded Real-Time Software and Systems Congress (ERTS)*, 2012.

Copty F., Irron A., Weissberg O., Kropp N. e Kamhi G. Efficient debugging in a formal verification environment. In *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference (CHARME)*, páginas 275–292. Springer, 2001.

Cordeiro L. e Fischer B. Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, páginas 331–340. ACM, 2011.

Cordeiro L., Fischer B. e Marques-Silva J. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In *Automated Software Engineering (ASE)*, páginas 137–148. IEEE, 2009.

Cordeiro L., Fischer B. e Marques-Silva J. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In *IEEE Transactions on Software Engineering (TSE)*, páginas 957–974. IEEE, 2012a.

Cordeiro L. C., Morse J., Nicole D. e Fischer B. Context-Bounded Model Checking with ESBMC 1.17 - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS. Proceedings*, páginas 534–537. Springer, 2012b.

Correnson L., Cuoq P., Puccetti A. e Signoles J. *Frama-C User Manual. Disponível em http://frama-c.com/download/user-manual-Carbon-20110201.pdf*. CEA LIST, 2009. Acessado em 13 de Janeiro de 2011.

Cousot P. Abstract interpretation. In *ACM Computing Surveys (CSUR)*, páginas 324–328. ACM, 1996.

Cousot P. Abstract interpretation based formal methods and future challenges. In *Informatics - 10 Years Back. 10 Years Ahead.*, páginas 138–156. Springer-Verlag, 2001.

Cousot P. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. In *Theoretical Computer Science*, páginas 47–103. Elsevier Science Publishers Ltd., 2002.

Cousot P. e Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL)*, páginas 238–252. ACM, 1977.

Cousot P. e Cousot R. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, páginas 269–282. ACM, 1979.

Cousot P. e Cousot R. *A gentle introduction to formal verification of computer systems by abstract interpretation*, páginas 1–29. NATO Science Series III: Computer and Systems Sciences. IOS Press, 2010.

Cousot P. e Halbwachs N. 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)*, páginas 84–96. ACM, 1978.

Cox D. A., Little J. e O'Shea D. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer Publishing Company, Incorporated, 3rd edition, 2010. ISBN 1441922571, 9781441922571.

C.Wang, Yang Z., Ivancic F. e Gupta A. Whodunit? causal analysis for counterexamples. In *4th International Symposium Automated Technology for Verification and Analysis (ATVA)*, páginas 82–95. Springer, 2006.

Daniel Kroening. Bounded Model Checking for ANSI-C. Disponível em http://www.cprover.org/cbmc/doc/manual.pdf, 2012. Acessado em 17 de Março de 2012.

De Moura L. e Bjørner N. Z3: an efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, páginas 337–340. Springer-Verlag, 2008.

Delahaye M., Kosmatov N. e Signoles J. Common Specification Language for Static and Dynamic Analysis of C Programs. In *28th Annual ACM Symposium on Applied Computing (SAC)*, páginas 1230–1235. ACM, 2013.

Delamaro M., Maldonado J. e Jino M. *Introdução ao teste de software*. Elsevier, 2007. ISBN 9788535226348.

Desoli G., Mateev N., Duesterwald E., Faraboschi P. e Fisher J. A. Deli: a new run-time control point. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, páginas 257–268. IEEE Computer Society Press, 2002.

Dhurjati D., Kowshik S., Adve V. e Lattner C. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems (LCTES)*, páginas 69–80. ACM, 2003.

Donaldson A. F., Haller L. e Kroening D. Strengthening induction-based race checking with lightweight static analysis. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, páginas 169–183. Springer-Verlag, 2011.

Donaldson A. F., Kroening D. e Ruemmer P. Automatic Analysis of Scratch-pad Memory Code for Heterogeneous Multicore Processors. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, páginas 280–295. Springer, 2010.

Dong Y., Ramakrishnan C. R. e Smolka S. A. Model checking and evidence exploration. In *IEEE Conference and Workshops on Engineering Computer Based Systems (ECBSW)*, páginas 214–223. IEEE, 2003.

D'Silva V., Kroening D. e Weissenbacher G. A survey of automated techniques for formal software verification. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, páginas 1165–1178. IEEE, 2008a.

D'Silva V., Kroening D. e Weissenbacher G. A survey of automated techniques for formal software verification. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, páginas 1165–1178. IEEE, 2008b.

Dudka K., Peringer P. e Vojnar T. Predator: A Shape Analyzer Based on Symbolic Memory Graphs. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, páginas 412–414. Springer Berlin Heidelberg, 2014.

Eén N. e Sörensson N. Temporal Induction by Incremental SAT Solving. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, páginas 543–560. Elsevier, 2003.

Elsevier. Digital Library ScienceDirect. Disponível em http://www.sciencedirect.com/, 2013a. Acessado em 18 de Março de 2013.

Elsevier. Scopus Library. Disponível em http://www.scopus.com, 2013b. Acessado em 20 de Fevereiro de 2013.

Elsevier. What does Scopus cover?. Disponível em http://www.info.sciverse.com/scopus/scopus-in-detail/facts, 2013c. Acessado em 20 de Fevereiro de 2013.

Ernst M. D., Cockrell J., Griswold W. G. e Notkin D. Dynamically discovering likely program invariants to support program evolution. In *IEEE Transactions on Software Engineering (TSE)*, páginas 99–123. IEEE, 2001.

Ernst M. D. Summary of Dynamically Discovering Likely Program Invariants. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, páginas 540–544. IEEE, 2001.

Ernst M. D. Static and dynamic analysis: Synergy and duality. In *ICSE Workshop on Dynamic Analysis (WODA)*, páginas 24–27, 2003.

Ernst M. D., Perkins J. H., Guo P. J., McCamant S., Pacheco C., Tschantz M. S. e Xiao C. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, páginas 35–45. Elsevier North-Holland, Inc., 2007.

Eureka. EUREKA Benchmark. Disponível em http://www.ai-lab.it/eureka/bmc.html, 2012. Acessado em 17 de Março de 2012.

Feautrier P. COMPilation of embedded SYStems (COMPSYS). Disponível em http://www.ens-lyon.fr/LIP/COMPSYS/tools/, 2015. Acessado em 20 de Janeiro de 2015.

Feautrier P. e Gonnord L. Accelerated Invariant Generation for C Programs with Aspic and C2fsm. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, páginas 3–13. Elsevier Science Publishers B. V., 2010.

Feret J. The arithmetic-geometric progression abstract domain. In *6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, páginas 42–58. Springer-Verlag, 2005a.

Feret J. Numerical Abstract Domains for Digital Filters. In *International workshop on Numerical & Symbolic Abstract Domains (NSAD 2005)*. Maison Des Polytechniciens, 2005b.

Fisteus J. A. *Definición de Un Modelo Para La Verificación Formal de Procesos de Negocio*. PhD thesis, Universidad Carlos III de Madrid, Departamento De Ingeniería Telemática, Setembro 2005.

Flanagan C. e Qadeer S. Predicate Abstraction for Software Verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, páginas 191–202. ACM, 2002.

Fouladgar H., Minaei-Bidgoli B. e Parvin H. On possibility of conditional invariant detection. In *Proceedings of the 15th international conference on Knowledge-based and intelligent information and engineering systems - Volume Part II*, páginas 214–224. Springer-Verlag, 2011.

Ghardallou W. Using invariant relations in the termination analysis of while loops. In *Proceedings of the International Conference on Software Engineering (ICSE)*, páginas 1519–1522. IEEE, 2012.

Godefroid P. Test generation using symbolic execution. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, páginas 24–33. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

Gonnord L. e Halbwachs N. Combining Widening and Acceleration in Linear Relation Analysis. In *Static Analysis Symposium (SAS)*, páginas 144–160. Springer, 2006.

Griesmayer A., Staber S. e Bloem R. Automated fault localization for c programs. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, páginas 95–111. Elsevier Science Publishers B. V., 2007.

Griesmayer A., Staber S. e Bloem R. Fault localization using a model checker. In *Software Testing Verification & Reliability*, páginas 149–173, 2010.

Groce A. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference (TACAS), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, páginas 108–122. Springer, 2004.

Groce A. e Joshi R. Extending Model Checking with Dynamic Analysis. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, páginas 142–156. Springer, 2008.

Groce A. e Kroening D. Making the most of bmc counterexamples. In *Electronic Notes in Theoretical Computer Science (ENTCS)*, páginas 67–81. Elsevier Science Publishers B. V., 2005.

Gulwani S., Srivastava S. e Venkatesan R. Constraint-based invariant inference over predicate abstraction. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, páginas 120–135. Springer-Verlag, 2009.

Gupta A., Majumdar R. e Rybalchenko A. From Tests to Proofs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS): Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, páginas 262–276. Springer-Verlag, 2009.

Gupta A. e Rybalchenko A. InvGen: An Efficient Invariant Generator. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, páginas 634–640. Springer-Verlag, 2009.

Gupta A. e Rybalchenko A. Test Suite of InvGen tool. Disponível em http://www.tcs.tifr.res.in/~agupta/invgen/invgen-c-examples.tar.gz, 2015. Acessado em 18 de Janeiro de 2015.

Hagen G. e Tinelli C. 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)*, páginas 109–117. IEEE, 2008.

Halpern J. Y. e Pearl J. Causes and Explanations: A Structural-Model Approach, Part I: Causes. páginas 843–887. British J. Philos. Sci, 2000.

Havelund K. Java PathFinder, A Translator from Java to Promela. In *SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking (SPIN)*, página 152, 1999.

Hoare C. A. R. An axiomatic basis for computer programming. In *Communications of the ACM (CACM)*, páginas 576–580. ACM, 1969.

Hoder K., Kovács L. e Voronkov A. Interpolation and symbol elimination in vampire. In *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR)*, páginas 188–195. Springer-Verlag, 2010.

Hoder K., Kovács L. e Voronkov A. Case studies on invariant generation using a saturation theorem prover. In *Proceedings of the 10th Mexican international conference on Advances in Artificial Intelligence - Volume Part I*, páginas 1–15. Springer-Verlag, 2011a.

Hoder K., Kovács L. e Voronkov A. Invariant generation in vampire. In *Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems (TACAS): part of the joint European conferences on theory and practice of software (ETAPS)*, páginas 60–64. Springer-Verlag, 2011b.

Holik L., Hruska M., Lengal O., Rogalewicz A., Simacek J. e Vojnar T. Forester. Disponível em http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/, 2015. Acessado em 05 de Janeiro de 2015.

Holzer A., Schallhart C., Tautschnig M. e Veith H. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *International Conference on Computer Aided Verification (CAV)*, páginas 209–213. Springer, 2008.

Holzmann G. J. The Model Checker SPIN. In *IEEE Transactions on Software Engineering*, páginas 279–295. IEEE, 1997.

Howden W. *Functional program testing and analysis*. Mac Graw-Hill series in software engineering and technology. McGraw-Hill Ryerson, Limited, 1987. ISBN 9780070305502.

IBM. Cell BE resource center. Disponível em http://www.ibm.com/developerworks/power/cell/, 2013. Acessado em 20 de Fevereiro de 2013.

IEEE. IEEE Xplore Digital Library. Disponível em http://ieeexplore.ieee.org, 2013. Acessado em 18 de Março de 2013.

INRIA. Jessie Plugin Tutorial. Disponível em http://frama-c.com/download/jessie-tutorial-Carbon-20101201-beta1.pdf, 2010. Acessado em 17 de Março de 2011.

Jain H., Ivančić F., Gupta A., Shlyakhter I. e Wang C. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, páginas 137–151. Springer-Verlag, 2006.

Jhala R. e Majumdar R. Software model checking. In *ACM Computing Surveys (CSUR)*, páginas 1–54. ACM, 2009.

Jia Y. e Harman M. An analysis and survey of the development of mutation testing. In *IEEE Transactions on Software Engineering (TSE)*, páginas 649–678. IEEE, 2010.

Jin H., Ravi K. e Somenzi F. Fate and free will in error traces. In *International Journal on Software Tools for Technology Transfer (STTT) - Special section on tools and algorithms for the construction and analysis of systems*, páginas 102–116. Springer-Verlag, 2004.

Kahlon V., Sankaranarayanan S. e Gupta A. Semantic Reduction of Thread Interleavings in Concurrent Programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS): Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, páginas 124–138. Springer-Verlag, 2009.

Kahsai T., Gurfinkel A. e Navas J. A. SeaHorn - A software verification tool. Disponível em https://bitbucket.org/lememta/seahorn/wiki/Home, 2015. Acessado em 05 de Janeiro de 2015.

Kebrt M. e Sery O. Unitcheck: Unit testing and model checking combined. In *International Symposium Automated Technology for Verification and Analysis (ATVA)*, páginas 97–103. Springer, 2009.

Kelly W., Pugh W., Rosser E. e Shpeisman T. Transitive closure of infinite graphs and its applications. *International Journal of Parallel Programming*, páginas 579–598, 1996.

Kerbrat A., Jéron T. e Groz R. Automated test generation from SDL specifications. In *SDL Forum*, páginas 135–152, 1999.

King J. C. Symbolic execution and program testing. In *Communications of the ACM (CACM)*, páginas 385–394. ACM, 1976.

Kirchner F. FramaC Software Analysis. Disponível em http://frama-c.com/, 2012. Acessado em 17 de Março de 2012.

Kitchenham B. e Brereton P. A systematic review of systematic review process research in software engineering. In *Information & Software Technology*, páginas 2049–2075. Butterworth-Heinemann, 2013.

Kitchenham B. e Charters S. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

Kitchenham B., Pearl Brereton O., Budgen D., Turner M., Bailey J. e Linkman S. Systematic Literature Reviews in Software Engineering - A Systematic Literature Review. In *Information and Software Technology*, páginas 7–15. Butterworth-Heinemann, 2009.

Kitchenham B. A. Procedures for performing systematic reviews. Technical Report Technical Report TR/SE-0401, Keele University and NICTA, 2004.

Kitchenham B. A., Dyba T. e Jorgensen M. Evidence-Based Software Engineering. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, páginas 273–281. IEEE, 2004.

Kong S., Jung Y., David C., Wang B.-Y. e Yi K. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems (APLAS)*, páginas 328–343. Springer-Verlag, 2010.

Labed Jilani L., Louhichi A., Mraihi O. e Mili A. Invariant relations, invariant functions, and loop functions. In *Innovations in Systems and Software Engineering*, páginas 195–212. Springer-Verlag, 2012.

Lahiri S. K., Bryant R. E. e Cook B. A symbolic approach to predicate abstraction. In *15th International Conference Computer Aided Verification (CAV)*, páginas 141–153. Springer, 2003.

Lattner C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, 2002. Disponível em http://llvm.cs.uiuc.edu. Acessado em 10 de Maio de 2015.

Lattner C. The LLVM Compiler Infrastructure. Disponível em http://llvm.org, 2015. Acessado em 18 de Março de 2015.

Lawson C. L. e Hanson R. J. *Solving least squares problems*, volume 15 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), 1995. ISBN 0-89871-356-0.

Lo R. W., Levitt K. N. e Olsson R. A. Validation of array accesses: Integration of flow analysis and program verification techniques. In *Software Testing Verification & Reliability*, páginas 201–227. John Wiley & Sons, 1997.

Löwe S., Mandrykin M. e Wendler P. Cpachecker with sequential combination of explicit-value analyses and predicate analyses. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, páginas 392–394. Springer Berlin Heidelberg, 2014.

Maebe J., Ronsse M. e De Bosschere K. Precise detection of memory leaks. In *Proceedings of the Second International Workshop on Dynamic Analysis (WODA)*, páginas 25–31, 2004.

Mafra S. N. e Travassos G. H. Estudos Primários e Secundários Apoiando a Busca por Evidência em Engenharia de Software. Disponível em www.cos.ufrj.br/uploadfiles/1149103120.pdf. Technical Report RT-ES 679/05, PESC - COPPP/UFRJ, Rio de Janeiro, RJ, Brasil, 2006. Acessado em 20 de Junho de 2011.

Magdaleno A. M. a., Werner C. M. L. e Araujo R. M. d. Reconciling software development models: A quasi-systematic review. *Journal of Systems and Software*, páginas 351–369, 2012.

Maisonneuve V., Hermant O. e Irigoin F. 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*, páginas 17–31. Elsevier, 2014.

Manna Z. e Pnueli A. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., 1995. ISBN 0-387-94459-1.

Marre B. e Arnould A. Test Sequences Generation from LUSTRE Descriptions: GATEL. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, páginas 93–111. IEEE Computer Society, 2000.

McMillan K. L. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *14th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, páginas 413–427. Springer, 2008.

Merz F., Falke S. e Sinz C. 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)*, páginas 146–161. Springer-Verlag, 2012.

Miné A. The octagon abstract domain. In *Higher-Order and Symbolic Computation*, páginas 31–100. Kluwer Academic Publishers, 2006.

Morse J., Cordeiro L., Nicole D. e Fischer B. Model checking LTL properties over ANSI-C programs with bounded traces. In *Software & Systems Modeling*, páginas 65–81. Springer Berlin Heidelberg, 2015.

Morse J., Cordeiro L. C., Nicole D. e Fischer B. Handling Unbounded Loops with ESBMC 1.20 - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS. Proceedings*, páginas 619–622. Springer, 2013.

Morse J., Ramalho M., Cordeiro L. C., Nicole D. e Fischer B. ESBMC 1.22 - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS. Proceedings*, páginas 405–407. Springer, 2014.

Mraihi O., Louhichi A., Jilani L., Desharnais J. e Mili A. Invariant assertions, invariant relations and invariant functions. Technical Report Technical Report, New Jersey Institute of Technology, 2012.

MRTC. WCET Benchmarks. Mälardalen Real-Time Research Center. Disponível em http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2012. Acessado em 17 de Março de 2012.

Musuvathi M. e Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, páginas 446–455. ACM, 2007.

Myers G., Sandler C. e Badgett T. *The Art of Software Testing*. ITPro collection. Wiley, 2011. ISBN 9781118133156.

Nagarakatte S., Zhao J., Martin M. M. e Zdancewic S. CETS: compiler enforced temporal safety for C. In *Proceedings of the International Symposium on Memory Management (SMM)*, páginas 31–40. ACM, 2010.

NEC Laboratories America, Inc. NECLA Verification Benchmarks. Disponível em http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php, 2012. Acessado em 17 de Março de 2012.

Nethercote N. Dynamic binary analysis and instrumentation. Technical Report UCAM-CL-TR-606, University of Cambridge, Computer Laboratory, nov 2004. Disponível em http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf. Acessado em 20 de Junho de 2011.

Nethercote N. e Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, páginas 89–100. ACM, 2007.

Nguyen T., Kapur D., Weimer W. e Forrest S. Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE)*, páginas 683–693. IEEE Press, 2012.

Nimmer J. W. e Ernst M. D. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of the First Workshop on Runtime Verification (RV) in International Conference Computer Aided Verification (CAV)*, páginas 255 – 276. Elsevier Science, 2001.

Nimmer J. W. e Ernst M. D. Invariant Inference for Static Checking. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT/FSE)*, páginas 11–20. ACM, 2002.

Parsa S., Minaei B., Daryabari M. e Parvin H. New efficient techniques for dynamic detection of likely invariants. In *Proceedings of the 10th international conference on Adaptive and natural computing algorithms - Volume Part I*, páginas 381–390. Springer-Verlag, 2011.

Pasareanu C. S., Pelánek R. e Visser W. Concrete Model Checking with Abstract Matching and Refinement. In *17th International Conference on Computer Aided Verification (CAV)*, páginas 52–66. Springer, 2005.

Perkins J. H. e Ernst M. D. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)*, páginas 23–32. ACM, 2004.

Pressman R. S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. ISBN 0072496681.

Ramalho M., Cordeiro L., Cavalcante A. e Jr V. L. Verificação Baseada em Indução Matemática de Programas C/C++. In *Brazilian Symposium on Computing Systems Engineering (SBESC)*, páginas 1–6. SBC, 2013.

Research M. An Efficient Theorem Prover. Disponível em http://research.microsoft.com/en-us/um/redmond/projects/z3/, 2012. Acessado em 17 de Março de 2012.

Riacs W. V., Groce A., Groce A., Visser W. e Visser W. What Went Wrong: Explaining Counterexamples. In *SPIN Workshop on Model Checking of Software (SPIN)*, páginas 121–135. Springer-Verlag, 2002.

Rocha H., Barreto R. e Cordeiro L. Caracterização do Estado da Arte sobre Invariantes para Inferência de Propriedades na Verificação de Software. Technical Report RT-GISE-005, Universidade Federal do Amazonas, Instituto de Computação, Laboratório de Sistemas Embarcados, Setembro 2013.

Rocha H., Barreto R. e Cordeiro L. Memory Management Test-Case Generation of C Programs using Bounded Model Checking. In *To appear in 13th edition of the International Conference on Software Engineering and Formal Methods (SEFM)*. Springer, 2015a.

Rocha H., Barreto R., Cordeiro L. e Neto A. D. Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples. In *9th International Conference on Integrated Formal Methods (IFM)*, páginas 128–142. Springer, 2012.

Rocha H., Ismail H., Cordeiro L. C. e Barreto R. S. Model Checking C Programs with Loops via k-Induction and Invariants. *Computing Research Repository (CoRR)*, abs/1502.02327, 2015b. Disponível em http://arxiv.org/abs/1502.02327. Acessado em 20 de Maio de 2015.

Rocha H. O., Cordeiro L., Barreto R. e Netto J. Exploiting Safety Properties in Bounded Model Checking for Test Cases Generation of C Programs. In *4th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, páginas 121–130. SBC, 2010.

Rocha H. O. Verificação e Comprovação de Erros em Códigos C usando Bounded Model Checker. Master's thesis, Universidade Federal do Amazonas, Instituto de Computação - Programa de Pós-Graduação em Informática, Fevereiro 2011.

Sagdeo P., Athavale V., Kowshik S. e Vasudevan S. Precis: Inferring invariants using program path guided clustering. In *26th International Conference on Automated Software Engineering (ASE)*, páginas 532–535. IEEE, 2011.

Sankaranarayanan S., Sipma H. B. e Manna Z. Non-linear loop invariant generation using Gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, páginas 318–329. ACM, 2004.

Schmitt M., Ebner M. e Grabowski J. Test Generation with Autolink and TestComposer. In *Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM)*, páginas 26–28, 2000.

Scott J., Lee L. H., Arends J. e Moyer B. Designing the Low-Power M*CORE Architecture. In *Power Driven Microarchitecture Workshop*, páginas 145–150, 1998.

Seward J. e Nethercote N. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)*, páginas 17–30. USENIX Association, 2005.

Shacham O., Sagiv M. e Schuster A. Scaling Model Checking of Dataraces Using Dynamic Information. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, páginas 107–118. ACM, 2005.

Sharma R., Dillig I., Dillig T. e Aiken A. Simplifying loop invariant generation using splitter predicates. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, páginas 703–719. Springer-Verlag, 2011.

Shen S., Qin Y. e Li S. A faster counterexample minization algorithm based on refutation analysis. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, páginas 672–677. IEEE, 2005.

Sherer S. A. A Cost-Effective Approach to Testing. In *IEEE Software*, páginas 34–40. IEEE Computer Society, 1991.

SIR Project. Software-artifact Infrastructure Repository. Disponível em http://sir.unl.edu/portal/index.php, 2012. Acessado em 17 de Março de 2012.

SNU. SNU Real-Time Benchmarks. Disponível em http://www.cprover.org/goto-cc/examples/snu.html, 2012. Acessado em 17 de Março de 2012.

Society B. C. The Chartered Institute for IT, Enabling the information society. Disponível em http://onlinelibrary.wiley.com/, 2013. Acessado em 18 de Março de 2013.

Sons J. W. . Wiley Online Library. Disponível em http://onlinelibrary.wiley.com/, 2013. Acessado em 18 de Março de 2013.

Springer. Springer Link, Part of Springer Science+Business Media. Disponível em http://link.springer.com/, 2013. Acessado em 18 de Março.

Srivastava S. e Gulwani S. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, páginas 223–234. ACM, 2009.

Srivastava S., Gulwani S. e Foster J. S. VS3: SMT Solvers for Program Verification. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, páginas 702–708. Springer-Verlag, 2009.

Staber S. e Bloem R. Fault localization and correction with QBF. In *10th International Conference Theory and Applications of Satisfiability Testing (SAT)*, páginas 355–368. Springer, 2007.

Ströder T., Giesl J., Brockschmidt M., Frohn F., Fuhs C., Hensel J. e Schneider-Kamp P. Proving Termination and Memory Safety for Programs with Pointer Arithmetic. In *International Journal of Computing Academic Research (IJCAR)*, páginas 208–223. Springer, 2014.

Suelflow A., Fey G., Bloem R. e Drechsler R. Using unsatisfiable cores to debug multiple design errors. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, páginas 77–82. ACM, 2008.

Tip F. A survey of program slicing techniques. In *Journal Programming Languages*, páginas 121–189. Centre for Mathematics and Computer Science (CWI), 1995.

Travassos G., dos Santos P., Neto P. e Biolchini J. An Environment to Support Large Scale Experimentation in Software Engineering. In *Engineering of Complex Computer Systems, 13th IEEE International Conference on*, páginas 193–202. IEEE, 2008.

Tretmans G. J. e Brinksma H. Torx: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*, páginas 31–43, 2003.

Unterkalmsteiner M., Gorschek T., Islam A. K. M. M., Cheng C. K., Permadi R. B. e Feldt R. Evaluation and Measurement of Software Process Improvement - A Systematic Literature Review. *IEEE Transactions on Software Engineering (TSE)*, páginas 398–424, 2012.

Verner J. M., Brereton O. P., Kitchenham B. A., Turner M. e Niazi M. Risks and risk mitigation in global software development: A tertiary study. In *Information & Software Technology*, páginas 54–78, 2014.

Williams N., Marre B., Mouy P. e Roger M. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *5th European Dependable Computing Conference (EDCC)*, páginas 281–292. Springer, 2005a.

Williams N., Marre B., Mouy P. e Roger M. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *European Dependable Computing Conference (EDCC)*, páginas 281–292. Springer, 2005b.

Wong W. E., Debroy V., Surampudi A., Kim H. e Siok M. F. Recent catastrophic accidents: Investigating how software was responsible. In *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, páginas 14–22. IEEE Computer Society, 2010.

Yeolekar A., Unadkat D., Agarwal V., Kumar S. e Venkatesh R. Scaling Model Checking for Test Generation Using Dynamic Inference. In *International Conference on Software Testing, Verification and Validation (ICST)*, páginas 184–191. IEEE, 2013.