UNIVERSIDADE FEDERAL DE RORAIMA

PRÓ-REITORIA DE ENSINO E EXTENSÃO

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

VICTOR DELUCA ALMIRANTE GOMES

Uma Abordagem para Geração de Invariantes de Programas Baseado em Templates para Verificação Eficiente de Programas em C

Boa Vista - RR

2019

VICTOR DELUCA ALMIRANTE GOMES

Uma Abordagem para Geração de Invariantes de Programas Baseado em Templates para Verificação Eficiente de Programas em C

Trabalho de conclusão de curso na área de Verificação de Software apresentada ao Departamento de Ciência da Computação da Universidade Federal de Roraima como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: DSc. Herbert Oliveira Rocha

Boa Vista - RR

2019

Dados Internacionais de Catalogação na publicação (CIP)
Biblioteca Central da Universidade Federal de Roraima

G633a Gomes, Victor Deluca Almirante.

Uma abordagem para geração de invariantes de programas baseado em templates para verificação eficiente de Programas em C / Victor Deluca Almirante Gomes. – Boa Vista, 2019.

53 f. : il.

Orientador: Prof. Dr. Herbert Oliveira Rocha.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Roraima, Curso de Ciência da Computação.

1 - Invariantes de programas. 2 - Pré e pós condições. 3 - Verificação de programas em C. I - Título. II - Rocha, Herbert Oliveira (orientador).

CDU - 681.3.066

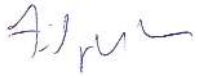
Ficha Catalográfica elaborada pela Bibliotecária/Documentalista:
Maria de Fátima Andrade Costa - CRB-11/453-AM

VICTOR DELUCA ALMIRANTE GOMES

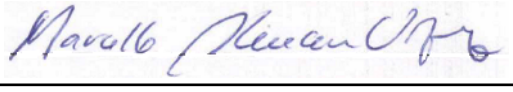
Uma Abordagem para Geração de Invariantes de Programas Baseado em Templates para Verificação Eficiente de Programas em C

Trabalho de conclusão de curso na área de Verificação de Software apresentada ao Departamento de Ciência da Computação da Universidade Federal de Roraima como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação
Defendido em 10 de julho de 2019 e aprovado pela seguinte banca examinadora:

Prof. DSc. Herbert Oliveira Rocha
Orientador / Curso de Ciência da Computação - UFRR



Prof. Msc. Filipe Dwan Pereira
Curso de Ciência da Computação - UFRR



Prof. Dsc. Marcelle Alencar Urquiza
Curso de Ciência da Computação - UFRR

"Enjoy the pain, it's yours for a while, girl; Don't run away, your fear, you're getting older; We are relying on you and your emotion; And as you go, promise of being bright" –Veela.

RESUMO

A verificação de software é parte vital do processo de desenvolvimento de software, especialmente em sistemas críticos onde erros podem ter consequências catastróficas. Contudo, verificar a corretude de um programa é um processo complexo: Há um *trade-off* enorme entre velocidade e precisão, e diversos trabalhos no meio acadêmico buscam formas de equilibrar ambos os aspectos. Neste trabalho, é apresentado um método para a melhoria da velocidade dos métodos de verificação formal sem perda de precisão, através do uso de invariantes geradas por *templates*, que representam de forma precisa a natureza do programa, facilitando o processo de verificação. O método é dividido em três abordagens diferentes, as quais são avaliadas por meio de um conjunto de programas incluindo *benchmarks* bem conceituados na área, apresentando resultados positivos, com uma acurácia de até 83%.

Palavras-chaves: Invariantes de Programas; Pré e Pós Condições; Verificação de Programas em C.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Atualmente, muitas companhias e organizações estão rotineiramente lidando com software de diferentes complexidades e que contêm milhares de linhas de código, escritos por diferentes pessoas, e que usam diferentes ferramentas e estilos (HODER et al., 2011). Adicionalmente, estes sistemas de software, devido ao curto espaço de tempo de liberação do produto ao mercado, precisam ser desenvolvidos rapidamente e atingir um alto nível de qualidade. Porém, os programadores cometem enganos. Um exemplo ocorre quando um programador acidentalmente escreve um dado requisito do sistema de forma incorreta, como a alteração de uma atribuição de $x \leq 5$ para $x < 5$(ROCHA, 2015). Bastante tempo e esforço são gastos em encontrar e corrigir estes erros (GUPTA; RYBALCHENKO, 2009).

Por exemplo, um erro de cálculo da dose de radiação no Instituto Nacional de Oncologia do Panamá resultou na morte de 23 pacientes (WONG et al., 2010). As aplicações de software em geral requerem qualidade na sua execução, mas essa necessidade é ainda maior em sistemas embarcados, os quais são usados em diversas aplicações sofisticadas, exemplo: eletrodomésticos, aviônicas, e aparelhos médicos. A flexibilidade exigida em tais aplicações aumenta significativamente o número de funções que são implementadas no software, em vez de no hardware (ROCHA, 2015).

Segundo Bensalem e Lakhnech (1999), model checking é uma técnica baseada em métodos formais conhecidos para provar propriedades de programas reativos, e tem sido amplamente adotado para validação de aplicações críticas (HODER et al., 2010; ROCHA et al., 2010; CORDEIRO et al., 2009; MERZ et al., 2012). Esta técnica gera uma busca exaustiva no espaço de estados do modelo para determinar se uma dada propriedade é válida ou não (BAIER; KATOEN, 2008). A principal razão para o sucesso da técnica model checking é que ela funciona de forma completamente automática. Contudo, a técnica model checking ainda possui alguns desafios, como, por exemplo, lidar com a explosão do espaço de estados do modelo (GRUMBERG; VEITH, 2008). Logo, os model checkers podem sofrer de falta de memória ou mesmo uma verificação incompleta. Uma das razões é devido ao fato de que eles manipulam estruturas complexas. Dois exemplos são:

- Os loops aninhados, onde tem-se cada iteração desenrolada, além da análise das estruturas contidas em suas possíveis intersecções;
- Múltiplos chaveamentos de contexto, que requerem uma análise de cada *interleaving* gerado, bem como suas iterações.

Visando contribuir com a verificação de sistemas computacionais, o contexto deste trabalho está situado no uso de metodologias e técnicas de verificação formal, focando principalmente

em *model checking* e técnicas baseadas em *templates* (combinações booleanas de desigualdades lineares sobre variáveis de programa com coeficientes mantidos como parâmetros para computar automaticamente invariantes de programas (HODER et al., 2011), pré- e pós-condições de programas). Segundo Fouladgar et al. (2011), invariantes em programas são fórmulas ou regras que surgem a partir do código fonte de um programa e permanecem únicas e inalteradas em relação à fase de execução de um programa com diferentes parâmetros.

Este trabalho tem seu foco especificamente na verificação de propriedades de segurança de programas (por exemplo, segurança de vetores), ou seja, testes de propriedades do programa que garantam que o programa não entre em um estado de erro ou bloqueio. O escopo deste trabalho é restrito à análise e verificação de códigos na linguagem de programação C. A motivação deste trabalho está em aprimorar o processo de verificação de um model checker, reduzindo o modelo do programa a ser verificado pela utilização de restrições para guiar a exploração dos estados, uma vez que devido as características do programa o número de estados do modelo pode crescer exponencialmente, por exemplo, em programas com estruturas não determinísticas em loops aninhados.

Visando exemplificar a motivação deste trabalho, no que trata do problema da explosão de estados sofridos pelos model checkers, será usado o exemplo apresentado por Rocha (2015) de um programa extraído do benchmark do SV-COMP (BEYER, 2013) apresentado na Figura 1, onde a variável a é uma constante inteira e observe que as variáveis i e sn são declaradas com um tipo maior do que o tipo da variável n para evitar *overfow* aritmético.

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

Figura 1 – Programa de soma do SV-COMP.

Matematicamente, compreende-se o código do programa como a implementação da soma dada pela seguinte equação:

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

No programa da Figura 1, a propriedade (representada pela assertiva na linha 10) deve ser verdadeira para qualquer valor de n (isto é, para qualquer desdobramento do programa). Contudo, as técnicas de *Bounded Model Checking* (BMC) têm dificuldades em provar a correção deste programa simples (CORDEIRO et al., 2009), uma vez que o valor do limite superior do loop,

representado por n, é escolhido de forma não determinística, ou seja, a variável n pode assumir qualquer valor a partir um tamanho do tipo `unsigned int`, que varia entre os diferentes tipos de computadores. Devido a está condição, o loop pode ser desenrolado $2^n - 1$ vezes (no pior caso $2^{32} - 1$ vezes em um inteiro de 32 bits) que é, portanto, pouco prático. Basicamente, um *Bounded Model Checker* simbolicamente executa várias vezes o incremento da variável i e computa a variável sn $4,294,967,295$ vezes. Um modo de resolver o problema de desenrolar o loop $2^n - 1$ vezes é guiar a verificação efetuada por um BMC utilizando invariantes de programas como restrições que auxiliam a exploração dos conjuntos de estados verificados, assim reduzindo o modelo a ser verificado por um BMC (ROCHA, 2015).

1.1 Definição do Problema

O problema considerado neste trabalho é expresso na seguinte questão: Como complementar e aprimorar a verificação de propriedades de segurança por *model checkers*, por meio da inferência de invariantes de programas por *templates* com aplicação na linguagem de programação C, de tal modo que as propriedades verificadas possam ser validadas?

1.2 Objetivos

O objetivo principal deste trabalho é apresentar um método para inferir invariantes de programas usando técnicas baseadas em combinações booleanas de desigualdades lineares sobre variáveis de programa, para a verificação formal de softwares escritos na linguagem de programação C, visando assim que a verificação seja aprimorada pela redução do espaço de estados de forma que as propriedades de segurança (incluindo segurança de memória) no programa possam ser validadas ou refutadas com sucesso.

Os objetivos específicos são:

1. Propor um método para definir um *template* de invariantes com uma conjunção finita de desigualdades;
2. Formular um método para gerar combinações booleanas de desigualdades lineares sobre variáveis de programas;
3. Validar a aplicação do método proposto sobre *benchmarks* públicos de programas em C, a fim de examinar a sua eficácia e aplicabilidade.

1.3 Organização do trabalho

Este trabalho é organizado em capítulos que objetivam fornecer uma base para a solução do problema de pesquisa abordado.

No **Capítulo 1: Introdução** foi apresentada uma contextualização do trabalho e seus objetivos, bem como a definição do problema.

No **Capítulo 2: Conceitos e definições** são apresentados os principais conceitos e definições necessários à compreensão deste trabalho.

No **Capítulo 3: Trabalhos correlatos** são apresentados outros trabalhos focados na geração de invariantes, e como estes trabalhos se comparam com o trabalho atual.

No **Capítulo 4: Método proposto** é proposto e descrito um método para se alcançar os objetivos do trabalho.

No **Capítulo 5: Resultados Experimentais** o método proposto é avaliado e os resultados obtidos são discutidos.

Finalmente, no **Capítulo 6: Considerações parciais** é realizada uma recapitulação do conteúdo abordado neste trabalho, bem como considerações relevantes à implementação do método proposto.

2 CONCEITOS E DEFINIÇÕES

Este capítulo tem como objetivo apresentar os principais conceitos e definições necessários para a compreensão deste trabalho.

2.1 Lógica Proposicional

Em verificação formal de software, as restrições e propriedades a serem validadas são expressas em termos próprios da lógica proposicional, sendo portanto o objetivo desta seção introduzir o leitor a esses conceitos. Bradley e Manna (2007) definem lógica proposicional como argumentos realizados sobre proposições, onde proposições são expressões que sempre assumirão um valor *booleano* (Ou seja, verdadeiro ou falso).

Ainda segundo Bradley e Manna (2007), a lógica proposicional utiliza cinco operadores principais para argumentar sobre proposições, chamados *conectivos*. Sejam P e Q duas proposições lógicas. A semântica de cada um dos operadores é descrita abaixo.

- $\neg$ denota o operador *not*. A expressão $\neg P$ é a *negação* da proposição P, sendo equivalente a afirmar que o oposto da proposição está correto;
- $\wedge$ denota o operador *and*. A expressão $P \wedge Q$ é a *conjunção* das proposições P e Q, sendo equivalente a afirmar que ambas as proposições estão corretas;
- $\vee$ denota o operador *or*. A expressão $P \vee Q$ é a *disjunção* das proposições P e Q, sendo equivalente a afirmar que ao menos uma das proposições está correta;
- $\Rightarrow$ denota o operador de implicação. A expressão $P \Rightarrow Q$ lê-se como "P implica em Q", sendo equivalente a afirmar que, se P está correta, Q está correta também;
- $\Leftrightarrow$ denota o operador se e somente se. A expressão $P \Leftrightarrow Q$ lê-se como "P equivale a Q", sendo equivalente a afirmar que a proposição P implica em Q ao mesmo tempo que a proposição Q implica em P.

A Figura 2 mostra um programa simples em C onde o valor de uma variável b depende do valor de outra variável, a. O programa apresentado na figura não pode ser representado por uma única expressão aritmética, pois ocorre uma disjunção da forma $if - else$. Contudo, é possível representá-lo na forma de uma expressão da lógica proposicional, conforme a Equação 2.1.

$$((a > 0) \wedge (b = 0)) \vee (\neg(a > 0) \wedge (b = 1)) \tag{2.1}$$

```
1  int main(int argc, char **argv)
2  {
3      int a, b;
4
5      if(a > 0){
6          b = 0;
7      } else {
8          b = 1;
9      }
10 }
```

Figura 2 – Exemplo de programa representável por expressão lógica

2.2 Teste e Verificação de Software

Duas noções frequentemente confundidas entre si são as de teste e de verificação de software. Para este trabalho, ambos os conceitos são importantes, sendo, portanto, necessário diferenciá-los. O trabalho de Cousot e Cousot (2010) define verificação formal de software como um método de provar matematicamente, de forma automática, certas propriedades de um programa. Ao mesmo tempo, afirma que testes não são um método de verificação formal, descrevendo estes como a exploração de algumas instâncias do programa, ao contrário da verificação, que estuda todas as instâncias do programa ao mesmo tempo. Afirma ainda que, enquanto testes podem falhar em encontrar um erro, a verificação de software pode gerar erros inexistentes, os chamados *alarmes falsos*. Ambos os métodos são considerados complementares, devendo portanto serem usados em conjunto.

```
1  int sum(int a, int b){
2      return a + b;
3  }
```

Figura 3 – Função de soma de duas variáveis

```
1  test_case_a(){
2      int result = sum(7,8);
3      assert(result == 15);
4  }
5
6  test_case_b(){
7      int result = sum(10000000000,10000000000);
8      assert(result == 20000000000);
9  }
```

Figura 4 – Casos de teste para o programa na Figura 3

A Figura 3 mostra uma simples função em C que recebe duas variáveis, a e b, e retorna o resultado de sua soma. A Figura 4 exemplifica a forma como uma aplicação de testes operaria sobre esta função: Múltiplos casos de teste são gerados, atribuindo valores para a e b, e buscando encontrar $bugs$ na função. A Figura 5, por outro lado, exemplifica a ideia de um sistema de verificação, onde se busca garantir que a função segue uma determinada propriedade (no caso, que a função retorne corretamente a soma dos dois valores).

```
1 verifica_soma(int a, int b){
2     int answer = soma(a,b);
3
4     assert(a + b == answer);
5 }
```

Figura 5 – Verificação da corretude do programa na Figura 3

2.3 Interpretação abstrata

Cousot e Cousot (2010) define interpretação abstrata como uma técnica que estuda a semântica de um programa, buscando definir algumas de suas propriedades em tempo de execução, podendo ser usada em verificação formal de software, embora não seja restrita a esse campo. As abstrações são necessárias para o estudo das propriedades de um sistema computacional, justificando assim o seu uso.

Para Cousot e Cousot (2010), existem três técnicas principais para verificação formal de software, sendo que todas utilizam de interpretação abstrata. As técnicas citadas são descritas abaixo:

- *Model Checking* abstrai o programa na forma de um modelo para então inferir propriedades sobre o mesmo através de uma busca exaustiva. Por utilizar de busca exaustiva, a técnica possui problemas graves de performance e escalabilidade, que serão descritos em uma seção posterior.
- A técnica de Análise Estática busca automatizar a abstração do programa, enquanto os outros métodos (Model Checking e dedutivos) demandam o auxílio do usuário na abstração. Embora possua finitude garantida, ou seja, sempre termina sua execução, é sujeita a falsos alarmes.
- Métodos dedutivos abstraem o programa sob a forma de expressões matemáticas, utilizando de ferramentas orientadas a provar teoremas para buscar corretude matemática. Requerem um grande auxílio do usuário, que deve prover argumentos indutivos e dicas para a prova da corretude do programa. Segundo Moy e Marché (2007), esses argumentos normalmente são dados na forma de invariantes, pré-condições e pós-condições.

Em geral, abstrações buscam representar computacionalmente um conjunto concreto de estados do programa (no caso, todos os estados que o programa pode alcançar) sem de fato executar o programa concretamente. Naturalmente, por não executar o programa concretamente, a abstração normalmente sofre de perda de precisão, incluindo estados que não pertencem ao programa (super-aproximação) ou removendo estados que pertencem ao programa (sub-aproximação) (COUSOT; COUSOT, 2010).

Para ilustrar o conceito de abstração, Cousot e Cousot (2010) descreve um conjunto de estados de um programa na forma de pontos em um plano, como a Figura 6. A Figura 7 apresenta duas abstrações para o conjunto de estados por super-aproximações: A primeira, em azul-claro, simplesmente representa o quadrante do plano que cobre o conjunto de estados. A segunda, em azul-escuro, é mais precisa, e representa o intervalo em x e y no qual os pontos estão contidos. Abstrações mais precisas representam os estados em formas mais compactas, reduzindo o número de estados que não pertencem ao programa. Exemplos são o domínio dos octógonos (MINÉ, 2006), o domínio dos poliedros (HENRY et al., 2012) e o domínio dos elipsóides (ROUX et al., 2012).

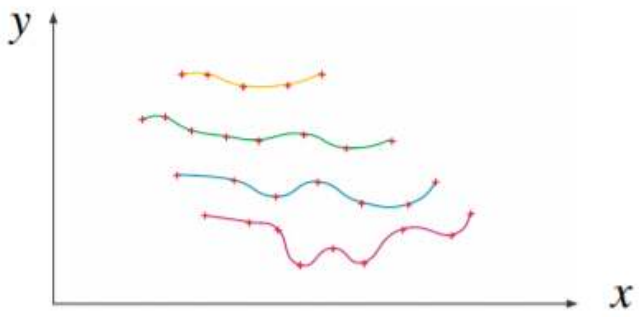


Figura 6 – Conjunto de estados de um programa.

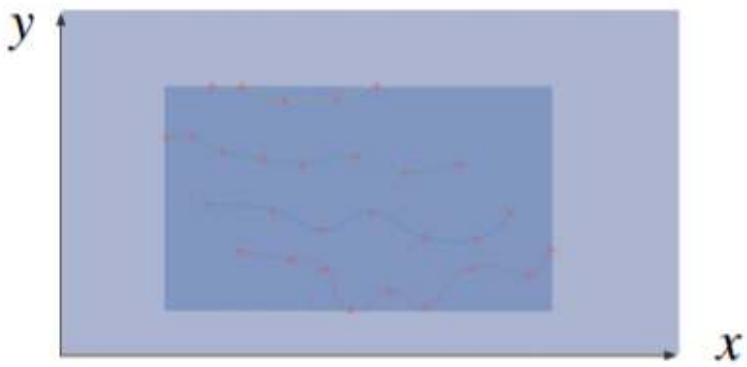


Figura 7 – Abstrações por quadrante e por intervalo do conjunto de estados.

2.4 Model Checking

Segundo Clarke e Schlingloff (2001), *Model Checking* é uma técnica de verificação de software que descreve programas na forma de *modelos*, os quais geralmente são grafos de estados finitos, e utiliza procedimentos de busca para verificar se a propriedade analisada é verdadeira ou não. Uma grande vantagem da técnica está no fato de ser completamente automatizada, o que facilita seu uso.

Contudo, por ser um método de busca exaustiva, a técnica de Model Checking está sujeita a explosões de estados, ou seja, uma variedade muito grande de estados a serem explorados tal que sua verificação seja impraticável em tempo de execução hábil. Para contornar este obstáculo, existem várias técnicas que tornam a técnica de Model Checking mais eficiente, como Bounded Model Checking, que limita a busca exaustiva a um dado número de passos (ROCHA et al., 2015), e CEGAR, que utiliza de contra-exemplos para limitar a busca (BEYER et al., 2007). Muitas dessas técnicas beneficiam-se do uso de *invariantes*, cuja geração é o foco deste trabalho.

É importante notar que, embora Cousot e Cousot (2010) considerem Bounded Model Checking (BMC) uma técnica de testes, ao invés de verificação, por não cobrir todos os estados em sua busca, trabalhos mais recentes são capazes de verificar propriedades de um programa cobrindo apenas um número limitado de estados utilizando k-indução, por exemplo (ROCHA et al., 2015), o que torna este método também uma técnica de verificação, segundo o conceito de Cousot e Cousot (2010).

2.4.1 K-indução

Uma vez que k-indução é uma importante parte da técnica de model checking, sendo responsável por uma melhoria significativa em sua eficiência, faz-se necessário apresentar seu conceito em detalhe. Seja p a propriedade do programa a ser verificada. Segundo Rocha et al. (2015), k-indução é uma técnica que limita a busca exaustiva do model checking a um número k de passos, onde k é um valor incrementado iterativamente, e dentro desses k passos busca alcançar uma das seguintes conclusões:

- Se p foi verdadeira para k passos, ela será verdadeira para qualquer número $n > k$ de passos, provando que a propriedade é verdadeira. Essa conclusão é alcançada através de prova por indução.
- Se for possível gerar um contra-exemplo para p em k passos, é verificado que a propriedade é falsa.
- É impossível alcançar uma conclusão em k passos, sendo portanto necessário incrementar o valor de k.

2.5 Invariantes

De acordo com o trabalho de Fouladgar et al. (2011), invariantes consistem de fórmulas ou regras que podem ser inferidas a partir do código-fonte de um programa e que se mantém imutáveis durante a sua execução, independentemente dos parâmetros que o programa venha a assumir. Através do uso de invariantes, é possível prever o comportamento do programa durante sua execução, o que as torna um importante aspecto da verificação de software, sendo uma de suas aplicações diretas a redução do espaço de busca em *Model Checking* (ROCHA et al., 2015).

Invariantes também possuem aplicações em outras áreas, como engenharia de software (KRKA et al., 2010), e reparo de software (PEI et al., 2014). De forma geral, áreas que operam sobre propriedades de programas beneficiam do uso de invariantes, conforme afirmam Fouladgar et al. (2011).

2.5.1 Invariantes Disjuntivas

Um dos grandes focos de trabalhos associados à inferência de invariantes é a geração de invariantes disjuntivas. Segundo Nguyen et al. (2014), invariantes disjuntivas são uma categoria de invariantes que utilizam de disjunções lógicas em suas fórmulas, servindo para captar relações exclusivas de alguns caminhos do programa. Srivastava e Gulwani (2009) afirmam que, embora a maioria dos programas não-triviais necessite de invariantes disjuntivas, muitas ferramentas de verificação de software são incapazes de inferí-las.

Um exemplo de programa representável por invariantes disjuntivas é descrito na Figura 8. Embora $(x \leq y)$, $(5 \leq y \leq 11)$ e $(x \leq 11)$ sejam invariantes válidas, a forma mais expressiva de representar o loop é através da disjunção $((x < 5) \wedge (y = 5)) \vee ((5 \leq x \leq 11) \wedge (y = x))$, mostrando a dependência que x exerce sobre o valor de y, conforme Nguyen et al. (2014) afirma em seu trabalho.

```
1  void ex1(int x){
2      int y = 5;
3      if(x > y) x = y;
4
5      while(x <= 10){
6          if(x >= 5) y = y+1;
7
8          x = x+1;
9      }
10     assert(y == 11);
11 }
```

Figura 8 – Exemplo de programa que requer uma invariante disjuntiva (NGUYEN et al., 2014)

2.5.2 Invariantes Quantificadas e Não-Quantificadas

O foco deste trabalho será a geração de um tipo específico de invariantes, as invariantes aritméticas lineares. Segundo Gupta e Rybalchenko (2009), invariantes aritméticas lineares utilizam apenas expressões de aritmética linear em seus predicados, sendo que uma invariante consiste em um ou mais predicados unidos por conjunções e/ou disjunções.

A Figura 9 apresenta um programa em C que pode ser representado por uma invariante aritmética linear. A expressão $assume(n > 0)$ garante que o valor de n sempre será maior que 0, de forma que inicialmente o valor de x seja sempre maior que n, ou seja, o loop deverá ser executado ao menos uma vez. A invariante associada a este loop é simplesmente $x \leq n$, que normalmente é escrita sob a forma $x - n \leq 0$.

```
1 void simple(int n){
2     if(n < 0) return;
3     int x = 0;
4
5     while(x < n){
6         x++;
7         assume(n - x > 0);
8     }
9 }
```

Figura 9 – Exemplo de programa representável por uma invariante aritmética linear (GUPTA; RYBALCHENKO, 2009)

Alternativamente, as invariantes aritméticas lineares podem ser chamadas de *invariantes não-quantificadas*, contrapondo-se a outro tipo de invariante, as *invariantes quantificadas*. Segundo Kong et al. (2010), invariantes quantificadas utilizam operadores de quantificação como o quantificador universal $\forall$ ou o quantificador existencial $\exists$. Embora invariantes quantificadas possam ser associadas a uma variedade maior de programas, computá-las é considerado um desafio (FOULADGAR et al., 2011). É importante notar que, embora toda invariante aritmética linear seja não-quantificada, nem todas as invariantes não-quantificadas são do tipo aritmético linear (KAHSAI et al., 2011).

```
1  void init_check(int *a, int n){
2      int i;
3
4      for(i=0;i<n;i++){
5          a[i] = 0;
6      }
7
8      for(i=0;i<n;i++){
9          assert(a[i] == 0);
10     }
11 }
```

Figura 10 – Exemplo de programa que requer uma invariante quantificada (BEYER et al., 2007)

A Figura 10 apresenta um programa que apenas pode ser representado por uma invariante quantificada, pois lida com *arrays*. Uma invariante que representa o primeiro loop da figura adequadamente é dada pela expressão $\forall k : 0 \leq k < i \Rightarrow a[k] = 0$, conforme Beyer et al. (2007). A invariante indica que, a qualquer momento do programa, todas as posições menores que i no array terão o valor 0.

2.5.3 Templates de invariantes

Templates são, segundo Srivastava e Gulwani (2009), um método para geração de invariantes que utiliza do auxílio do usuário. O usuário deve prover um *template*, uma expressão incompleta com incógnitas que devem ser preenchidas, mas que fornece à ferramenta de inferência uma dica para a forma que a invariante gerada deve assumir. Invariantes geradas por

templates são, em geral, mais expressivas, porém possuem a desvantagem de estarem restritas ao *template* fornecido pelo usuário.

Templates podem assumir as mais diversas formas. Gupta e Rybalchenko (2009), por exemplo, utilizam inequações parametrizadas lineares da forma $c_1v_1 + c_2v_2 + ... + c_nv_n \leq c$ como *templates*, onde as incógnitas a serem preenchidas são as constantes das inequações, c_1, c_2...c_n. Adicionalmente, embora *templates* geralmente sejam providos pelo usuário, Kong et al. (2010) afirma que é possível inferí-los automaticamente através de técnicas como análise estática, a qual será discutida em uma seção posterior.

Geralmente, o uso de *templates* implica no uso de ferramentas auxiliares chamadas *solucionadores de restrições*, como SAT (KROENING et al., 2008) e SMT (SRIVASTAVA et al., 2009) Solvers. A função destas ferramentas no contexto da geração de invariantes por meio de templates é encontrar valores para as incógnitas dos *templates*, ou afirmar que tais invariantes não existem.

2.6 Análise estática

Para Møller e Schwartzbach (2018), a análise estática é a arte de analisar o comportamento de programas sem executá-los, sendo a verificação de software uma possível aplicação dessa técnica. Para realizar essa análise, ferramentas estudam o código-fonte do programa em busca de informações sobre uma propriedade a ser verificada. Muito frequentemente, estas informações são dadas na forma de *invariantes*.

Embora existam diversos domínios abstratos próprios para análise estática, apenas três deles são relevantes para o escopo deste trabalho: O domínio de intervalos, o domínio poliedral e o domínio octagonal. Estes domínios são os mais comumente utilizados, sendo empregados em ferramentas no estado da arte, como Seahorn (SEAHORN, 2018c), DepthK (ROCHA et al., 2017) e CPAChecker (BEYER et al., 2015). Miné (2006) define cada um destes domínios da seguinte forma:

- O domínio de intervalos é apenas capaz de gerar invariantes da forma $v \in [x,y]$, onde *x e y* são constantes e *v* é uma variável. Este domínio é bastante limitado em termos de expressividade, ou seja, muitos programas estão além de seu escopo. Contudo, sua eficiência computacional é a melhor entre os domínios abstratos, possuindo custo linear.
- O domínio poliedral gera invariantes da forma: $c_1v_1 + c_2v_2 + ... + c_nv_n \leq c$, onde $c, c_1, c_2...c_n$ são constantes e $v_1, v_2...v_n$ são variáveis. É o domínio mais expressivo e, embora o custo para realizar uma busca em todo o domínio possua custo exponencial, ferramentas recentes são capazes de reduzir este custo através de diversas otimizações (FEAUTRIER; GONNORD, 2010; HENRY et al., 2012).

- O domínio octagonal gera invariantes da forma $\pm x \pm y \leq c$, onde x *e* y são variáveis e c é uma constante. É um intermediário entre o domínio dos intervalos e o domínio poliedral que busca equilibrar expressividade e eficiência.

```
1 void simple(int n){
2     if(n <= 0) n = 1;
3     int x = 0;
4
5     while(x < n){
6         x++;
7     }
8 }
```

Figura 11 – Programa ilustrativo para exemplificar os domínios abstratos.

Seja, por exemplo, o programa descrito na Figura 11. Uma análise no domínio dos intervalos apenas seria capaz de inferir invariantes como $x \in [0, \infty]$ e $n \in [1, \infty]$. Uma invariante no domínio octagonal, por outro lado, seria capaz de inferir relações entre x e n, como $x - n < 0$.

```
1  void simple(int n){
2      void forward(int n) {
3      int i, n, a, b;
4      assume(n >= 0);
5      i = 0; a = 0; b = 0;
6      while(i<n){
7          if(...) {
8              a = a+1;
9              b = b+2;
10         } else {
11             a = a+2;
12             b = b+1;
13         }
14         i = i+1;
15     }
16         assert( a+b == 3*n );
17     }
18 }
```

Figura 12 – Programa ilustrativo para exemplificar o domínio poliedral. Fonte: Beyer et al. (2007).

Seja agora o código na Figura 12. Por operar sobre múltiplas variáveis, o domínio poliedral é capaz de gerar as invariantes $a + b - 3n \leq 0$ e $0 \leq a + b - 3n$, que são suficientes para verificar a propriedade desejada.

2.7 Análise dinâmica

Segundo Ernst et al. (2007), a análise dinâmica executa diversas instâncias de um programa com diferentes parâmetros, observa o comportamento das variáveis e então reporta propriedades comuns a todas as instâncias. A ideia é que, assumindo que um número suficiente de

instâncias tenha sido observado, supõe-se que as propriedades observadas possam ser estendidas para as instâncias não-observadas, o que torna essas propriedades invariantes.

Ernst et al. (2007) ilustra o conceito com um exemplo em Java, que implementa uma pilha conforme a Figura 13.

```
1  Object[] theArray;
2  int topOfStack;
3
4  void push(Object x); //Insert x
5  void pop(); //Remove most recently inserted element
6  Object top(); //Get most recently inserted element
7  Object topNPop(); /Integrates pop and top
8  boolean isEmpty() // Return true if empty; else false
9  boolean isFull() // Return true if full; else false
10 void makeEmpty() // Remove all items
```

Figura 13 – Estrutura de pilha descrita por Ernst et al. (2007)

Assuma que o código descrito na figura seja analisado por uma técnica de análise dinâmica, que em todas as instâncias de sua execução nunca obteve um valor nulo para o objeto $theArray$. Logo, $theArray! = null$ é assumido como invariante. Da mesma forma, qualquer outra expressão que seja verdadeira para todas as instâncias estudadas é assumida como verdadeira para as instâncias não estudadas também.

Assim como a análise estática está associada à verificação de software, a análise dinâmica está associada a testes de software: Não existe uma prova formal de que as propriedades são válidas para qualquer instância (KING, 1976). De acordo com Nguyen et al. (2014), a vantagem da análise dinâmica é sua eficiência em comparação com os métodos de análise estática.

A análise dinâmica pode ser efetuada de duas formas: Por execução concreta ou execução simbólica. A execução concreta de um programa é a forma mais ingênua de se realizar análise dinâmica. Para cada instância de um programa, instancia-se as variáveis de entrada com valores diferentes, e executa-se o programa normalmente. A execução simbólica, por outro lado, é uma técnica mais sofisticada que utiliza restrições sobre as variáveis do programa (adotando valores simbólicos) em forma de expressões booleanas para cada condição de caminho (em inglês *path condition*), que é uma fórmula livre de quantificadores sobre as entradas simbólicas que codificam todas as decisões condicionais (exemplo estruturas `IFs`) tomadas no programa.

2.7.1 Execução simbólica

De acordo com King (1976), a técnica de execução simbólica consiste em uma melhoria da execução concreta, onde, ao invés de se testar instâncias com variáveis de entrada arbitrárias, se testam *classes* de entradas. Uma classe de entrada é definida pelo fluxo de controle do programa: Cada *caminho* diferente que o programa pode tomar é descrito por uma classe diferente.

King (1976) afirma que, embora essa técnica ainda esteja sujeita a explosões de estados (quando se lida com *loops*, por exemplo, que podem chegar a infinitos caminhos diferentes), o

número de estados sobre o qual ela opera é significativamente menor que a execução concreta. Isso ocorre, pois, embora o custo de uma verificação completa utilizando essa técnica ainda seja exponencial em relação ao número de classes, cada classe pode representar um grande número de instanciações de variáveis, reduzindo o espaço de busca significativamente. De forma geral, a execução simbólica promete prover resultados melhores e em menos tempo que a execução concreta para a maioria dos programas na prática.

A Figura 14 mostra o comportamento de uma aplicação de execução simbólica simples ao analisar um programa em C. À esquerda, é apresentado o código-fonte do programa. As variáveis x, y e z são inicializadas com valores reais, enquanto a, b e c não foram inicializadas, e portanto recebem valores simbólicos. Para cada estrutura if que avalia uma variável simbólica, dois caminhos são abertos: Um que assume que a instrução if resulta em $true$, e outro que assume que a instrução resulta em $false$.

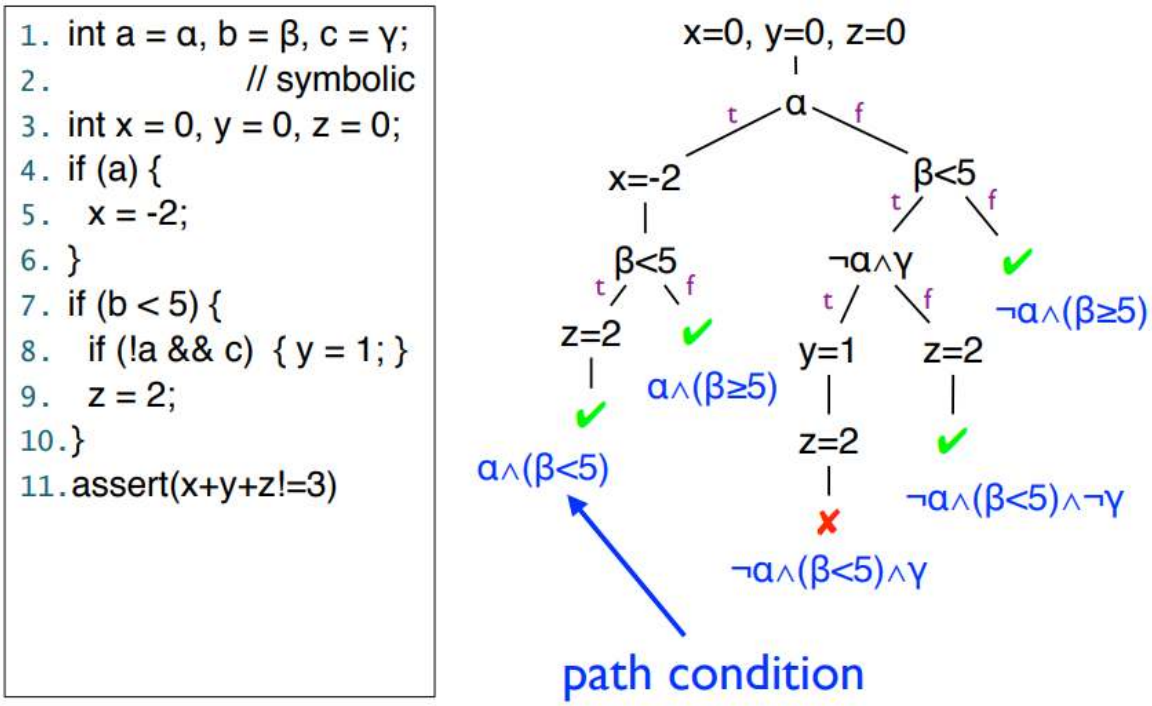


Figura 14 – Fluxo de execução simbólica de um programa em C (FOSTER, 2011)

.

Devido a sua superior eficiência em comparação com a execução concreta, a execução simbólica é utilizada em diversos contextos da análise dinâmica, como testes de software (MENEZES et al., 2018a; KING, 1976) e geração dinâmica de invariantes (CSALLNER et al., 2008).

2.8 Solucionadores de restrições

Rossi et al. (2006) define satisfabilidade de restrições como o problema de encontrar valores para um conjunto de variáveis, dado um conjunto de restrições sobre essas variáveis. Em sua forma mais básica, essas restrições declaram que alguns subconjuntos de valores não podem ser utilizados juntos.

Segundo Rossi et al. (2006), embora a satisfabilidade de restrições seja considerado um problema de busca, é possível representar muitos problemas de diversas áreas na forma de satistabilidade de restrições. Exemplos são a modelagem geométrica (KLEIN, 1998) e agendamento de atividades (PAPE; A, 2005). Essa capacidade de representar uma amplitude tão grande de problemas tornou a satisfabilidade de restrições um novo paradigma da programação, sendo alvo de diversas pesquisas e métodos específicos para resolvê-la de forma eficiente (MINTON et al., 1990; BADROS et al., 2001).

Uma das aplicações da satisfabilidade de restrições é a verificação de software, onde diversas ferramentas chamadas *constraint solvers*, ou solucionadores de restrições, foram desenvolvidas com o objetivo específico de resolver este problema. Dois tipos específicos de *constraint solvers* se destacam na área de verificação de software: Os SAT Solvers e os SMT Solvers, que serão apresentados nas próximas seções.

2.8.1 SAT Solvers

Claessen et al. (2008) define o problema de satisfabilidade booleana, também conhecido como SAT, como o problema de, dada uma expressão da lógica proposicional com diversas variáveis, encontrar valores *true* ou *false* para essas variáveis tais que a expressão assuma o valor *true*. Sob o ponto de vista da satisfabilidade de restrições, a própria expressão lógica é a restrição sobre as variáveis. Por ser um problema clássico de difícil solução em tempo hábil, diversas técnicas para resolvê-lo foram desenvolvidas, como DPLL, uma técnica baseada em backtracking (OUYANG, 1996), e PPSZ, um algoritmo probabilístico (SCHEDER; STEINBERGER, 2017).

Seja, por exemplo, a expressão $(a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee \neg a)$. Uma valoração que satisfaz a expressão é dada por $a = false$, $b = true$ e $c = true$, onde cada uma das cláusulas assume o valor $true$ e, portanto, a expressão inteira assume o valor $true$. Por outro lado, não existem valores para a e b que satisfaçam a expressão $(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b)$. Denota-se este tipo de expressão por *insatisfatível*.

SAT *solvers* são ferramentas especializadas na resolução do problema da satisfabilidade (PIPATSRISAWAT; DARWICHE, 2007; BIERE, 2011; MOSKEWICZ et al., 2001). As diversas peculiaridades da lógica proposicional permitem aos SAT Solvers diversas otimizações, descritas em detalhe por Claessen et al. (2008). Essas otimizações os tornam muito mais eficientes que solucionadores de restrições comuns, que precisam de soluções mais generalizadas por tratarem de um problema mais generalizado.

2.8.2 SMT Solvers

Moura e Bjørner (2008) define SMT (*Satisfiability Modulo Theory*) como uma generalização da satisfabilidade booleana, onde são inseridos conceitos como aritmética, vetores e quantificadores, com o objetivo de tornar o problema da satisfabilidade mais expressivo em relação a sistemas computacionais. Um SMT Solver é uma ferramenta capaz de decidir a satisfabilidade de fórmulas nesse domínio estendido, onde o conceito de satisfabilidade é análogo ao utilizado para definir a satisfabilidade booleana: uma valoração das variáveis de uma dada fórmula, tal que a fórmula assuma o valor de *true* sob essa valoração.

SMT Solvers operam sobre *teorias*. Teorias consistem em sistemas de expressões em primeira ordem (i.e. consistindo em predicados, expressões que assumem valores *true* ou *false* dependendo de sua entrada), as quais descrevem as regras de um *domínio* específico (GANESH, 2007). Este domínio pode ser o domínio dos números inteiros e reais (RANISE et al., 2006), dos vetores (GANESH, 2007) ou mesmo estruturas complexas como listas e mapas (KRöNING; WEISSENBACHER, 2016).

Sendo uma extensão dos SAT Solvers, SMT Solvers são capazes de resolver problemas de satisfabilidade tal qual os SAT Solvers, não sendo incomum que contem com o auxílio de um SAT Solver para tal (MOURA; BJØRNER, 2008; BRUMMAYER; BIERE, 2009). Seu diferencial está na sua expressividade, sendo um SMT Solver capaz de lidar com sistemas mais complexos, ao custo de ser ele próprio um sistema mais complexo (CLAESSEN et al., 2008).

3 TRABALHOS CORRELATOS

Este capítulo apresenta trabalhos relacinados à geração de invariantes através de *templates*, descrevendo sua metodologia e como esta se compara com o método proposto por este TCC, bem como, quando adequado, as contribuições oferecidas por cada um para a criação do trabalho atual.

3.1 *Automatically Inferring Quantified Loop Invariants by Algorithmic Learning from Simple Templates*

No trabalho de Kong et al. (2010), é abordado o problema da geração de invariantes através do uso de *templates* por meio de um modelo de aprendizagem de máquina chamado "Estudante e Professor". Nesse modelo, duas aplicações diferentes simulam um estudante, responsável por fazer perguntas, com o objetivo de aprender sobre a natureza das invariantes do programa, e um professor, que deve ser capaz de responder a essas perguntas. É utilizada uma aplicação específica desse modelo, o algoritmo CDNF (BSHOUTY, 1993).

Embora a aplicação original do modelo seja orientada à geração de fórmulas booleanas, o trabalho foca na geração de expressões (Invariantes) quantificadas. Para realizar essa conversão, o algoritmo utiliza o auxílio de um *template*, que deve conter um conjunto não-vazio de quantificadores existenciais, e expressões atômicas (expressões aritméticas lineares sobre variáveis do programa), que substituirão as variáveis na expressão booleana. O *template* e as expressões atômicas devem ser manualmente fornecidas pelo usuário. Finalmente, o método proposto utiliza de respostas aleatórias para gerar invariantes de forma não-determinística, possibilitando a inferência de várias invariantes diferentes para um único programa.

Um protótipo do método proposto por Kong et al. (2010) foi implementado utilizando a linguagem de programação OCaml (OCAML, 2018), com o objetivo de realizar experimentos comparativos. Para os experimentos, foram utilizados dois *benchmarks* da literatura (SRIVASTAVA; GULWANI, 2009) e quatro trechos de código do kernel do Linux. Os experimentos realizados demonstram a eficácia do método, capaz de gerar diversas invariantes para *benchmarks* considerados complexos, embora uma análise do tempo de execução não tenha sido disponibilizada.

O método proposto neste TCC difere do trabalho de Kong et al. (2010) em diversos pontos. Embora ambos sejam orientados à geração de invariantes, o trabalho de Kong et al. (2010) gera invariantes quantificadas, enquanto a solução proposta neste TCC busca apenas gerar invariantes não-quantificadas, cuja expressividade é menor. Em contrapartida, o método proposto requer menos influência do usuário, não necessitando de expressões atômicas. Adicionalmente,

o método proposto não necessita que o usuário provenha uma pós-condição, e incorpora o uso de ferramentas mais atuais, possuindo maior integrabilidade.

A principal contribuição do trabalho Kong et al. (2010) para o desenvolvimento deste é o uso do algoritmo CDNF. Acredita-se que é possível utilizar esse algoritmo para a inferência da pós-condição do programa analisado, evitando assim que o usuário necessite provê-la, sendo este um dos diferenciais do método proposto.

3.2 *Instantiation-Based Invariant Discovery*

O método proposto por Kahsai et al. (2011) consiste em um método baseado em força bruta para gerar invariantes através de *templates* binários, i.e. *templates* com duas variáveis. A ideia básica do método consiste em gerar todas as instâncias possíveis do *template*, formando uma conjunção de todas essas instâncias e assumindo inicialmente que a expressão resultante é verdadeira. Contra-exemplos são utilizados para eliminar todas as conjunções inválidas, de forma que a expressão resultante seja, obrigatoriamente, uma invariante.

Kahsai et al. (2011) propõe que a conjunção seja gerada em tempo inferior a $O(n^2)$, propondo para tanto a limitação do método a dois domínios específicos: o domínio das expressões aritméticas lineares, e o domínio das expressões booleanas. Também são descritas as otimizações aplicáveis a cada domínio de forma a reduzir o custo de execução do algoritmo.

Para avaliar a eficácia e eficiência do método proposto, Kahsai et al. (2011) implementou seu método a partir de um *model checker* já existente, o KIND(CHAMPION et al., 2016), gerando uma nova ferramenta chamada KIND-INV. Foram utilizados 941 *benchmarks* traduzidos para a linguagem de programação LUSTRE (HALBWACHS et al., 1991), com o objetivo de analisar a precisão do *model checker* com e sem o auxílio das invariantes geradas pelo método proposto. Os resultados foram positivos, mostrando uma melhoria de 24% na precisão do *model checker* com o auxílio das invariantes.

Em relação ao método proposto neste TCC, a ferramenta KIND-INV objetiva gerar invariantes não-quantificadas, sendo capaz de gerar invariantes no domínio aritmético linear e no domínio booleano, enquanto o método proposto limita-se ao domínio aritmético linear. Contudo, a ferramenta KIND-INV demonstra-se incapaz de gerar invariantes disjuntivas, possuindo portanto menor representatividade que o método proposto por este trabalho, além de carecer das diversas otimizações que ferramentas já existentes podem fornecer, como remoção de código morto e propagação de constante.

3.3 *InvGen - An Efficient Invariant Generator*

No trabalho de Gupta e Rybalchenko (2009) apresentado um método para a geração de invariantes aritméticas lineares com o auxílio de técnicas de análise estática e análise dinâmica.

O método consiste em gerar restrições sobre as variáveis do programa, e a partir dessas restrições computar uma invariante através do uso de um solucionador de restrições.

A análise dinâmica é realizada através de execução simbólica, enquanto a análise estática utiliza de interpretação abstrata. Ambas as técnicas inferem invariantes simples/fracas, que são convertidas em restrições e então utilizadas para encontrar invariantes mais expressivas. Quanto ao solucionador de restrições, é utilizado na implementação inicial o paradigma de *constraint programming*, sendo posteriormente implementada uma versão que utiliza o *SMT Solver* Z3 (GUPTA et al., 2009).

Experimentos foram realizados em *benchmarks* com o objetivo de demonstrar a eficiência do método. Os *benhmarks* incluíam *loops* simples, *loops* aninhados e algoritmos de ordenação, inspirados pelo trabalho de Ku et al. (2007). Os resultados foram positivos, sendo a ferramenta capaz de inferir invariantes complexas em períodos de tempo relativamente curtos. Por exemplo, testes com o algoritmo de *heap sort* custaram apenas 13.3 segundos para verificar várias propriedades, embora o número exato de propriedades não tenha sido descrito. A ferramenta foi também aplicada ao refinamento de abstração de predicados, e através de comparações mostrou-se que as invariantes geradas pelo InvGen possuem grande impacto na eficiência da abstração de predicados. A análise objetivou exemplificar o impacto causado pelas invariantes da ferramenta na eficiência de ferramentas de verificação de software em geral.

O método proposto neste TCC é baseado na abordagem utilizada pelo InvGen, objetivando melhorá-lo ao integrar a metodologia com ferramentas atuais como o LLVM para melhor suporte a estruturas e otimizações de programas em C; Crab-LLVM para utilização de diferentes domínios abstratos (SEAHORN, 2018b); e o SMT solver Z3 para análise de restrições (MOURA; BJØRNER, 2008). Adicionalmente, também gerando pós-condições automaticamente e utilizando aprendizagem de máquina, adotando pré-condições geradas e valores de contra-exemplos gerados, para melhorar a expressividade das invariantes.

3.4 *Path Invariants*

O trabalho de Beyer et al. (2007) objetiva a melhoria de uma técnica de verificação já existente, chamada CEGAR (CLARKE et al., 2003) (*Counter-Example Guided Abstraction Refinement*), a qual utiliza de contra-exemplos para refinar a técnica de interpretação abstrata. A melhoria consiste em escrever contra-exemplos como subprogramas, objetivando representar vários contra-exemplos em uma única instância. Para efetivar essa melhoria, é necessário alterar a forma como os contra-exemplos afetam a abstração de um programa. Nesse ponto, é introduzido o conceito de *Path Invariants*, ou invariantes de caminhos, que consistem em invariantes computadas a partir desses subprogramas. É através dessas invariantes que a refinação da abstração do programa é efetuada.

Em relação à geração das invariantes de um programa, é utilizada uma técnica baseada na

geração de restrições sobre as variáveis do programa, a qual conta com o auxílio de um *template* manualmente provido pelo usuário. O *template* conta com expressões aritméticas lineares e pode conter um quantificador universal, com o objetivo de representar invariantes quantificadas.

Experimentos foram realizados em quatro programas descritos ao longo do artigo, segundo o qual a verificação desses programas seria desafiadora para técnicas CEGAR tradicionais. Os programas incluíam um programa com condição if não-determinística, dois programas que operam sobre *arrays* e utilizam invariantes quantificadas, e um programa com uma invariante disjuntiva. Os resultados foram positivos, sendo o método proposto capaz de verificar, com baixo custo de tempo (menos de três segundos para os piores casos, menos de um segundo em média), as propriedades desejadas para todos os programas descritos.

Em comparação com o método proposto por este trabalho de TCC, o método de Beyer et al. (2007) apresenta maior poder de representatividade, sendo capaz de gerar invariantes quantificadas, além de invariantes aritméticas lineares. Contudo, sua aplicabilidade é direcionada à técnica de verificação CEGAR, enquanto o método proposto por este trabalho de TCC é aplicável a qualquer técnica de verificação que beneficie do uso de invariantes para guiar a verificação na exploração dos estados do sistema analisado.

3.5 *Program Verification using Templates over Predicate Abstraction*

Srivastava e Gulwani (2009) descrevem três algoritmos diferentes para a geração de invariantes quantificadas, todos utilizando as técnicas de abstração de predicados e *templates* de invariantes. Dois dos métodos utilizam de algoritmos iterativos, enquanto o terceiro utiliza uma abordagem baseada em restrições, mas a ideia geral por trás de todos é a de obter instanciações ótimas dos *templates* que gerem uma invariante. O usuário deve prover um *template* e um conjunto de predicados que serão utilizados pelos algoritmos.

Uma vez que a ideia central dos algoritmos é a geração de invariantes *ótimas*, faz-se necessário definir uma invariante ótima. Uma invariante ótima, no contexto do trabalho de Srivastava e Gulwani (2009), é uma invariante tal que a remoção (ou adição, em alguns contextos) de qualquer predicado descaracterize a expressão como invariante. Em termos simplificados, a invariante ótima é a invariante que melhor expressa a natureza do programa analisado.

No trabalho de Srivastava e Gulwani (2009), foram realizados experimentos com *benchmarks* de manipulação de *arrays* e listas, incluindo algoritmos de ordenação, comparando a performance de algoritmos propostos em trabalhos anteriores (BEYER et al., 2007; JHALA; MCMILLAN, 2007; HALBWACHS; PÉRON, 2008; GULWANI et al., 2008) com a dos algoritmos propostos em seu trabalho. Os resultados foram positivos, sendo que através dos algoritmos propostos foi-se possível verificar vários *benchmarks* que ferramentas propostas em trabalhos

anteriores eram incapazes de verificar, além de diversos ganhos em tempo de execução em comparação com outras ferramentas. Também verificou-se que os três algoritmos apresentavam vantagens diferentes, apresentando melhor performance que os outros para determinados *benchmarks*.

A técnica apresentada em Srivastava e Gulwani (2009) é capaz de gerar invariantes quantificadas e disjunções, sendo possível ainda estendê-la de forma que gere pré-condições e/ou pós-condições para um programa.

Em contrapartida, o método proposto por este trabalho de TCC é mais simples e foca em automatizar a geração das invariantes, não necessitando de predicados providos pelo usuário e limitando-se à necessidade de um *template*, que também não precisará ser provido pelo usuário.

3.6 VS3: *SMT Solvers for Program Verification*

O trabalho de Srivastava et al. (2009) apresenta uma ferramenta para a inferência de invariantes chamada VS^3. A ferramenta é capaz de gerar invariantes quantificadas a partir de um *template* e um conjunto de predicados, que devem ser providos pelo usuário.

Quanto à geração de invariantes, a ferramenta apresenta dois métodos possíveis. O primeiro é um método iterativo, que mantém um conjunto de soluções candidatas e iterativamente melhora essas soluções inserindo mais predicados, ao final escolhendo a melhor solução que constitui uma invariante. O segundo é um método baseado em restrições, que consiste em gerar restrições booleanas que impliquem na condição de verificação do programa e então resolver essas restrições através de um SAT Solver.

Srivastava et al. (2009) apresenta testes que foram efetuados utilizando programas que operam sobre *arrays*, incluindo algoritmos de ordenação, contudo detalhes como descrição dos *benchmarks* utilizados e tempo de execução da ferramenta não foram descritos. O trabalho apenas afirma que os resultados foram positivos, sendo a ferramenta capaz de verificar as propriedades desejadas em todos os programas analisados.

Embora o método aplicado na ferramenta VS^3 seja mais expressivo, sendo capaz de gerar invariantes quantificadas e disjuntivas, este apresenta uma leve desvantagem em relação ao trabalho atual por necessitar de uma maior contribuição do usuário, que deve prover um conjunto de predicados além do *template*.

3.7 *Speeding Up the Constraint-Based Method in Difference Logic*

Candeago et al. (2016) propõem uma técnica para inferência de invariantes em um domínio específico, a lógica diferencial. Duas metodologias são descritas, uma para o domínio

da lógica diferencial, e um para o domínio mais geral da aritmética linear, sendo a primeira metodologia uma especialização da segunda. *Templates* aritméticos lineares são inseridos em cada localidade não-inicial do programa, de forma que ao final da execução do método o programa analisado tenha uma invariante em cada localidade.

Para avaliar a técnica proposta, Candeago et al. (2016) a implementam na ferramenta VeryMax, realizando uma experimentação com um *benchmark* de 3270 consultas, onde cada consulta é dada por um programa e uma propriedade que deve ser verificada. A técnica foi comparada com implementações do lema de Farkas, mostrando grande superioridade tanto em tempo de execução como em precisão.

O método proposto por este TCC integra características da técnica proposta por Candeago et al. (2016), como o uso de sistemas de transição para representar o programa analisado e a integração de *templates* a cada ponto de controle do mesmo, porém não se limita ao domínio da lógica diferencial, focando no domínio mais amplo da aritmética linear e utilizando uma abordagem alternativa.

4 MÉTODO PROPOSTO

Nesse capítulo, será descrito em detalhes o funcionamento do método proposto nesse trabalho. Também será descrita a forma como pretende-se, a partir do método proposto, alcançar os objetivos deste trabalho.

4.1 Visão Geral do Método

O método proposto objetiva a geração de invariantes para programas em C com o auxílio de *templates* e restrições inferidas no programa. Para este fim, três abordagens diferentes são propostas e apresentadas a seguir. Na primeira abordagem, as restrições são utilizadas em conjunto com as invariantes geradas, diretamente durante a verificação. Para a geração da restrições, o programa deve ser convertido para uma linguagem intermediária e então otimizado com o objetivo de simplificar o código. Técnicas de análise estática e análise dinâmica são aplicadas para coletar dados sobre o programa analisado, e inferir restrições a partir desses dados. Esta abordagem foi proposta no trabalho de Gupta e Rybalchenko (2009).

Paralelamente à geração de restrições, o programa é convertido em um autômato de fluxo de controle (FTSRG, 2019), de forma a facilitar a identificação de seus pontos de controle. A partir deste autômato, o programa é reescrito na forma de um conjunto de fórmulas de satisfabilidade baseadas em suas transições e anotadas com *templates*. Cada fórmula é resolvida individualmente, e os valores obtidos são substituídos no *template*. As expressões resultantes são invariantes. Finalmente, o programa é anotado com ambas, as invariantes e restrições geradas para cada ponto de controle. O fluxo de execução desta primeira abordagem é ilustrado pela Figura 15.

Na segunda abordagem, as restrições são utilizadas para guiar a geração das invariantes, objetivando a geração de invariantes mais precisas. Seu fluxo de controle é similar à primeira abordagem, porém segue uma sequência linear de passos. O primeiro passo é a geração de restrições, análoga à primeira abordagem. O programa é, então, convertido em um autômato de fluxo de controle e reescrito como fórmulas de satisfabilidade. No entanto, estas fórmulas são anotadas não somente com *templates*, como também com as restrições inferidas no primeiro passo. Cada fórmula é resolvida individualmente de forma a gerar uma invariante. O fluxo de execução da segunda abordagem é ilustrado pela Figura 16.

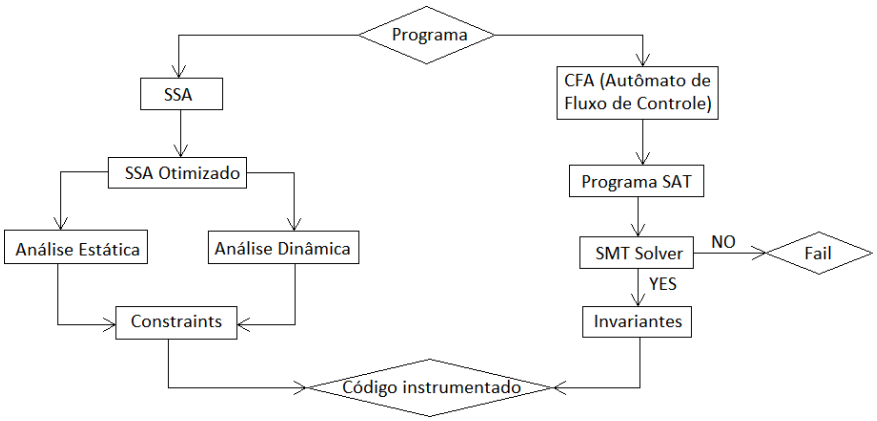


Figura 15 – Fluxo de execução da primeira abordagem. Entradas e saídas do fluxo são indicadas por losangos, enquanto etapas internas são identificadas por caixas. Setas indicam relações de dependência, onde cada etapa recebe o resultado da etapa anterior.

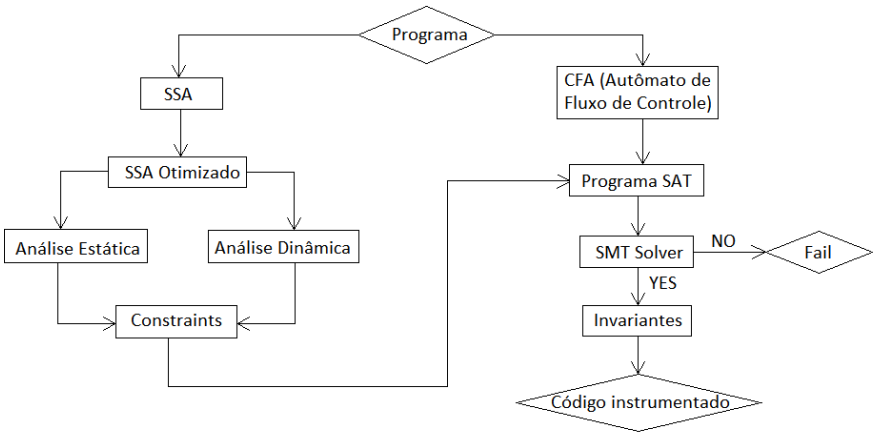


Figura 16 – Fluxo de execução da segunda abordagem. Entradas e saídas do fluxo são indicadas por losangos, enquanto etapas internas são identificadas por caixas. Setas indicam relações de dependência, onde cada etapa recebe o resultado da etapa anterior.

Finalmente, a terceira abordagem é uma união das duas abordagens anteriores, onde as restrições são utilizadas em dois pontos: Na geração das invariantes, como na segunda abordagem, e na verificação do programa, como na primeira abordagem. O fluxo de execução da terceira abordagem é ilustrado pela Figura 17.

Visando automatizar a aplicação do método proposto é utilizado o auxílio de algumas ferramentas, como o SMT solver Z3 (MOURA; BJØRNER, 2008). As ferramentas utilizadas, bem como detalhes adicionais sobre as fases do método proposto, serão descritas em detalhe em seções posteriores.

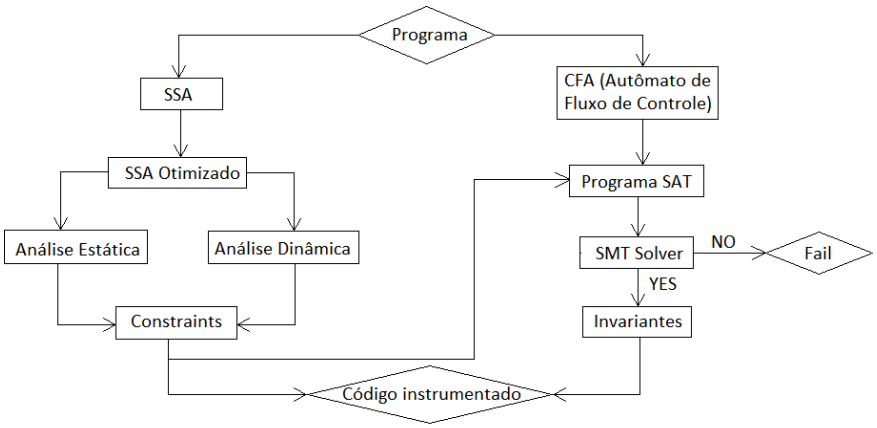


Figura 17 – Fluxo de execução da terceira abordagem. Entradas e saídas do fluxo são indicadas por losangos, enquanto etapas internas são identificadas por caixas. Setas indicam relações de dependência, onde cada etapa recebe o resultado da etapa anterior.

4.2 Representação Intermediária

A linguagem de programação C possui diversos elementos semânticos e sintáticos complexos que dificultam a sua análise direta. Portanto, ferramentas de análise de código (incluindo geração de invariantes) costumam traduzir o programa para uma linguagem intermediária. O método proposto utiliza um tipo específico de linguagem intermediária no formato *Static Single Assignment* (SSA).

Normalmente, ferramentas de geração de invariantes assumem que o código já esteja traduzido para a linguagem intermediária, como ocorre, por exemplo, nos trabalhos de Srivastava e Gulwani (2009) e Gupta e Rybalchenko (2009), sendo portanto responsabilidade do usuário realizar a conversão. No método proposto por este trabalho, objetiva-se que a conversão esteja dentro do escopo da própria ferramenta, sem a necessidade de executar manualmente uma aplicação secundária.

4.2.1 Infraestrutura de Compilação LLVM

O LLVM é uma coleção de tecnologias para compiladores, que conta inclusive com um compilador próprio, chamado Clang (CLANG, 2018). Entre as funcionalidades que o LLVM oferece, encontram-se diversas otimizações, como propagação de constante e remoção de código morto (LLVM, 2018). A ferramenta é orientada à manipulação de códigos na forma SSA, adequando-se perfeitamente ao método proposto.

Neste trabalho, utiliza-se o LLVM e suas funcionalidades para não somente converter códigos em C diretamente para códigos na forma SSA, como também aplicar diversas otimizações a estes códigos a nível de compilador, com o objetivo de reduzir o custo de execução da análise do programa e a posterior geração de invariantes. A Figura 18 apresenta uma simples função em

C. A Figura 19 apresenta o equivalente desta função na forma SSA nativa do LLVM, denominada de LLVM-IR.

```
1 unsigned square_int(unsigned a) {
2     return a*a;
3 }
```

Figura 18 – Simples função de exponenciação em C. Recebe um valor n e retorna n^2. Fonte: Majek (2018).

```
1 define i32 @square_unsigned(i32 %a) {
2   %1 = mul i32 %a, %a
3   ret i32 %1
4 }
```

Figura 19 – Forma intermediária do código apresentado na Figura 18.

4.3 Análise Estática

Uma vez que o programa esteja convertido na forma SSA, são aplicadas as análises estática e dinâmica sobre o programa convertido. Para tanto, é necessário o uso de uma ferramenta de análise estática capaz de operar sobre programas na forma SSA. A Seção 4.3.1 descreve uma ferramenta que cumpre essa especificação.

4.3.1 Crab-LLVM

O Crab (CoRnucopia of ABstractions) é uma ferramenta de análise estática baseada em interpretação abstrata que analisa o fluxo de controle de programas, sendo possível a aplicação de vários domínios, tais como: intervalos; poliedral; octógono; e zonas. Vale ressaltar que Crab foi desenvolvido sobre a ferramenta Ikos (*Inference Kernel for Open Static Analyzers*), desenvolvida pela NASA Ames Research Center. Como a ferramenta analisa o grafo do fluxo de controle, é independente de linguagem de programação, assumindo que exista uma ferramenta que traduza o programa em um grafo de fluxo de controle (SEAHORN, 2018a).

O Crab-LLVM é a integração da ferramenta Crab com o LLVM. Através dessa integração, torna-se possível realizar a análise abstrata de programas em linguagens suportadas pelo LLVM, após ser realizada a sua conversão para a forma SSA (SEAHORN, 2018b). Para este trabalho em particular, são utilizados os domínios de intervalos, octagonal e poliedral, conforme indicado por Gupta e Rybalchenko (2009) em seu trabalho.

4.4 Análise Dinâmica

Similarmente à análise estática, será necessária uma ferramenta para efetuar a análise dinâmica a partir de códigos em LLVM-IR. Para tanto, será utilizado o KLEE, cuja descrição será efetuada a seguir.

4.4.1 KLEE

KLEE é uma ferramenta para execução simbólica que opera sobre programas LLVM-IR bitcode. A ferramenta apresenta diversas otimizações sobre a técnica de execução simbólica, como o uso de heurísticas, representação compacta de estados de um programa e otimizações sobre o uso de *constraint solvers* (CADAR et al., 2008).

O KLEE é uma ferramenta completa para verificação de software, enquanto o foco deste trabalho é a geração de invariantes através de um método mais elaborado. Contudo, para efetuar a verificação de propriedades de um programa, o KLEE gera *restrições* sobre esse programa, sendo possível utilizar as restrições geradas pelo KLEE para cobrir a seção de análise dinâmica da ferramenta.

4.5 Geração de Restrições

Com o objetivo de facilitar a geração de restrições a partir do uso das ferramentas supracitadas, utiliza-se uma terceira ferramenta que já possui integrada a si essa funcionalidade, descrita na subseção 4.5.1.

4.5.1 Map2Check

Map2Check é uma ferramenta para verificação de programas em C que, para facilitar a verificação, os converte em LLVM-IR bitcode (MENEZES et al., 2018b). Sua metodologia inclui o uso de restrições inferidas por técnicas de análise dinâmica por execução simbólica através da ferramenta KLEE e, mais recentemente, análise estática por meio da ferramenta Crab-LLVM. Sendo uma ferramenta completa para verificação de software, o Map2Check realiza operações após a geração de restrições que não integram o método proposto por este trabalho. Contudo, modificações foram efetuadas na ferramenta de forma a permitir que a geração de restrições possa ser efetuada como um passo separado, sem efetuar seus próprios passos de verificação, conforme o método proposto.

4.6 Autômato de Fluxo de Controle

Para a geração de invariantes, utiliza-se o auxílio de SMT Solvers e *templates*. Contudo, SMT Solvers operam melhor sobre modelos matemáticos formais, ao invés de um código-fonte

tradicional (FTSRG, 2019). Assim, o método proposto baseia-se no trabalho de Candeago et al. (2016), convertendo o programa em um sistema de transições que melhor se adequa às necessidades de um SMT Solver.

Uma forma de realizar essa conversão é utilizando autômatos de fluxo de controle (CFA). Ao contrário de um CFA comum, onde transições podem ser atribuições e predicados, aqui estamos interessados apenas nos predicados do programa (e.g. headers de loops e estruturas condicionais). Portanto, transições por atribuições são convertidas em transições vazias. A Figura 21 mostra um exemplo de programa na forma CFA segundo o método proposto, cujo código-fonte pode ser visualizado na Figura 20.

```
1  #include <assert.h>
2  int main(){
3    int i=0, n=__VERIFIER_nondet_int();
4    while(i < n){
5      i++;
6    }
7    assert(i>=n);
8
9    return 0;
10 }
```

Figura 20 – Um programa simples em C

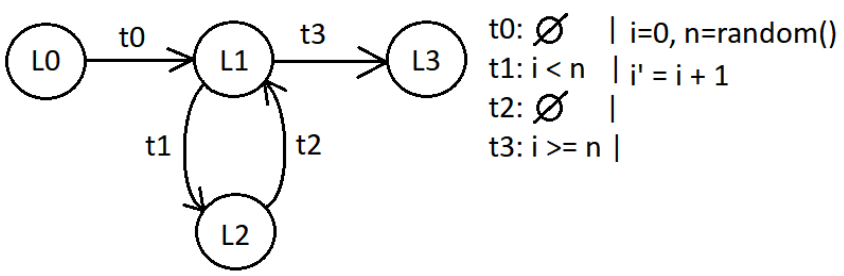


Figura 21 – Conversão do programa descrito na Figura 20 para a forma CFA. Transições são representadas apenas por predicados, porém atribuições são mostradas em separado para melhor compreensibilidade do CFA.

4.7 Programa SAT

Para representar computacionalmente o CFA gerado pelo passo anterior, escreve-se um novo programa em C. O programa deve conter cada uma das transições do CFA e, dependendo da abordagem, as restrições geradas. Cada transição é representada por uma fórmula de satisfabilidade, e complementada com um *template*, descrito em detalhe na Seção 4.7.1.

4.7.1 Template de invariantes

A cada expressão de satisfabilidade representando um ponto de controle do CFA, é associado um *template* da forma $T_j = c_{j1}v_1 + c_{j2}v_2 + ... + c_{jn}v_n \leq d$, onde v_1, v_2...v_n corresponde

ao conjunto de variáveis do programa, c_{j1}, c_{j2}...c_{jn} são constantes de valor desconhecido intervalo [-1,1] e d é uma constante de restrição no conjunto Z^+. Vale ressaltar que este template é baseado no trabalho de Gupta e Rybalchenko (2009) e Candeago et al. (2016). Quando valores são atribuídos a todas as constantes e as inequações construidas são resolvidas, uma invariante será gerada.

Uma expressão A pode também "herdar"um template de uma expressão B, se e somente se houver uma relação de implicação da forma $B \rightarrow A$ (por exemplo, em loops aninhados). A ideia é que estruturas condicionais mais complexas possam ser associadas a invariantes mais expressivas. A Figura 22 descreve a estrutura de um programa SAT.

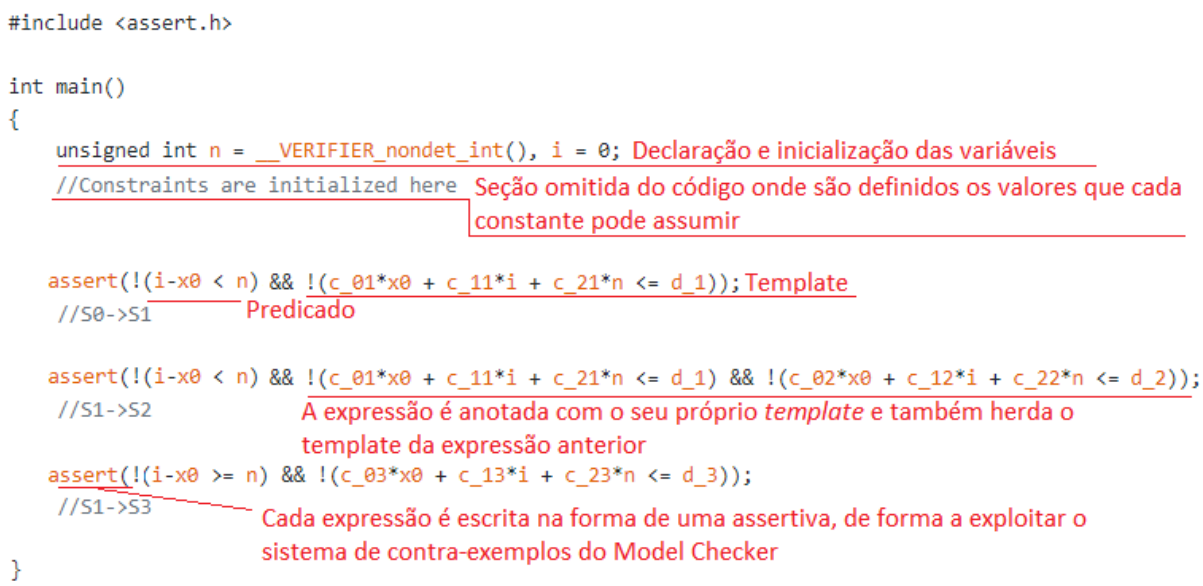


Figura 22 – Exemplo de programa SAT e sua estrutura.

4.8 Solucionador de Restrições

Finalmente, basta apenas gerar a invariante a partir do *template* previamente estabelecido e, dependendo da abordagem, das restrições obtidas. Para tanto, deve-se utilizar um *constraint solver* para obter valores para as constantes do *template*. Uma vez instanciadas as constantes do *template*, a expressão resultante será uma invariante do programa analisado.

Neste trabalho, optou-se pelo uso de um SMT Solver, devido à robustez e relativa simplicidade que os SMT Solvers apresentam em geral, além de ser o método predominante nos trabalhos correlatos analisados.

4.8.1 Z3 - SMT Solver

O Z3 destaca-se por sua robustez, sendo a ferramenta de escolha para vários trabalhos, e.g. Gupta e Rybalchenko (2009) e Srivastava et al. (2009). Foi desenvolvido como ferramenta de suporte para aplicações maiores, e é utilizado em diversos projetos, entre os quais se encontram projetos que utilizam o LLVM, como KLEE (CADAR et al., 2008) e SeaHorn (SEAHORN, 2018c).

Neste trabalho, objetiva-se aproveitar a robustez do Z3, já garantida por diversos trabalhos, e.g. (MOURA; BJØRNER, 2008), para focar os esforços deste trabalho na geração de restrições e em otimizações aplicadas ao próprio código.

4.8.2 ESBMC - Model Checker

Como muitas ferramentas na ciência da computação, o Z3 possui uma linguagem própria, sendo a conversão de códigos C para sua linguagem uma tarefa árdua e complexa. De forma a facilitar esta fase, utiliza-se um Model Checker como front-end para o uso do SMT Solver.

Para este fim, é utilizada a ferramenta ESBMC (GADELHA et al., 2018), que já inclui o Z3 como seu SMT Solver. A ideia é que, ao reescrever o CFA como um código em C, é possível explorar a técnica de geração de contra-exemplos do ESBMC para gerar valores para as incógnitas na forma de um contra-exemplo. A Figura 23 mostra um exemplo de soluções gerado pelo ESBMC para um programa SAT, e as invariantes resultantes.

```
#include <assert.h>

int main()
{
unsigned int n = __VERIFIER_nondet_int(), i = 0;
//Constraints are initialized here

assert((i-x0 < n) && !(c_01*x0 + c_11*i + c_21*n <= d_1)); c_01 = 0, c_11 = -1, c_21 = 1, d_1 = 4 --> -i + n <= 4
//S0->S1

                                                                                    c_01 = 0, c_11 = -1, c_21 = 1, d_1 = 4 -> -i+n <= 4
assert((i-x0 < n) && !(c_01*x0 + c_11*i + c_21*n <= d_1) && !(c_02*x0 + c_12*i + c_22*n <= d_2)); c_02 = 0, c_12 = 0, c_22 = 1, d_2 = 2 -> n <= 2
//S1->S2

assert((i-x0 >= n) && !(c_03*x0 + c_13*i + c_23*n <= d_3)); c_03 = 0, c_13 = 0, c_23 = 1, d_3 = 2 --> n <= 2
//S1->S3

}
```

Figura 23 – Soluções obtidas pelo ESBMC para um programa SAT.

4.9 Instrumentação do código

Um dos diferenciais deste trabalho é a forma como se opera com as invariantes após sua geração. Muitos trabalhos se limitam à geração de invariantes, e.g. Gupta e Rybalchenko (2009) e Kong et al. (2010), devendo o usuário instrumentar o programa com as invariantes

manualmente. Outros trabalhos geram ferramentas completas de verificação, e.g. Srivastava et al. (2009), sendo portanto mais completos, porém limitando seu método à ferramenta utilizada.

Este trabalho oferece uma solução intermediária, limitando-se à geração de invariantes, porém instrumentando o programa com as invariantes automaticamente. Através dessa abordagem, o método mantém-se integrável a ferramentas de verificação, ao mesmo tempo que reduz o esforço manual do usuário.

A Figura 24 ilustra a ideia de instrumentação de código, utilizando a instrução *assume* para inserir a invariante no código. A adição dessa linha de código possui impacto positivo em ferramentas de verificação de software (ROCHA et al., 2015).

```
1  int main()
2  {
3   unsigned int n = __VERIFIER_nondet_uint();
4   unsigned int x=n, y=0;
5
6   __VERIFIER_assume(y + n <= 10);
7   __VERIFIER_assume(y <= 10);
8   while(x>0)
9   {
10    x--;
11    y++;
12    __VERIFIER_assume(y + n <= 10);
13    __VERIFIER_assume(y <= 10);
14  }
15  __VERIFIER_assume(y <= 10);
16  assert(y==n);
17 }
```

Figura 24 – Instrumentação de invariante em código C, na forma instruções *assume*

5 RESULTADOS EXPERIMENTAIS

Essa seção descreve o planejamento, projeto, execução e análise dos resultados de um estudo experimental para avaliar o método proposto neste trabalho.

5.1 Planejamento e projeto dos experimentos

Esta avaliação experimental objetiva analisar a eficácia das três variantes do método proposto sob dois ângulos: Individualmente, no que diz respeito à capacidade de cada invariante gerada de reduzir o espaço de estados a um ponto que torne possível a verificação de programas ; e comparativamente, onde busca-se descobrir, dentre as três variantes do método proposto, qual possui a melhor performance. Tendo em vista estes objetivos, foram definidas as seguintes questões de pesquisa:

QP1 - As invariantes geradas pelo método reduzem o espaço de busca o suficiente para que a verificação do programa analisado seja executada em tempo hábil?

QP2 - Qual é a forma mais eficaz de integrar o uso de restrições à técnica de geração de invariantes?

QP3 - Qual das variantes do método proposto mais acelera o processo de verificação?

Para responder a estas questões, foram utilizados 10 programas da categoria *Reach-Safety*, subcategoria *Loops*, do benchmark da *Competition on Software Verification (SV-COMP)* (BEYER, 2019), além de 2 programas adicionais proposto pelo autor deste trabalho. O SV-COMP é uma competição internacional que objetiva avaliar diversas ferramentas de verificação com base em *benchmarks* próprios (incluindo programas de domínios externos, como drivers e o kernel do Linux). Na categoria Reach-Safety, é dado um programa e uma propriedade ou conjunto de propriedades associadas a este programa, sendo o objetivo da ferramenta de verificação gerar um dos seguintes vereditos: `True`, se todas as propriedades forem verdadeiras, ou `False`, se alguma das propriedades for incorreta. Contudo, em alguns casos, a ferramenta pode gerar um terceiro veredito, `Unknown`, se for impossível para a ferramenta obter uma resposta.

A análise experimental foi restrita a programas da subcategoria Loops, pois, conforme descrito em seções anteriores, a maior causa de explosões de estados são programas com *loops*. Os programas selecionados incluíam programas com quantidades relativamente grandes de variáveis, loops aninhados, loops infinitos (cuja execução nunca termina) e loops não-determinísticos (que podem ter um fim ou não). Contudo, uma vez que a literatura afirma que casos de teste onde todas as propriedades são verdadeiras são mais difíceis de avaliar do que casos de teste que

possuam alguma propriedade falsa (BEYER, 2019), todos os programas escolhidos esperam o veredito `True`.

O método proposto foi simulado manualmente, de forma que cada variante do método constituiu um experimento diferente. Para realizar a avaliação, foi empregado o auxílio da ferramenta de verificação ESBMC (GADELHA et al., 2018). Cada experimento instanciava uma nova versão dos códigos originais do *benchmark*, anotada com templates e/ou restrições, e a avaliação consistiu em observar o desempenho da ferramenta ESBMC sobre cada uma dessas versões.

Os experimentos foram efetuados em uma máquina virtual com as seguintes configurações: Sistema Operacional Linux Mint, versão 19.1 Cinnamon, 64 bits; 1660 MB de memória RAM, 126.50 GB de HD, processador Intel Core I5, sem nenhuma aplicação externa sendo executada com o objetivo de evitar imprecisão nos dados. Todos os experimentos e resultados estão disponíveis em <https://github.com/VictorDeluca/Inv_Generation_Tests>.

5.2 Execução e análise dos experimentos

O primeiro experimento buscou avaliar a eficácia do ESBMC em relação ao *benchmark* descrito, sem o auxílio de invariantes ou restrições, servindo como base para comparar os experimentos posteriores. Para tanto, foi utilizado um limite de desdobramentos maior que as outras instâncias, sendo cada programa limitado a um máximo de 100 desdobramentos. Os resultados são descritos na Tabela 1, onde cada linha descreve um programa do *benchmark*. Foi definido, na verificação, um limite de 30 segundos como tempo máximo (`Timeout`).

Tabela 1 – Experimento 1: Avaliação do ESBMC pleno

ID	**Nome do programa**	**Tempo de verificação**	**Resultado da verificação**	**Resultado esperado**
1	count-up-down-1.c	0.040s	Unknown	True
2	Mono1-1-2.c	0.152s	Unknown	True
3	sample-loop-1.c	0.002s	Unknown	True
4	sample-loop-2.c	0.013s	Unknown	True
5	while-infinite-loop-1.c	0.000s	Unknown	True
6	eq1.c	Timeout	N/A	True
7	half.c	0.069s	Unknown	True
8	nested-1.c	Timeout	N/A	True
9	gauss-sum.c	0.040s	Unknown	True
10	even.c	0.002s	Unknown	True
11	id-trans.c	0.021s	Unknown	True
12	gr2006.c	0.000	Unknown	True

O experimento indica que, conforme esperado, o *model checker por* ESBMC adotando somente a técnica BMC, foi incapaz de lidar com a explosão de estados causada por um loop,

mesmo em casos de teste simples. Na maioria dos casos, a ferramenta se declara incapaz de verificar o programa, porém em alguns é impossível obter qualquer resposta, resultando em um Timeout.

O segundo experimento objetivou avaliar a performance das restrições geradas por análise estática e dinâmica, sem o auxílio das invariantes, com o objetivo de observar se estas possuem impacto na verificação e, posteriormente, comparar sua performance individual com sua integração ao método proposto.

Contudo, as análises estática e dinâmica falharam em gerar restrições para alguns dos códigos do *benchmark*. Como consequência, os códigos modificados não diferem dos originais, já avaliados no experimento anterior, e portanto sua análise foi omitida deste experimento. Os resultados estão descritos na Tabela 2.

Tabela 2 – Experimento 2: Avaliação do ESBMC com restrições

ID	**Nome do programa**	**Geração de restrições**	**Tempo de verificação**	**Resultado da verificação**	**Resultado esperado**
1	count-up-down-1.c	0.06s	0.028s	True	True
2	Mono1-1-2.c	0.07s	0.000s	Unknown	True
6	eq1.c	0.13s	0.131s	Unknown	True
7	half.c	0.06s	0.069s	True	True
8	nested-1.c	0.11s	0.232s	Unknown	True
9	gauss-sum.c	0.05s	0.090s	Unknown	True
10	even.c	0.05s	0.011s	Unknown	True
11	id-trans.c	0.05s	0.077s	Unknown	True
12	gr2006.c	0.07s	0.000s	Unknown	True

Pode-se notar que há uma grande melhoria na performance do ESBMC. Dois programas que antes resultavam em `Unknown` agora geram `True`, a resposta correta, enquanto os programas que davam Timeout passam a gerar uma resposta em tempo hábil, embora a resposta gerada seja `Unknown`. Os resultados mostram que as restrições por si sós reduzem visivelmente o espaço de busca dos programas analisados, porém a redução ainda não atinge um nível satisfatório, uma vez que a ferramenta não foi capaz de obter uma resposta para a maioria dos programas analisados.

O terceiro experimento objetivou avaliar a precisão das invariantes geradas a partir do método proposto, sem o auxílio de restrições. Similarmente ao experimento anterior, seu propósito foi avaliar a influência das invariantes geradas na verificação, e compará-la com as diferentes abordagens do método proposto, que integram o uso de restrições. Seus resultados podem ser analisados na Tabela 3.

Os resultados obtiveram resultados muito positivos, mostrando uma acurácia de 83% e obtendo sucesso em verificar 10 dentre os 12 programas selecionados. Em outras palavras, mesmo sem o auxílio de restrições, as invariantes geradas causam uma grande melhoria na eficiência da verificação dos programas analisados.

O quarto experimento é referente à primeira abordagem do método proposto, em que as restrições são incorporadas ao código em C em conjunto com as invariantes geradas, e seu objetivo é verificar a eficácia da abordagem. Nesse ponto, é importante lembrar que os programas de IDs 3 a 5 não possuem restrições associadas a si, devido a limitações das ferramentas utilizadas. Portanto, estes programas foram omitidos da tabela, tendo em vista que já foram analisados no experimento anterior. Além disso, o tempo de geração de invariantes e restrições foi omitido,

Tabela 3 – Experimento 3: Avaliação do ESBMC com invariantes

ID	**Nome do programa**	**Geração de invariantes**	**Tempo de verificação**	**Resultado da verificação**	**Resultado esperado**
1	count-up-down-1.c	0.004s	0.088s	True	True
2	Mono1-1-2.c	0.004s	0.008s	True	True
3	sample-loop-1.c	0.034s	0.051s	True	True
4	sample-loop-2.c	0.043s	0.038	True	True
5	while-infinite-loop-1.c	0.003	0.000s	Unknown	True
6	eq1.c	0.172s	1.780s	True	True
7	half.c	0.039s	0.046s	True	True
8	nested-1.c	0.107s	0.723s	True	True
9	gauss-sum.c	0.029s	0.067s	True	True
10	even.c	0.047s	0.010s	True	True
11	id-trans.c	0.154s	0.064s	True	True
12	gr2006.c	0.009s	0.000s	Unknown	True

pois já foi mencionado nos experimentos anteriores. A Tabela 4 descreve os resultados da experimentação.

Tabela 4 – Experimento 4: Avaliação da primeira abordagem

ID	**Nome do programa**	**Tempo de verificação**	**Resultado da verificação**	**Resultado esperado**
1	count-up-down-1.c	0.048s	True	True
2	Mono1-1-2.c	0.009s	True	True
6	eq1.c	0.936s	True	True
7	half.c	0.062s	True	True
8	nested-1.c	1.222s	True	True
9	gauss-sum.c	0.115s	True	True
10	even.c	0.018s	True	True
11	id-trans.c	0.102s	True	True
12	gr2006.c	0.000s	Unknown	True

Surpreendentemente, os resultados não mostram grandes melhorias em comparação com o uso individual de invariantes. Enquanto alguns programas tiveram seu tempo de verificação reduzido, outros sofreram um leve aumento no tempo de verificação, e não houve nenhuma alteração no número de programas verificados corretamente. Conjectura-se que as instâncias em que o desempenho piora se deve ao overhead de grandes quantidades de instruções *assume*, ao ponto de não compensar o ganho no número de estados reduzidos.

O quinto experimento, por sua vez, refere-se à segunda abordagem do método proposto, em que as restrições geradas são incorporadas ao CFA do programa, de forma a gerar um conjunto mais expressivo de invariantes. A Tabela 5 descreve os resultados obtidos.

Tabela 5 – Experimento 5: Avaliação da segunda abordagem

ID	**Nome do programa**	**Geração de invariantes**	**Tempo de verificação**	**Resultado da verificação**	**Resultado esperado**
1	count-up-down-1.c	0.028s	0.037s	True	True
2	Mono1-1-2.c	0.009s	N/A	Unknown	True
6	eq1.c	0.134s	0.347s	True	True
7	half.c	0.025s	0.029s	True	True
8	nested-1.c	0.120s	0.539s	True	True
9	gauss-sum.c	0.002s	N/A	Unknown	True
10	even.c	0.005s	0.012s	Unknown	True
11	id-trans.c	0.091s	0.088s	True	True
12	gr2006.c	0.004s	0.000s	Unknown	True

Por um lado, observa-se que esta abordagem é claramente superior às outras em termos de velocidade. Ganhos em tempo de verificação são notáveis, em particular, nos programas 6 e 8. Ainda, o tempo para geração de invariantes não sofre alterações notáveis. Contudo, a precisão é inferior à das abordagens anteriores (invariantes puras e invariantes com restrições).

O motivo para a precisão reduzida é que, em alguns casos, o ESBMC não consegue preencher os templates com valores que respeitem as restrições inferidas, resultando na omissão de invariantes em diversos trechos de código de alguns programas. Conjectura-se que a conversão do programa para uma forma SAT já configura uma redução suficiente no espaço de estados do programa para uma geração de invariantes eficiente, enquanto restrições adicionais limitam exageradamente as soluções possíveis dentro desta configuração adotada pelo método proposto.

Finalmente, o sexto e último experimento consiste em avaliar a terceira abordagem, que busca unir os dois métodos anteriores ao utilizar as restrições no progama SAT e na verificação ao mesmo tempo. A Tabela 6 mostra os resultados obtidos. Uma vez que os programas 2 e 10 não geram invariantes pela segunda abordagem, estes também não foram inclusos na tabela.

Novamente, o uso de restrições na verificação não gerou uma diferença notável. Embora em alguns casos o tempo tenha melhorado, em outros o tempo piorou em igual medida, sendo difícil decidir entre ambos qual deles é a melhor opção para acelerar a verificação.

Analisando os resultados obtidos nota-se que o uso de restrições na verificação possua algum impacto, este não é muito impactante em comparação com as invariantes geradas pelo método usando a ferramenta Crab-LLVM. Como consequência, o uso ou não de restrições neste ponto é irrelevante. Além disso, a geração de invariantes guiada por restrições apresenta um *trade-off*. Por um lado, as invariantes geradas usando *templates* são mais precisas, o que facilita a verificação. Por outro, existe o risco de as restrições limitarem demais a geração das invariantes,

Tabela 6 – Experimento 6: Avaliação da terceira abordagem

ID	**Nome do programa**	**Tempo de verificação**	**Resultado da verificação**	**Resultado esperado**
1	count-up-down-1.c	0.037s	True	True
6	eq1.c	0.156s	True	True
7	half.c	0.056s	True	True
8	nested-1.c	1.193s	True	True
10	even.c	0.013s	Unknown	True
11	id-trans.c	0.088s	True	True
12	gr2006.c	0.000s	Unknown	True

ao ponto de torná-la impossível. Finalmente, conclui-se que todas as três abordagens para o método proposto são utilizáveis, porém o uso de invariantes sem restrições, na configuração adotada no método proposto, apresenta também bons resultados. Contudo, vale ressaltar que uma experimentação com um número maior de casos de teste deverá ainda ser implementada, para se identificar limitações e melhorias.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Este trabalho apresentou um método para gerar e instrumentar invariantes de programas em C por meio de *templates*, com o objetivo de auxiliar ferramentas de verificação de software. A metodologia proposta apresenta embasamento teórico, com trabalhos similares apresentando sucesso, e busca integrar-se a ferramentas recentes para renovar técnicas antigas.

O método proposto baseia-se na ideia de integrar o uso de restrições, baseadas em análise estática e dinâmica, a invariantes geradas por meio de *templates*. São propostas e avaliadas três abordagens que realizam esta integração de formas diferentes. Os resultados obtidos validam a eficácia do método proposto, mostrando que as invariantes geradas por este possuem de fato grande influência na verificação, atingindo uma acurácia de 83% nas melhores abordagens.

Como trabalhos futuros, recomenda-se a automatização do método proposto, uma vez que uma de suas qualidades é ser automatizável. Também seria interessante realizar uma avaliação do método com um conjunto de testes maior, inclusive com o objetivo de compará-lo a trabalhos similares. Para tanto, a automatização do método é essencial, uma vez que a simulação manual do método implica em grandes custos de tempo e esforço. Outra possibilidade é um estudo mais profundo do emprego de restrições por análise estática e dinâmica, e como estas poderiam ser aplicadas de forma a melhorar a performance da verificação, uma vez que não ofereceram um grande impacto nas abordagens propostas por este trabalho.

REFERÊNCIAS

BADROS, G. J.; BORNING, A.; STUCKEY, P. J. The cassowary linear arithmetic constraint solving algorithm. **ACM Trans. Comput.-Hum. Interact.**, ACM, New York, NY, USA, v. 8, n. 4, p. 267–306, dez. 2001. ISSN 1073-0516. Disponível em: <http://doi.acm.org/10.1145/504704.504705>. Citado na página 25.

BAIER, C.; KATOEN, J.-P. **Principles of Model Checking (Representation and Mind Series)**. [S.l.]: The MIT Press, 2008. ISBN 026202649X, 9780262026499. Citado na página 10.

BENSALEM, S.; LAKHNECH, Y. Automatic generation of invariants. **Form. Methods Syst. Des.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 15, n. 1, p. 75–92, jul. 1999. ISSN 0925-9856. Disponível em: <https://doi.org/10.1023/A:1008744030390>. Citado na página 10.

BEYER, D. Second competition on software verification. In: **Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer-Verlag, 2013. (TACAS'13), p. 594–609. ISBN 978-3-642-36741-0. Disponível em: <http://dx.doi.org/10.1007/978-3-642-36742-7_43>. Citado na página 11.

BEYER, D. Automatic verification of c and java programs: Sv-comp 2019. In: BEYER, D. et al. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. Cham: Springer International Publishing, 2019. p. 133–155. ISBN 978-3-030-17502-3. Citado 2 vezes nas páginas 42 e 43.

BEYER, D.; DANGL, M.; WENDLER, P. Combining k-induction with continuously-refined invariants. **CoRR**, abs/1502.00096, 2015. Disponível em: <http://arxiv.org/abs/1502.00096>. Citado na página 21.

BEYER, D. et al. Path invariants. In: **Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2007. (PLDI '07), p. 300–309. ISBN 978-1-59593-633-2. Disponível em: <http://doi.acm.org/10.1145/1250734.1250769>. Citado 6 vezes nas páginas 5, 18, 20, 22, 29 e 30.

BIERE, A. Lingeling and friends at the sat competition 2011. In: . [S.l.: s.n.], 2011. Citado na página 25.

BRADLEY, A. R.; MANNA, Z. **The Calculus of Computation: Decision Procedures with Applications to Verification**. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN 3540741127. Citado na página 14.

BRUMMAYER, R.; BIERE, A. Boolector: An efficient smt solver for bit-vectors and arrays. In: **Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,**. Berlin, Heidelberg: Springer-Verlag, 2009. (TACAS '09), p. 174–177. ISBN 978-3-642-00767-5. Disponível em: <http://dx.doi.org/10.1007/978-3-642-00768-2_16>. Citado na página 26.

BSHOUTY, N. H. Exact learning via the monotone theory. In: **Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science**. [S.l.: s.n.], 1993. p. 302–311. Citado na página 27.

CADAR, C.; DUNBAR, D.; ENGLER, D. 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**. Berkeley, CA, USA: USENIX Association, 2008. (OSDI'08), p. 209–224. Disponível em: <http://dl.acm.org/citation.cfm?id=1855741.1855756>. Citado 2 vezes nas páginas 37 e 40.

CANDEAGO, L. et al. Speeding up the constraint-based method in difference logic. In: CREIGNOU, N.; BERRE, D. L. (Ed.). **Theory and Applications of Satisfiability Testing – SAT 2016**. Cham: Springer International Publishing, 2016. p. 284–301. ISBN 978-3-319-40970-2. Citado 4 vezes nas páginas 31, 32, 38 e 39.

CHAMPION, A. et al. The kind 2 model checker. In: **CAV**. [S.l.: s.n.], 2016. Citado na página 28.

CLAESSEN, K. et al. Sat-solving in practice. In: **2008 9th International Workshop on Discrete Event Systems**. [S.l.: s.n.], 2008. p. 61–67. Citado 2 vezes nas páginas 25 e 26.

CLANG. **Clang**. Clang, 2018. Disponível em: <https://clang.llvm.org/>. Citado na página 35.

CLARKE, E. et al. Counterexample-guided abstraction refinement for symbolic model checking. **J. ACM**, ACM, New York, NY, USA, v. 50, n. 5, p. 752–794, set. 2003. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/876638.876643>. Citado na página 29.

CLARKE, E. M.; SCHLINGLOFF, B.-H. Chapter 24 - model checking. In: ROBINSON, A.; VORONKOV, A. (Ed.). **Handbook of Automated Reasoning**. Amsterdam: North-Holland, 2001, (Handbook of Automated Reasoning). p. 1635 – 1790. ISBN 978-0-444-50813-3. Disponível em: <http://www.sciencedirect.com/science/article/pii/B9780444508133500266>. Citado na página 17.

CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. Smt-based bounded model checking for embedded ansi-c software. In: **Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. (ASE '09), p. 137–148. ISBN 978-0-7695-3891-4. Disponível em: <https://doi.org/10.1109/ASE.2009.63>. Citado 2 vezes nas páginas 10 e 11.

COUSOT, P.; COUSOT, R. A gentle introduction to formal verification of computer systems by abstract interpretation. In: ESPARZA, J.; GRUMBERG, O.; BROY, M. (Ed.). **Logics and Languages for Reliability and Security**. IOS Press, 2010, (NATO Science Series III: Computer and Systems Sciences). p. 1–29. Disponível em: <https://hal.inria.fr/inria-00543886>. Citado 4 vezes nas páginas 15, 16, 17 e 18.

CSALLNER, C.; TILLMANN, N.; SMARAGDAKIS, Y. Dysy: Dynamic symbolic execution for invariant inference. In: **Proceedings of the 30th International Conference on Software Engineering**. New York, NY, USA: ACM, 2008. (ICSE '08), p. 281–290. ISBN 978-1-60558-079-1. Disponível em: <http://doi.acm.org/10.1145/1368088.1368127>. Citado na página 24.

ERNST, M. D. et al. The daikon system for dynamic detection of likely invariants. **Sci. Comput. Program.**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 69, n. 1-3, p. 35–45, dez. 2007. ISSN 0167-6423. Disponível em: <http://dx.doi.org/10.1016/j.scico.2007.01.015>. Citado 3 vezes nas páginas 5, 22 e 23.

FEAUTRIER, P.; GONNORD, L. Accelerated invariant generation for c programs with aspic and c2fsm. **Electronic Notes in Theoretical Computer Science**, v. 267, n. 2, p. 3 – 13, 2010. ISSN 1571-0661. Proceedings of the Tools for Automatic Program AnalysiS (TAPAS). Disponível em: <http://www.sciencedirect.com/science/article/pii/S157106611000143X>. Citado na página 21.

FOSTER, J. **Symbolic Execution**. 2011. University Lecture. Citado 2 vezes nas páginas 5 e 24.

FOULADGAR, H.; MINAEI-BIDGOLI, B.; PARVIN, H. On possibility of conditional invariant detection. In: **Proceedings of the 15th International Conference on Knowledge-based and Intelligent Information and Engineering Systems - Volume Part II**. Berlin, Heidelberg: Springer-Verlag, 2011. (KES'11), p. 214–224. ISBN 978-3-642-23862-8. Disponível em: <http://dl.acm.org/citation.cfm?id=2041341.2041366>. Citado 4 vezes nas páginas 11, 18, 19 e 20.

FTSRG. **Software Verification**. 2019. Citado 2 vezes nas páginas 33 e 38.

GADELHA, M. R. et al. ESBMC 5.0: An industrial-strength C model checker. In: 33^{rd} **ACM/IEEE Int. Conf. on Automated Software Engineering (ASE'18)**. New York, NY, USA: ACM, 2018. p. 888–891. Citado 2 vezes nas páginas 40 e 43.

GANESH, V. **Decision Procedures for Bit-vectors, Arrays and Integers**. Tese (Doutorado), Stanford, CA, USA, 2007. AAI3281841. Citado na página 26.

GRUMBERG, O.; VEITH, H. (Ed.). **25 Years of Model Checking: History, Achievements, Perspectives**. Berlin, Heidelberg: Springer-Verlag, 2008. ISBN 978-3-540-69849-4. Citado na página 10.

GULWANI, S.; MCCLOSKEY, B.; TIWARI, A. Lifting abstract interpreters to quantified logical domains. In: **Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 2008. (POPL '08), p. 235–246. ISBN 978-1-59593-689-9. Disponível em: <http://doi.acm.org/10.1145/1328438.1328468>. Citado na página 30.

GUPTA, A.; MAJUMDAR, R.; RYBALCHENKO, A. From tests to proofs. In: **Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,**. Berlin, Heidelberg: Springer-Verlag, 2009. (TACAS '09), p. 262–276. ISBN 978-3-642-00767-5. Disponível em: <http://dx.doi.org/10.1007/978-3-642-00768-2_24>. Citado na página 29.

GUPTA, A.; RYBALCHENKO, A. Invgen: An efficient invariant generator. In: **Proceedings of the 21st International Conference on Computer Aided Verification**. Berlin, Heidelberg: Springer-Verlag, 2009. (CAV '09), p. 634–640. ISBN 978-3-642-02657-7. Disponível em: <http://dx.doi.org/10.1007/978-3-642-02658-4_48>. Citado 11 vezes nas páginas 5, 10, 19, 20, 21, 28, 33, 35, 36, 39 e 40.

HALBWACHS, N. et al. The synchronous data flow programming language lustre. **Proceedings of the IEEE**, v. 79, n. 9, p. 1305–1320, Sept 1991. ISSN 0018-9219. Citado na página 28.

HALBWACHS, N.; PÉRON, M. Discovering properties about arrays in simple programs. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 43, n. 6, p. 339–348, jun. 2008. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1379022.1375623>. Citado na página 30.

HENRY, J.; MONNIAUX, D.; MOY, M. Pagai: A path sensitive static analyser. **Electron. Notes Theor. Comput. Sci.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 289, p. 15–25, dez. 2012. ISSN 1571-0661. Disponível em: <http://dx.doi.org/10.1016/j.entcs.2012.11.003>. Citado 2 vezes nas páginas 17 e 21.

HODER, K.; KOVáCS, L.; VORONKOV, A. Interpolation and symbol elimination in vampire. In: **Proceedings of the 5th International Conference on Automated Reasoning**. Berlin, Heidelberg: Springer-Verlag, 2010. (IJCAR'10), p. 188–195. ISBN 3-642-14202-8, 978-3-642-14202-4. Disponível em: <http://dx.doi.org/10.1007/978-3-642-14203-1_16>. Citado na página 10.

HODER, K.; KOVÁCS, L.; VORONKOV, A. Case studies on invariant generation using a saturation theorem prover. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)|Lect. Notes Comput. Sci.** [S.l.: s.n.], 2011. v. 7094, p. 1–15. ISBN 9783642253232. Citado 2 vezes nas páginas 10 e 11.

JHALA, R.; MCMILLAN, K. L. Array abstractions from proofs. In: DAMM, W.; HERMANNS, H. (Ed.). **Computer Aided Verification**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 193–206. ISBN 978-3-540-73368-3. Citado na página 30.

KAHSAI, T.; GE, Y.; TINELLI, C. Instantiation-based invariant discovery. In: **Proceedings of the Third International Conference on NASA Formal Methods**. Berlin, Heidelberg: Springer-Verlag, 2011. (NFM'11), p. 192–206. ISBN 978-3-642-20397-8. Disponível em: <http://dl.acm.org/citation.cfm?id=1986308.1986326>. Citado 2 vezes nas páginas 20 e 28.

KING, J. C. Symbolic execution and program testing. **Commun. ACM**, ACM, New York, NY, USA, v. 19, n. 7, p. 385–394, jul. 1976. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/360248.360252>. Citado 2 vezes nas páginas 23 e 24.

KLEIN, R. The role of constraints in geometric modelling. In: ____. **Geometric Constraint Solving and Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 3–23. ISBN 978-3-642-58898-3. Disponível em: <https://doi.org/10.1007/978-3-642-58898-3_1>. Citado na página 25.

KONG, S. et al. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In: **Proceedings of the 8th Asian Conference on Programming Languages and Systems**. Berlin, Heidelberg: Springer-Verlag, 2010. (APLAS'10), p. 328–343. ISBN 3-642-17163-X, 978-3-642-17163-5. Disponível em: <http://dl.acm.org/citation.cfm?id=1947873.1947904>. Citado 5 vezes nas páginas 20, 21, 27, 28 e 40.

KRKA, I. et al. Using dynamic execution traces and program invariants to enhance behavioral model inference. In: **2010 ACM/IEEE 32nd International Conference on Software Engineering**. [S.l.: s.n.], 2010. v. 2, p. 179–182. ISSN 1558-1225. Citado na página 19.

KROENING, D. et al. Loop summarization using abstract transformers. In: **Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis**. Berlin, Heidelberg: Springer-Verlag, 2008. (ATVA '08), p. 111–125. ISBN 978-3-540-88386-9. Disponível em: <http://dx.doi.org/10.1007/978-3-540-88387-6_10>. Citado na página 21.

KRöNING, D.; WEISSENBACHER, G. **A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard**. 2016. Citado na página 26.

KU, K. et al. A buffer overflow benchmark for software model checkers. In: **Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: ACM, 2007. (ASE '07), p. 389–392. ISBN 978-1-59593-882-4. Disponível em: <http://doi.acm.org/10.1145/1321631.1321691>. Citado na página 29.

LLVM. **LLVM**. LLVM, 2018. Disponível em: <https://llvm.org/>. Citado na página 35.

MAJEK. **IR is better than assembly**. 2018. Disponível em: <https://idea.popcount.org/2013-07-24-ir-is-better-than-assembly/>. Citado 2 vezes nas páginas 5 e 36.

MENEZES, R. et al. Map2check using llvm and klee. In: BEYER, D.; HUISMAN, M. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. Cham: Springer International Publishing, 2018. p. 437–441. ISBN 978-3-319-89963-3. Citado na página 24.

MENEZES, R. et al. Map2check using llvm and klee. In: BEYER, D.; HUISMAN, M. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. Cham: Springer International Publishing, 2018. p. 437–441. ISBN 978-3-319-89963-3. Citado na página 37.

MERZ, F.; FALKE, S.; SINZ, C. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In: ____. **Verified Software: Theories, Tools, Experiments**. [s.n.], 2012. p. 146–161. Exported from https://app.dimensions.ai on 2018/11/30. Disponível em: <https://app.dimensions.ai/details/publication/pub.1006497645andhttp://baldur.iti.kit.edu/~falke/papers/VSTTE12b.pdf>. Citado na página 10.

MINÉ, A. The octagon abstract domain. **Higher Order Symbol. Comput.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 19, n. 1, p. 31–100, mar. 2006. ISSN 1388-3690. Disponível em: <http://dx.doi.org/10.1007/s10990-006-8609-1>. Citado 2 vezes nas páginas 17 e 21.

MINTON, S. et al. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In: **Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 1**. AAAI Press, 1990. (AAAI'90), p. 17–24. ISBN 0-262-51057-X. Disponível em: <http://dl.acm.org/citation.cfm?id=1865499.1865502>. Citado na página 25.

MØLLER, A.; SCHWARTZBACH, M. I. **Static Program Analysis**. 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`. Citado na página 21.

MOSKEWICZ, M. W. et al. Chaff: Engineering an efficient sat solver. In: **Proceedings of the 38th Annual Design Automation Conference**. New York, NY, USA: ACM, 2001. (DAC '01), p. 530–535. ISBN 1-58113-297-2. Disponível em: <http://doi.acm.org/10.1145/378239.379017>. Citado na página 25.

MOURA, L. D.; BJØRNER, N. Z3: An efficient smt solver. In: **Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer-Verlag, 2008. (TACAS'08/ETAPS'08), p. 337–340. ISBN 3-540-78799-2, 978-3-540-78799-0. Disponível em: <http://dl.acm.org/citation.cfm?id=1792734.1792766>. Citado 4 vezes nas páginas 26, 29, 34 e 40.

MOY, Y.; MARCHÉ, C. Checking memory safety with abstract interpretation and deductive verification . In: . [S.l.: s.n.], 2007. Citado na página 16.

NGUYEN, T. et al. Using dynamic analysis to generate disjunctive invariants. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: ACM, 2014. (ICSE 2014), p. 608–619. ISBN 978-1-4503-2756-5. Disponível em: <http://doi.acm.org/10.1145/2568225.2568275>. Citado 3 vezes nas páginas 5, 19 e 23.

OCAML. **OCaml**. OCaml, 2018. Disponível em: <https://ocaml.org/>. Citado na página 27.

OUYANG, M. **How Good Are Branching Rules in DPLL?** [S.l.], 1996. Citado na página 25.

PAPE, C. L.; A, I. S. **Constraint-Based Scheduling: A Tutorial**. 2005. Citado na página 25.

PEI, Y. et al. Automated fixing of programs with contracts. **IEEE Transactions on Software Engineering**, v. 40, n. 5, p. 427–449, May 2014. ISSN 0098-5589. Citado na página 19.

PIPATSRISAWAT, K.; DARWICHE, A. **RSat 2.0: SAT Solver Description**. [S.l.], 2007. Citado na página 25.

RANISE, S.; LORIA; TINELLI, C. **The SMT-LIB Standard: Version 1.2**. [S.l.], 2006. Citado na página 26.

ROCHA, H. et al. **Exploiting Safety Properties in Bounded Model Checking for Test Cases Generation of C Programs**. 2010. Citado na página 10.

ROCHA, H. et al. Model checking embedded C software using k-induction and invariants (extended version). **CoRR**, abs/1509.02471, 2015. Disponível em: <http://arxiv.org/abs/1509.02471>. Citado 2 vezes nas páginas 18 e 41.

ROCHA, H. O. **Verificação de Sistemas de Software baseada em Transformações de Código usando Bounded Model Checking**. Tese (Doutorado) — Universidade Federal do Amazonas, Biblioteca Digital de Teses e Dissertações da UFAM, 2015. Citado 3 vezes nas páginas 10, 11 e 12.

ROCHA, W. et al. Depthk: A k-induction verifier based on invariant inference for c programs. In: . [S.l.]: Springer, Berlin, Heidelberg, 2017. (Lecture Notes in Computer Science, v. 10206), p. 360–364. Citado na página 21.

ROSSI, F.; BEEK, P. v.; WALSH, T. **Handbook of Constraint Programming (Foundations of Artificial Intelligence)**. New York, NY, USA: Elsevier Science Inc., 2006. ISBN 0444527265. Citado na página 25.

ROUX, P. et al. A generic ellipsoid abstract domain for linear time invariant systems. In: **HSCC**. [S.l.: s.n.], 2012. Citado na página 17.

SCHEDER, D.; STEINBERGER, J. P. Ppsz for general k-sat: Making hertli's analysis simpler and 3-sat faster. In: **Proceedings of the 32Nd Computational Complexity Conference**. Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. (CCC '17), p. 9:1–9:15. ISBN 978-3-95977-040-8. Disponível em: <https://doi.org/10.4230/LIPIcs.CCC.2017.9>. Citado na página 25.

SEAHORN. **Crab**. Github, 2018. Disponível em: <https://github.com/seahorn/crab>. Citado na página 36.

SEAHORN. **Crab-LLVM**. Github, 2018. Disponível em: <https://github.com/seahorn/crab-llvm>. Citado 2 vezes nas páginas 29 e 36.

SEAHORN. **Seahorn**. Github, 2018. Disponível em: <https://github.com/seahorn/seahorn>. Citado 2 vezes nas páginas 21 e 40.

SRIVASTAVA, S.; GULWANI, S. Program verification using templates over predicate abstraction. In: **Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2009. (PLDI '09), p. 223–234. ISBN 978-1-60558-392-1. Disponível em: <http://doi.acm.org/10.1145/1542476.1542501>. Citado 6 vezes nas páginas 19, 20, 27, 30, 31 e 35.

SRIVASTAVA, S.; GULWANI, S.; FOSTER, J. S. Vs3: Smt solvers for program verification. In: **Proceedings of the 21st International Conference on Computer Aided Verification**. Berlin, Heidelberg: Springer-Verlag, 2009. (CAV '09), p. 702–708. ISBN 978-3-642-02657-7. Disponível em: <http://dx.doi.org/10.1007/978-3-642-02658-4_58>. Citado 4 vezes nas páginas 21, 31, 40 e 41.

WONG, W. E. et al. Recent catastrophic accidents: Investigating how software was responsible. In: **2010 Fourth International Conference on Secure Software Integration and Reliability Improvement**. [S.l.: s.n.], 2010. p. 14–22. Citado na página 10.