Geração de Invariantes Baseada em Templates para Verificação Eficiente de Programas em C

Victor Deluca Almirante Gomes

Orientador: Prof. Dr. Herbert Oliveira Rocha

10 de julho de 2019

Universidade Federal de Roraima

Departamento de Ciência da Computação

UFRR

1. **Introdução**
2. Conceitos e Definições
3. Trabalhos Correlatos
4. Método Proposto
5. Avaliação Experimental
6. Considerações Finais

- Erros no Desenvolvimento de Software podem causar catástrofes.
- Porém alguns erros podem ser difíceis de identificar.
- A Verificação de Software busca apontar tais erros.

```c
#include <stdio.h>

int somar(int a, int b){
    return a*b;
}

int main(){
    int a=4,b=5;
    int soma = somar(a,b);
    assert(soma == 9);
}
```

- A área de verificação automática de software apresenta espaço para melhorias, sendo alvo de vários estudos (SV-COMP, 2019).
- Model Checking, uma abordagem automática para a verificação de software, pode sofrer com explosões de estados (GRUMBERG; VEITH, 2008).
- Técnicas para aprimorar a performance de Model Checkers incluem o uso de invariantes, porém a geração de invariantes é outro gargalo da área (GUPTA; RYBALCHENKO, 2009).

Como aprimorar a verificação de propriedades de segurança por model checkers, por meio da inferência de invariantes de programas em C com o auxílio de templates, de modo que as propriedades analisadas possam ser validadas?

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 no programa possam ser validadas ou refutadas com sucesso**.

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. Introdução
2. **Conceitos e Definições**
3. Trabalhos Correlatos
4. Método Proposto
5. Avaliação Experimental
6. Considerações Finais

- Objetivam evitar falhas no Software (COUSOT; COUSOT, 2010).
- Testes exploram algumas instâncias do programa, procurando *bugs*, porém sem garantir corretude.
- Verificação explora todas as possíveis instâncias do programa, buscando *garantir* que este siga alguma propriedade.

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

Figura: Função de soma em C

```
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: Casos de teste para a função acima (Testando instâncias específicas, como [A = 7, B = 8])

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

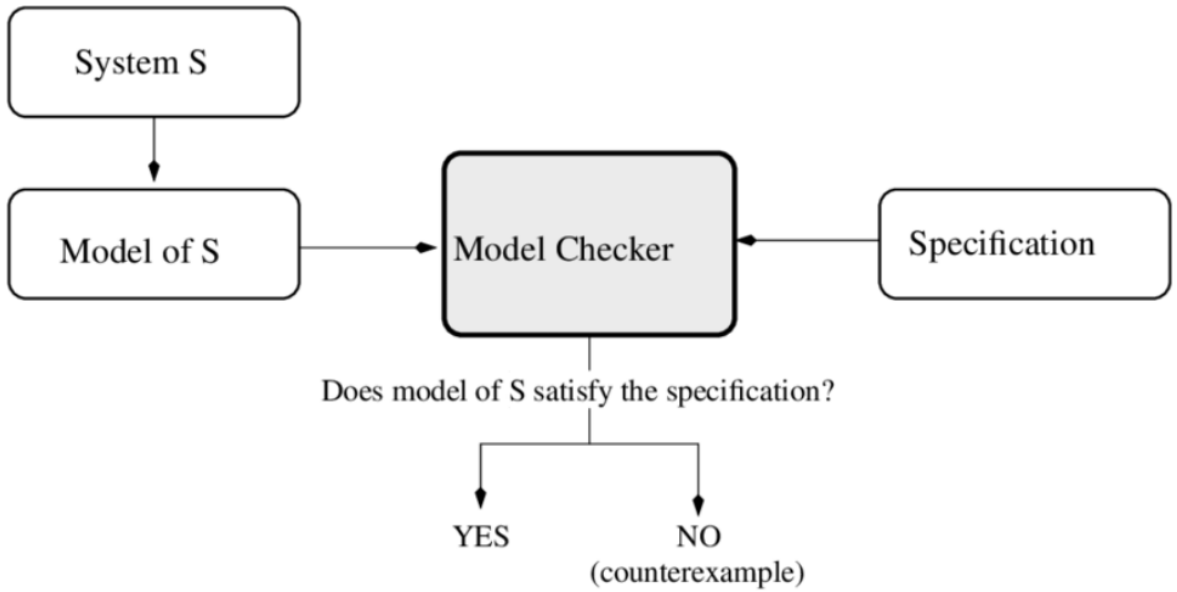
Figura: Função de soma em C

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

a e *b* são valores não-determinísticos, i.e. podem assumir qualquer valor

Figura: Verificação da função acima (Buscando garantir que o resultado retornado sempre será a soma de A com B)

- Técnica completamente automática para verificação de software (CLARKE; SCHLINGOFF, 2011).
- Descreve o programa como modelos, geralmente grafos de estados finitos, e realiza a verificação por busca exaustiva.
- Sujeita a explosão de estados (Quantias impraticáveis de estados para analisar).



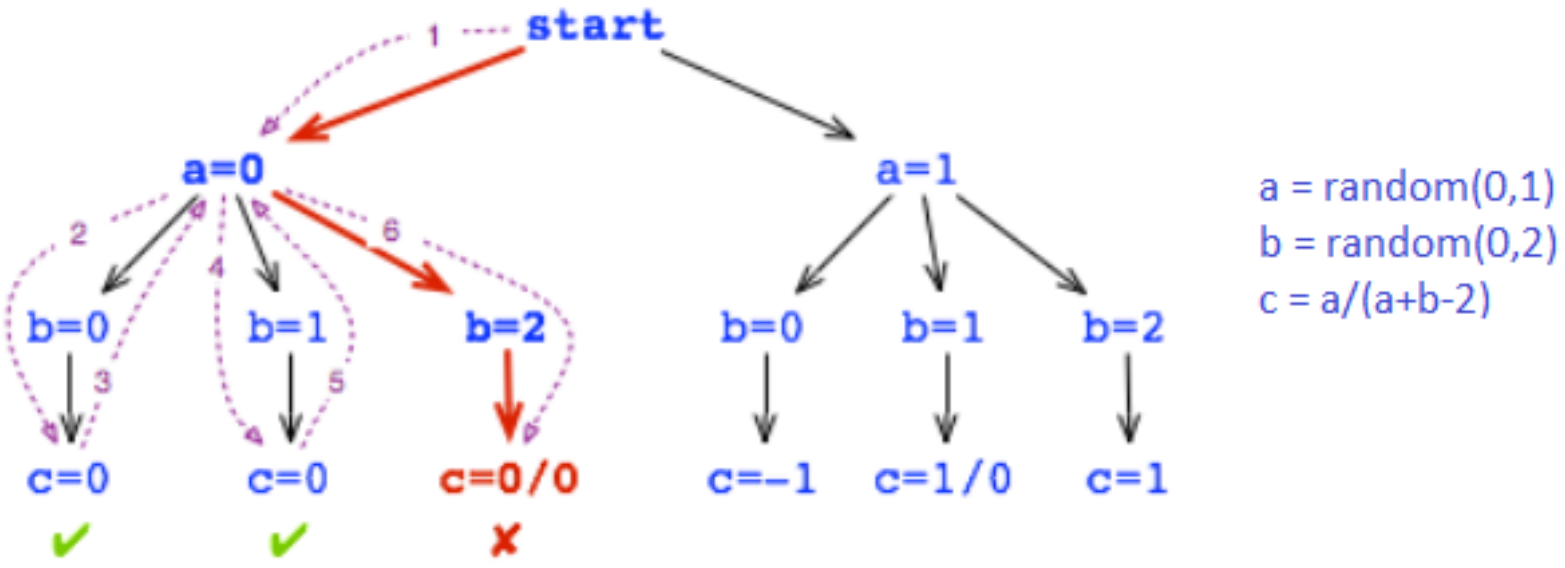


Figura: Procedimento de busca exaustiva de um Model Checker em um programa simples

- Fórmulas ou regras inferidas a partir do código-fonte de um programa (FOULADGAR et al., 2011).
- Se mantém imutáveis durante a execução, independentemente dos parâmetros que o programa venha a assumir.
- Através de seu uso, é possível reduzir o espaço de busca em Model Checking (ROCHA et al., 2015).

```
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); Invariante
8     }
9 }
```

- Método para auxiliar a geração de invariantes (SRIVASTAVA; GULWANI, 2009).
- Expressões incompletas com incógnitas que devem ser preenchidas pela ferramenta de geração de invariantes.
- Ditam a forma que a invariante gerada deve assumir.
- Geralmente providos pelo usuário, porém é possível gerá-los automaticamente (KONG et al., 2010).

Exemplo: $c_1v_1 + c_2v_2 + \ldots + c_nv_n \leq c$

- c, c_1, c_2, ... , c_n são as incógnitas a serem preenchidas.
- v_1, v_2, ... , v_n são variáveis do programa analisado.

- Analisa o comportamento de programas sem executá-los (MOLLER; SCHWARTZBACH, 2018).
- Estuda o código-fonte do programa.
- Informações sobre o programa analisado podem ser dadas na forma de invariantes.

- Analisa o comportamento de programas durante sua execução (ERNST et al., 2007).
- Executa várias instâncias, assume que comportamentos das instâncias observadas se repetem em instâncias não observadas.
- Informações sobre o programa analisado podem ser dadas na forma de invariantes.

- Solução automática de satisfabilidade de restrições.
- Satisfabilidade de restrições: "Dado um conjunto de variáveis e restrições sobre estas variáveis, encontrar valores para as variáveis que satisfaçam as restrições."
- Exemplo: $(a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee \neg a) \leftrightarrow \mathit{True}$
- Solução: $a = \mathit{false}$, $b = \mathit{true}$ e $c = \mathit{true}$

- Solucionadores de Restrições robustos.
- Operam com restrições sobre lógica booleana, aritmética, vetores, entre outros (MOURA; BJORNER, 2008).
- Amplamente utilizados pela literatura, exemplo: GUPTA e RYBALCHENKO, 2009.

1. Introdução
2. Conceitos e Definições
3. **Trabalhos Correlatos**
4. Método Proposto
5. Avaliação Experimental
6. Considerações Finais

- **Automatically Inferring Quantified Loop Invariants by Algorithmic Learning from Simple Templates**: Ferramenta para geração de invariantes por aprendizagem de máquina, modelo "estudante-professor" (KONG et al., 2010). Difere do método proposto por abranger uma gama superior de invariantes, porém não é automatizável.
- **Instantiation-Based Invariant Discovery**: Método para geração de invariantes por força bruta otimizada, a partir de templates binários. (KAHSAI et al., 2011). Limita-se a uma gama de invariantes inferior ao método proposto, e não é automatizável.

- **InvGen - An Efficient Invariant Generator**: Ferramenta para geração de invariantes guiada por *restrições* inferidas por análises estática e dinâmica. Utiliza templates não-quantificados (GUPTA; RYBALCHENKO, 2009). Base para a estrutura do método proposto por este trabalho.

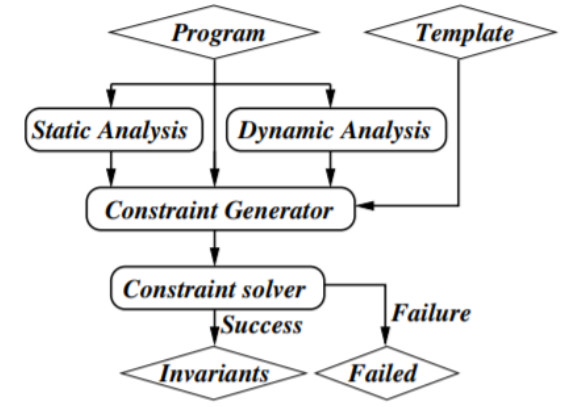


Figura: Estrutura da ferramenta InvGen.

- **Speeding Up the Constraint-Based Method in Difference Logic**: Duas abordagens para geração de invariantes no domínio da lógica diferencial. Modela o programa na forma de um autômato de fluxo de controle, CFA (CANDEAGO et al., 2016). A modelagem do programa como um CFA é aplicada no método proposto por este trabalho.

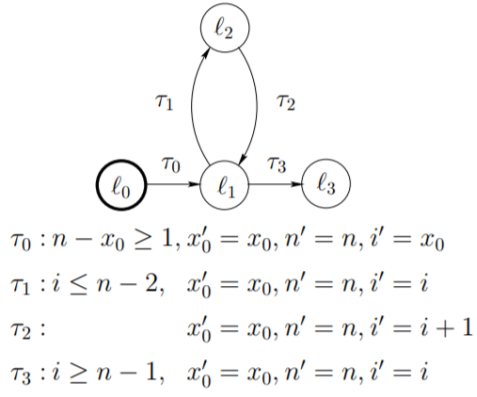


Figura: Autômato de fluxo de controle de controle do método de Candeago et al.

1. Introdução
2. Conceitos e Definições
3. Trabalhos Correlatos
4. **Método Proposto**
5. Avaliação Experimental
6. Considerações Finais

- Entrada: Um programa arbitrário em C.
- Saída: Programa da entrada, anotado com invariantes geradas pelo método.
- Objetivo: Ao ser anotado com as invariantes geradas pelo método, o programa torna-se mais fácil de ser verificado por um Model Checker, haja visto que o número de estados analisados é reduzido.

- Integra a geração de invariantes baseada em restrições ("Constraints"), proposta por GUPTA e RYBALCHENKO (2009), à modelagem de programas como autômatos de fluxo de controle, utilizada por CANDEAGO et al. (2016).
- Restrições são, em conceito, invariantes. No entanto, geralmente não são capazes de realizar grandes reduções no espaço de estados analisado por um Model Checker por si sós (GUPTA; RYBALCHENKO, 2009).

- No entanto, estas podem ser utilizadas em conjunto com técnicas mais poderosas (Exemplos: Refinamento de invariantes (GUPTA; RYBALCHENKO, 2009) e K-Indução (ROCHA et al., 2015)).
- Três abordagens diferentes são propostas para realizar a integração entre ambas as abordagens.
- Para manter uma nomenclatura padrão, as invariantes geradas por análise estática e dinâmica serão chamadas *restrições*, sendo o termo *invariante* reservado para as invariantes geradas pelo método proposto.

Fluxo de execução do método

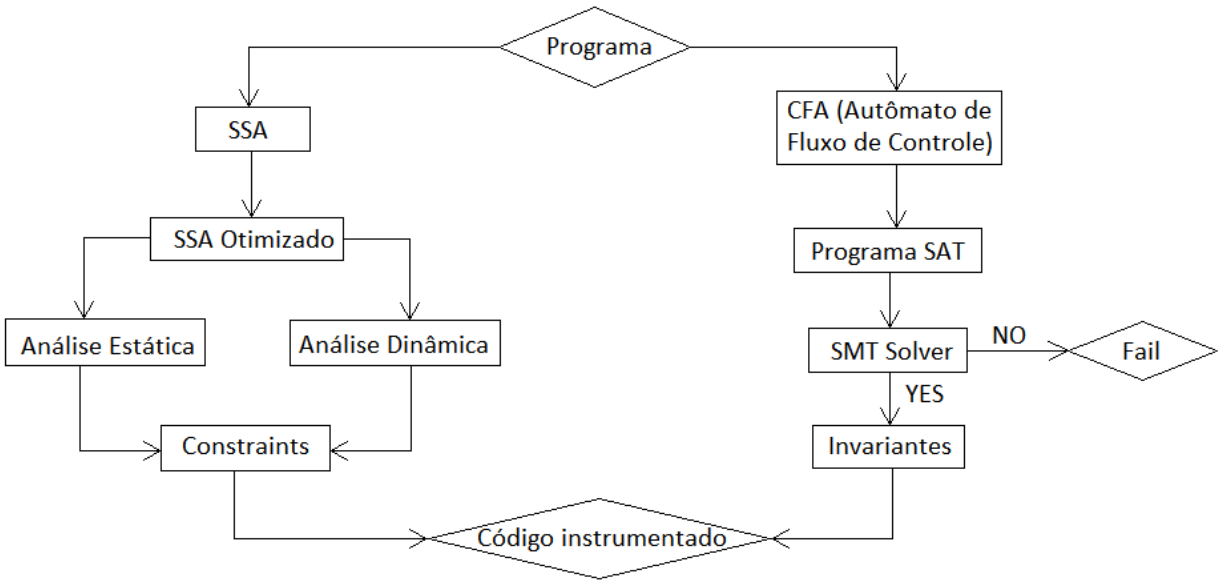


Figura: Primeira abordagem: Realiza a geração das invariantes e das restrições independentemente. Ambas são integradas à verificação.

Fluxo de execução do método

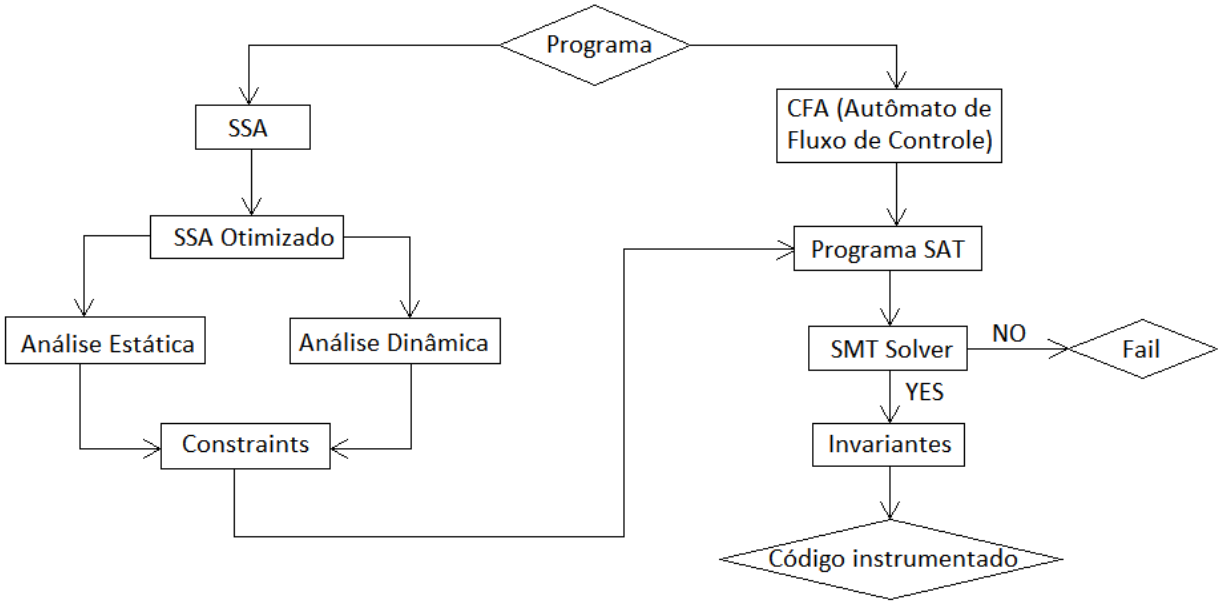


Figura: Segunda abordagem: Utiliza as restrições geradas para gerar invariantes mais precisas. Apenas as invariantes são integradas à verificação.

Fluxo de execução do método

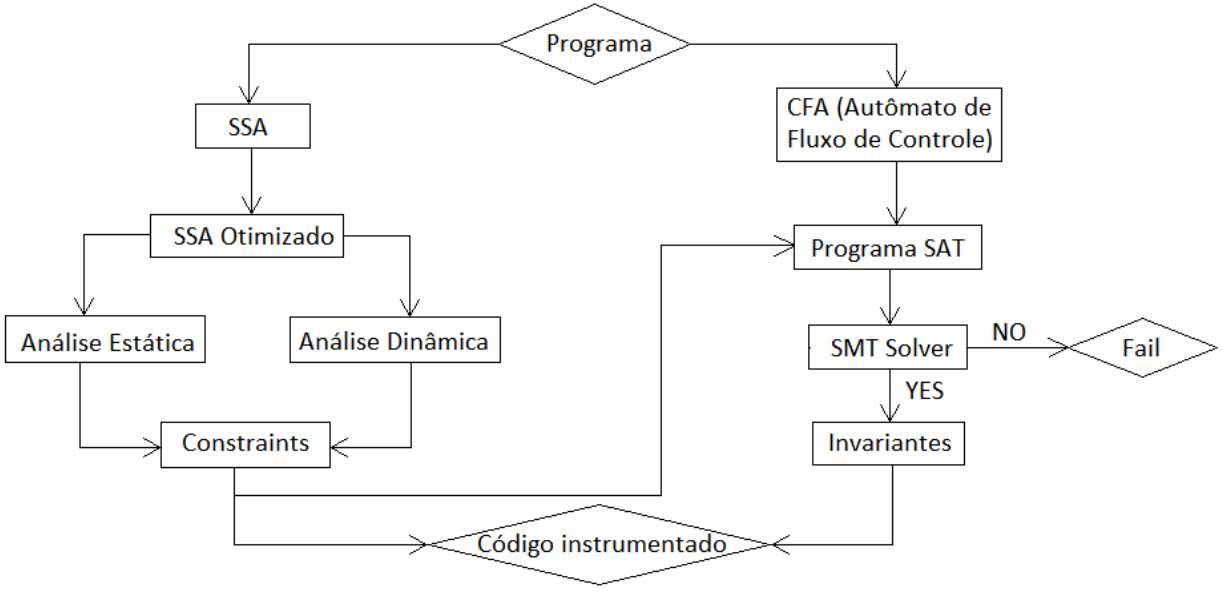


Figura: Terceira abordagem: Híbrida. Utiliza as restrições geradas para gerar invariantes mais precisas, como a segunda. No entanto, ambas, restrições e invariantes, são integradas ao código final, como na primeira abordagem.

Representação Intermediária

- C possui elementos semânticos e sintáticos complexos para ferramentas de análise.
- Necessária uma linguagem intermediária para a análise estática/dinâmica do código.
- Framework de compilação LLVM (https://llvm.org/)

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

Figura: Função de multiplicação em C

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

Figura: Equivalente em código intermediário LLVM-IR

- Ferramenta que realize Análise Estática sobre LLVM-IR.
- Crab-LLVM (https://github.com/seahorn/crab-llvm).

- Ferramenta que realize Análise Dinâmica sobre LLVM-IR.
- KLEE (https://klee.github.io/).

- Map2Check (https://github.com/hbgit/Map2Check/).
- Ferramenta de verificação, modificada para realizar a geração de restrições.
- Integra a conversão para LLVM-IR, e análise pelo Crab-LLVM e o KLEE.

- A ferramenta retorna um código em LLVM-IR anotado com as restrições.
- Assim, o código deve ser traduzido de volta para C, de forma a obter restrições sobre as variáveis originais.
- Atualmente, a tradução é efetuada manualmente, pelo próprio programador.

```
_.02:                                             ; preds = %_9, %_call2
  %.02 = phi i32 [ 0, %_call2 ], [ %spec.select, %_9 ]
  %.01 = phi i32 [ 0, %_call2 ], [ %_call4, %_9 ]
  %crab_8 = zext i1 %or.cond to i64, !dbg !12
  %crab_9 = add i64 0, %crab_8, !dbg !12
  %crab_10 = icmp eq i64 %crab_9, 1, !dbg !12
  call void @llvm.assume(i1 %crab_10), !dbg !12, !crabllvm !2
  %crab_11 = zext i32 %.02 to i64, !dbg !12
  %crab_12 = sub i64 0, %crab_11, !dbg !12
  %crab_13 = icmp sle i64 %crab_12, 0, !dbg !12
  call void @llvm.assume(i1 %crab_13), !dbg !12, !crabllvm !2
  %crab_14 = zext i32 %.01 to i64, !dbg !12
  %crab_15 = sub i64 0, %crab_14, !dbg !12
  %crab_16 = icmp sle i64 %crab_15, 0, !dbg !12
  call void @llvm.assume(i1 %crab_16), !dbg !12, !crabllvm !2
```

Exemplo de restrição gerada, em LLVM-IR

Figura: Trecho de código LLVM-IR anotado com restrições geradas pelo Map2Check

- A ferramenta retorna um código em LLVM-IR anotado com as restrições.
- Assim, o código deve ser traduzido de volta para C, de forma a obter restrições sobre as variáveis originais.
- Atualmente, a tradução é efetuada manualmente, pelo próprio programador.

Autômato de Fluxo de Controle

- Como as ferramentas de análise, SMT Solvers também precisam de uma linguagem intermediária.
- Modelo formal matemático (FTSRG, 2019).
- Autômatos de Fluxo de Controle (CFA).
- Apenas transições por predicados (Headers de loops, estruturas condicionais) são consideradas. Atribuições são consideradas transições vazias.

```
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: Um programa em C.

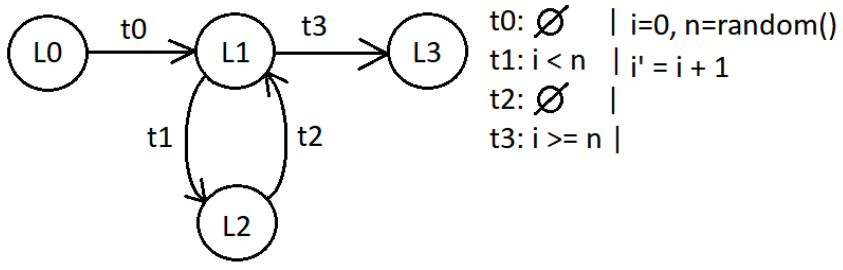


Figura: CFA associado ao programa descrito.

- Representação computacional do CFA. Deve conter as transições do CFA e, dependendo da abordagem, as restrições geradas.
- Cada transição é representada por uma fórmula de satisfabilidade e a ela é atribuído um *template* da forma $T_j = c_{j1}v_1 + c_{j2}v_2 + \ldots + c_{jn}v_n \leq d$.
  - $c_{j2}...c_{jn}$ são constantes de valor desconhecido no intervalo [-1,1]
  - v_1, $v_2...v_n$ corresponde ao conjunto de variáveis do programa
  - d é uma constante de restrição no conjunto Z^+

```
#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));
    //S0->S1

    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));
    //S1->S2

    assert(!(i-x0 >= n) && !(c_03*x0 + c_13*i + c_23*n <= d_3));
    //S1->S3
}
```

Declaração e inicialização das variáveis

Seção omitida do código onde são definidos os valores que cada constante pode assumir

Template Predicado

A expressão é anotada com o seu próprio *template* e também herda o template da expressão anterior

Cada expressão é escrita na forma de uma assertiva, de forma a exploitar o sistema de contra-exemplos do Model Checker

Figura: Estrutura básica de um programa SAT.

- Necessário para preencher as incógnitas do template.
- SMT Solver Z3 (https://github.com/Z3Prover/z3)
  - Ferramenta de escolha para diversos outros trabalhos (Exemplo: Srivastava et al., 2009).
  - Utilizado em projetos maiores (Exemplos: KLEE, Seahorn)

- Z3 possui uma linguagem própria, para a qual é difícil converter um código C.
- Utiliza-se o Model Checker ESBMC (esbmc.org) como front-end para esta conversão.

```
#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: Soluções obtidas pelo ESBMC para um programa SAT e invariantes correspondentes.

1. Introdução
2. Conceitos e Definições
3. Trabalhos Correlatos
4. Método Proposto
5. **Avaliação Experimental**
6. Considerações Finais

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?

- Vale lembrar que cada variante do método proposto gera como resultado final um programa anotado com invariantes.
- Assim, a forma mais direta de avaliar a performance do método proposto consiste em observar o impacto das invariantes geradas no processo de verificação de um Model Checker, conforme inicialmente proposto por este trabalho.

- Para auxiliar na avaliação do método proposto, é novamente empregado o Model Checker ESBMC (esbmc.org), porém dessa vez atuando diretamente como Model Checker.
- É importante notar que, embora o ESBMC possa ser utilizado com K-Indução, o escopo destes experimentos se limita a seu uso como pleno BMC (Força bruta) e a performance das invariantes neste cenário.
- Embora a proposta do trabalho seja um método completamente automatizável, no momento os experimentos foram executados manualmente.
- No entanto, a automatização do método proposto já está em progresso, com alguns módulos já desenvolvidos.

- 10 programas originados da competição internacional de verificação de Software, SV-COMP + 2 programas propostos pelo próprio autor.
- Possíveis vereditos:
  - True, se a propriedade a ser verificada for verdadeira.
  - False, se a propriedade a ser verificada for falsa.
  - Unknown, se a ferramenta se declarar incapaz de chegar a um resultado.
  - Timeout, se a ferramenta falhar em produzir um output dentro de um tempo de 30 segundos.

- Experimentos simulados em uma máquina virtual com as seguintes configurações:
  - Sistema Operacional Linux Mint
  - 1660 MB de Memória RAM
  - 126.50 GB de HD
  - Processador Intel Core I5
- Nenhuma aplicação externa foi executada durante os experimentos.
- Experimentos disponíveis em https://github.com/VictorDeluca/Inv_Generation_Tests/

Experimento 1

- Experimento: Verificação de programas sem invariantes ou restrições.
- Objetivo: Avaliação da própria ferramenta ESBMC, como base para os próximos experimentos.
- Resultados: 0/12 verificações corretas, dois Timeouts.

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

Figura: Experimento 1: Performance do ESBMC sem o auxílio de invariantes ou restrições

Experimento 2

- Experimento: Verificação de programas com restrições, mas sem invariantes.
- Objetivo: Avaliação da subrotina de geração de restrições, e sua performance individual.
- Resultados: 2/12 verificações corretas.

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

Figura: Experimento 2: Performance do ESBMC com o auxílio de restrições; Nota: Programas 3,4,5 não geraram restrições;

Experimento 3

- Experimento: Verificação de programas sem restrições, mas com invariantes.
- Objetivo: Avaliação da subrotina de geração de invariantes, e sua performance individual.
- Resultados: 10/12 verificações corretas.

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

Figura: Experimento 3: Performance do ESBMC com o auxílio de invariantes

Experimento 4

- Experimento: Análise da primeira abordagem do método proposto.
- Objetivo: Avaliação da primeira abordagem do método proposto
- Resultados: 10/12 verificações corretas, performance similar ao experimento 3 em tempos de execução.

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

Figura: Experimento 4: Avaliação da primeira abordagem (Programas 3,4,5 ignorados, pois não geram restrições);

Experimento 5

- Experimento: Análise da segunda abordagem do método proposto.
- Objetivo: Avaliação da segunda abordagem do método proposto
- Resultados: 5/12 verificações corretas, grande salto em tempo de execução (Mais rápida)

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

Figura: Experimento 5: Avaliação da segunda abordagem (Programas 3,4,5 ignorados, pois não geram restrições; Programas 2 e 9 não geram uma invariante pela terceira abordagem);

Experimento 6

- Experimento: Análise da terceira abordagem do método proposto.
- Objetivo: Avaliação da terceira abordagem do método proposto
- Resultados: 5/12 verificações corretas, performance similar ao experimento 5 em tempo de execução.

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

Figura: Experimento 6: Avaliação da terceira abordagem (Programas 3,4,5 ignorados, pois não geram restrições; Programas 2 e 9 ignorados, pois não geram uma invariante pela terceira abordagem);

Análise dos Experimentos

- O uso de restrições diretamente na verificação não possui grande impacto. Uma possível razão seria falha humana, uma vez que as restrições foram traduzidas manualmente.
- Porém, o mais provável é que estes resultados reflitam as pesquisas de GUPTA e RYBALCHENKO (2009), que afirmam que as restrições por si sós não possuem influência suficiente no Model Checker pleno.

Análise dos Experimentos

- Quanto às questões de pesquisa, pode-se observar que as três abordagens propostas apresentam resultados positivos, embora não superiores à geração de invariantes sem restrições.
- Sendo a primeira abordagem a mais eficaz, com uma acurácia de 83%.
- Enquanto a segunda e a terceira abordagens geram maiores ganhos em tempo de execução.

1. Introdução
2. Conceitos e Definições
3. Trabalhos Correlatos
4. Método Proposto
5. Avaliação Experimental
6. **Considerações Finais**

- Resultados preliminares validam a eficácia do método proposto, com uma acurária máxima de 83%.
- Trabalhos futuros:
  - Automatização do método proposto.
  - Análise do método com um benchmark mais amplo, possivelmente comparando a outras metodologias.
  - Estudo do uso de restrições para melhorias mais significativas na verificação.
  - Aplicação do método proposto a técnicas além do Model Checking (Exemplo: K-indução).

Agradeço a atenção. Dúvidas?