GERAÇÃO DE CASOS DE TESTE USANDO BOUNDED MODEL CHECKING

RAFAEL MENEZES

GERAÇÃO DE CASOS DE TESTE USANDO BOUNDED MODEL CHECKING

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Instituto de Ciências Exatas da Universidade Federal do Amazonas como requisito para a obtenção do grau de Mestre em Informática.

Orientador: Lucas Carvalho Cordeiro
Coorientador: Herbert Oliveira Rocha

Manaus - AM
Junho de 2021

Ficha Catalográfica

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

M543g

Menezes, Rafael Sá
Geração de casos de teste usando Bounded Model Checking / Rafael Sá Menezes . 2021
71 f.: il. color; 31 cm.

Orientador: Lucas Carvalho Cordeiro
Coorientador: Herbert Oliveira Rocha
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. bounded model checking. 2. caching. 3. software. 4. verification. 5. testcase generation. I. Cordeiro, Lucas Carvalho. II. Universidade Federal do Amazonas III. Título

PODER EXECUTIVO
MINISTÉRIO DA EDUCAÇÃO
INSTITUTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

FOLHA DE APROVAÇÃO

"Geração de casos de teste usando Bounded Model Checking"

RAFAEL SÁ MENEZES

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Lucas Carvalho Cordeiro - PRESIDENTE

Prof. Raimundo da Silva Barreto - MEMBRO INTERNO

Dr. Eddie Batista de Lima Filho - MEMBRO EXTERNO

Manaus, 08 de Junho de 2021

Este trabalho é dedicado ao meu irmão Lucas, que me mostrou seu jeito único de pensar.

Agradecimentos

Primeiramente, quero agradecer aos meus pais, por sempre me apoiarem em todas as minhas decisões, aconselhando-me mesmo nos momentos difíceis e sempre visando meu futuro.

Gostaria de agradecer ao meu orientador, professor Lucas Cordeiro, que durante o mestrado não só me orientou a ser um pesquisador mas também se tornou um amigo, dando-me suporte e ajuda nos momentos difíceis.

Agradeço também ao meu coorientador, professor Herbert Oliveira, que desde a graduação já me introduziu no universo da pesquisa e, agora no mestrado, voltou a me ajudar com contribuições valiosas durante a pesquisa.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico, por ter me cedido auxílio financeiro. Graças a isso pude me dedicar mais às atividades acadêmicas.

Resumo

A geração automática de casos de teste consiste na geração de entradas para um algoritmo que sejam capazes de explorar todos os caminhos de um programa, maximizando a cobertura. Comumente, técnicas como fuzzing são utilizadas para isso por sua performance alta e de custo baixo. Entretanto, técnicas formais, como execução simbólica, vêm ganhando espaço nesse domínio, entre elas, a Bounded Model Checking (BMC). BMC é uma técnica de verificação que codifica uma propriedade em caminhos de um sistema de transição até um limite k, resultando em uma única fórmula lógica que, caso contenha uma violação, irá gerar uma prova. Essa prova - embora reduzida - contém informações que podem ser utilizadas como base para a geração de um caso de teste. A dificuldade de usar BMC em casos de teste está na complexidade de tempo/espaço da técnica e nas simplificações feitas para tentar contornar isso (simplificações, estratégias). Incremental BMC é uma estratégia que, de forma incremental, aumenta esse limite k até que uma violação seja encontrada ou que o sistema seja completamente verificado. Entretanto, cada caminho do sistema é reverificado a cada incremento, o que adiciona tempo e consumo de memória para resolver a instância atual. Para resolver esses problemas, este trabalho propõe duas contribuições: (1) algoritmo de geração de casos de teste a partir de provas e (2) um sistema de cache que utiliza soluções de instâncias menores através de (A) uso direto das instâncias anteriores; e (B) uso indireto através de fórmulas de outros sistemas. Essas contribuições foram implementadas no Efficient SMT-based Bounded Model Checker (ESBMC), sendo testadas em situações de Integração Contínua (CI) e sobre benchmarks públicos de programas em C, obtendo a primeira colocação na competição TestComp'21 na categoria reach-error e reduzindo o tempo do CI.

Palavras-chave: bounded model checking; caching; software; verification; SMT solving; testcase generation.

Abstract

Testcase generation consists in generating inputs for an algorithm that are able to explore all paths of a program, maximizing its coverage. Usually, fuzzing techniques are used because of its great performace and low cost. However, formal techniques such as Symbolic Execution are gaining notority in this domain, one of those is Bounded Model Checking (BMC). BMC is a verification technique that encodes a property in paths of transition system up to a bound k to one single logic formula such that if the system contains a flaw the formula will generate a proof. This proof contain information for the testcase generation. The difficult of using BMC for testcase is at the time/space complexity and at the optimizations trying to workaround this issue. Incremental BMC is a strategy that increases the k limit until the violation is found or the system is completely verified. The issue is that at each increment every path is verified again, which add time and memory consuption to solve the current instance. To solver those issue this work proposes two main contributions: (1) algorithm for testcase generation using proofs; and (2) a caching system that uses small instances by: (A) direct use; and (B) indirect use of other systems. Those contributions were implemented into the Efficient SMT-based Bounded Model Checker (ESBMC), being tested over Continuos Integration and over public benchmarks of C programs, obtaining the first place at Test-Comp'21 in the Cover-error category and reducing the time of the CI.

Keywords: bounded model checking; caching; software verification ; SMT solving ; testcase generation.

Lista de Figuras

1.1 Fluxo de Incremental BMC . . . 4
1.2 Fluxo de Incremental BMC com *caching* . . . 7
1.3 Arquitetura de *caching* . . . 8

2.1 Exemplo de teste . . . 13
2.2 Programa para análise estática . . . 14
2.3 Exemplo de Execução Simbólica . . . 16
2.4 Demonstração de $k = 1$ . . . 17
2.5 Fluxo do ESBMC . . . 18
2.6 Programa de exemplo com erro . . . 19
2.7 Exemplo de GOTO . . . 19
2.8 Exemplo de SSA (k=1) . . . 20
2.9 Exemplo de SSA (k=2) . . . 22

3.1 Arquitetura do método . . . 23
3.2 Fluxo da geração de um caso de teste . . . 24
3.3 Exemplo de Reordenação . . . 31
3.4 Algoritmo utilizado para reordenação . . . 32

Lista de Tabelas

3.1 Regras de transoformação T 27
3.2 Regras de geração U 28

4.1 Ferramentas de teste 35

5.1 Subcategorias do Test-Comp 39
5.2 Test-Comp'21 41
5.3 Test-Comp'20 41

Sumário

Agradecimentos **ix**

Resumo **xi**

Abstract **xiii**

Lista de Figuras **xv**

Lista de Tabelas **xvii**

1 Introdução **1**

1.1 Descrição do Problema 5
1.2 Objetivos de Pesquisa 6
1.3 Metodologia Proposta 7
1.4 Contribuições 8

2 Fundamentação Teórica **11**

2.1 Verificação Formal de Software 11
2.1.1 Testes de *Software* 12
2.1.2 Análise Estática 13
2.1.3 Análise Dinâmica 14
2.2 Execução Simbólica 15
2.2.1 Árvore de execução simbólica 15
2.3 Bounded Model Checking 16
2.3.1 ESBMC 17
2.4 Resumo 20

3 Método Proposto **23**

3.1 Geração de Casos de Teste 24
3.1.1 Demonstração 26

3.2 Caching de fórmulas UNSAT 26
3.2.1 Simplificação de Assertivas 27
3.2.2 Maximização de *Hits* 30
3.2.3 Canonização 30
3.3 Resumo 32

4 Trabalhos Relacionados **35**
4.1 Geração de Casos de Teste 35

5 Resultados Experimentais **37**
5.1 Planejamento e projeto dos experimentos 37
5.1.1 Descrição dos Benchmarks 38
5.1.2 Ambiente de execução 38
5.2 Resultados dos Experimentos 40
5.3 Resumo 42

6 Conclusões e Trabalhos Futuros **43**

Referências Bibliográficas **45**

Anexo A Publicações **53**
A.1 Relacionadas 53
A.2 Contribuições em outras pesquisas 53

Capítulo 1

Introdução

O teste de software tem como objetivo encontrar erros em um programa antes do seu lançamento. Ao executar um teste, o testador executa um programa por meio de dados artificiais, buscando anomalias e erros [Roger, 2019]. Ao utilizar teste de software, aumenta-se a confiança sobre a qualidade do produto antes da distribuição. Assim, é possível determinar os riscos associados ao software antes de chegar ao usuário final.

Entre as modalidades para teste de um software existe o *white-box*, em que os casos de teste são criados com base em informações sobre como a implementação ocorreu. Um testador se utiliza de metodologias e de sua experiência para construir casos de teste que exponham um erro [O'Regan, 2017]. Porém, muitos softwares dependem de sistemas complexos com implementações escritas por diferentes pessoas, com diferentes metodologias e estilos [Hoder et al., 2011].

Dado que o teste muitas vezes é a parte mais custosa de um projeto, é fundamental reduzir seu custo e melhorar sua eficiência com automatizações do processo. Existem técnicas focadas na geração de testes através de ferramentas como *fuzzing* e execução simbólica, sendo capazes de encontrar vulnerabilidades e maximizar a cobertura de linhas de código [Alshmrany et al., 2021, Gadelha et al., 2020b, Menezes et al., 2018, Rocha et al., 2015a, 2020].

O *Fuzzing* consiste em gerar entradas utilizando técnicas probabilísticas e de Aprendizagem de Máquina de forma que as entradas maximizem (ou minimizem) um objetivo [Serebryany, 2016]. Alguns exemplos comuns de objetivos são: quantidade de instruções executadas com a entrada, quantidade de erros encontrados, e erros alcançados [Chowdhury et al., 2019]. Em contraste, a execução simbólica faz uma execução do programa em função de símbolos, uma operação como $x = -y$ não irá computar o valor de y, apenas manterá a associação de que x contém o valor de $-y$.

Ao fazer esse tipo de execução, ao alcançar uma estrutura de controle de fluxo, é possível analisar quais seriam as condições que precisam ser verdadeiras para mudar o fluxo, gerando uma condição de caminho. Caso essa condição de caminho tenha uma solução, é possível usar os valores da solução para gerar entradas e explorar o programa [Cadar et al., 2008]. Em comparação, um *fuzzer* é mais rápido, pois as entradas são apenas executadas diretamente, porém quanto mais complexo for o caminho de execução, mais difícil será para o *fuzzer* encontrar o erro. Já a execução simbólica consegue gerar entradas para programas com caminhos com muitas restrições, porém encontrar soluções em condições é um problema NP-Completo e, portanto, mais lento.

Ao mesmo tempo técnicas como *model checking* vêm ganhando popularidade na parte de Verificação de Software, e já foram demonstradas como eficazes para encontrar vulnerabilidades e provar ausência delas em programas em C [Cordeiro et al., 2012, Cordeiro, 2010]. Essas técnicas utilizam propriedades fundamentais que não podem ser violadas e que podem envolver gerenciamentos de memória (*memory-leak, dereference*) [Rocha et al., 2016] até problemas de alcançabilidade (*assertions*) [Rocha et al., 2016] e são definidas através de uma formalização [Chong et al., 2021] e, caso encontrem alguma violação, geram um contraexemplo que contém as informações sobre como houve a violação. Esses contraexemplos podem ser utilizados como base para a geração de novos testes para a replicação do erro [Rocha et al., 2012].

Para fazer uso de ferramentas, é necessária uma generalização da propriedade que está sendo testada, podendo ser feita através de *proof harness* [Chong et al., 2021], que é uma prova (proof) da propriedade (harness), que é estruturada de maneira similar a testes unitários (Subseção 2.1.1), a AWS utiliza model-checking para verificar seus códigos [Cook et al., 2018], possuindo diversas provas para a sua biblioteca base de C99 que inclui primitivas e estruturas[1]. O Algoritmo 1 contém um exemplo de prova extraído do repositório da AWS, nele todas as funções iniciadas por `nondet` representam a criação de um valor não-determinístico, as funções de `aws_add_*_checked` verificam se a soma resultaria em um *overflow*. A prova consiste em checar se a função acusa *overflows* e se retorna a soma corretamente.

Essas provas podem ser validadas por uma ferramenta de BMC [Cordeiro et al., 2019, Gadelha et al., 2018b, Pereira et al., 2016]. O BMC é um método de análise estática que simbolicamente explora o espaço de estados de um programa para checar a negação de uma dada propriedade ϕ em uma profundidade k, onde k limita o número de regiões visitadas de estruturas de dados, iterações de *loops* e chamadas recursivas. Entretanto, ferramentas BMC podem sofrer com exaustão de tempo ou limites de

[1] `https://github.com/awslabs/aws-c-common`

Algoritmo 1 Proof harness aws_add_size_checked_harness.c

```
/*
* Assumptions:
*     - given 2 non-deterministics unsigned integers
*
* Assertions:
*     - r does not overflow, if aws_add_u32_checked or
*       aws_add_u64_checked functions return AWS_OP_SUCCESS
*/
void aws_add_size_checked_harness() {
  if (nondet_bool()) {
    uint64_t a = nondet_uint64_t();
    uint64_t b = nondet_uint64_t();
    uint64_t r = nondet_uint64_t();
    int rval = aws_add_u64_checked(a, b, &r);
    if (!rval) {
      assert(r == a + b);
    } else {
      assert((b > 0) && (a > (UINT64_MAX - b)));
    }
  } else {
    uint32_t a = nondet_uint32_t();
    uint32_t b = nondet_uint32_t();
    uint32_t r = nondet_uint32_t();
    if (!aws_add_u32_checked(a, b, &r)) {
      assert(r == a + b);
    } else {
      assert((b > 0) && (a > (UINT32_MAX - b)));
    }
  }
}
```

memória por programas que contenham *loops* com limites muito grandes, essa limitação faz com que BMC seja mais utilizado em um estilo de verificação incremental [Gadelha et al., 2017, 2019, Morse et al., 2013, Rocha et al., 2015b]. Como o melhor valor de k normalmente é desconhecido, a ferramenta BMC é reexecutada com valores crescentes de k, e a cada aumento do k aumenta o número de iterações em *loops*, a profundidade de recursão e os requisitos de tempo e memória. Para checar se o valor de k é grande o suficiente, as ferramentas BMC inserem assertivas de desenrolamento após cada *loop*, ao falhar, significa que a verificação está incompleta.

A Figura 1.1 demonstra o fluxo de uma ferramenta de Incremental BMC, consistindo em: gerar o CFG de um programa, fazer a execução simbólica (adicionam-se as assertivas, guardas e cláusulas) e o SSA, o qual é convertido em uma fórmula SMT que é resolvida por um solucionador, caso a fórmula seja válida, verifica-se se as condições de desenrolamento foram satisfeitas, se sim o BMC encerra, e se não foram, o BMC é reexecutado com um k maior.

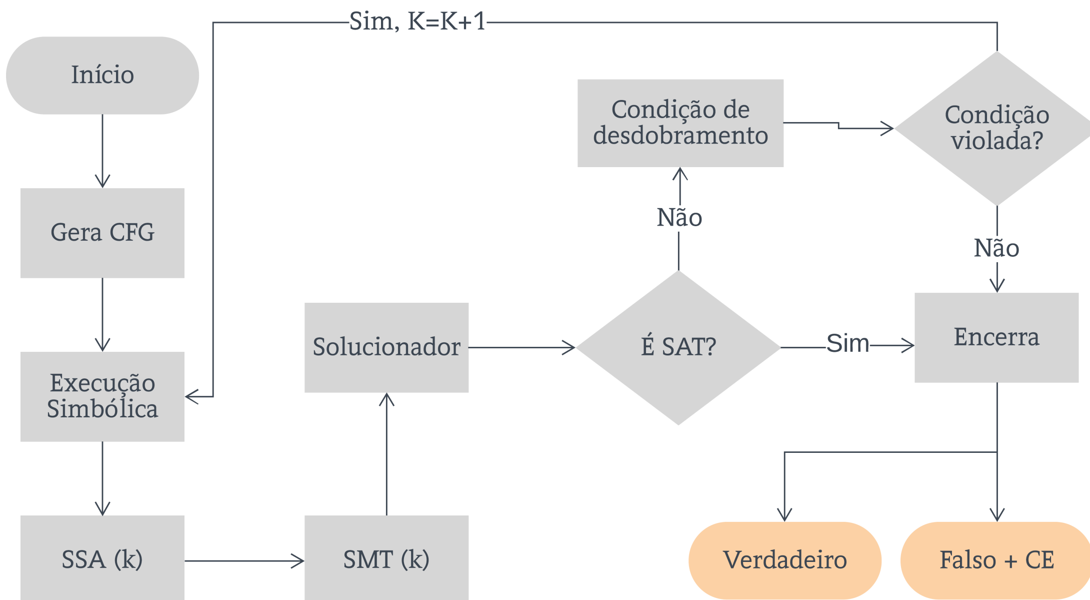


Figura 1.1: Fluxo de Incremental BMC

Ferramentas BMC podem tentar determinar um valor de k de modo estático de forma que ele desenrole todos os *loops*, se esse valor for encontrado, então não existe a necessidade de assertivas de desenrolamento [Cordeiro et al., 2012]. Entretanto, determinar esses valores nem sempre é possível (*loops* infinitos ou condições que dependem da entrada do usuário), e buscar por esse valor pode fazer com que as ferramentas BMC tentem desenrolar esses *loops* para o maior valor possível, o que causa exaustão de recursos e falhas no seu propósito original que é encontrar *bugs*.

Nesta dissertação é desenvolvido e avaliado um método para utilizar ferramentas de verificação formal para geração de casos de teste, sendo utilizados contraexemplos das ferramentas *Efficient SMT-based Bounded Model Checking* (ESBMC) e Map2Check. O método foi posteriormente integrado a um *fuzzer*, e obteve premiações no TestComp'21 (FuSeBMC) [2] e TestComp'20 (ESBMC)[3]. Além disso, foram feitas algumas contribuições nas ferramentas para otimizar o suporte a casos de teste, em especial um sistema de *caching* para o ESBMC na modalidade Incremental.

[2] https://test-comp.sosy-lab.org/2021/results/results-verified/
[3] https://test-comp.sosy-lab.org/2020/results/results-verified/

1.1 Descrição do Problema

Esta dissertação de mestrado busca investir no problema de Teste de Software: (I) Geração automática de casos de teste usando BMC; e (II) Explosão de estados em Incremental-BMC.

Embora estudos mostrem que o BMC pode ser utilizado de forma eficiente para a criação de testes de software [Beyer & Lemberger, 2017], ele tem dificuldades, como: explosão de estados, aproximação e incompletude. Os métodos tradicionais para lidar com esses problemas não afetam a geração de provas, porém dificultam a criação de testes automáticos.

Por exemplo, o programa do Algoritmo 2 contém um *variable-length array*(VLA) - disponível no C99 - de tamanho não determinístico que após ser inicializado (com valores não-determinísticos) encontraria uma assertiva que sempre iria falhar. Ao usar uma ferramenta como o ESBMC com a estratégia Incremental BMC, a violação seria encontrada no *bound* 6, e a prova apenas conteria o valor gerado para o `SIZE`. Isso acontece devido a otimizações internas que desconsideram os estados que não contribuem na alcançabilidade do erro.

Algoritmo 2 Programa de exemplo (incomplete)

```
1  int main() {
2    int SIZE = __VERIFIER_nondet_int();
3    __ESBMC_assume(SIZE > 10);
4    int arr[SIZE];
5    for(int i = 0; i < SIZE; i++) {
6      arr[i] = __VERIFIER_nondet_int();
7      __ESBMC_assert(1 == 1, "success");
8    }
9    __ESBMC_assert(1 == 0, "will fail");
10   return 0;
11 }
```

Incremental BMC [Whittemore et al., 2001] apareceu primeiramente no início do século, o programa é desenrolado de maneira incremental até um *bug* ser encontrado ou um limite de completude ser alcançado, garantindo que problemas menores sejam resolvidos em ordem incremental [Günther & Weissenbacher, 2014] . O algoritmo incremental, porém, ainda contém suas limitações: o BMC tem que refazer todo o *parsing* do programa, geração das restrições, e resolver para cada limite *k*, e nenhuma recordação dos passos anteriores *1* até *k-1* é utilizada quando solucionando para *k*, voltando ao exemplo da Fig 2, para cada incremento de *k* a condição da linha 7 é verificada novamente. Embora solucionar SAT através de incrementos tenha aparecido na década de

90, o problema sobre reutilizar informações aprendidas de instâncias anteriores de BMC continua [Günther & Weissenbacher, 2014, Hooker, 1993, Whittemore et al., 2001].

Uma instância de BMC consiste em um programa desenrolado (até uma profundidade k), traduzido para um conjunto de restrições e representado na forma de *single static assignment* (SSA). Além disso, quando um verificador BMC executa de maneira simbólica o programa, ele codifica todos os caminhos possíveis de execução em uma fórmula SMT única, o que resulta em um grande número de restrições que precisam ser verificadas. Há trabalhos que definem condições para o reuso de informação aprendida, mas uma análise de dependência detalhada deve ser feita, o que enfraquece o benefício de solução incremental [Whittemore et al., 2001].

Considerando que uma das motivações de BMC é encontrar *bugs* em programas e a dificuldade de encontrar esses problemas em programas maiores, o problema de pesquisa deste trabalho pode ser descrito como: *De que maneira adaptar o BMC para que o custo de solucionar novas instâncias em Incremental BMC seja minimizado e ele possa ser utilizado para gerar testes de programas mais complexos?*

1.2 Objetivos de Pesquisa

O objetivo principal deste trabalho é definido como: Projetar e avaliar um método para conversão de contraexemplos originados de técnicas de BMC para a geração automática de testes de programas em C de forma eficiente, utilizando-se instâncias anteriores.

Os objetivos específicos são:

1. Pesquisar e propor um algoritmo para conversão de contraexemplos em casos de teste, o qual deve ser capaz de funcionar em outros verificadores formais;

2. Demonstrar um método para caching de instâncias de BMC, de forma que programas não necessitem repetir as mesmas provas ao tentar alcançar um estado de erro;

3. Implementar o método proposto em um software que utiliza a técnica BMC;

4. Validar o método via uma avaliação experimental utilizando aplicações de software opensource com vulnerabilidades conhecidas para analisar a eficiência do método proposto.

1.3 Metodologia Proposta

Para gerar os casos de teste, uma abordagem simples seria a conversão de witness gerados por ferramentas de verificação[4], isso permitiria comparação e complementação com outros métodos. Além disso, otimizações que normalmente são *unsound* podem ser utilizadas para a criação de testes de maneira mais rápida e eficiente.

Quanto escalonabilidade em verificadores BMC, a proposta consiste em aproveitar a solução da instância anterior como base para geração da nova fórmula SMT - um *caching* - a cada novo incremento do k. Isso diminuiria a exploração de estado-espaço utilizando um conhecimento prévio. A Figura 1.2 mostra o fluxo proposto, onde o *cache* entraria entre a geração da fórmula SMT a partir do SSA até a profundidade k.

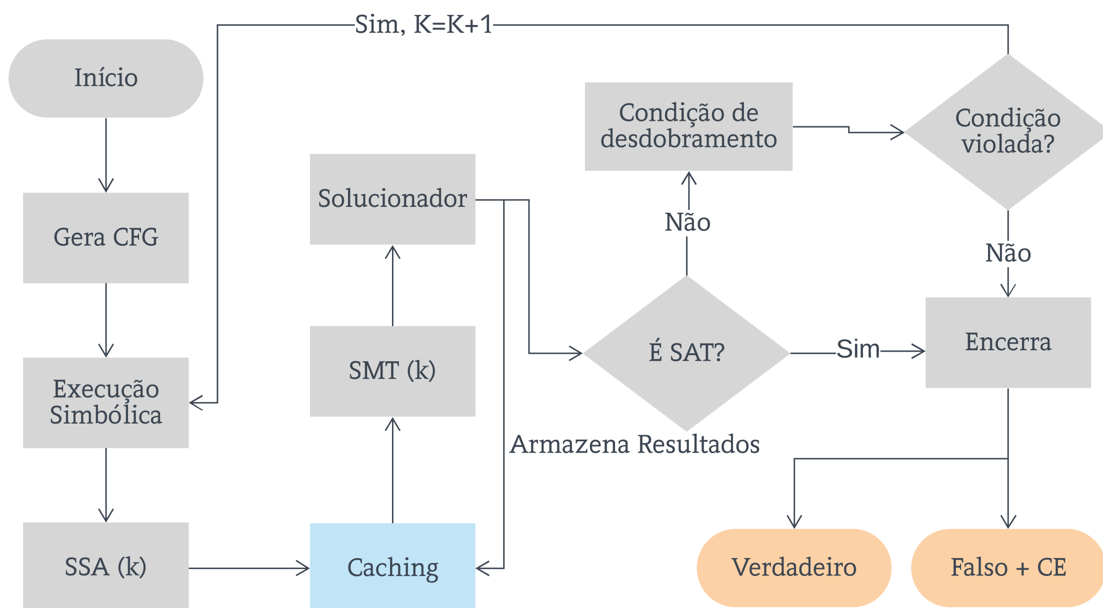


Figura 1.2: Fluxo de Incremental BMC com *caching*

A Figura 1.3 contém a proposta de arquitetura para um sistema de em que, ao receber um *SSA Step*, o algoritmo irá passar por: (a) uma interface de *caching* independente do solver; (b) uma canonização da fórmula; (c) Uma heurística que será capaz de determinar fórmulas semelhantes, e (d) um *slicer* capaz de verificar se as fórmulas semelhantes podem ser aproveitadas.

As hipóteses de pesquisa podem ser descritas como:

[4] `https://github.com/sosy-lab/sv-witnesses`

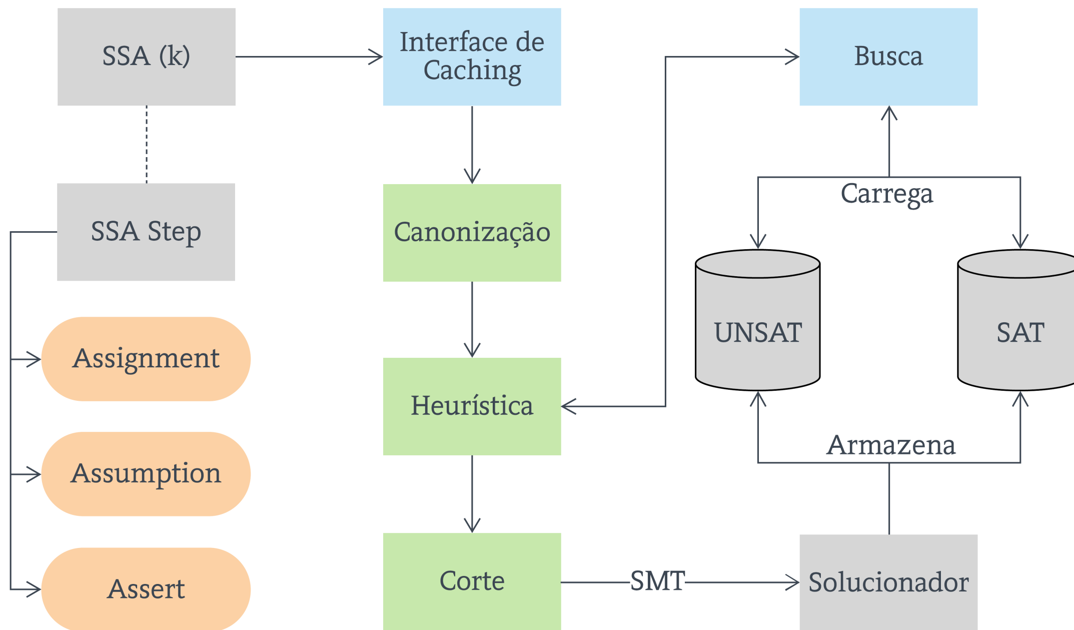


Figura 1.3: Arquitetura de *caching*

1. *soluções de proof harness quando falham podem ser utilizadas como base para a geração de casos de teste.*

2. *instâncias BMC são grandes, porém as diferenças no SSA geradas entre as iterações são mínimas. Dessa forma, as instâncias anteriores podem ser aproveitadas em novas iterações.*

1.4 Contribuições

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

- Investigacão, implementação e avaliação de uma metodologia para a geração de casos de testes;

- Investigacão, implementação e avaliação de um algoritmo de caching em Incremental BMC.

Durante o desenvolvimento do trabalho foi observada a oportunidade de utilizar os testes gerados como entrada para um *fuzzer* externo, resultando em uma contribuição com a ferramenta FuSeBMC [5]

[5] https://github.com/kaled-alshmrany/FuSeBMC

Capítulo 2

Fundamentação Teórica

Este capítulo irá introduzir os conceitos necessários para o entendimento desta dissertação, ele está subdividido nas seguintes seções:

Verificação Formal de Software: esta seção irá explicar e descrever o que é verificação formal de Software, juntamente com uma descrição sobre o que são casos de teste e como eles podem ser gerados.

Execução Simbólica: esta seção irá descrever a técnica de Execução Simbólica e como ela pode ser usada para encontrar violações em programas.

Bounded Model Checking: a seção irá definir *Bounded Model Checking*, uma técnica de Análise Estática que utiliza Execução Simbólica para gerar uma fórmula lógica que será posteriormente verificada. Além disso, a seção mostra a modalidade incremental da técnica e o principal problema, que é a explosão exponencial de estados. Por último, a subseção 2.3.1 irá explicar o funcionamento do BMC que será utilizoado no Método.

2.1 Verificação Formal de Software

Métodos formais podem ser considerados, como: *matemática aplicada para modelagem e análise de sistemas de informação* [Baier & Katoen, 2008]. A verificação formal busca validar a corretude de um programa utilizando lógica matemática [Clarke et al., 2009]. Neste contexto, um programa é um objeto matemático (usualmente composto por uma série de conjuntos), com um comportamento definido, assim, a lógica matemática pode ser utilizada para descrever precisamente o que é um comportamento correto do programa. Isso torna possível utilizar modelos matemáticos de um programa [Monteiro

et al., 2017] e, assim, provar a corretude de seu comportamento, até mesmo para sistemas digitais [Chaves et al., 2018, de Bessa et al., 2014].

A criação de métodos automatizados para validar a corretude de programas tem um impacto significante, visto que a construção manual de uma prova de corretude pode ser complexa. Essa prova manual pode não funcionar corretamente para programas grandes, devido à enumeração exaustiva de estados em sistemas complexos.

2.1.1 Testes de *Software*

Teste de *software* são métodos para checar a corretude da implementação de um sistema experimentando-o. Basicamente, é o processo de execução de um *software* procurando comportamentos incorretos. Os testes não garantem observar todos os comportamentos de um programa, porém, se o programa contiver apenas um caminho, é possível [Ding et al., 2008]. A corretude então é determinada por forçar o programa a percorrer um conjunto de caminhos de execução que abrangem a maior cobertura de execução do programa sem resultar na violação de propriedades [Baier & Katoen, 2008].

Um dos tipo de teste é o teste unitário, nele o processo de testar consiste em validar componentes de um programa, como métodos e classes de objetos. Os testes são chamadas dessas rotinas com diferentes parâmetros de entrada [Pressman, 2001]. Para exemplificar esse teste, tomemos a situação em uma loja que vende batatas ao quilograma: o preço normal é de R$$1,5$/kg, porém para compras acima de 50kg, o preço ficaria R$$1,25$/kg. Ao desenvolver uma função que calculasse o custo de uma compra, os casos de teste deveriam ser entradas da função. Nesse exemplo, a entrada seria a quantidade de quilogramas, como: $0, 49$ e 51. A Figura 2.1 exibe um exemplo de implementação de uma função de custo, caso alguma entrada inválida seja feita, é retornado -1. Os casos de teste também foram implementados e a função `COMPARE_FLOAT_EQ` é usada para fazer comparações entre pontos flutuantes.

As técnicas de teste estão divididas em três categorias: **especificação** (caixa-preta), **estrutural** (caixa-branca) e **experiência**. Testes de caixa-preta se concentram nas entradas e saídas do programa sem analisar a sua estrutura interna, já as técnicas de caixa-banca focam em como o processamento interno funciona. Testes de experiência são desenvolvidos com base no conhecimento prévio do código e dos requisitos [Alaqail & Ahmed, 2018, Ali & Yue, 2015, Reid, 2013].

Métodos formais podem ser utilizados em teste de *software*, os casos de teste podem ser gerados com eficiência e eficácia quando derivados de especificações formais [Ding et al., 2008]. Na prática, qualidade nem sempre se refere a um *software* totalmente correto em projeto, pois isso aumenta o custo de produção consideravelmente. Porém,

```c
1  double custo(double entrada)
2  {
3    if (entrada < 0)
4    {
5      return -1;
6    }
7    if ((entrada * 1.5) <= 0))
8    {
9      return -1;
10   }
11   if(entrada > 50)
12   {
13     return 1.25*entrada;
14   }
15   return 1.5*entrada;
16 }
17
```

(a) Código C

```c
1  TEST_CASE_1()
2  {
3    double result;
4    result = custo(0.49)
5    COMPARE_FLOAT_EQ(result,
       0.735);
6  }
7  TEST_CASE_2()
8  {
9    double result;
10   result = custo(51);
11   COMPARE_FLOAT_EQ(result,
       63.75);
12 }
```

(b) Código de Teste

Figura 2.1: Exemplo de teste

garantir que falhas específicas estarão ausentes é uma tarefa realística e é considerada uma boa métrica para garantir qualidade [D'silva et al., 2008].

Alguns métodos envolvem a geração automática utilizando técnicas como *random test*, *metamorphic*, e *fuzzing* [Chen et al., 2020, Jiang et al., 2009, Pacheco et al., 2007, Paraskevopoulou et al., 2015, Wunderlich, 1990, Zhou et al., 2004], podendo utilizar de técnicas formais como Execução Simbólica para gerar os caminhos de teste [Cadar et al., 2008].

2.1.2 Análise Estática

A análise estática é a análise de um programa sem a sua execução, através de análises no código-fonte e analisando também a semântica da linguagem (como tipagem ou acesso de elemento fora do índice). Compiladores utilizam esse tipo de análise durante o processo de compilação para verificar erros (como de tipos) e otimização de código. As ferramentas que fazem essa análise só precisam ler o código para efetuá-la [Rocha, 2015]. O programa da Figura 2.2 contém dois erros que podem ser detectados através de uma análise estática: (a) na linha 2, tenta-se carregar o conteúdo de `a` no índice 7 sendo que na linha 1 ele foi definido como de tamanho 3; (b) na linha 5, tenta-se atribuir um valor de uma variável do tipo flutuante para um inteiro, o que pode gerar erros futuros.

A vantagem dessa abordagem é que ela pode ser usada para programas grandes

```
1 int a[3];
2 int b = a[7];
3
4 long c = 5.5432;
5 a[0] = c;
6
```

Figura 2.2: Programa para análise estática

.

(com mais de 220 mil linhas de código C) sem interação do usuário e as abstrações escolhidas podem ser implementadas em bibliotecas que podem ser reutilizadas em diferentes linguagens [Cousot, 2001]. Já a desvantagem dessa abordagem é que as especificações e a maioria das propriedades são simples, geralmente propriedades de segurança elementares como erros de execução [Cousot, 2001].

Um exemplo de aplicação que utiliza esse tipo de análise seria o ASTRÉE [Cousot et al., 2005] que é um analisador estático que prova automaticamente a falha de erros em tempo de execução para programas escritos em C, e que foi aplicado com sucesso em vários sistemas embarcados críticos. Astrée se baseia na teoria da interpretação abstrata [Cousot & Cousot, 1977] e prossegue calculando uma aproximação das propriedades semânticas de traços de execução do programa analisado e provando que essas propriedades abstratas implicam na ausência de erros em tempo de execução.

2.1.3 Análise Dinâmica

A análise dinâmica infere conclusões a partir da execução do programa, derivando assim propriedades de segurança para uma ou mais execuções, podendo detectar a violação destas. Para a análise dinâmica, as ferramentas devem instrumentar o programa com análises [Rocha, 2015]. Algumas utilizações dessa abordagem são: evitar *bugs*, depuração, teste e verificação. Com a análise dinâmica é possível executar testes e verificar se suas assertivas não são violadas [Ernst et al., 2007].

O VALGRIND [Nethercote & Seward, 2007] é um *framework* que faz uso de instrumentação binária para análise dinâmica. O módulo *memcheck* VALGRIND verifica por erros de memória em tempo de execução, por exemplo, vazamentos de memória [Cheng et al., 2006]. A verificação efetuada pelo VALGRIND envolve sombrear/marcar cada bit de dados em registos e memória com um segundo *bit*, que indica se o bit tem um valor definido. Cada operação de determinação de valor é instrumentada com uma operação de marcação que propaga os *bits* adequadamente. O *Memcheck* usa

esses *bits* de marcação para detectar usos de valores indefinidos que poderiam afetar adversamente o comportamento de um programa.

2.2 Execução Simbólica

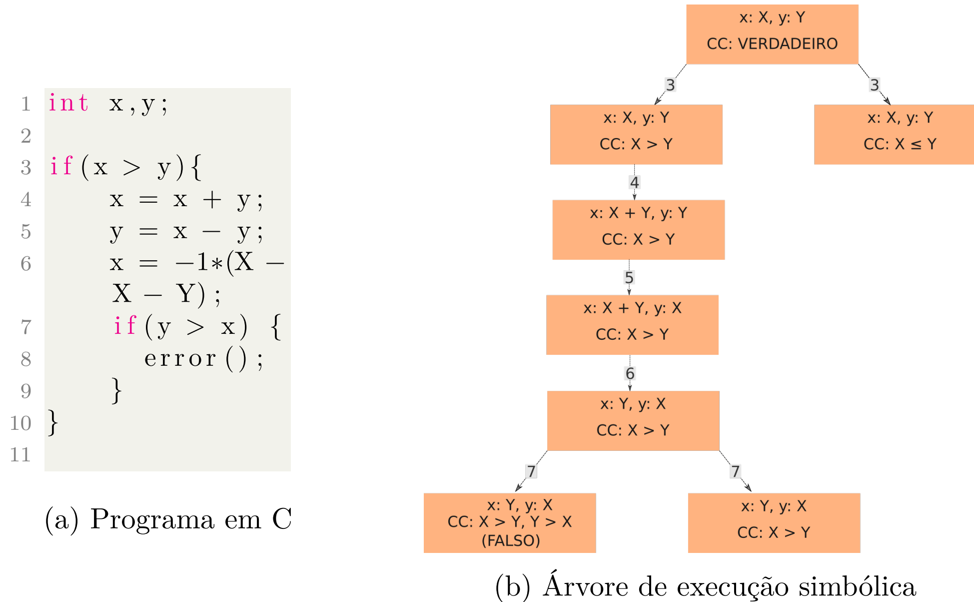
A execução simbólica, em verificação e teste de *software*, é uma técnica para gerar dados de entrada de programas utilizando valores simbólicos ao invés de valores concretos e, assim, representar os valores das variáveis com expressões simbólicas. Em teste de *software*, a execução simbólica é utilizada para gerar entradas para cada caminho de execução viável de um programa, gerando casos de teste para valores simbólicos que gerem um erro [Cadar et al., 2011]. Um caminho de execução viável é uma sequência de valores booleanos. Todos esses caminhos podem ser representados a partir de uma árvore de execução do programa [Cadar & Sen, 2013].

O estado de um programa executado de forma simbólica inclui: (a) os valores simbólicos das variáveis, que são valores representativos das variáveis, por exemplo um programa contendo duas variáveis, x e y, e havendo uma atribuição em x com o valor de $y + 1$ geraria a seguinte informação $x : Y + 1, y : Y$; (b) Uma condição de caminho (CC), que é uma condição lógica para o caminho ser executado, um exemplo seria uma estrutura de condição, em que a condição é que x seja maior que y; e (c) um *program counter* (PC), que aponta para o próximo estado a ser executado. A CC é uma fórmula booleana sem quantificadores que utiliza as entradas simbólicas, a fórmula aglomera todas as restrições que as entradas devem satisfazer para que a execução siga o caminho associado. O PC define a próxima instrução a ser executada. Uma árvore de execução simbólica exibe todos os caminhos seguidos durante a execução simbólica de um programa. Segundo o objetivo da execução simbólica pode ser descrito como obter um conjunto de entradas tais que todos os caminhos de execução viáveis possam ser explorados uma única vez ao se executar o programa com essas entradas [Cadar & Sen, 2013], na Seção 2.2.1 será apresentado um exemplo da utilização.

2.2.1 Árvore de execução simbólica

A árvore é um grafo cujos nós representam os estados do programa e as arestas representam as transições entre os estados [Khurshid et al., 2003], a árvore é uma forma de demonstrar os caminhos possíveis dentro de um programa. A Figura 2.3b mostra um exemplo de uma árvore para o programa da Figura 2.3a. O programa contém duas variáveis que para este exemplo, vamos assumir que foram inicializadas com valores aleatórios, na linha 3 há uma condição de $x > y$ e, logo em seguida, os valores de x e y

são trocados, e caso $y > x$, chegamos a um estado de erro. Com a árvore de execução simbólica da Figura 2.3b, podemos perceber que a seguinte condição: $x : Y$, $y : X$, e a CC $X > Y$. Para a linha 7 dar falso, a condição de caminho será $(X > Y) \wedge (Y > X)$, o que é uma contradição.



(a) Programa em C

(b) Árvore de execução simbólica

Figura 2.3: Exemplo de Execução Simbólica

2.3 Bounded Model Checking

Bounded Model Checking (BMC) é a verificação de uma dada propriedade em uma determinada 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 satisfeito se e somente se ϕ tem um contra-exemplo de profundidade menor ou igual a k [Rocha, 2015]. Um exemplo de ferramenta BMC é o ESBMC [Cordeiro et al., 2012], que em especial utiliza fórmulas (SMT) para codificar a satisfatibilidade.

Técnicas de *Bounded Model Checking* (BMC) foram aplicadas com sucesso para encontrar *bugs*, checar propriedades, geração de entradas para casos de teste, detecção de vulnerabilidades de segurança e síntese de programas, incluindo programas *multi-threaded* escritos em linguagens como C/C++ e Java [Cordeiro et al., 2018, Gadelha et al., 2018a, Ramalho et al., 2013].

A Figura 2.4 contém um trecho de código em C (a) juntamente com um *unwinding* para $k = 1$, a figuras contém também a geração de SMT.

```
1 int x = 2;
2 int y = *;
3 assume(y>=0 && y
     <3);
4 while(*) {
5   x = x*y;
6   assert(x != 8)
7 }
8
```

(a) Código C com erro

```
1 x1 = 1;
2 y1 = *;
3 assume(y1 >= 0 &&
     y1 < 3);
4 x2 = x1 * y1;
5 assert(x2 != 8);
6
```

(b) K=1

```
1 (= x1 1)
2 (< y1 0)
3 (> y1 3)
4 (= x2 (* x1 y1))
5 (assert (= x2 8))
6
```

(c) SMT Gerado

Figura 2.4: Demonstração de $k = 1$

2.3.1 ESBMC

O ESBMC (Efficient SMT-based Bounded Model Checker) é uma ferramenta de BMC capaz de verificar programas em C utilizando técnicas incrementais e de indução para provar a corretude de um programa[1]. Esta seção irá descrever apenas os processos internos da ferramenta (na versão 6) necessários para o entedimento deste trabalho.

A Figura 2.5, irá servir como base para explicar o caminho de um programa no ESBMC. Para demonstrar o método, o Programa na Figura 2.6 será utilizado durante esta seção. Ele contém um erro na Linha 7, pois caso o *loop* seja executado mais de uma vez com $y = 2$, a assertiva irá falhar. As funções `__ESBMC_assume` e `__VERIFIER_nondet_X` são intrínsecas ao ESBMC e a primeira apenas monta um intervalo de possibilidades para a variável y, a segunda gera um valor não determinístico para uma variável. Primeiramente, em **A** o usuário entrará com um programa em C/C++ juntamente com as opções de propriedades que querem ser avaliadas (como segurança de memória ou assertivas), e como o BMC ocorrerá (estático, incremental ou indução), neste exemplo o ESBMC será executado apenas com a flag `-incremental-bmc`, o que fará com que ele execute de forma incremental.

Em **B** temos a primeira fase, que é a geração de um CFG, internamente o ESBMC utiliza a linguagem GOTO como CFG e representação intermediária [Cordeiro et al., 2012]. Nessa fase, o código que está em C/C++ é parseado, utilizado o **Clang**[2] como *frontend* e gerando uma AST do programa [Gadelha et al., 2019]. Essa AST é então analisada e é gerado um programa em GOTO juntamente com as assertivas necessárias para verificar o programa. Por exemplo, uma operação de `load` sobre um ponteiro iria adicionar uma assertiva para verificar se o endereço de memória é válido. A Figura 2.7 contém um exemplo de como o `main` do Programa ficaria em GOTO, além dele, o

[1]Disponível em `https://github.com/esbmc/esbmc`
[2]`https://llvm.org/`

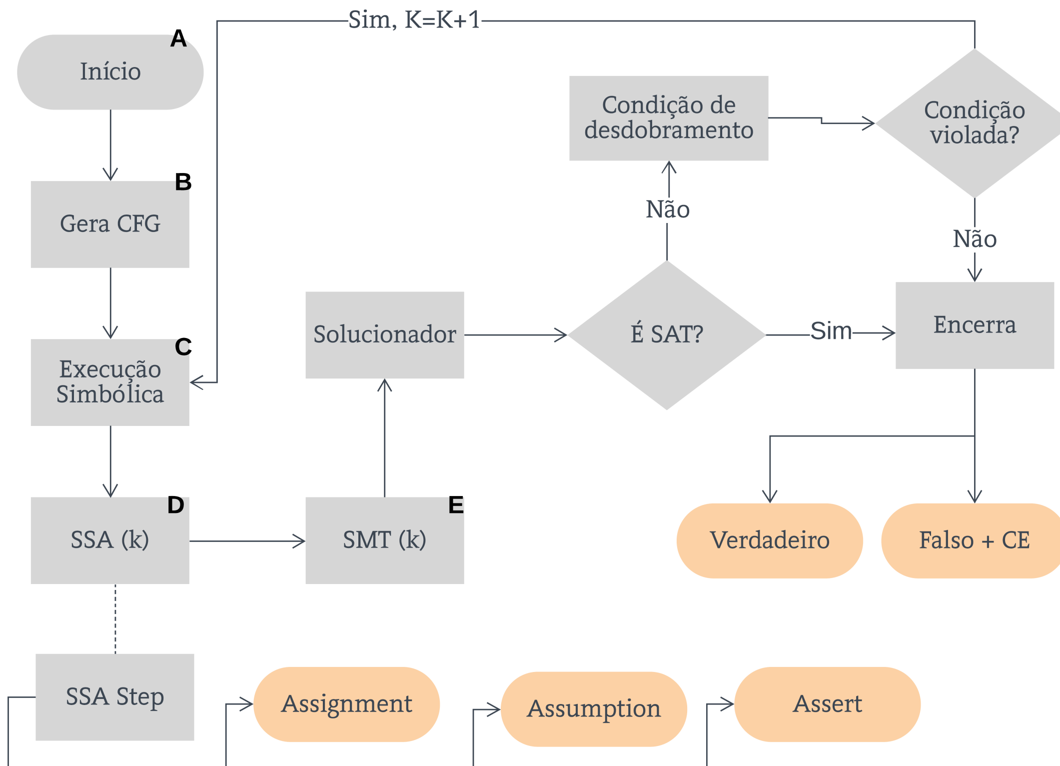


Figura 2.5: Fluxo do ESBMC

programa inteiro também teria variáveis globais e funções intrínsecas do ESBMC. Nessa fase também é aplicada a técnica de propagação de constantes, podendo já acusar falhas em assertivas.

Após a geração do GOTO, na etapa **C**, onde a execução simbólica começa, a execução do programa é simulada através de modelos operacionais que aproximam o comportamento real de um programa em função de símbolos, gerando um SSA representado pela etapa **D**. As figuras Figura 2.8 e Figura 2.9 mostram o SSA para profundidade 1 e 2, respectivamente. O SSA é uma representação na qual variáveis tem valores constantes, então a cada alteração de uma variável em um CFG, o SSA associado irá criar uma nova variável. No ESBMC, esse SSA pode ser um **Assignment** para atribuições, **Assume** para delimitações de intervalo (a violação desse intervalo para a análise e não gera um erro), e **Assert** para as assertivas. Em especial, as assertivas são definidas no formato $guard \implies \neg P$. Finalmente, na etapa **E** esse SSA é convertido para uma lógica SMT, e então mandado a um solucionador, recebendo a

```
1 int main() {
2   int x = 2;
3   int y = __VERIFIER_nondet_int();
4   __ESBMC_assume(y>=0 && y<3);
5   while(__VERIFIER_nondet_int()) {
6     x = x*y;
7     assert(x != 8);
8   }
9 }
```

Figura 2.6: Programa de exemplo com erro

.

```
1        main (c:@F@main):
2        signed int x;
3        x=2;
4        signed int y;
5        signed int return_value$___VERIFIER_nondet_int$1;
6        return_value$___VERIFIER_nondet_int$1=NONDET(signed int);
7        y=return_value$___VERIFIER_nondet_int$1;
8        ASSUME (_Bool)((signed int)(y >= 0 && y < 3))
9    1:  signed int return_value$___VERIFIER_nondet_int$2;
10       return_value$___VERIFIER_nondet_int$2=NONDET(signed int);
11       IF !(_Bool)return_value$___VERIFIER_nondet_int$2 THEN GOTO 2
12       x=x * y;
13       ASSERT (_Bool)(x != 8) // x is 8
14       GOTO 1
15   2:  dead return_value$___VERIFIER_nondet_int$2;
16       dead y;
17       dead return_value$___VERIFIER_nondet_int$1;
18       dead x;
19       RETURN: NONDET(signed int)
20       END_FUNCTION // main
21
22
```

Figura 2.7: Exemplo de GOTO

.

satisfatibilidade da fórmula.

No contexto de qualidade de Software, o ESBMC pode ser utilizado de forma automatizada dentro de um processo de *Continuous Integration* (CI), nesse processo para cada versão do produto, uma bateria de provas é executada novamente.

```
Thread 0 file program.c line 2 function main
ASSIGNMENT ()
c:program.c@15@F@main@x?1!0&0#1 == 2
Thread 0 file program.c line 3 function main
ASSIGNMENT (HIDDEN)
return_value$____VERIFIER_nondet_int$1?1!0&0#1 == nondet_symbol(symex::
    nondet2)
Thread 0 file program.c line 3 function main
ASSIGNMENT ()
c:program.c@28@F@main@y?1!0&0#1 == return_value$____VERIFIER_nondet_int$1
    ?1!0&0#1
Thread 0 file program.c line 4 function main
ASSUME
!(((signed int)(c:program.c@28@F@main@y?1!0&0#1 >= 0 && c:program.
    c@28@F@main@y?1!0&0#1 < 3)) == 0)
Thread 0 file program.c line 5 function main
ASSIGNMENT (HIDDEN)
return_value$____VERIFIER_nondet_int$2?1!0&0#1 == nondet_symbol(symex::
    nondet3)
Thread 0 file program.c line 6 function main
ASSIGNMENT (HIDDEN)
guard?0!0&0#1 == !(return_value$____VERIFIER_nondet_int$2?1!0&0#1 == 0)
Thread 0 file program.c line 6 function main
ASSIGNMENT ()
c:program.c@15@F@main@x?1!0&0#2 == 2 * c:program.c@28@F@main@y?1!0&0#1
Not unwinding loop 1 iteration 1 file program.c line 5 function main
Thread 0 file program.c line 5 function main
ASSERT
\guard_exec?0!0 => !guard?0!0&0#1
unwinding assertion loop

```

Figura 2.8: Exemplo de SSA (k=1)

.

2.4 Resumo

Neste capítulo foi descrita a fundamentação teórica necessária para entender o método proposto. O referencial foi dividido em três seções: *Verificação Formal de Software*, *Execução Simbólica* e *Bounded Model Checking*.

A Seção de Verificação Formal de Software descreveu a diferença entre Verificação Formal e Teste de Software, introduzindo o processo para criação de um teste unitário. Em seguida, foram adicionadas as técnicas clássicas para análise de algoritmos: Análise Estática e Análise Dinâmica. A diferença entre as duas é que na dinâmica o algoritmo é executado e o estado real do ambiente é avaliado, já na estática, o programa não é executado diretamente, sendo analisado por sua estrutura.

Na Seção de Execução Simbólica, foi explicado o conceito de execução de um

algoritmo em função de símbolos. Essa execução simbólica permite a construção de fórmulas que podem ser usadas por algoritmos de Análise Estática e Dinâmica para obter os valores necessários para explorar os caminhos de um programa.

Finalmente na Seção de Bounded Model Checking, foi apresentado uma ferramenta de BMC: o ESBMC. O fluxo do ESBMC foi explicado: Geração do CFG, estratégia incremental, adição de assertivas, execução simbólica, conversão para SMT e geração de um contraexemplo.

```
Thread 0 file program.c line 2 function main
ASSIGNMENT ()
c:program.c@15@F@main@x?1!0&0#1 == 2
Thread 0 file program.c line 3 function main
ASSIGNMENT (HIDDEN)
return_value$___VERIFIER_nondet_int$1?1!0&0#1 == nondet_symbol(symex::
    nondet4)
Thread 0 file program.c line 3 function main
ASSIGNMENT ()
c:program.c@28@F@main@y?1!0&0#1 == return_value$___VERIFIER_nondet_int$1
    ?1!0&0#1
Thread 0 file program.c line 4 function main
ASSUME
!(((signed int)(c:program.c@28@F@main@y?1!0&0#1 >= 0 && c:program.
    c@28@F@main@y?1!0&0#1 < 3)) == 0)
Thread 0 file program.c line 5 function main
ASSIGNMENT (HIDDEN)
return_value$___VERIFIER_nondet_int$2?1!0&0#1 == nondet_symbol(symex::
    nondet5)
Thread 0 file program.c line 6 function main
ASSIGNMENT (HIDDEN)
guard?0!0&0#1 == !(return_value$___VERIFIER_nondet_int$2?1!0&0#1 == 0)
Thread 0 file program.c line 6 function main
ASSIGNMENT ()
c:program.c@15@F@main@x?1!0&0#2 == 2 * c:program.c@28@F@main@y?1!0&0#1
Thread 0 file program.c line 7 function main
ASSERT
\guard_exec?0!0 => (guard?0!0&0#1 => c:program.c@15@F@main@x?1!0&0#2 !=
    8)
x is 8
Unwinding loop 1 iteration 1 file program.c line 5 function main
Thread 0 file program.c line 5 function main
ASSIGNMENT (HIDDEN)
return_value$___VERIFIER_nondet_int$2?2!0&0#1 == nondet_symbol(symex::
    nondet6)
Thread 0 file program.c line 6 function main
ASSIGNMENT (HIDDEN)
guard?0!0&0#2 == !(return_value$___VERIFIER_nondet_int$2?2!0&0#1 == 0)
Thread 0 file program.c line 6 function main
ASSIGNMENT ()
c:program.c@15@F@main@x?1!0&0#3 == c:program.c@15@F@main@x?1!0&0#2 * c:
    program.c@28@F@main@y?1!0&0#1
Thread 0 file program.c line 7 function main
ASSERT
\guard_exec?0!0 => (guard?0!0&0#1 && guard?0!0&0#2 => c:program.
    c@15@F@main@x?1!0&0#3 != 8)
x is 8
Not unwinding loop 1 iteration 2 file program.c line 5 function main
Thread 0 file program.c line 5 function main
ASSUME
!(guard?0!0&0#1 && guard?0!0&0#2)
Thread 0 file program.c line 10 function main
ASSIGNMENT (HIDDEN)
c:program.c@15@F@main@x?1!0&0#4 == c:program.c@15@F@main@x?1!0&0#2
Thread 0 file program.c line 10 function main
ASSIGNMENT (HIDDEN)
c:program.c@15@F@main@x?1!0&0#5 == (!guard?0!0&0#1 ? 2 : c:program.
    c@15@F@main@x?1!0&0#4)
```

Figura 2.9: Exemplo de SSA (k=2)

Capítulo 3

Método Proposto

Esta seção descreve as etapas desenvolvidas para: (a) adaptar as ferramentas de verificação formal para a geração de casos de teste; e (b) utilização de um sistema de *caching* no ESBMC. A Figura 3.1 contém o arquitetura do ESBMC com a adição do método. Na nova arquitetura, a geração do SMT é afetada por conta de *patches* no *slicer* (facilitando a geração de casos de teste). Para o *caching* há alterações na geração da fórmula em SMT e na resposta do *solver*.

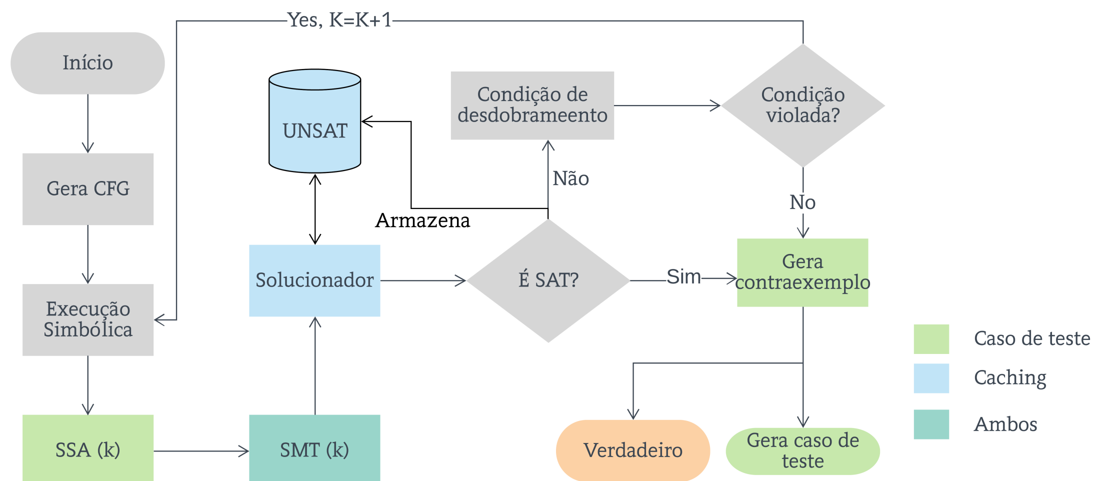


Figura 3.1: Arquitetura do método

3.1 Geração de Casos de Teste

A geração de casos de teste consiste em utilizar uma ferramenta de verificação de programas em busca de uma prova de violação, a partir desta prova é gerado um caso de teste. A Figura 3.2 contém como entrada uma prova de violação, em que os vértices representam o estado, as arestas representam mudanças no estado e o contexto são informações extras sobre o estado: arquitetura do computador, linha do código-fonte, e *thread*. Na figura, a partir de uma prova (A) todas as suas arestas são verificadas em busca atribuições não-determinísticas, extraindo seus valores (B) e depois disso, armazenado-os para a geração de caso de teste (C). Para isso foi desenvolvido um algoritmo, a fim de determinar todas as chamadas não-determinísticas de um programa e associá-las a um valor do contraexemplo. O algoritmo funciona usando uma Witness **W** e um código-fonte **S**.

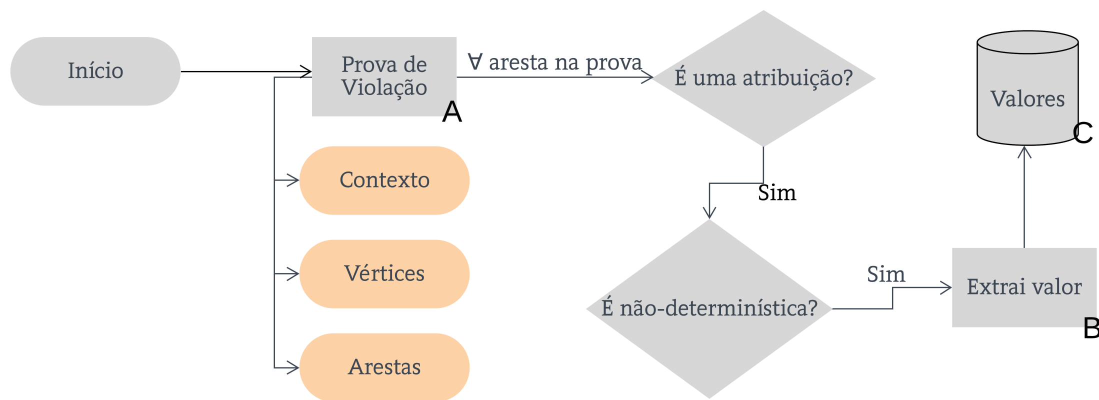


Figura 3.2: Fluxo da geração de um caso de teste

Algoritmo 3 Conversão de Witness

```
 1: função EXTRACTWITNESS(W, S)
 2:     Assumptions ← [∀a ∈ W.nodes, a.isAssumption()]
 3:     result ← []
 4:     para todo a ∈ Assumptions faça
 5:         se IsNondet(a, S) então
 6:             r ← simplify(a)
 7:             result ← result ⋃ r
 8:         fim se
 9:     fim para
10: fim função
```

O Algoritmo 3 contém a função `ExtractWitness`, que primeiramente extrai todas as *Assumptions* (linha 2) da Witness, já que contém metadados sobre linha e de atribuições de um programa. Em posse dessas Assumptions, o algoritmo busca no código-fonte por chamadas não-determinísticas (linha 5). O valor não-determinístico é extraído da Witness. Com essas informações, é possível construir o caso de teste.

Porém, alguns valores não-determinísticos ficam em funções intrínsecas. Por exemplo, `byte_extract_little_endian` é uma função que extrai um byte específico de um *array* de bits e precisa ser simplificada para gerar um valor para o caso de teste.

Algoritmo 4 *slicer* Modifications

```
 1: função ContainsNondet(expr)    ▷ Verifica se uma expressão contém um valor
    não-determinístico
 2:     se IsConstant(expr) então
 3:         retorne false
 4:     fim se
 5:     se IsNil(expr) então
 6:         retorne false
 7:     fim se
 8:     se IsSymbol(expr) então
 9:         se expr.name.contains("nondet$symex") então
10:             retorne true
11:         fim se
12:         retorne false
13:     fim se
14:     para todo e ∈ expr.sub_expr() faça
15:         se ContainsNondet(e) então
16:             retorne true
17:         fim se
18:     fim para
19:     retorne false
20: fim função
```

Diferentemente de provas, para a geração de teste é necessário que todos os valores não-determinísticos sejam setados para algum valor. Ao implementar o método, outra dificuldade observada foi os *slicers* que removem todas as expressões que não contribuem em provar uma violação, por isso se fez necessário modificar *slicers* de forma que o *slicer* não corte nenhuma expressão que contenha um símbolo não determinístico.

O Algoritmo 4 contém a função **ContainsNondet**, que dada uma expressão, checa se ela representa um símbolo não-determinístico (Linhas 8-12), caso não seja e a estrutura seja complexa (**struct**, **array**), então o algoritmo é aplicado recursivamente para cada um de seus membros (Linhas 14-18). Esse Algoritmo é utilizado em dois

lugares: (1) durante o *slicing* de atribuições; e (2) durante a geração do contra-exemplo, certificando-se de que nenhum valor não-determinístico fique oculto.

3.1.1 Demonstração

Para exemplificar o uso da técnica, vamos considerar o programa do Algoritmo 5. Nesse programa, são construídas três variáveis não-determinísticas: a, b, e c. A Witness original iria conter apenas a atribuição $a = 43$, sem os valores das outras variáveis. Com a modificação no *slicer*, a Witness resultante contém valores para os outros símbolos.

Algoritmo 5 Exemplo de geração de teste

```
1 int main() {
2   int a = nondet_int();
3   int b = nondet_int();
4   int c = nondet_int();
5   __ESBMC_assert(a == 42, "this will fail");
6   return 0;
7 }
```

O teste gerado contém os seguintes valores: <41,0,0>, que resultaria em um erro para esse programa.

3.2 Caching de fórmulas UNSAT

O *caching* consiste em armazenar fórmulas que são conhecidas como UNSAT, com o intuito de otimizar assertivas quando gerar a fórmula final para o SAT. Por exemplo, a Equação 3.1 é UNSAT pois não existe nenhum valor para X que satisfaça a fórmula, uma vez sabendo que a Equação 3.2 contém - em uma conjunção - todos os elementos da Equação 3.1, podemos dizer que ela é UNSAT.

$$(X < 0) \wedge (X > 0) \tag{3.1}$$

$$(X < 0) \wedge (X > 0) \wedge (X = 0) \tag{3.2}$$

Isso pode ser demonstrado com a intuição de que dada uma fórmula $\psi = P_0 \vee P_1 \wedge ... \wedge P_N$ onde $\exists i / 0 \leq i \leq N, P_i =$ Falso (F), portanto $\psi = P_0 \wedge P_1 \wedge ... \wedge F \wedge ... \wedge P_N$. Logo, $\psi = F$. Em BMC, essa propriedade pode ser utilizada para simplificar assertivas que contenham algum elemento que seja conhecido como F.

3.2.1 Simplificação de Assertivas

As assertivas são compostas por: *guardas*, *expressões* e *propriedades*. Inicialmente, um *HashMap* de guardas **T** é construído com intuito de identificar essas guardas, utilizando o algoritmo de *hashing* **H**. Em **T** o índice é o *hash* de uma guarda **G**, e seu valor é o conjunto de *hashes* das conjunções que compõem a guarda. A Tabela 3.1 contém as regras para geração do *HashMap* T.

Tabela 3.1: Regras de transoformação T

#	Regra	Transformação
1	$G \rightarrow e$	$T(G) \rightarrow \{H(e)\}$
2	$G \rightarrow \neg G$	$T(G) \rightarrow \{H(\neg G)\}$
3	$G \rightarrow G_1 \wedge G_2$	$T(G) \rightarrow T(G_1) \bigcup T(G_2)$
4	$G \rightarrow G_1 \vee G_2$	$T(G) \rightarrow \{H(G_1 \vee G_2)\}$
5*	$G \rightarrow G_1 \implies \psi$	$T(G) \rightarrow \{H(G_1), H(\neg\psi)\}$

Fonte: Própria

Para aplicar o caching, além de **T** também é necessário um conjunto **U** que contenha todos os conjuntos de *hashes* que são UNSAT. Uma vez com **T** e **U**, pode-se dizer que uma guarda **G** será UNSAT se e somente se:

$$\exists u \in U / u \subseteq T(G)$$

Com isso, é possível determinar que uma guarda é UNSAT, possibilitando simplificações na fórmula ao substituir a guarda por **F** e depois as simplificações naturais podem ser utilizadas. Ao final, as guardas de **T** que são assertivas serão enviadas ao solver que decidirá se elas são UNSAT. A Tabela 3.2 contém as regras pra determinar o conjunto de *hashes* que torna uma assertiva UNSAT (lembrando que uma assertiva é negada quando enviada ao solucionador).

Considerando o custo de checar $\exists u \in U / u \subseteq T(G)$ todas as vezes para cada guarda de um programa, foi implementado um LRU **L** (uma estrutura que de Cache que mantém elementos que foram recentemente usados) e um Bloom Filter **B** (uma estrutura probabilística que verifica se um elemento pertence a um conjunto). **L** armazena o resultado de UNSAT para guardas vistas recentemente, retornando **T** caso a guarda seja UNSAT (utilizando as informações de **U**) e **F** caso não seja. Já **B** armazena todas as guardas que Independentemente de contexto ou conjunção são UNSAT.

Tabela 3.2: Regras de geração U

#	Regra	Transformação
1	$A \to \neg e$	$\{h(e)\}$
2	$A \to \neg G$	$\{T(G)\}$
3	$A \to G \implies P$	$\{H(G), H(\neg P)\}$

Fonte: Própria

O Algoritmo 6 contém a função **IsUnsat**, que checa se uma guarda **G** é UNSAT utilizando: *HashMap* de guardas **T**, conjunto **U** de UNSAT, LRU **L** e Bloom Filter **B**. Inicialmente, busca-se no LRU pela guarda (linhas 2-4). Em seguida, busca-se todos os *hashes* que compõem a guarda no Bloom Filter (linhas 5-10). Caso alguma *hash* não esteja presente, é possível deduzir que não será possível determinar que a guarda é UNSAT. Finalmente, é feita a busca exaustiva pra determinar se a guarda é UNSAT ou não. Retornando *Verdadeiro* caso a guarda seja UNSAT, caso retorne *Falso* significa que não é possível determinar que a guarda é UNSAT. Para o cálculo de complexidade, assume-se que, no pior caso, **U** é implementado como uma matriz em que U_c é o número de colunas e U_r é o número de linhas, generalizando para um n: $\exists n, |T(G)| \leq n, U_c \leq n, U_r \leq n$. Portanto, a complexidade para o pior caso é $O(n^3)$.

3.2.1.1 Exemplo

Assumindo as seguintes guardas e assertivas para um sistema:

$$
\begin{aligned}
g_0 &\implies 1 == 0 \\
g_1 &\implies 2 > 7
\end{aligned}
$$

$$
\begin{aligned}
T(g_0) &= H(1 == 0) = x \\
T(g_1) &= H(2 > 7) = y
\end{aligned} \tag{3.3}
$$

$$
\begin{aligned}
a_0 &\implies g_0 \land g_1 \to \phi \\
F(a_0) &= T(g_0) \cup T(g_1) \cup H(\phi) = \{x, y, H(\phi)\}
\end{aligned}
$$

As guardas g_0 e g_1 são UNSAT, x e y são dois números naturais. Considerando o conjunto $U = \{\{x\}\}$, podemos deduzir que a assertiva a_0 é UNSAT.

Voltando para o programa visto anteriormente (Figura 2.6), adaptado para o

Algoritmo 6 Verificação de UNSAT

1: **função** IsUnsat(G, T, U, L, B) ▷ Checa se **G** é UNSAT
2:     **se** $G \in L$ **então** ▷ Checa LRU $O(1)$
3:         **retorne** $L[G]$
4:     **fim se**
5:     **para todo** $t \in T(G)$ **faça** ▷ Checa Bloom Filter $O(n)$
6:         **se** $t \notin B$ **então**
7:             $LRU[G] \leftarrow F$
8:             **retorne** F
9:         **fim se**
10:     **fim para**
11:     **para todo** $u \in U$ **faça** ▷ Checa em U $O(n^3)$
12:         **para todo** $t \in T(G)$ **faça**
13:             **se** $t \notin u$ **então** ▷ $T(G) \nsubseteq u$
14:                 **continue**
15:             **fim se**
16:         **fim para**
17:         $LRU[G] \leftarrow T$ ▷ $T(G) \subseteq u$
18:         **retorne** T
19:     **fim para**
20:     $LRU[G] \leftarrow F$
21:     **retorne** F
22: **fim função**

Algoritmo 7, ele contém uma assertiva na linha 7 que será desenrolada até a localização do erro.

Algoritmo 7 Programa com instâncias repetidas

```
1  int main() {
2      int x = 2;
3      int y = nondet_int();
4      __ESBMC_assume(y>0 && y<3);
5      while(nondet_int()) {
6          x = x*y;
7          __ESBMC_assert(x != 8, "");
8      }
9      return 0;
10 }
```

O Algoritmo 8 contém o SSA desse programa no *bound* 1. Na linha 14, o Assert é composto por uma guarda implicando na propriedade de segurança, a guarda representa a condição de entrar no loop. Considerando que o Assert não falha, ele é marcado como UNSAT, e na próxima iteração de *bound* 2, esse Assert será substituído pelo valor 1.

Algoritmo 8 K=1 (Adaptado)

```
Thread 0 file cache.c line 3 function main
ASSIGNMENT ()
y?1 == nondet_int$1
Thread 0 file cache.c line 4 function main
ASSUME
y?1 > 0 && y?1 < 3
Thread 0 file cache.c line 6 function main
ASSIGNMENT (HIDDEN)
goto_symex::guard?0 == !(nondet_int$2 == 0)
Thread 0 file cache.c line 6 function main
ASSIGNMENT ()
x?1 == 2 * y?1
Thread 0 file cache.c line 7 function main
ASSERT
\guard_exec?0!0 => (guard?0 => x?1 != 8)
```

3.2.2 Maximização de *Hits*

Levando em consideração como o BMC se comporta, o caching pode ser executado com vários objetivos:

1. Verificação de um programa com diferentes estratégias. Por exemplo, a exploração incremental gera as mesmas assertivas já vistas por k menores. Essa ideia também vale para validação de diferentes contextos de threads e induções.
2. Verificação de um programa com objetivos diferentes. Por exemplo, verificar outras propriedades de segurança em um mesmo programa: ponto de entrada diferente, alcançabilidade de outros alvos, etc.
3. Verificação de programas diferentes. Por exemplo, programas similares porém com estruturas e variáveis diferentes.

Os objetivos 1 e 2 são os mais triviais, pois a guarda já vai contar com os elementos e estruturas similares, além disso, 2 também está sujeito a pequenas mudanças no programa (como a adição de um *nested if*). Dentre eles, o objetivo 3 é mais difícil, pois outro programa não só vai ter estruturas diferentes, mas o contexto também. Para esse tipo de programa será necessário um processo de *canonização* (subseção 3.2.3)

3.2.3 Canonização

Durante o processo de Canonização, a fórmula será reescrita e reordenada com o objetivo de normalizar e facilitar a busca por fórmulas semelhantes. Os algoritmos são

adaptados do método utilizado pelo Green [Visser et al., 2012] para: **Reordenação**, **Rearranjo** e **Renomeio**. A etapa de reordenação coloca as variáveis em ordem alfabética; a de rearranjo coloca a fórmula em uma forma normal e, finalmente, a de renomeio renomeia as fórmulas com um novo nome baseado na sua posição atual.

O algoritmo de reordenação - mostrado pela Figura 3.4 - funciona lendo todas as expressões de um código SSA e obtendo uma lista de variáveis e constantes com a mesma precedência e depois substituindo de maneira ordenada, em que as variáveis são ordenadas de forma lexicográfica. A equação da Figura 3.3 contém um exemplo de como a reordenação funcionaria para uma expressão, os dois lados da relação e cada operação binária é aplicada de maneira recursiva para facilitar a implementação já que no ESBMC as expressões são implementadas a partir da utilização da notação polonesa.

$$\begin{gathered} c + a + b + 4 + d * b == b + a \\ 4 + a + b + c + b * d == a + b \end{gathered} \tag{3.4}$$

Figura 3.3: Exemplo de Reordenação

Para o Rearranjo em uma forma normal, todas as fórmulas devem manter o formato em qual o lado direito da equação é a constante 0 e as relações utilizadas são $[=, \neq, \leq]$. A conversão de inequações tem a limitação de funcionarem apenas para inteiros, pois consiste em adicionar a constate 1 em um dos lados da equação. Abaixo são definidas as regras de conversão, nessas regras, k e y representam constantes e as letras maiúsculas representam variáveis.

$$A + B + C + \ldots + k = y \Longrightarrow A + B + C + \ldots + (k - y) = 0 \tag{3.5}$$

$$A + B + C + \ldots + k \neq y \Longrightarrow A + B + C + \ldots + (k - y) \neq 0 \tag{3.6}$$

$$A + B + C + \ldots + k < y \Longrightarrow A + B + C + \ldots + (k - y + 1) \leq 0 \tag{3.7}$$

$$A + B + C + \ldots + k \leq y \Longrightarrow A + B + C + \ldots + (k - y) \leq 0 \tag{3.8}$$

$$A + B + C + \ldots + k > y \Longrightarrow -(A + B + C + \ldots + (k + y - 1)) \leq 0 \tag{3.9}$$

$$A + B + C + \ldots + k \geq y \Longrightarrow -(A + B + C + \ldots + (k + y - 1)) \geq 0 \tag{3.10}$$

Finalmente, na fase de renomeação, as variáveis são renomeadas com base na sua posição na fórmula, com o intuito de continuar gerando contra-exemplos corretos

```
 1: função REORDENAÇÃO(SSA)
 2:     para todo expr ∈ SSA faça
 3:         escolha expr.tipo faça
 4:             caso BinaryOperation                                  ▷ [+, *]
 5:                 ReordenarBinOp(expr)
 6:             caso Relation                          ▷ [==, !=, <, <=, >, >=]
 7:                 ReordenarRelation(expr)
 8:     fim para
 9: fim função
10:
11: função REORDENARBINOP(expr)
12:     Reordenação(expr.side1)
13:     Reordenação(expr.side2)
14:     values ← ValuesSamePrecedence(expr)
15:     sort(values)
16:     ReplaceSamePrecedence(values, expr)
17: fim função
18:
19: função REORDENARRELATION(expr)
20:     Reordenação(expr.side1)
21:     Reordenação(expr.side2)
22: fim função
```

Figura 3.4: Algoritmo utilizado para reordenação

é utilizado um mapa para salvar o nome antigo.

$$y + x < z \land x = z \land 10 + x > y \tag{3.11}$$

$$x + y < z \land x = z \land x + 10 > y \tag{3.12}$$

$$x + y - z + 1 \leq 0 \land x - z = 0 \land -x + y - 9 \leq 0 \tag{3.13}$$

$$V_0 + V_1 - V_2 + 1 \leq 0 \land V_0 - V_2 = 0 \land -V_0 + V_1 - 9 \leq 0 \tag{3.14}$$

3.3 Resumo

Neste capítulo, foi apresentado um método para converter *Witness* em casos de teste que pode ser aplicado em diversas ferramentas de verificação (como ESBMC e Map2Check) e um método de *caching* que é capaz de aproveitar fórmulas durante incrementos e de programas diferentes diminuindo o recurso necessário para análise.

O método para geração de casos de teste consistiu em usar a prova juntamente com o programa original para determinar onde os valores deveriam ser utilizados. Para

obtenção de resultados melhores, fez-se também uma modificação no *slicer* para que ele não gere provas incompletas. Também foi construído um *framework* para construção destes casos.

Para *caching*, o método consistiu em adaptar a técnica utilizada pelo Green para canonizar as expressões lógicas e, em seguida, verificar a satisfatibilidade dessa expressão canonizada através de *solvers*. A cada iteração, essa expressão é salva e as novas expressões são verificadas para checar se há repetição, cortando assertivas da fórmula, caso haja.

Capítulo 4

Trabalhos Relacionados

Neste capítulo serão apresentadas algumas das soluções atuais para geração automática de casos de teste e *caching* de soluções. Os trabalhos revisados abordam a descrição, comparação e aplicação das diferentes técnicas para geração usando *fuzzing* e execução simbólica.

4.1 Geração de Casos de Teste

Os trabalhos selecionados foram Klee, LibKluzzer e VeriFuzz, que são ferramentas que obtiveram pontuações altas em competições de teste.

O KLEE [Cadar et al., 2008] é uma ferramenta de execução simbólica capaz de

Tabela 4.1: Ferramentas de teste

Ferramenta	Execução Simbólica	White-box Fuzzing	Coverage guided fuzzing
VeriFuzz		×	×
KLEE	×		
LibKluzzer	×	×	×

Fonte: Própria

gerar casos de teste substituindo valores simbólicos e substituindo todas as computações que as usam em operações que usam valores simbólicos. Ao chegar em uma ramificação (*if* ou *loop*), o sistema segue os dois caminhos mantendo as restrições. Ao encontrar um *bug*, um caso de teste é gerado ao solucionar essas restrições.

Uma ferramenta que usa *fuzzing* é o VeriFuzz, que usa a ideia de fazer uma instrumentação inicial para analisar o comportamento do programa. Após essa execução, a informação é usada para gerar mais testes utilizando um algoritmo evolutivo com o critério de obter uma cobertura maior.

Finalmente, o LibKluzzer que é uma ferramenta híbrida que combina o KLUZZER (KLEE com extensões para *fuzzing*) com o libFuzzer. As extensões consistem em adições de *checkpoints* no motor de execução simbólica, onde é possível fazer várias execuções com entradas promissoras (usando cobertura como métrica).

Capítulo 5

Resultados Experimentais

Esta seção descreve o planejamento, projeto, execução e análise dos resultados de um estudo experimental para avaliar o método proposto neste trabalho, verificando a eficácia do método para programas em C. Os experimentos consistem num conjunto de *benchmarks* públicos e na suíte de regressão do ESBMC. Todos os *scripts* e ferramentas utilizados estão disponíveis de maneira *open-source* no GitHub[1].

5.1 Planejamento e projeto dos experimentos

Os experimentos tiveram o foco de analisar a *eficiência* de gerar casos de teste:

QP1: O método utilizado é capaz de gerar casos de teste de maneira eficiente e eficaz quando comparado com outras ferramentas?

QP2: O método de caching torna o ESBMC mais eficiente para geração de casos de teste?

Para a QP1, a forma mais direta é utilizando os *benchmarks* do Test-Comp. Eles consistem em um conjunto de tarefas de diversos domínios que são resolvidas por ferramentas comparadas pela quantidade de casos de teste confirmados e também por eficiência. Em especial, será usada a categoria **Cover-Error**, pois nem todas as ferramentas de verificação possuem suporte para cobertura nativamente. Para avaliação dos resultados, será feita a comparação com *Verifuzz* e *libKluzzer*, que receberam premiações nas edições de 2020 e 2021 na categoria **Cover-Error**. Foram feitos dois experimentos: o primeiro utilizando apenas o ESBMC e o segundo utilizando o FuSeBMC (com o mesmo método de geração de teste).

[1] `https://github.com/esbmc/esbmc/tree/caching2`

Já a QP2 é mais complexa de responder, pois ela não envolve só a performance de encontrar o erro. Existem casos como CI, no qual a mesma prova (ou similar) é repetida a cada *patch* e situações em que ao se repetir o experimento k vezes o estado de erro é alcançado. Ao salvar essas provas é possível fazer com que o ESBMC seja mais eficiente em recriar provas para pequenas alterações de código. Para responder essa questão, será utilizada a suíte de regressão usada pelo ESBMC, sendo executada três vezes: (1) sem cache; (2) gerando o cache inicial; (3) com cache pronto.

5.1.1 Descrição dos Benchmarks

A Tabela 5.1 contém todas as subcategorias de **Cover-Error** do Test-Comp com uma descrição do tipo de teste que está incluso. As colunas **2020** e **2021** se referem a quantidade de *benchmarks* para a subcategoria, totalizando 607 (2021) e 699 (2020). A suíte é pública e mantida através do *sv-benchmarks*[2]

O outro *benchmark* é a própria suíte do ESBMC, que contém cerca de 1600 programas em C que validam o suporte de diversas propriedades de segurança e estratégias suportadas. Infelizmente, o *benchmark* não conta com muitos testes para as estratégias incrementais e poucos testes nos quais o tempo do solver é longo, sendo o pior caso do *cache*.

5.1.2 Ambiente de execução

Para realização dos experimentos do Test-Comp será usada o `benchexec`[3], que é o programa utilizado para avaliar as ferramentas no Test-Comp e SV-COMP [Beyer, 2020a,b]. Esses experimentos foram realizados durante as competições de teste e os resultados estão disponíveis publicamente no site oficial[4].

Para os experimentos de regressão, foi utilizado o próprio ambiente de CI utilizado pelos desenvolvedores, **Github Actions**. Na ferramenta de teste foram adicionadas todas as *flags* necessárias para forçar a estratégia incremental e adicionar o caching. O experimento então funciona em 3 etapas: (1) execução da suíte; (2) execução da suíte e gerando o cache; (3) execução da suíte usando o cache. Para análise do tempo, foi utilizado o tempo que cada passo da *action* levou (informado pela API).

[2] `https://github.com/sosy-lab/sv-benchmarks`
[3] `https://github.com/sosy-lab/benchexec`
[4] `https://test-comp.sosy-lab.org/2021/results/results-verified/`

Tabela 5.1: Subcategorias do Test-Comp

Subcategoria	**Descrição**	**2020**	**2021**
ReachSafety-Arrays	Tarefas nas quais é necessário tratamento de *arrays* para determinar a alcançabilidade	30	100
ReachSafety-BitVectors	Tarefas nas quais é necessário tratamento de operações de bit para determinar a alcançabilidade	10	10
ReachSafety-ControlFlow	Programas que dependem de controle de fluxo e variáveis inteiras	8	32
ReachSafety-ECA	Programas que representam sistemas de Event Condition Action (ECA)	412	18
ReachSafety-Floats	Tarefas que contêm aritmética de ponto flutuante	32	33
ReachSafety-Heap	Tarefas que contêm análise em estruturas sobre o *heap* como estruturas de dados e ponteiros	54	57
ReachSafety-Loops	Tarefas onde a análise do loop é necessária	30	158
ReachSafety-Recursive	Tarefas nas quais a análise de recursão é necessária.	16	20
ReachSafety-Sequentialized	Contém programas concorrentes sequencializados com a adição de um *scheduler*	107	107
ReachSafety-XCSP	Tarefas originadas do benchamark XCSP_to_C	-	59
SoftwareSystems-BusyBox-MemSafety	Tarefas vindas do Software BusyBox.	-	11
SoftwareSystems-DeviceDriversLinux64-ReachSafety	Tarefas que requerem análise de ponteiros e ponteiros de função.	-	2

Adaptado de [Beyer, 2020b]

5.2 Resultados dos Experimentos

Os experimentos envolvendo geração de testes foram realizados durante as competições de teste Test-Comp'20 e Test-Comp'21 [Beyer, 2020b, 2021], na Categoria *Cover-Error*, que consiste em alcançar uma localização de erro. O ESBMC obteve terceiro lugar no Test-Comp'20 e, quando combinado com outros métodos (FuSeBMC), obteve primeiro lugar (com 80% dos casos vindos do ESBMC) [Alshmrany et al., 2021] no Test-Comp'21. Além disso, no Test-Comp'21 o FuSeBMC participou em outras categorias: *Cover-Branches* (obtendo quarto lugar), e *Overall* (obtendo o terceiro lugar). A Tabela 5.2 e a Tabela 5.3 contêm os resultados das edições 21 e 20, respectivamente.

No Test-Comp'20 foram observadas algumas vantagens em utilizar BMC para geração de testes em categorias específicas. Em especial, em categorias que dependem de fluxo e *loops* (incluindo recursão) o método se saiu muito bem, isso se deve principalmente às estratégias e execução simbólica que fornecem uma forma de fazer *reasoning* sobre essas estruturas. No total, o ESBMC gerou treze (13) casos de teste que não foram confirmados. Desses: onze (11) se deram por limitações no *script* que converte a *Witness*; um (1) se deu por uma limitação do ESBMC com construção de provas a partir de *arrays*[5]; um (1) se deu por um *undefined behaviour* no *benchmark*, em que os valores de um array eram usados antes de ser inicializado (um PR já foi submetido e aceito[6]). Em geral, pode-se observar que os piores resultados foram em *Arrays* (pelo *bug*) e *ECA* (por ser uma categoria que o ESBMC tem dificuldades).

Já no Test-Comp'21 o método foi utilizado através do FuSeBMC que, em resumo, combina BMC e técnicas de fuzzing para gerar casos de teste [Alshmrany et al., 2021]. No FuSeBMC, o motor BMC é o ESBMC, que é executado diversas vezes sob diferentes configurações, gerando múltiplos casos de teste para o mesmo problema. O FuSeBMC, por se utilizar também de análise dinâmica, foi capaz de aumentar o número de testes consideravelmente e, assim, obtendo o primeiro lugar. A principal fraqueza do método está na *engine* de *fuzzing* que ainda não está otimizada o suficiente para lidar com programas complexos, pois não conta com técnicas como *slicing* e nem *reasoning* sobre *loops*.

A competição é capaz de responder a **QP1**, mostrando que o método se equipara ao estado da arte em eficiência e eficácia. O método pôde ser integrado a outras ferramentas de verificação como o Map2Check e utilizando-se de *fuzzing* foi capaz de alcançar os tipos de erros difíceis para BMC.

[5] `https://github.com/esbmc/esbmc/pull/362`
[6] `https://github.com/sosy-lab/sv-benchmarks/pull/1073`

Tabela 5.2: Test-Comp'21

Subcategoria	**FuSeBMC**	**LibKluzzer**	**VeriFuzz**
ReachSafety-Arrays	93 (19000s)	96 (29000s)	95 (500s)
ReachSafety-BitVectors	10 (1400s)	9 (8100s)	9 (500s)
ReachSafety-ControlFlow	8 (950s)	11 (7500s)	9 (380s)
ReachSafety-ECA	8 (1400s)	11 (9900s)	16 (1600s)
ReachSafety-Floats	32 (4500s)	30 (27000s)	30 (490s)
ReachSafety-Heap	45 (2300s)	47 (29000s)	47 (280s)
ReachSafety-Loops	131 (32000s)	138 (120000s)	136 (2900s)
ReachSafety-Recursive	19 (910s)	17 (1500s)	13 (310s)
ReachSafety-Sequentialized	101 (13000s)	83 (7500s)	99 (2100s)
ReachSafety-XCSP	53 (3100s)	3 (2700s)	25 (540s)
SoftwareSystems-BusyBox-MemSafety	0 (0s)	0 (0s)	0 (0s)
SoftwareSystems-DeviceDriversLinux64-ReachSafety	0 (0s)	0 (0s)	0 (0s)
Pontuação (normalizada)	405 (78000s)	359 (320000s)	385 (9300s)

Adaptado de [Beyer, 2020b]

Tabela 5.3: Test-Comp'20

Subcategoria	**ESBMC**	**LibKluzzer**	**VeriFuzz**
ReachSafety-Arrays	7 (180s)	26 (23000s)	24 (110s)
ReachSafety-BitVectors	8 (3.8s)	10 (9000s)	10 (620s)
ReachSafety-ControlFlow	5 (3.1s)	8 (7200s)	7 (580s)
ReachSafety-ECA	254 (32000s)	288 (26000s)	376 (53000s)
ReachSafety-Floats	30 (980s)	29 (26000s)	29 (780s)
ReachSafety-Heap	38 (120s)	51 (33000s)	50 (320s)
ReachSafety-Loops	23 (19s)	28 (21000s)	27 (730s)
ReachSafety-Recursive	16 (350s)	16 (1400s)	15 (1500s)
ReachSafety-Sequentialized	89 (840s)	81 (73000s)	100 (4800s)
Pontuação (normalizada)	506 (34000 s)	630 (470000 s)	636 (63000 s)

Adaptado de [Beyer, 2021]

Os resultados da suíte do ESBMC foram variados. Em especial, programas incrementais com tempo de solução alto tiveram os maiores ganhos, pois instâncias que demoravam vários minutos não precisaram ser analisadas novamente. O tempo de execução da suíte foi de 57 minutos (sem cache), 54 minutos (com cache incremental), e 45 minutos (com cache salvo). Foi observado, porém, que nos casos em que a solução do problema é muito rápida (<100ms), o uso do *caching* deixou o programa mais lento, enquanto para situações em que o tempo de solução foi longo ($>$ 1s), o cache foi eficiente.

Quanto a **QP2**; os experimentos mostraram que é situacional, mesmo para programas que contêm as mesmas assertivas, nem sempre há um ganho grande, pois o tempo de resolução as vezes é menor do que o ganho com caching. Entretanto, em situações em que o tempo do solver é longo, o ganho é significativo. Por exemplo, em situações com incrementos pequenos o uso do cache foi capaz de reduzir entre 50-90% do tempo total de análise.

5.3 Resumo

Neste capítulo foi apresentada uma análise experimental sobre o método proposto em duas ferramentas: FuSeBMC e ESBMC. Para geração de casos de teste, os experimentos foram realizados no TestComp'20 e TestComp'21 em que o método rendeu premiações na categoria **CoverError**, consistindo em gerar um caso de teste que leve a uma localização de erro.

O *caching* foi testado sob a regressão do ESBMC e percebeu-se que em situações onde o tempo de solucionar a expressão lógica é longo, o tempo de análise para execuções posteriores diminui. No uso de casos de teste com incrementos pequenos ou com limitações no ambiente o *caching* consegue diminuir a quantidade de tempo/espaço necessário para a análise do programa.

Capítulo 6

Conclusões e Trabalhos Futuros

Nesta proposta, foram abordados os problemas de geração de casos de teste com BMC e caching de soluções. Esses problemas pertencem a diferentes etapas do fluxo do BMC que dificultam a utilização do método durante o desenvolvimento.

Em BMC, um sistema é avaliado sob uma propriedade de segurança e, em seguida, é sistematicamente explorado em busca de provar (ou não) a violação do mesmo. Caso seja encontrada uma violação, o algoritmo irá produzir uma prova contendo estados e valores que fizeram com que o sistema alcançasse um estado de erro. Porém, resolver uma instância de BMC tem um custo de tempo e memória alto (exponencial), portanto são feitas simplificações (slicing) com o intuito de otimizar a busca da solução. Essas otimizações fazem com que as provas não possam ser utilizadas diretamente para gerar um caso de teste, pois alguns valores e estados serão suprimidos.

O processo de *slicing* consiste na remoção de valores que não são usados em nenhum tipo de restrição. Para solucionar esta situação, foi proposta uma variação do algoritmo de *slicing*, em que variáveis não-determinísticas não são removidas da análise. Isso faz com que a prova final contenha todos os valores que, mesmo não impactando na alcançabilidade do erro, são usados para gerar um caso de teste.

Durante a extração dos valores das provas, muitos destes ainda necessitaram ser computados. Isso ocorreu pois a prova é feita em um código intermediário que suporta computações mais simples para análise. Essas operações envolvem manipulação de pontos flutuantes e operações bit a bit. O método de gerar teste consistiu em utilizar as provas geradas como base para criação do teste, adaptando as chamadas não-determinísticas e modificando o *slicer* para manter casos replicáveis. O método foi integrado no ESBMC e FuSeBMC, obtendo premiações na competição TestComp.

Outro problema que afeta o BMC é que a cada iteração o programa é convertido em uma fórmula lógica. Essas fórmulas podem ser repetidas nas próximas iterações,

resultando na mesma computação sendo realizada sucessivamente. Para solucionar esse problema foi proposto um algoritmo de *caching* capaz de reconhecer e remover partes da fórmulas que já são conhecidas. Isso elimina computações repetidas.

Para o caching de soluções, foi desenvolvido um algoritmo capaz de identificar uma composição de guardas que são UNSAT. Para isso, foi utilizado o princípio de que se uma premissa em uma fórmula na forma normal conjuntiva for UNSAT, a expressão inteira será UNSAT. Ou seja, a partir de cache vazio, as iterações de BMC em que uma fórmula resultar em UNSAT serão armazenadas para serem aproveitadas.

O algoritmo foi testado sob a regressão do ESBMC, obtendo resultados significantes (50%-90%) para fórmulas cujo tempo de solução eram maiores que 1s. O uso do *caching*, porém, cai no dilema de saber o que seria mais rápido: solucionar a fórmula diretamente ou analisar a fórmula em busca de *matches*.

Respondendo o problema inicial, o *caching* pode ser utilizado para diminuir o custo para solucionar novas instâncias, gerando assim, casos de teste para programas mais complexos, principalmente se o programa tiver um custo de solução alto.

Além do método proposto, ficam como sugestões de trabalhos futuros:

- Integrar outros métodos de *caching* ao *framework* como Julia [Aquino et al., 2017] e GreenTrie [Jia et al., 2015] para melhorar a eficiência do método.
- Melhorar o sistema de armazenamento na memória secundária e construir uma codificação mais eficiente para serialização das guardas, como utilização de ProtocolBuffers[1].
- Uso de técnicas de compilação (como *loop-unrolling* para otimizar a identificação de erros. Em especial, foi observado um ganho significativo para *arrays*.
- Otimizar o algoritmo com utilização de paralelismo e concorrência durante a aplicação dos algoritmos sobre a representação intermediária.

[1] `https://developers.google.com/protocol-buffers`

Referências Bibliográficas

Alaqail, H. & Ahmed, S. (2018). Overview of software testing standard iso/iec/ieee 29119. *International Journal of Computer Science and Network Security (IJCSNS)*, 18(2):112--116.

Ali, S. & Yue, T. (2015). Formalizing the iso/iec/ieee 29119 software testing standard. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 396--405. IEEE.

Aljaafari, F.; Cordeiro, L. C.; Mustafa, M. A. & Menezes, R. (2021). EBF: A hybrid verification tool for finding software vulnerabilities in iot cryptographic protocols. *CoRR*, abs/2103.11363.

Alshmrany, K. M.; Menezes, R. S.; Gadelha, M. R. & Cordeiro, L. C. (2021). Fusebmc: A white-box fuzzer for finding security vulnerabilities in C programs (competition contribution). In *24th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 12649 of *LNCS*, pp. 363--367.

Aquino, A.; Denaro, G. & Pezzè, M. (2017). Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 427--437. IEEE.

Baier, C. & Katoen, J.-P. (2008). *Principles of Model Checking (Representation and Mind Series)*. The MIT Press. ISBN 026202649X, 9780262026499.

Beyer, D. (2020a). Advances in automatic software verification: Sv-comp 2020. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 347--367. Springer.

Beyer, D. (2020b). Second competition on software testing: Test-comp 2020. In *FASE*, pp. 505--519.

Beyer, D. (2021). Status report on software testing: Test-comp 2021. *Fundamental Approaches to Software Engineering*, 12649:341.

Beyer, D. & Lemberger, T. (2017). Software verification: Testing vs. model checking. In *Haifa Verification Conference*, pp. 99--114. Springer.

Cadar, C.; Dunbar, D. & Engler, D. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pp. 209--224. USENIX Association.

Cadar, C.; Godefroid, P.; Khurshid, S.; Păsăreanu, C. S.; Sen, K.; Tillmann, N. & Visser, W. (2011). Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 1066--1071, New York, NY, USA. ACM.

Cadar, C. & Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82--90. ISSN 0001-0782.

Chaves, L. C.; Bessa, I.; Ismail, H.; dos Santos Frutuoso, A. B.; Cordeiro, L. C. & de Lima Filho, E. B. (2018). Dsverifier-aided verification applied to attitude control software in unmanned aerial vehicles. *IEEE Trans. Reliab.*, 67(4):1420--1441.

Chen, T. Y.; Cheung, S. C. & Yiu, S. M. (2020). Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543*.

Cheng, W.; Zhao, Q.; Yu, B. & Hiroshige, S. (2006). Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *11th IEEE Symposium on Computers and Communications (ISCC'06)*, pp. 749–754. ISSN 1530-1346.

Chong, N.; Cook, B.; Eidelman, J.; Kallas, K.; Khazem, K.; Monteiro, F. R.; Schwartz-Narbonne, D.; Tasiran, S.; Tautschnig, M. & Tuttle, M. R. (2021). Code-level model checking in the software development workflow at amazon web services. *Software: Practice and Experience*, 51(4):772--797.

Chowdhury, A. B.; Medicherla, R. K. & Venkatesh, R. (2019). Verifuzz: Program aware fuzzing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 244--249. Springer.

Clarke, E. M.; Emerson, E. A. & Sifakis, J. (2009). Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74--84. ISSN 0001-0782.

Cook, B.; Khazem, K.; Kroening, D.; Tasiran, S.; Tautschnig, M. & Tuttle, M. R. (2018). Model checking boot code from aws data centers. In Chockler, H. & Weissenbacher, G., editores, *Computer Aided Verification*, pp. 467--486, Cham. Springer International Publishing.

Cordeiro, L.; Fischer, B. & Marques-Silva, J. (2012). Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38(4):957–974. ISSN 0098-5589.

Cordeiro, L.; Kesseli, P.; Kroening, D.; Schrammel, P. & Trtik, M. (2018). JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *CAV*, volume 10981 of *LNCS*, pp. 183--190.

Cordeiro, L. C. (2010). Smt-based bounded model checking for multi-threaded software in embedded systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 373--376. ACM.

Cordeiro, L. C.; Kroening, D. & Schrammel, P. (2019). JBMC: bounded model checking for java bytecode - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 11429 of *LNCS*, pp. 219--223. Springer.

Cousot, P. (2001). Abstract interpretation based formal methods and future challenges. In *Informatics*, pp. 138--156. Springer.

Cousot, P. & Cousot, R. (1977). 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*, pp. 238--252. ACM.

Cousot, P.; Cousot, R.; Feret, J.; Mauborgne, L.; Miné, A.; Monniaux, D. & Rival, X. (2005). *The ASTREÉ Analyzer*, pp. 21--30. Springer Berlin Heidelberg, Berlin, Heidelberg.

de Bessa, I. V.; Ismail, H. I.; Cordeiro, L. C. & Filho, J. E. C. (2014). Verification of delta form realization in fixed-point digital controllers using bounded model checking. In *2014 Brazilian Symposium on Computing Systems Engineering (SBESC)*, pp. 49--54. IEEE Computer Society.

Ding, Z.; Zhang, K. & Hu, J. (2008). A rigorous approach towards test case generation. *Inf. Sci.*, 178(21):4057--4079. ISSN 0020-0255.

D'silva, V.; Kroening, D. & Weissenbacher, G. (2008). A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165--1178.

Ernst, M. D.; Perkins, J. H.; Guo, P. J.; McCamant, S.; Pacheco, C.; Tschantz, M. S. & Xiao, C. (2007). The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35--45.

Gadelha, M. R.; Menezes, R. S. & Cordeiro, L. C. (2020a). Esbmc 6. 1: automated test case generation using bounded model checking. *INTERNATIONAL JOURNAL ON SOFTWARE TOOLS FOR TECHNOLOGY TRANSFER*.

Gadelha, M. R.; Monteiro, F. R.; Morse, J.; Cordeiro, L. C.; Fischer, B. & Nicole, D. A. (2018a). ESBMC 5.0: An industrial-strength C model checker. In *ASE*, pp. 888--891.

Gadelha, M. Y. R.; Ismail, H. I. & Cordeiro, L. C. (2017). Handling loops in bounded model checking of C programs via k-induction. *Int. J. Softw. Tools Technol. Transf.*, 19(1):97--114.

Gadelha, M. Y. R.; Menezes, R.; Monteiro, F. R.; Cordeiro, L. C. & Nicole, D. A. (2020b). ESBMC: scalable and precise test generation based on the floating-point theory - (competition contribution). In *23rd International on Fundamental Approaches to Software Engineering (FASE)*, volume 12076 of *LNCS*, pp. 525--529.

Gadelha, M. Y. R.; Monteiro, F. R.; Cordeiro, L. C. & Nicole, D. A. (2019). ESBMC v6.0: Verifying C programs using k-induction and invariant inference - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, volume 11429 of *LNCS*, pp. 209--213.

Gadelha, M. Y. R.; Monteiro, F. R.; Morse, J.; Cordeiro, L. C.; Fischer, B. & Nicole, D. A. (2018b). ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pp. 888--891. ACM.

Günther, H. & Weissenbacher, G. (2014). Incremental Bounded Software Model Checking. In *SPIN*, pp. 40--47.

Hoder, K.; Kovács, L. & Voronkov, A. (2011). Case studies on invariant generation using a saturation theorem prover. In *Mexican International Conference on Artificial Intelligence*, pp. 1--15. Springer.

Hooker, J. N. (1993). Solving The Incremental Satisfiability Problem. *The Journal of Logic Programming*, 15(1):177--186.

Jia, X.; Ghezzi, C. & Ying, S. (2015). Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 177--187. ACM.

Jiang, B.; Zhang, Z.; Chan, W. K. & Tse, T. H. (2009). Adaptive random test case prioritization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 233–244.

Khurshid, S.; PĂsĂreanu, C. S. & Visser, W. (2003). *Generalized Symbolic Execution for Model Checking and Testing*, pp. 553--568. Springer Berlin Heidelberg, Berlin, Heidelberg.

Menezes, R.; Rocha, H.; Cordeiro, L. C. & Barreto, R. S. (2018). Map2check using LLVM and KLEE - (competition contribution). In *24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10806 of *LNCS*, pp. 437--441.

Monteiro, F. R.; Garcia, M.; Cordeiro, L. C. & de Lima Filho, E. B. (2017). Bounded model checking of C++ programs based on the qt cross-platform framework. *Softw. Test. Verification Reliab.*, 27(3).

Morse, J.; Cordeiro, L. C.; Nicole, D. A. & Fischer, B. (2013). Handling unbounded loops with ESBMC 1.20 - (competition contribution). In *19th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *LNCS*, pp. 619--622.

Nethercote, N. & Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pp. 89--100, New York, NY, USA. ACM.

O'Regan, G. (2017). *Concise guide to software engineering*. Springer.

Pacheco, C.; Lahiri, S. K.; Ernst, M. D. & Ball, T. (2007). Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pp. 75–84.

Paraskevopoulou, Z.; HriŢcu, C.; Dénès, M.; Lampropoulos, L. & Pierce, B. C. (2015). Foundational property-based testing. In Urban, C. & Zhang, X., editores, *Interactive Theorem Proving*, pp. 325--343, Cham. Springer International Publishing.

Pereira, P. A.; Albuquerque, H. F.; Marques, H.; da Silva, I.; Carvalho, C. B.; Cordeiro, L. C.; Santos, V. & Ferreira, R. (2016). Verifying CUDA programs using smt-based context-bounded model checking. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM.

Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edição. ISBN 0072496681.

Ramalho, M.; Freitas, M.; Sousa, F. R. M.; Marques, H.; Cordeiro, L. C. & Fischer, B. (2013). SMT-Based Bounded Model Checking of C++ Programs. In *ECBS*, pp. 147--156.

Reid, S. (2013). Iso/iec/ieee 29119: the new international software testing standards. *Testing Solutions Group*.

Rocha, H.; Barreto, R. S. & Cordeiro, L. C. (2015a). Memory Management Test-Case Generation Of C Programs Using Bounded Model Checking. In *SEFM*, volume 9276 of *LNCS*, pp. 251--267.

Rocha, H.; Barreto, R. S.; Cordeiro, L. C. & Neto, A. D. (2012). Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In *9th International Conference Integrated Formal Methods (IFM)*, volume 7321 of *LNCS*, pp. 128--142.

Rocha, H.; Ismail, H.; Cordeiro, L. C. & Barreto, R. S. (2015b). Model checking embedded C software using k-induction and invariants. In *2015 Brazilian Symposium on Computing Systems Engineering, SBESC*, pp. 90--95. IEEE Computer Society.

Rocha, H.; Menezes, R.; Cordeiro, L. C. & Barreto, R. S. (2020). Map2check: Using symbolic execution and fuzzing - (competition contribution). In *26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12079 of *LNCS*, pp. 403--407.

Rocha, H. O. (2015). Verificação de sistemas de software baseada em transformações de código usando bounded model checking.

Rocha, H. O.; Barreto, R. S. & Cordeiro, L. C. (2016). Hunting memory bugs in C programs with Map2Check. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 934--937. Springer.

Roger, S. P. (2019). *Software engineering: a practitioner's approach*. McGraw-Hill Education.

Serebryany, K. (2016). Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pp. 157–157.

Visser, W.; Geldenhuys, J. & Dwyer, M. B. (2012). Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 58. ACM.

Whittemore, J.; Kim, J. & Sakallah, K. (2001). SATIRE: A New Incremental Satisfiability Engine. In *DAC*, pp. 542--545.

Wunderlich, H.-J. (1990). Multiple distributions for biased random test patterns. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):584–593.

Zhou, Z. Q.; Huang, D.; Tse, T.; Yang, Z.; Huang, H. & Chen, T. (2004). Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, pp. 346--351. Software Engineers Association Xian, China.

Anexo A

Publicações

A.1 Relacionadas

- Gadelha, M., **Menezes, R.**, Monteiro, F., Cordeiro, L., Nicole D. *ESBMC: Scalable and Precise Test Generation based on the Floating-Point Theory.* In International Conference on Fundamental Approaches to Software Engineering, 2020. **(Publicado)** [Gadelha et al., 2020b]

- Gadelha, M., **Menezes, R.**, Cordeiro, L. *ESBMC 6. 1: automated testcase generation using bounded model checking.* In INTERNATIONAL JOURNAL ONSOFTWARE TOOLS FOR TECHNOLOGY TRANSFER, 2020. **(Publicado)** [Gadelha et al., 2020a]

- Rocha, H., **Menezes, R.**, Cordeiro, L., Barreto, R. *Map2Check: Using Symbolic Execution and Fuzzing.* In Tools and Algorithms for the Construction and Analysis of Systems, 2020. **(Publicado)** [Rocha et al., 2020]

- Alshmrany, K., **Menezes, R.**, Gadelha, M., Cordeiro, L. *FuSeBMC: A white-box fuzzer for finding security vulnerabilities in C programs (competition contribution).* In Fundamental Approaches to Software Engineering, 2021. **(Publicado)** [Alshmrany et al., 2021]

A.2 Contribuições em outras pesquisas

- Aljaafari, F., **Menezes, R.**, Mustafa, M., Cordeiro, L. *Finding Security Vulnerabilities in IoT Cryptographic Protocol and Concurrent Implementations.* In

INTERNATIONAL SYMPOSIUM ON AUTOMATED TECHNOLOGY FOR VERIFICATION AND ANALYSIS, 2021. **(Submetido)** [Aljaafari et al., 2021]