Verificação Baseada em Indução Matemática para Programas C/C++

Mikhail Y. R. Gadelha, Lucas C. Cordeiro, André L. D. Cavalcante, Vicente F. Lucena Jr.
Universidade Federal do Amazonas, Manaus, AM, Brasil
Departamento de Eletrônica e Computação
Email: {mikhailramalho,lucascordeiro,andrecavalcante,vicente}@ufam.edu.br

***Resumo*—O método *k-induction* tem sido aplicado com sucesso para verificar projetos de *hardware* (representados como máquinas de estado finito) usando um solucionador de Satisfatibilidade Booleana. No entanto, as primeiras tentativas para aplicar esta técnica a projetos de *software* são recentes. Neste artigo, será apresentado um novo algoritmo de prova por indução, que utiliza uma abordagem de aprofundamento iterativo e checa, para cada passo até um valor máximo, se uma dada propriedade de segurança faz parte do sistema. Para lidar com o aumento da complexidade do *software*, um solucionador das teorias do módulo da satisfatibilidade é usado para resolver as condições de verificação geradas pelo algoritmo *k-induction*. Os resultados experimentais mostram que o algoritmo proposto é capaz de verificar uma grande variedade de propriedades de segurança, extraídas de *benchmarks* padrões, que vão desde propriedades de alcançabilidade à restrições temporais.**

I. Introdução

Atualmente, as linguagens C/C++ são algumas das principais linguagens de programação para desenvolvimento de sistemas em geral e, em especial, sistemas embarcados. Sistemas embarcados consistem de um conjunto de componentes de *hardware/software*, que juntos implementam um conjunto específico de funcionalidades, enquanto satisfazem restrições físicas reais (por exemplo, de tempo, de dissipação de potência, ou referentes aos custos) [1]. Os sistemas embarcados são utilizados em um grande número de aplicações, de reatores nucleares e controles automotivos a sistemas de entretenimento, de tal forma que tais sistemas vêm se tornando indispensáveis para a vida do homem moderno.

Devido a importância dos sistemas embarcados e, consequentemente, da linguagem com que são desenvolvidos tais sistemas, é que se mostra evidente a importância de garantir a robustez na execução dos mesmos. Em especial, a verificação formal, uma técnica baseada em formalismos matemáticos para especificação e verificação de sistemas de *hardware* e *software*, surge para facilitar a busca por defeitos [2]. Em linhas gerais, a verificação formal consiste em provar e/ou falsificar a corretude de sistemas em relação a uma determinada especificação (ou propriedade), utilizando métodos formais.

A técnica de verificação de modelo limitada (*Bounded Model Checking*, BMC) baseadas em Satisfatibilidade Booleana (*Boolean Satisfiability*, SAT) ou Teorias do Módulo da Satisfatibilidade (*Satisfiability Modulo Theories*, SMT) foi aplicada com sucesso para verificação de programas sequenciais e paralelos e na descoberta de defeitos sutis [3]. A ideia básica da técnica BMC é checar a negação de uma propriedade em uma dada profundidade, ou seja, dado um sistema de transições M, uma propriedade ϕ, e um limite de iterações k, BMC desdobra o sistema k vezes e o transforma em uma condição de verificação (*Verification Condition*, VC) ψ de tal forma que ψ é verdadeiro se, e se somente se, ϕ possui um contraexemplo de profundidade igual ou menor do que k.

Note que, a técnica BMC consiste somente em falsificar propriedades até uma dada profundidade k, porém a mesma não é capaz de provar a corretude do sistema, a não ser que um limite superior de k seja conhecido (ou seja, um limite que desdobra suficientemente os laços e funções recursivas). Uma das técnicas empregadas para provar propriedades, para qualquer que seja a profundidade da análise, é a indução matemática. O algoritmo utilizado para a prova por indução do limite k, chamado de *k-induction*, foi aplicado com sucesso para garantir que programas não possuíssem defeitos de corridas de dados [4], [5], e que respeitassem restrições temporais descritas em fase de projeto do sistema [6].

O presente artigo apresenta um novo algoritmo para realizar prova por indução matemática de programas C/C++. A ideia principal do algoritmo consiste em usar uma abordagem de aprofundamento iterativo e checar, para cada passo k até um valor máximo, três diferentes casos chamados caso base, condição adiante e passo indutivo. Intuitivamente, no caso base, pretende-se encontrar um contraexemplo de ϕ com até k desdobramentos do laço. Na condição adiante, verifica-se a validade da propriedade ϕ em todos os estados alcançáveis dentro de k desdobramentos. E, no passo indutivo, verifica-se sempre que ϕ é válido para k desdobramentos, ϕ também será válido para o próximo desdobramento do sistema.

Tal algoritmo foi implementado na ferramenta *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC), que utiliza técnicas de BMC e as teorias de SMT para verificar sistemas embarcados escritos em C/C++ [7], [8]. No entanto, em vez de falsificar uma determinada propriedade em uma dada profundidade, o presente algoritmo *k-induction* estende o verificador ESBMC tentando provar a validade da propriedade para qualquer profundidade. Para validar a implementação do algoritmo, foram utilizados os *benchmarks* da competição internacional em verificação de *software* (*Competition on Software Verification*, SV-COMP) [9], onde o algoritmo *k-induction* conquistou o terceiro lugar no *ranking* geral, além de uma aplicação real do computador de bordo de uma bicicleta [10]. Os resultados experimentais mostram que a ferramenta é capaz de verificar se as propriedades do usuário (por exemplo, restrições temporais) estão sendo respeitadas, além de indicar se o programa possui defeitos relacionados a própria linguagem de programação (por exemplo, estouro de *buffer* e aritmético, divisão por zero e segurança de ponteiros).

II. PRELIMINARES

Esta seção fornece conceitos básicos para o entendimento da técnica BMC e da Lógica Hoare, que será utilizada para formalização do algoritmo *k-induction*.

A. A Técnica BMC

No ESBMC, o programa que está sendo analisado é modelado por um sistema de transição de estados, que é gerado a partir de um grafo de fluxo de controle do programa (*Control-Flow Graph*, CFG) [11]. O grafo de fluxo de controle do programa é gerado automaticamente, durante o processo de verificação. Um nó no CFG representa uma atribuição (determinística ou não-determinística) ou uma expressão condicional, enquanto que uma aresta representa uma mudança no fluxo do programa.

Um sistema de transição de estados $M = (S, T, S_0)$ é uma máquina abstrata, que consiste em um conjunto de estados S, onde $S_0 \subseteq S$ representa um conjunto de estados iniciais e $T \subseteq S \times S$ é a relação de transição. Um estado $s \in S$ consiste no valor do contador de programa *pc* e também nos valores de todas as variáveis do *software*. Um estado inicial s_0 atribui a localização inicial do programa no CFG. As transições são identificadas como $\gamma = (s_i, s_{i+1}) \in T$ entre dois estados s_i e s_{i+1}, com uma fórmula lógica $\gamma(s_i, s_{i+1})$ que contém as restrições dos valores das variáveis do *software* e do contador de programa.

Dado um sistema de transição M, uma propriedade ϕ e um limite k, o ESBMC desdobra o sistema k vezes e transforma o resultado em uma VC ψ, de tal forma que ψ é satisfeita se, e se somente se, ϕ possuir um contraexemplo de comprimento menor ou igual a k [3]. O problema da técnica BMC é então formulado como:

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

onde ϕ é uma propriedade, I é o conjunto de estados iniciais de M e $\gamma(s_j, s_{j+1})$ é a relação de transição de M entre os passos j e $j+1$. Logo, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ representa a execução de M, i vezes, e a Fórmula (1) só poderá ser satisfeita se, e somente se, para um $i \leq k$, existir um estado alcançável no passo em que ϕ é violada. Se a Fórmula (1) é satisfeita, então o ESBMC mostra um contraexemplo, definindo quais os valores das variáveis necessárias para reproduzir o erro. O contraexemplo para uma propriedade ϕ é uma sequência de estados $s_0, s_1, \cdots, s_k$ com $s_0 \in S_0$, e $\gamma(s_i, s_{i+1})$ com $0 \leq i < k$. Se a Fórmula (1) não for satisfeita, pode-se concluir que nenhum estado com erro é alcançável em k ou menos passos.

B. Lógica Hoare

A ideia básica por trás da Lógica Hoare é que um programa de computador é uma ciência exata e todas as consequências de execução em qualquer ambiente podem, em princípio, ser encontrados a partir do próprio código desse programa, por meio de pensamento dedutivo [12].

Em diversos casos, a validade do resultado de um programa dependerá dos valores assumidos pelas variáveis antes do código executar. A notação proposta por Hoare cria uma relação entre as pré-condições (*P*), um determinado código, composto por uma sequência de comandos (*Q*), e o resultado da execução (*R*):

$$\{\ P\ \}\ Q\ \{\ R\ \} \tag{2}$$

A sentença pode ser interpretada como "se a assertiva *P* é verdadeira antes de *Q*, então *R* será verdadeira ao término de *Q*". Em caso particular, onde não existem pré-condições, a sentença se torna:

$$\{\ \textbf{verdadeiro}\ \}\ Q\ \{\ R\ \} \tag{3}$$

A atribuição é a funcionalidade mais comum em programas. O axioma da atribuição define que, o valor de uma variável *x*, depois de executar um comando de atribuição *x:=t*, se torna o valor da expressão *t* no estado antes da atribuição. O axioma de atribuição pode ser definido como:

$$\{\ p\ [t/x]\ \}\ x\ :=\ t\ \{\ p\ \} \tag{4}$$

onde *x* é o identificador de uma variável, *p* é uma assertiva e a notação *p*[*t*/*x*] denota o resultado de substituir o termo t por todas as ocorrências livres de *x* em *p*.

A regra da composição define uma sequência de operações, executadas uma após a outra. As expressões devem ser separadas por um símbolo ou equivalente de forma a denotar uma composição procedural: $(Q_1; Q_2; ...; Q_n)$. A regra da composição é definida como:

$$\frac{\{\ p\ \}\ S_1\ \{\ r\ \},\ \{\ r\ \}\ S_2\ \{\ q\ \}}{\{\ p\ \}\ S_1,\ S_2\ \{\ q\ \}} \tag{5}$$

onde *p, q, r* são assertivas (na primeira expressão, *p* é a pré-condição e *r* é a pós-condição, enquanto que na segunda expressão, *r* é a pré-condição e *q* é a pós-condição) e S_1 e S_2 são códigos ou sequências de comandos.

A regra do se-então-senão define a operação lógica de decisão binária, em que uma condição define qual trecho de código deve ser executado. Esta regra é definida como:

$$\frac{\{\ p\ \wedge\ e\ \}\ S_1\ \{\ q\ \}, \{\ p\ \wedge\ \neg\, e\ \}\ S_2\ \{\ q\ \}}{\{\ p\ \}\ \textbf{se}\ e\ \textbf{então}\ S_1\ \textbf{senão}\ S_2\ \textbf{fim-se}\ \{\ q\ \}} \tag{6}$$

onde *p, q* são assertivas, S_1 e S_2 são códigos ou sequências de comandos e *e* é um condicionante, de forma que assegura a execução de *q* após S_1, caso a fórmula $(p \wedge e)$ seja satisfeita, e assegura a execução de *q* após S_2, caso a fórmula $\{p \wedge \neg e\}$ seja satisfeita.

A regra do enquanto define as operações iterativas, compostas por um valor inicial, uma condição de parada e um incremento sobre a condição inicial. Esta regra é definida como:

$$\frac{\{\ p\ \wedge\ e\ \}\ S\ \{\ p\ \}}{\{\ p\ \}\ \textbf{enquanto}\ e\ \textbf{faça}\ S\ \textbf{fim-faça}\ \{\ p\ \wedge\ \neg\, e\ \}} \tag{7}$$

onde p é uma assertiva, S é um código ou sequência de comandos e e é um condicionante, de forma que, caso S mantenha a expressão $\{p \wedge e\}$ satisfeita, então a execução de S manterá a expressão satisfeita em qualquer número de iterações. A regra do enquanto também expressa o fato de que, após o comando enquanto terminar, a expressão $\{\ p \wedge \neg e\ \}$ deverá ser satisfeita, ou seja, o condicionante e passa a ser falso.

III. Verificação Baseada em Indução para Programas C/C++

O algoritmo descrito nesta seção é chamado *k-induction* e consiste de três passos: caso base, condição adiante e passo indutivo. As transformações em cada passo do algoritmo são descritas usando a notação Hoare (cf. Seção II-B).

A. O Algoritmo k-induction

A Figura 1 mostra o algoritmo *k-induction*. No caso base, o algoritmo tenta encontrar um contraexemplo de até um dado número máximo de iterações *k*. Na condição adiante, o algoritmo checa se todos os estados foram alcançados dentro das *k* iterações e, no passo indutivo, o algoritmo checa, se a propriedade é verdadeira em *k* iterações, então também deve ser verdadeira para as próximas iterações. Caso os três passos falhem para um valor de k, então o valor de k é incrementado e os três passos são executados novamente.

```
1  k = numero de iteracoes inicial
2  enquanto true faca
3    se caso_base(k) entao
4      retorne contraexemplo s[0..k]
5    senao
6      se condicao_adiante(k) entao
7        retorne true
8      senao
9        se passo_indutivo(k) entao
10         retorne true
11       fim-se
12     fim-se
13   fim-se
14   k=k+1
15 fim-enquanto
```

Fig. 1. O algoritmo *k-induction*.

Para os três passos do algoritmo, uma série de transformações são realizadas. Em especial, os laços *for* e *do while* são transformados em laços *while*, e são convertidos utilizando a regra do enquanto da seguinte forma:

$$for(A; c; B) \wedge D \wedge E \iff A \wedge enquanto(c) \wedge D \wedge B \wedge E$$

onde A é a condição inicial do laço, c é a condição de parada do laço, B é o incremento de cada iteração sobre A, D é o código dentro do laço *for* e E é um código após o laço *for*. O laço *do while* é transformado da seguinte forma:

$$do\ A\ while(c) \wedge B \iff A \wedge enquanto(c) \wedge B$$

onde A é o código dentro do laço *do while*, c é a condição de parada do laço e B é um código após o laço *do while*. Os laços *do while* são convertidos em laços *while*, com uma única diferença, o código dentro do laço deve executar pelo menos uma vez antes da condição de parada ser checada.

Além dessas transformações, os três passos do algoritmo *k-induction* inserem suposições e assertivas com o intuito de provar a validade das propriedades. Como exemplo de aplicação do algoritmo *k-induction*, será utilizado um programa extraído dos *benchmarks* do SV-COMP [9], conforme mostrado na Figura 2.

```
1 int main() {
2   unsigned int i, n, sn=0;
3   assume (n>=1);
4   for(i=1; i<=n; i++)
5     sn = sn + a;
6   assert(sn==n*a);
7 }
```

Fig. 2. Código exemplo para prova por indução matemática.

Matematicamente, o código acima representa simplesmente a implementação do somatório dado pela seguinte fórmula:

$$S_n = \sum_{i=1}^{n} a = na, n \geq 1 \tag{8}$$

Note que no código da Figura 2, a propriedade (representada pela assertiva da linha 6) deve ser verdadeira para qualquer valor de *n*. Além das transformações (dos laços) descritas acima, o caso base inicializa os limites do condicionante do laço com valores não-determinísticos com o intuito de explorar todos os possíveis estados implicitamente. Sendo assim, as pré- e pós-condições do código presente na Figura 2 tornam-se:

$$P \iff n_1 = nondet\_uint \wedge n_1 \geq 1 \wedge sn_1 = 0 \wedge i_1 = 1$$
$$R \iff i_k > n_k \Rightarrow sn_k = n_1 \times a$$

onde P e R são as pré- e pós-condições respectivamente e $nondet\_uint$ é uma função não-determinística, que pode retornar qualquer valor do tipo *unsigned int*. Nas pré-condições, n_1 representa a primeira atribuição para a variável n, de um valor não-determinístico maior do que ou igual a um. Note que, esta atribuição de valores não-determinísticos garante que o verificador explore todos os possíveis estados implicitamente. Além disso, sn_1 representa a primeira atribuição para a variável sn com valor zero e i_1 representa a condição inicial do laço. Nas pós-condições, sn_k representa a atribuição $n + 1$ para a variável sn da Figura 2, que é verdadeira se $i_k > n_1$. O trecho de código, que não é pré- ou pós-condição, é representado pela variável Q e este não sofre nenhuma alteração durante as transformações do caso base.

O código resultante das transformações do caso base pode ser visto na Figura 3. A instrução *assume* (linha 9), contendo o inverso da condição de parada, elimina todos os caminhos de execução que não satisfazem a restrição $i > n$. Isso assegura que o caso base encontre um contraexemplo de profundidade k sem reportar nenhum falso negativo.

Na condição adiante, o algoritmo *k-induction* tenta provar se o laço foi suficientemente desenrolado e se a propriedade é válida em todos os estados alcançáveis até a profundidade *k*. As pré- e pós-condições do código presente na Figura 2, durante a condição adiante, são definidas como:

$$P \iff n_1 = nondet\_uint \wedge n_1 \geq 1 \wedge sn_1 = 0 \wedge i_1 = 1$$
$$R \iff i_k > n_k \wedge sn_k = n_1 \times a$$

As pré-condições do caso adiante são idênticas à do caso base. Porém, nas pós-condições R, existe a verificação se o laço foi suficientemente expandido, representado pela comparação $i_k > n_k$, onde i_k representa o valor da variável i na iteração $n + 1$ e n_k representa o valor da variável n na

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

Fig. 3. Código exemplo para prova por indução, durante o caso base.

iteração $n+1$ (devido às atribuições que ocorrem no laço *for* da Figura 2 que possui n iterações). O código resultante das transformações do caso adiante pode ser visto na Figura 4. O caso adiante tenta provar se o laço foi suficientemente desenrolado (verificando o laço invariante na linha 9) e se a propriedade é válida em todos os estados alcançáveis dentro de *k* desdobramentos (através da assertiva na linha 10).

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

Fig. 4. Código exemplo para prova por indução, durante o *caso adiante*.

No passo indutivo, o algoritmo *k-induction* tenta provar que, se a propriedade é válida até a profundidade k, a mesma deve ser válida para o próximo valor de k. Diversas transformações são feitas no código original. Primeiramente, é definido um tipo de estrutura chamada *statet*, contendo todas as variáveis presentes dentro do laço e na condição de parada desse laço. Em seguida, é declarado uma variável do tipo *statet*, chamada cs (*current state*), que é responsável por armazenar os valores das variável presentes no laço em uma determinada iteração. É declarado também um vetor de estados de tamanho igual ao número de iterações do laço, chamado sv (*state vector*), que irá armazenar os valores de todos as variáveis do laço em cada iteração.

Antes do início do laço, todas as variáveis são inicializadas com valores não-determinísticos e armazenadas no vetor de estados, durante a primeira iteração do laço. Dentro do laço, após o armazenamento do estado atual e o código do laço, todas as variáveis do estado atual são atualizadas com os valores da iteração atual. Uma instrução *assume* é inserida com a condição de que o estado atual é diferente do estado anterior, para evitar que estados redundantes sejam inseridos no vetor de estados (neste caso a comparação entre todos os estados é feita de forma incremental). Por fim, após o laço, tem-se uma instrução *assume* igual à instrução inserida no caso base. As pré- e pós-condições do código presente na Figura 2, durante o passo indutivo, são definidas como:

$$\begin{aligned} P \iff & n_1 = nondet\_uint \wedge n_1 \geq 1 \wedge sn_1 = 0 \wedge i_0 = 1 \\ & \wedge cs_1.v_0 = nondet\_uint \\ & \ldots \\ & \wedge cs_1.v_m = nondet\_uint \\ R \iff & i_k > n \Rightarrow sn_k = n_1 \times a \end{aligned}$$

Nas pré-condições P, além da inicialização dos valores das variáveis, é necessário iniciar o valor de todas as variáveis contidas no estado atual cs, com valores não-determinísticos, onde m é o número de variáveis (automáticas e estáticas) que são utilizadas no programa. As pós-condições não mudam em relação ao caso base, e contém somente a propriedade que se deseja provar. No conjunto de instruções Q, são feitas alterações no código para salvar o valor das variáveis na iteração i atual, da seguinte maneira:

$$\begin{aligned} Q \iff & sv[i-1] = cs_i \wedge S \\ & \wedge cs_i.v_0 = v_{0i} \wedge \ldots \wedge cs_i.v_m = v_{mi} \end{aligned}$$

No conjunto de instruções Q, $sv[i-1]$ representa a posição no vetor para salvar o estado atual cs_i, S representa o código dentro do laço e a série de atribuições, semelhantes às da pré-condição, que representam o estado atual cs_i salvando o valor das variáveis na iteração i. O código modificado pelo passo indutivo pode ser visto na Figura 5. Assim como no caso base, o passo indutivo também inclui uma instrução *assume*, contendo o inverso da condição de parada. Diferentemente do caso base, o passo indutivo tenta provar que a propriedade contida no *assert* é verdadeira para qualquer valor de n.

```
1  //variaveis presentes no loop
2  typedef struct state {
3  unsigned int i, n, sn;
4  } statet;
5  int main() {
6    unsigned int i, n=nondet_uint(), sn=0;
7    assume (n>=1);
8    i=1;
9    //declara estado atual e vetor de estados
10   statet cs, sv[n];
11   //atribui valores nao-deterministicos
12   cs.i=nondet_uint(); cs.sn=nondet_uint();
13   cs.n=n;
14   while(i<=n) {
15     sv[i-1]=cs;    //armazena estado atual
16     sn = sn + a;   //codigo dentro do loop
17     //atualiza as variaveis do estado atual
18     cs.i=i; cs.sn=sn; cs.n=n;
19     //remove estados redundantes
20     assume(sv[i-1]!=cs);
21     i++;
22   }
23   assume(i>n); //suposicao de desdobramento
24   assert(sn==n*a);
25 }
```

Fig. 5. Código exemplo para prova por indução, durante o passo indutivo.

IV. Resultados Experimentais

Para avaliar o algoritmo *k-induction*, foram utilizados 79 casos de teste da categoria *loops*[1] do SV-COMP [9] e um

[1]https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp13

software de monitoramento de uma bicicleta [10]. A categoria *loops* foi escolhida pelo fato de possuir diversos programas que requerem análise da condição de parada dos laços. A comparação dos resultados do ESBMC (usando a implementação do algoritmo *k-induction*) com outras ferramentas, não será apresentada neste artigo, uma vez que a edição do SV-COMP 2013 já compara os resultados da categoria *loops* usando diferentes ferramentas de verificação (conforme apresentado por Dirk Beyer [9]).

Todos os experimentos foram conduzidos em um computador com processador Intel Core i7-2600K, 3.40Ghz com 16GB de memória RAM com Ubuntu 12.04 64-bits. Para cada caso de teste, foram ajustados um limite de tempo de verificação e um limite de memória, de 900 segundos (15 minutos) e 15GB de memória, respectivamente. A ferramenta utilizada, com a implementação do *k-induction*, é o ESBMC v1.20 [13].

A. Resultados da Categoria Loops

O conjunto de programas da categoria *loops* está dividido da seguinte forma: 43 casos de testes contém propriedades válidas, ou seja, o ESBMC deve conseguir provar a propriedade; 36 casos de testes contém propriedades inválidas, ou seja, o ESBMC deve ser capaz de falsificar a propriedade. Dos 79 casos de teste verificados, somente 3 falharam com resultados incorretos: 1 caso de teste (*count_up_down_safe.i*) com propriedade inválida (falso negativo) e 2 casos de teste (*linear_search_unsafe.i* e *s3_unsafe.i*) com propriedades válidas (falso positivo). O ESBMC não foi capaz de verificar corretamente esses casos de teste, pois os mesmos utilizam alocação dinâmica de memória, funcionalidade ainda não suportada pelo algoritmo *k-induction*.

Todos os casos de teste foram verificados em 6800 segundos, sendo verificado corretamente 74 de 79 (93%).[2] Note que o ESBMC não abortou devido ao esgotamento de memória, porém falhou devido a um erro interno (durante a construção de expressões passadas ao solucionador) no caso de teste *n.c24_safe.i*, e extrapolou o limite de tempo para outro caso de teste (*toy_safe.i*), pois o mesmo possui diversos laços aninhados, o que aumenta a complexidade das VCs geradas e, consequentemente, o tempo necessário para o solucionador checar a satisfatibilidade das VCs. Dentre seis ferramentas participantes, o algoritmo *k-induction* obteve a segunda posição no *ranking* da categoria *loops*[3] [9].

B. Verificação do Código da Bicicleta

Uma prova de conceito foi realizada utilizando o ESBMC com o algoritmo *k-induction*, em um sistema de computador de bordo de uma bicicleta, que utiliza a plataforma embarcada *Raspberry Pi* [14] com o processador ARM1176JZF-S [15]. O sistema consiste basicamente de uma tela e dois botões, um para circular nos modos de funcionamento do computador e outro botão para reiniciar o modo atual do computador. Os modos de funcionamento do computador incluem: viagem (mostra na tela a quilometragem atual da viagem), velocidade (mostra a velocidade atual), total (mostra a quilometragem total percorrida) e tempo (mostra o tempo total da viagem).

Para cada modo de funcionamento desse sistema, existem restrições temporais entre o tempo de processamento das informações e a exibição na tela. No modo viagem, a quilometragem deve ser mostrada a cada 200ms, com uma tolerância de 100ms. No modo velocidade, a velocidade atual deve ser mostrada a cada 100ms. No modo total, a quilometragem total deve ser mostrada a cada 500ms e no modo tempo, o tempo de viagem deve ser mostrado a cada segundo.

Para realizar a verificação das restrições temporais do programa, foi necessário inserir assertivas no programa sobre o tempo de processamento de cada modo de viagem. Por exemplo, caso o computador esteja no modo velocidade, o tempo de leitura do sensor e o cálculo da velocidade deve ser menor do que 100ms, que é o tempo de atualização da tela nesse modo. Para identificar o tempo de processamento das informações de cada modo, o código foi transformado em *assembly* e, a partir do número de instruções, foi calculado o tempo de CPU. O tempo de CPU foi calculado da seguinte forma [16]:

$$T.\,de\,CPU = N_{ins} \times CPI \times T_c \tag{9}$$

onde N_{ins} é o número de instruções gerado, CPI é o número médio de ciclos do processador por número de instruções e o T_c é o tempo de ciclo do processador (o inverso do relógio do processador). Nos experimentos, os valores de CPI e T_c foram obtidos a partir do manual técnico do processador ARM1176JZF-S, presente na plataforma *Raspberry Pi*, utilizada pelo sistema. Os valores são os seguintes: $CPI = 1,1$ e $T_c = 1,42857 \times 10^{-9}$ segundos (i.e., inverso de 700MHz).

O programa modificado com as assertivas sobre as propriedades temporais é mostrado na Figura 6, onde Tp é o tempo de processamento, Tb é o tempo que o botão foi pressionado (mudando o modo do sistema) e Rt representa a restrição temporal do modo atual. A função *current_time()* retorna o tempo atual do sistema. A assertiva checa se o tempo de processamento é menor do que a restrição do modo atual.

```
1 ...
2 Tb=current_time();
3 /* Processing and display informations */
4 Tp=current_time();
5 assert(Tp - Tb < Rt);
6 ...
```

Fig. 6. Programa alterado com as assertivas.

O ESBMC foi capaz de verificar corretamente o código em menos de um segundo e nenhum defeito foi encontrado[4]. Notou-se também que o tempo de processamento e de impressão na tela são extremamente pequenos em relação às restrições temporais. Por exemplo, no modo viagem são necessárias 55 instruções para calcular a velocidade e imprimir na tela. O tempo de CPU calculado para esse caso é:

$$\begin{aligned} T.\,de\,CPU &= 55 \times 1,1 \times 1,42857 \times 10^{-9} \\ &= 86ns \end{aligned} \tag{10}$$

ou seja, o processador utilizado para resolver o problema é extremamente superior ao necessário para o sistema.

[2]Os casos de teste foram verificados usando o comando: esbmc file.c −−k-induction −−k-step 100 −−memlimit 15g −−timeout 900s

[3]http://sv-comp.sosy-lab.org/2013/results/index.php

[4]Os casos de teste foram verificados usando o comando: esbmc file.c −−k-induction −−k-step 10

V. Trabalhos Relacionados

A aplicação do método *k-induction* vem ganhando popularidade na comunidade de verificação de *software*, principalmente devido ao advento de sofisticados solucionadores SMT construídos sobre eficientes solucionadores SAT. Trabalhos anteriores já exploraram a técnica de prova por indução de sistemas de *hardware* e *software* com algumas limitações, por exemplo, exigência de anotações no código (dificultando a automação do processo de verificação) e utilização de solucionadores SAT em vez de SMT (dificultando a escalabilidade do algoritmo) [4], [5], [17].

Daniel et al. descrevem uma forma de provar propriedades de projetos TLM (*Transaction Level Modeling*) na linguagem SystemC [17]. A abordagem consiste de três passos; transformação do código em SystemC para código C, a geração e adição de lógicas para monitorar propriedades TLM, e a verificação do código C utilizando indução matemática. Um programa produtor-consumidor é testado, porém o aumento do número de produtores e consumidores, faz com que o algoritmo não finalize no tempo determinado, devido à limitação no número de iterações dos laços.

Sheeran et al. descrevem uma ferramenta chamada Lucifer [18] para verificação utilizando indução matemática [19]. A ferramenta fornece uma forma rápida e eficaz de provar corretude de projetos de *hardware*, incluindo *field-programmable gate array* (FPGA), durante a fase de projeto. No entanto, os autores não tratam de projetos de software escritos em C/C++ (o sistema deve ser convertido para a linguagem Lustre). Além disso, os autores não exploram o uso de solucionadores SMT.

Alastair et al. descrevem uma ferramenta de verificação chamada Scratch [4], para detectar corrida de dados durante acesso direto de memória (*Direct Memory Access*, DMA) em processadores CELL BE da IBM [4]. A abordagem utilizada para verificar os programas, utiliza-se da técnica *k-induction*. A ferramenta é capaz de provar a ausência de corrida de dados, porém está restrita a verificar essa classe específica de defeitos para um tipo específico de *hardware*.

Em outro trabalho relacionado, Alastair et al. descrevem duas ferramentas para provar corretude de programas: K-Boogie e *K-Inductor* [5]. O K-Boogie é uma extensão para o verificador da linguagem Boogie e permite a prova de corretude utilizando o método *k-induction* de diversas linguagens de programação, incluindo Boogie, Spec, Dafny, Chalice, VCC e Havoc. K-Inductor é um verificador de programas C, construído sobre a ferramenta de verificação CBMC [20]. As duas ferramentas apresentam bons resultados, porém para ambas as ferramentas, existe a necessidade de alteração no código por parte do desenvolvedor, ou seja, a verificação não é um processo totalmente automatizado.

O algoritmo que mais se aproxima da abordagem proposta é aquele apresentado por Alastair et al. [5]. No entanto, as técnicas utilizadas diferem tanto no algoritmo de indução quanto no processo de verificação. O algoritmo de indução utilizado no *K-Inductor* não possui o passo da condição adiante, o que impossibilita a verificação se todos os estados foram alcançados em k desdobramento. Além disso, no *K-Inductor* o desenvolvedor precisa alterar o código original com o intuito de inserir suposições adicionais sobre os laços, enquanto que o processo de verificação do ESBMC é completamente automático.

VI. Conclusões

O presente trabalho apresentou um novo algoritmo para realizar prova por indução matemática de programas C/C++. O algoritmo *k-induction* foi implementado na ferramenta de verificação de modelos ESBMC. A principal contribuição do trabalho consiste na concepção e implementação do algoritmo *k-induction* em uma ferramenta de verificação de modelos, assim como a utilização da técnica para verificação de restrições temporais em programas. Para validação do algoritmo, foram feitos experimentos envolvendo vários casos de teste, dos quais o algoritmo foi capaz de verificar corretamente 93%. Além disso, o algoritmo foi capaz de provar e falsificar propriedades de alcançabilidade e de restrições temporais em um sistema de computador de bordo de bicicleta. Para trabalhos futuros, a criação de um modelo de memória para suportar a alocação dinâmica de memória deve ser desenvolvido. Além disso, o algoritmo pode ser paralelizado, de tal forma que cada passo do processo seja executado de forma concorrente.

Bibliografia

[1] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 2011.

[2] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[3] L. Cordeiro, *SMT-Based Bounded Model Checking of Multi-threaded Software in Embedded Systems*. PhD thesis, University of Southampton, UK, 2011.

[4] A. Donaldson, D. Kroening, and P. Rummer, "Automatic Analysis of Scratch-pad Memory Code for Heterogeneous Multicore Processors," in *TACAS, LNCS 6015*, pp. 280–295, 2010.

[5] A. Donaldson, L. Haller, D. Kroening, and P. Rummer, "Software Verification using k-Induction," in *Static Analysis*, pp. 351–268, 2011.

[6] N. EÉN and N. Sörensson, "Temporal Induction by Incremental SAT Solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, 2003.

[7] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Trans. Software Eng.*, vol. 38, no. 4, pp. 957–974, 2012.

[8] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *ICSE*, pp. 331–340, 2011.

[9] D. Beyer, "Second competition on software verification - (summary of sv-comp 2013)," in *TACAS, LNCS 7795*, pp. 594–609, 2013.

[10] J. Morse, L. Cordeiro, D. Nicole, and B. Fischer, "Model Checking LTL Properties over C Programs with Bounded Traces," in *Journal of Software and Systems Modeling, Springer (to appear)*, 2013.

[11] S. Muchnick, *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[12] T. Hoare, "An axiomatic basis for computer programming," in *Proceedings of ACM 12*, pp. 576–580, 583, 1969.

[13] J. Morse, L. Cordeiro, D. Nicole, and B. Fischer, "Handling Unbounded Loops with ESBMC 1.20 - (Competition Contribution)," in *TACAS, LNCS 7795*, pp. 619–622, 2013.

[14] J. Kiepert, "Creating a raspberry pi-based beowulf cluster," in *Boise State University*, pp. 1–17, 2013.

[15] ARM, *ARM1176JZF-S Technical Reference Manual*. 2009.

[16] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series, Academic Press, 2012.

[17] D. Groβe, H. Le, and R. Drechsler, "Proving Transaction and System-level Properties of Untimed SystemC TLM Designs," in *MEMOCODE, LNCS 1954*, pp. 108–125, 2000.

[18] M. Ljung, *Formal Modeling and Automatic Verification of Lustr Programs using NP-Tools*. 1999.

[19] M. Sheeran, S. Singh, and G. Stalmarck, "Checking Safety Properties using Induction and a SAT-Solver," in *FMCAD, LNCS 1954*, pp. 108–125, 2000.

[20] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS, LNCS 2988*, pp. 168–176, 2004.