BMCLua: Verificação de Programas Lua com *Bounded Model Checking*

Francisco A. P. Januario, Lucas C. Cordeiro,
and Vicente F. de Lucena Jr.
Universidade Federal do Amazonas – UFAM
Manaus, Amazonas, Brasil
Email: {franciscojanuario,lucascordeiro,vicente}@ufam.edu.br

Eddie B. L. Filho
Centro de Ciência, Tecnologia e Inovação
do Pólo Industrial de Manaus – CT-PIM
Manaus, Amazonas, Brasil
Email: eddie@ctpim.org.br

***Resumo*—O desenvolvimento de programas escritos na linguagem de programação Lua, a qual é muito utilizada em aplicações para TV digital e jogos, pode gerar erros, como por exemplo, bloqueio fatal, estouro aritmético e divisão por zero. Este trabalho tem como objetivo propor uma nova metodologia para a verificação de modelos de programas escritos na linguagem de programação Lua usando a ferramenta *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC). A metodologia proposta consiste na tradução de programas escritos na linguagem Lua para a linguagem ANSI-C. O programa resultante desta tradução é posteriormente verificado pelo ESBMC. Os resultados experimentais mostram que a metodologia proposta pode ser muito eficaz e eficiente para verificar propriedades de segurança na linguagem de programação Lua.**

Palavras-chave*—*linguagem lua, TV digital, verificação de modelos.

I. Introdução

Lua, é uma linguagem de *script* poderosa e leve, que foi projetada com a finalidade de estender funcionalidades para linguagens [19], como por exemplo, NCL [2], C/C++, Java, Ada, entre outras linguagens. Sua aplicação vai desde jogos até programas interativos para a TV digital [3], sendo também utilizado em aplicações críticas.

Assim como em outras linguagens de programação, erros podem ocorrer, como por exemplo, bloqueio fatal, estouro aritmético e divisão por zero, dentre outras violações. Durante (e após) o desenvolvimento, testes são realizados sobre o software para detectar possíveis erros que podem ocorrer durante a execução do programa. Uma das técnicas utilizadas em testes de programas é a verificação por métodos formais [4]. O método formal aplica modelos matemáticos para analisar sistemas complexos, fornecendo um verificação eficiente e reduzindo o tempo de validação de programas. Os modelos são baseados em teorias matemáticas, como por exemplo, Satisfatibilidade Booleana (SAT, do inglês Boolean Satisfiability) ou Teorias do Módulo de Satisfatibilidade (SMT, do inglês *Satisfiability Modulo Theories*).

Um das principais ferramentas utilizadas para a verificação formal de sistemas complexos em C/C++ é o *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC) [8], cuja arquitetura é baseada nas teorias do módulo de satisfatibilidade. A escolha do ESBMC deve-se ao fato de ser uma das ferramentas BMC mais eficiente, ganhando destaque nas últimas competições de verificação de software [5], [6], [7]. Outras ferramentas, como por exemplo, CBMC [9] e LLBMC [10] são baseados em SAT e SMT, respectivamente.

Este trabalho de pesquisa é motivado pela necessidade de se poder estender os benefícios da verificação de modelos, baseada nas teorias de satisfatibilidade, para programas escritos na linguagem de programação Lua. Assim, será possível detectar as causas, provavelmente violações no código, como por exemplo o estouro aritmético, que podem ocasionar falhas durante a execução do programa. Os resultados experimentais mostram que o desempenho do BMCLua [11] é semelhante ao ESBMC, em cerca de 70% dos *benchmarks* utilizados na avaliação experimental.

Este artigo está organizado da seguinte forma. A Seção II apresenta a ferramenta ESBMC e sua arquitetura, destacando suas principais características. Na Seção III é fornecido um estudo teórico sobre a linguagem Lua e suas estruturas básicas. Em seguida, na Seção IV, é abordada a ferramenta ANTLR, utilizada nesse trabalho para o desenvolvimento do tradutor da ferramenta BMCLua, descrita na Seção V. A Seção VI apresenta os resultados experimentais. Finalmente, as conclusões e trabalhos futuros são expostas na Seção VII.

II. ESBMC

Cordeiro et al. [8] apresentam a ferramenta ESBMC com uma abordagem eficiente para a verificação de programas embarcados ANSI-C utilizando BMC (do inglês, *Bounded Model Checking*) e solucionadores SMT. A ferramenta ESBMC faz uso dos componentes do *C Bounded Model Checker* (CBMC) [9], que é um verificador de modelos que utiliza solucionadores SAT. Cordeiro et al. [8] estendem e melhoram os benefícios da ferramenta CBMC para variáveis de tamanho finito de bits, operações de vetor de bits, vetores, estruturas e ponteiros. Para melhorar a escalabilidade e precisão de forma automática, os autores utilizaram várias teorias de *background* e os solucionadores SMT tais como CVC3 [12], Boolector [13] e Z3 [14].

Em particular, o ESBMC é um verificador de modelos, baseado em SMT, para programas ANSI-C/C++. Com o uso do ESBMC, é possível realizar a validação de programas sequenciais ou multi-tarefas (*multi-thread*) e também verificar bloqueio fatal, estouro aritmético, divisão por zero, limites de *array* e outros tipos de violações.

Considere um sistema de transição de estados definido como $M = (S, T, S_0)$, onde S representa o conjunto de estados, $S_0 \subseteq S$ representa o conjunto de estados iniciais e $T \subseteq S \times S$ é a transição de relação. O ESBMC é capaz de modelar o programa a ser analisado a partir de um sistema de transição de estados. Como resultado, este gera um gráfico

de fluxo de controle (*Control-Flow Graph* - CFG), que depois é verificado, de forma simbólica, a partir de um programa *GOTO*. Desta forma, 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 condição de verificação (*verification condition* - VC) ψ, de tal forma que ψ é satisfeita se, e somente se, ϕ possuir um contraexemplo de comprimento menor ou igual a *k*.

O processo de verificação do ESBMC é completamente automatizado, tornando-o ideal para testes eficientes de *software* embarcado de tempo real. Na Figura 1, é possível visualizar a arquitetura do ESBMC [8].

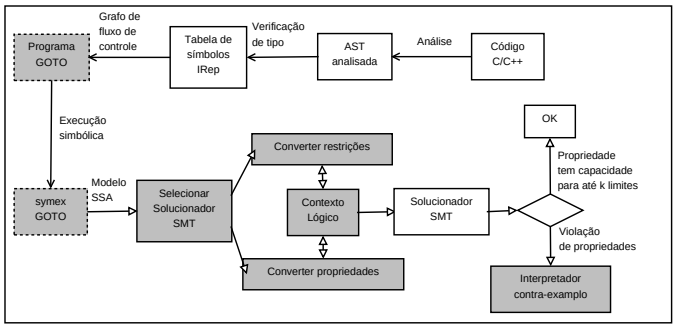

Fig. 1. A arquitetura do ESBMC.

Conforme pode ser observado, com essa abordagem é possível gerar condições de verificação para se checar estouro aritmético, realizar uma análise no CFG de um programa, para determinar o melhor solucionador para um dado caso particular e simplificar o desdobramento de uma fórmula. Em linhas gerais, o ESBMC converte um programa ANSI-C/C++ em um programa *GOTO*, ou seja, transformando expressões como *switch* e *while* em instruções *goto*, que é então simulado simbolicamente pelo *symex GOTO*. Então, um modelo em *Single Static Assignment* (SSA) é gerado, com a atribuição de valores estáticos às propriedades, para ser verificado por um solucionador SMT adequado. Se existe uma violação na propriedade, a interpretação do contraexemplo é realizada e o erro encontrado é informado. Caso contrário, a propriedade é válida até o limite de iterações *k*.

III. Linguagem Lua

Lua é uma linguagem de programação muito popular no desenvolvimento de jogos e aplicações para TV digital [15]. Na realidade, ela é uma linguagem de extensão que pode ser utilizada por outras linguagens, como C/C++ [16] e NCL [17], [2]. Lua é interpretada, sendo que o próprio interpretador foi desenvolvido em ANSI-C, o que a torna compacta, rápida e capaz de executar em uma vasta gama de dispositivos, que vão desde pequenos dispositivos até servidores de rede de alto desempenho [19].

A linguagem é poderosa devido em parte ao conjunto de bibliotecas disponíveis, que permitem estender funcionalidades, tanto por meio de código Lua ou através de bibliotecas externas escritas em código C. Essa característica facilita a integração da linguagem Lua com outras linguagens de programação como por exemplo C/C++, Ada e Java [18]. É uma linguagem portável para qualquer ambiente computacional Windows ou Unix, processadores ARM e mainframes.

Diferente de muitas linguagens de programação, Lua é ao mesmo tempo rápida e fácil de codificar. Isso torna a linguagem muito atrativa, por exemplo, para aplicações interativas voltadas para TV digital, que exigem facilidade de programação e resposta em tempo real.

A. Sintaxe e Estruturas

Na linguagem Lua, as variáveis não possuem declarações de tipos, dado que essa associação pode ser inferida dos valores armazenados nas variáveis (ver a primeira linha da Figura 2). Assim, a mesma variável pode armazenar dados diferentes em momentos distintos, durante a execução do programa. Outra característica da linguagem Lua é a múltipla atribuição de variáveis, onde vários valores são atribuídos à diferentes variáveis simultaneamente. No trecho de código Lua da Figura 2, observam-se algumas instruções que demonstram a forma de sintaxe da linguagem.

```
1 N, F = 1, 1
2 repeat
3    print(N.."! e "..F.."\n")
4    N = N + 1
5    F = F * N
6 until F >= 100
```

Fig. 2. Trecho de código Lua.

Os tipos de dados para a linguagem Lua são: *nil*, *boolean*, *number*, *string*, *table* e *function*. Vetores são declarados a partir do tipo *table*, enquanto funções em Lua são variáveis de primeira classe. Isso significa que, como qualquer outro valor, uma função pode ser criada e armazenada em uma variável (local ou global), ou no campo de uma *table* e ser passada como parâmetro ou valor de retorno de outra função. Os parâmetros da função não declaram o tipo, assim como uma variável, e a função pode retornar uma lista de valores. Lua também define as co-rotinas, que são funções que executam tarefas concorrentes de forma similar as *threads*. Sendo funções, as co-rotinas também são valores que podem ser definidos em variáveis declaradas na linguagem Lua.

Para o desenvolvimento de aplicações interativas para TV digital [2], foi criada a extensão NCLua que permite interagir à documentos NCL, por exemplo, através do controle remoto. A extensão NCLua disponibiliza alguns módulos, além da biblioteca padrão da linguagem Lua: *event*, *canvas*, *setting* e *persistent*.

Durante o desenvolvimento deste trabalho, algumas dificuldades foram encontradas para a verificação de programas Lua. Diferente de outras linguagens, como por exemplo C/C++, Lua não é uma linguagem fortemente tipada, ou seja, não é obrigatório associar um tipo de dado durante a declaração da variável. E como uma mesma variável pode assumir tipos de valores diferentes, a tradução torna-se particularmente complexa. O tipo *table* em Lua, também traz um grau de complexidade para sua tradução, pois a tabela é utilizada em Lua para criar outras estruturas, como por exemplo, os vetores e as *structs* utilizadas na linguagem ANSI-C. As funções são particularmente difíceis de traduzir, pois são consideradas tipos de valores em Lua, sendo utilizadas como objeto ou elemento de uma *table*.

IV. ANTLR

O ANTLR (*ANother Tool for Language Recognition*) [20] é uma ferramenta que permite gerar analisadores léxico e de sintaxe, possibilitando automatizar a construção de reconhecedores de linguagens. Essa ferramenta permite criar gramáticas para especificar a sintaxe da linguagem. Uma gramática é um conjunto de regras, cada uma expressando a estrutura de uma frase. A frase representa um fluxo de dados (*stream*) correspondente, por exemplo, as instruções de um programa escrito em uma linguagem de programação. A partir de uma gramática, como o exemplo ilustrado na Figura 3, o ANTLR pode gerar automaticamente classes escritas na linguagem Java, que possuem as funcionalidades do analisador léxico (*lexer*) e do analisador sintático (*parser*).

```
grammar T;

r  : ID '=' INT ';'
     {System.out.println("int" +
             $ID.text + "=" + $INT.text + ";");};

ID : 'a'..'z'+;
INT : '0'..'9'+;

WS : (' ' | '\t' | '\n' | '\r') + {$channel=HIDDEN;};
```

Fig. 3. Exemplo da gramática no ANTLR.

A. Tradução

O funcionamento básico de tradução, ilustrado na Figura 4, é realizado em duas etapas básicas: a análise léxica e a análise sintática.

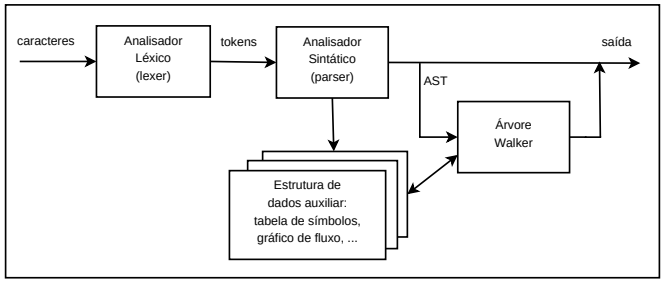

Fig. 4. Fluxo básico de tradução no ANTLR.

Na análise léxica, através da classe *lexer* gerada pelo ANTLR, um conjunto de caracteres de entrada, nesse caso, o código-fonte em Lua, produz na saída uma sequência de símbolos chamados *tokens*. Os *tokens* são segmentos de textos padrões que se repetem em um texto e possuem um significado comum, como por exemplo, o símbolo da barra (/) que separa o dia, mês e ano de uma data. Na fase da análise sintática, a classe *parser*, também gerada pelo ANTLR a partir da gramática, utiliza o fluxo de *tokens* produzidos pelo *lexer* para gerar uma resposta de saída conforme a sintaxe do código, podendo gerar uma árvore AST (*Abstract Symbol Tree*) com a qual é possível realizar a análise e manipulação de instruções de linguagem através da técnica de caminhamento em profundidade (DFS, do inglês *Depth-First Search*) [21]. Na Figura 5 são mostradas as classes associadas a árvore AST gerada pelo *parser*.

Para realizar a tradução de trechos de códigos, o ANTLR utiliza mecanismos de busca capazes de responder à eventos

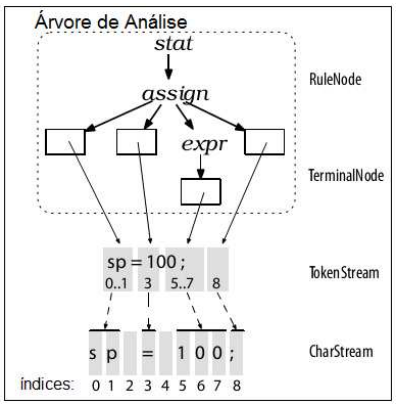

Fig. 5. Classes geradas a partir da AST [20].

desencadeados pelo caminhamento em profundidade (DFS) realizado na AST. O mecanismo de busca *visitor*, utilizado neste trabalho e ilustrado na Figura 6, permite controlar os eventos em uma DFS, tornando possível gerar um texto estruturado de saída através do método *println* da classe *System.out* da linguagem Java.

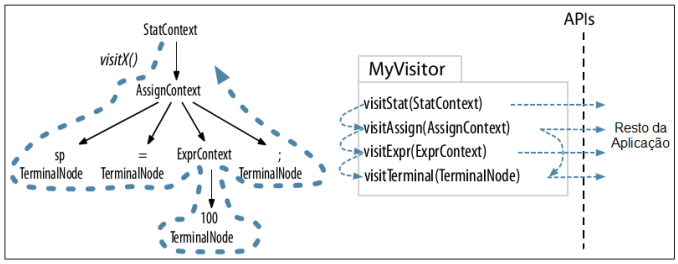

Fig. 6. Exemplo de busca com o *visitor*. [20]

V. BMCLua

O BMCLua, referência a *Bounded Model Checking - Lua*, é um verificador formal de modelos para programas Lua, em cuja versão preliminar [11], o tradutor foi desenvolvido utilizando técnicas não formais para a construção de aplicações de linguagens, sem o uso dos analisadores léxico e sintático. Na versão atual da ferramenta BMCLua, o módulo tradutor é desenvolvido utilizando o ANTLR, permitindo assim gerar automaticamente os analisadores *lexer* e *parser* a partir de uma gramática formal BNF (*Backus Naur Form*) [23], que é amplamente utilizada no desenvolvimento de compiladores. Essa nova abordagem simplifica e padroniza o desenvolvimento do tradutor. Na Figura 7 é apresentada a arquitetura do verificador BMCLua com o uso de uma gramática BNF.

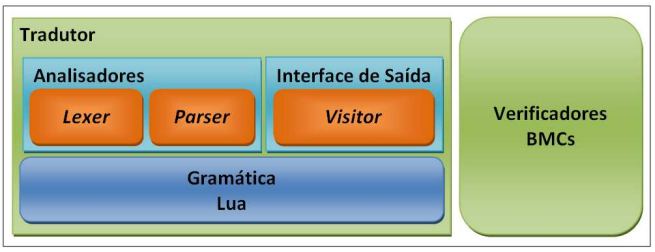

Fig. 7. Arquitetura do Verificador BMCLua.

A "Gramática Lua" é a base do tradutor e descreve o conjunto de regras válidas da sintaxe da linguagem Lua. A partir da gramática são construídos os analisadores "Lexer" e "Parser". A interface de saída "Visitor" permite gerar o código-fonte ANSI-C a partir do reconhecimento do código-fonte Lua. Finalizando a arquitetura, os "Verificadores BMCs", como por exemplo o ESBMC e o CBMC, representam ferramentas de verificação de modelos capazes de verificar código-fonte ANSI-C gerado pelo tradutor.

A. *Tradução*

Para que a verificação formal de um programa Lua seja realizada, é necessária a utilização de uma representação intermediária do código original, baseado nas teorias do módulo de satisfatibilidade, que permita a verificação das propriedades definidas dentro do código. Com o uso da ferramenta ESBMC, que realiza a verificação de códigos C/C++, o tradutor do BMCLua simplesmente realiza a tradução de programas Lua em seu equivalente funcional ANSI-C, que é posteriormente verificado automaticamente pelo ESBMC.

O tradutor foi desenvolvido utilizando a ferramenta ANTLR [20], que permitiu gerar classes Java para os analisadores, facilitando a integração com a ferramenta BMCLua.

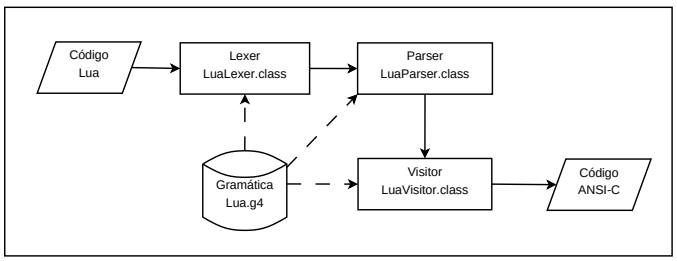

Fig. 8. Fluxo de tradução do BMCLua.

Na Figura 8 é mostrado o fluxo de tradução. Nesse diagrama pode-se observar as classes Java associadas aos analisadores *lexer* e *parser*, construídos a partir da gramática BNF Lua.g4 [24].

$$
\begin{array}{rcl}
block & ::= & stat \\
stat & ::= & varlist \text{ “=” } explist \\
 & & \mid functioncall \mid label \mid \text{“}break\text{”} \\
 & & \mid \text{“}do\text{” } block \text{ “}end\text{”} \\
 & & \mid \text{“}while\text{” } exp \text{ “}do\text{” } block \text{ “}end\text{”} \\
exp & ::= & \text{“}nil\text{”} \mid \text{“}false\text{”} \mid \text{“}true\text{”} \mid number \mid string \\
 & & \mid exp\ binop\ exp \mid unop\ exp
\end{array}
$$

Fig. 9. Exemplo de notação para a grámatica Lua.

A Figura 9 mostra um exemplo de notação para a gramática Lua, que consiste em um conjunto de regras que descrevem a sintaxe da linguagem Lua. A regra *block*, representa um bloco de instruções executáveis da linguagem Lua. Uma regra *block* pode ter uma ou várias instruções, cuja a sintaxe é especificada na regra *stat*. Uma regra *stat*, definida na gramática Lua, especifica a sintaxe de uma instrução da linguagem Lua. Por exemplo, a sintaxe da estrutura *while* é especificada como {**"while" exp "do" block "end"**}, onde *while*, *do* e *end* são palavras-chave que definem a estrutura de repetição e as palavras *exp* e *block* representam regras da gramática. É possível observar que a regra *block* é chamada de forma recursiva para formar a estrutura da linguagem Lua.

O mecanismo de busca do ANTLR utilizado no tradutor é o *visitor* [20], que permite gerar uma saída estruturada correspondente ao código-fonte ANSI-C. A classe *LuaVisitor* utiliza a estrutura AST para gerar um texto na saída com a estrutura correspondente a sintaxe ANSI-C e funcionalmente equivalente à sintaxe da linguagem Lua. Um exemplo da tradução de um código-fonte Lua para código-fonte ANSI-C é ilustrado na Figura 10.

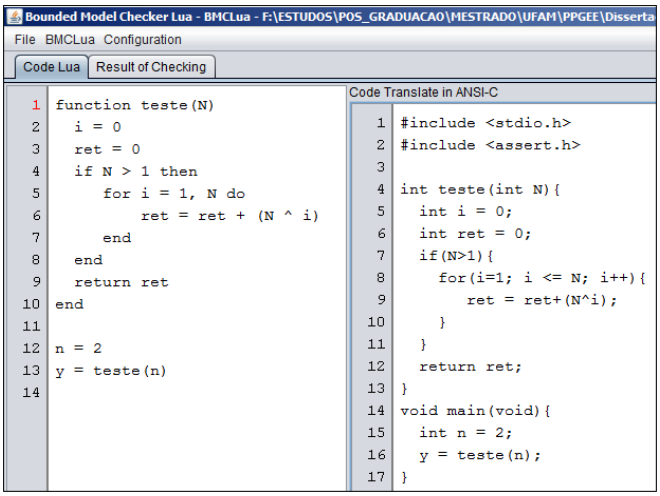

Fig. 10. Exemplo de tradução no BMCLua.

A ferramenta BMCLua, em sua versão atual, traduz um subconjunto de estruturas que abrange em torno de 50% da sintaxe da linguagem de programação Lua. Por exemplo, a tradução de instruções de atribuição múltipla, na linguagem Lua, gera um conjunto de instruções de atribuições simples, com declaração de tipo conforme o valor atribuído, na linguagem ANSI-C.

A Tabela I mostra as estruturas suportadas no todo ou parcialmente suportadas pela ferramenta BMCLua. As estruturas parcialmente suportadas estão em fase de desenvolvimento.

TABLE I. ESTRUTURAS LUA TRADUZIDAS PARA O BMCLUA.

Estrutura Lua	**Suportada**
Tipos primitivos (*boolean*, *number*, *string*)	SIM
Operadores relacional e lógicos	SIM
Operadores unários	PARCIAL
Tabelas	SIM
Atribuições simples e múltiplas	PARCIAL
Estruturas de controle	PARCIAL
Definição de função	SIM

Além das estruturas parcialmente implementadas, o tradutor ainda não suporta a conversão das funções das bibliotecas *coroutine*, *metatable*, *package*, *math*, *i/o* e *debug*. A ideia básica é acrescentar a declaração da função utilizada no código Lua, durante a conversão para ANSI-C, sem implementar o código da função que não é verificada pelo solucionador BMC.

A equivalência entre os códigos ANSI-C e o código original Lua é garantida quando a execução do programa

ANSI-C produz o mesmo resultado quando comparado com o resultado gerado pela execução do programa Lua. Assim, para cada código convertido de Lua para ANSI-C, foi realizado uma verificação funcional do programa C gerado pelo tradutor.

B. Verificação

A última etapa da metodologia BMCLua é a verificação do código-fonte ANSI-C gerado pelo tradutor. O módulo verificador, com o uso de uma ferramenta BMC , é encarregado de realizar a verificação do código-fonte ANSI-C, gerando um contraexemplo quando for encontrado alguma possível violação, como por exemplo, uma erro causado por divisão por zero. Caso seja detectada alguma violação no código, o BMCLua mostra uma lista de verificações realizadas dentro de um determinado limite de iterações, a violação que foi encontrada e a linha no código-fonte Lua onde foi detectado o erro.

Na Figura 11, um exemplo do resultado da verificação de um programa escrito na linguagem Lua é mostrado. É possível observar que na linha 3 do código ocorrerá uma violação na execução causada por uma divisão por zero. Essa violação foi detectada e informada no contraexemplo, mostrando que a variável "m" deve ser diferente de zero, e o número da linha no código-fonte Lua onde foi encontrada a violação.

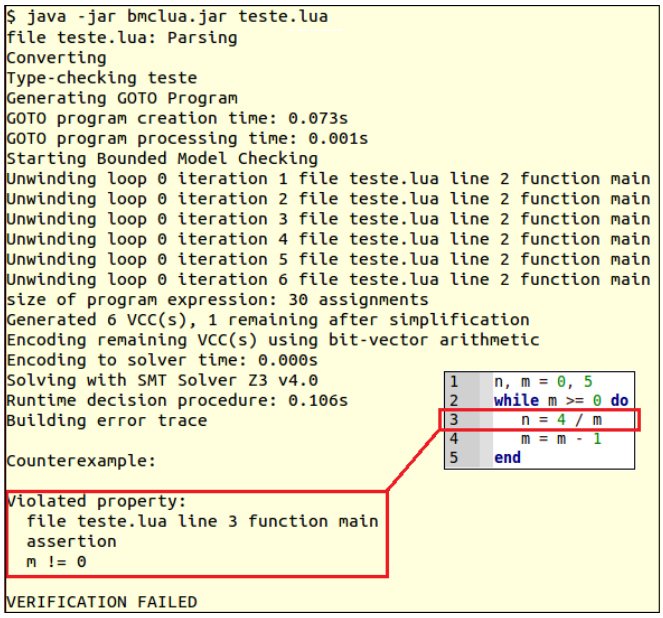

Fig. 11. Exemplo do resultado de verificação no BMCLua.

VI. Avaliação Experimental

Para avaliar a eficiência da ferramenta BMCLua, foram realizados experimentos, que consistiam em verificar algoritmos padrões usados para testes de desempenho de software, conhecidos como *benchmarks*. Foram utilizados os *benchmarks* *Bellman-Ford*, *Prim*, *BublleSort* e *SelectionSort*. O algoritmo *Bellman-Ford* é aplicado na solução do problema do caminho mais curto (mínimo), com aplicação em roteadores de rede de computadores, para se determinar a melhor rota para pacotes de dados. Assim como o *Bellman-Ford*, o algoritmo *Prim* é um caso especial do algoritmo genérico de árvore espalhada mínima, cujo objetivo é localizar caminhos mais curtos em um grafo. Os algoritmos *Bubblesort* e *SelectionSort* ordenam objetos, através da permutação iterativa de elementos adjacentes, que estão inicialmente desordenados. Estes *benchmarks* são os mesmos utilizados na avaliação de desempenho e precisão do verificador ESBMC [8].

A. O Ambiente de Testes

Os experimentos foram realizados na plataforma Linux, em um computador Intel Core i3 2.5 GHz, com 2 GB de RAM. Os tempos de desempenho dos algoritmos foram medidos em segundos, utilizando-se a classe *ManagementFactory* do pacote *java.lang*, da linguagem Java [25]. Essa classe Java permite mensurar o tempo de CPU, descontando os tempos de E/S e de outras tarefas compartilhadas.

B. Resultados

Para que a avaliação de desempenho do BMCLua fosse adequada, foram testados limites de *loops* diferentes, para cada algoritmo do conjunto de *benchmarks*. Por exemplo, para o algoritmo *Bellman-Ford*, iterações (*bounds*) para vetores variando de 5 a 20 elementos foram realizadas. Dessa forma, foi possível avaliar o comportamento do tempo de processamento devido ao aumento de elementos por vetor, com base no número de iterações.

Na Tabela II, os resultados gerados são exibidos, onde **E** identifica o total de elementos do vetor, **L** é o total de linhas de código Lua, **B** mostra o limite de iterações de *loops* realizadas, **P** significa o total de propriedades verificadas, **TL** é o tempo de processamento total, em segundos de verificação do código Lua na ferramenta BMCLua e **TE** é o tempo de processamento total, em segundos, de verificação do código ANSI-C, na ferramenta ESBMC. No tempo de processamento **TL**, deve-se considerar o tempo de tradução do código Lua para ANSI-C, além do tempo de verificação do código convertido. Os tempos **TL** e **TE** são utilizados como base de comparação de desempenho entre o código traduzido para ANSI-C e o código original do *benchmark* escrito em C, respectivamente.

TABLE II. Resultados de desempenho do BMCLua

Algoritmo	E	L	B	P	TL	TE
Bellman-Ford	5	43	6	1	< 1	< 1
	10	43	11	1	< 1	< 1
	15	43	16	1	< 1	< 1
	20	43	21	1	< 1	< 1
Prim	4	64	5	1	< 1	< 1
	5	64	6	1	< 1	< 1
	6	64	7	1	< 1	< 1
	7	64	8	1	< 1	< 1
	8	64	9	1	< 1	< 1
BubbleSort	12	30	13	1	< 1	< 1
	35	30	36	1	3	2
	50	30	51	1	6	5
	70	30	71	1	12	10
	140	30	141	1	68	52
	200	30	201	1	248	163
SelectionSort	12	33	13	1	< 1	< 1
	35	33	36	1	1	1
	50	33	51	1	3	2
	70	33	71	1	6	4
	140	33	141	1	42	25
	200	33	201	1	177	89

Os *benchmarks* padrões utilizam a função *assert*, disponível tanto em Lua como em C/C++, para verificar uma determinada propriedade. Assim, durante o processo de

tradução do código Lua para código ANSI-C, a instrução *assert* foi convertida adequadamente. Em todos os testes realizados não observou-se violação da propriedade verificada pela função *assert*.

Os resultados obtidos estão dentro do esperado, para o desempenho do tempo de processamento da ferramenta BMCLua, o que pode ser confirmado observando-se os valores obtidos nas colunas **TL** e **TE**, da Tabela II. Entretanto, é possível notar que, para os *benchmarks BubbleSort* e *SelectionSort*, o tempo de verificação do BMCLua é bem mais lento do que o ESBMC, se os limites de iterações de *loops* de 140 e 200, respectivamente, forem considerados. Isso é devido ao fato da conversão de código Lua para ANSI-C envolver mais variáveis do que o *benchmark* original, em ANSI-C. Algumas técnicas de otimização de algoritmos, como por exemplo, eliminação de expressões comuns CSE (do inglês, *Common Subexpression Elimination*) e propagação de constantes [22], serão utilizados para otimizar o código ANSI-C gerado pelo tradutor, permitindo reduzir o tempo de verificação pelas ferramentas BMC.

VII. Conclusão

O trabalho realizado alcançou o objetivo esperado, que consistia no desenvolvimento de uma ferramenta capaz de traduzir códigos escritos na linguagem Lua para a linguagem ANSI-C, além de validar o código traduzido através do verificador de modelos ESBMC. Desta forma, este artigo marca a primeira aplicação da técnica BMC para programas escritos na linguagem de programação Lua.

Os resultados dos experimentos comprovaram a eficácia do tradutor, mesmo que o desempenho tenha sido reduzido devido ao aumento do tempo de processamento total de verificação de programas Lua. De todos os *benchmarks* verificados, somente 21% apresentaram tempo de verificação no BMCLua maior quando comparado com a ferramenta ESBMC. Isso pode ser minimizado através da otimização do módulo tradutor da ferramenta BMCLua. Além disso, o BMCLua foi capaz de detectar todas as violações de propriedades nos *benchmarks* verificados.

Para trabalhos futuros, serão suportadas as estruturas de conversão de tipos, chamada de função e objeto, declaração das funções da biblioteca Lua e NCLua. Além disso, será também incorporada a API SMT-LIB [26] à ferramenta BMCLua, que permitirá incrementar o número de solucionadores SMT, como por exemplo, o Z3. Finalmente, a ferramenta BMCLua será integrada ao *IDE* Eclipse, através de um *plug-in*, que permita aos desenvolvedores validar os códigos escritos em linguagem Lua, utilizando o verificador ESBMC.

Agradecimentos

Esta pesquisa foi apoiada por Samsung, CNPq e FAPEAM.

Referencias

[1] J. Kurt e B. Aaron, *Beginning Lua Programming*. Indianapolis: Wiley Publishing, 2007. 644 p.

[2] L. Soares e S. Barbosa, *Programando em NCL 3.0: Desenvolvimento de Aplicações para Middleware Ginga, TV digital e Web*. São Paulo: Editora Campus, 2009. 341 p.

[3] A. Megrich, *Televisão digital: princípios e técnicas*. São Paulo: Érica, 2009. 336 p.

[4] C. Baier e K. Katoen, *Principles of Model Checking*. The MIT Press, 2008. 984 p.

[5] L. Cordeiro, J. Morse, D. Nicole e B. Fischer: *Context-Bounded Model Checking with ESBMC 1.17 - (Competition Contribution)*. In TACAS, LNCS 7214, pp. 534–537, 2012.

[6] J. Morse, L. Cordeiro, D. Nicole e B. Fischer, *Handling unbounded loops with ESBMC 1.20 - (Competition Contribution)*. In TACAS, LNCS 7795, pp. 621–624, 2013.

[7] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole e B. Fischer, *ESBMC 1.22 - (Competition Contribution)*. In TACAS, LNCS 8413, pp. 405–407, 2014.

[8] L. Cordeiro, B. Fischer, e J. Maraques-Silva. *SMT-Based Bounded Model Checking for Embedded ANSI-C Software*. In IEEE TSE, v. 38, n. 4, pp. 957–974, 2012.

[9] E. Clarke, D. Kroening, e F. Lerda. *A Tool for Checking ANSI-C Programs*. In 10th International Conference, TACAS 2004. Springer, v. 2988, pp. 168–176, 2004.

[10] S. Falke, M. Florian, e C. Sinz. *LLBMC: Improved Bounded Model Checking of C Programs Using LLVM*. In 19th International Conference, TACAS 2013. Springer, v. 7795, pp. 623–626, 2013.

[11] F. Januário, L. Cordeiro, e E. Filho. *Verificação de Códigos Lua Utilizando BMCLua*. Fortaleza – Brasil. XXXI Simpósio Brasileiro de Telecomunicações, 2013.

[12] A. Stump, C. Barrett, e D. Dill. *CVC: A Cooperating Validity Checker*. In 14th International Conference on Computer-Aided Verification. Springer, v. 2404, pp. 500–504, 2002.

[13] R. Brummayer e A. Biere. *Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays*. In 15th International Conference, TACAS 2009. Springer, v. 5505, pp. 174–177, 2009.

[14] L. Moura e N. Bjørner. *Z3: An efficient SMT solver*. In 14th International Conference, TACAS 2008. Springer, v. 4963, pp. 337–340, 2008.

[15] R. Brandão, G. Filho, C. Batista, e L. Soares. *Extended Features for the Ginga-NCL Environment: Introducing the LuaTV API*. zurich: Computer Communications and Networks (ICCCN), pp. 1–6, 2010.

[16] A. Hiischi. *Traveling Light, the Lua Way*. IEEE Software, pp. 31–38, 2007.

[17] ABNT (Brazilian Association of Technical Standards), *NBR 15606-2:2007: Digital terrestrial television − Data coding and transmission specification for digital broadcasting − Part 2: Ginga-NCL for fixed and mobile receivers − XML application language for application coding*. Rio de Janeiro: ABNT, 2007.

[18] R. Ierusalimschy, *Programming in Lua*. 2. ed. Rio de Janeiro: PUC-Rio, 2006. 308 p.

[19] J. Kurt e B. Aaron, *Beginning Lua Programming*. Indianapolis: Wiley Publishing, 2007. 644 p.

[20] T. Parr. *The Definitive ANTLR 4 Reference*. Texas: The Pragmatic Bookshelf, 2013. 328 p.

[21] T. Cormen, C. Leiserson, R. Rivest, e C. Stein. *Algoritmos: teoria e prática*. 6. ed. Rio de Janeiro: Editora Campus, 2002. 916 p.

[22] A. Aho, M. Lam, R. Sethi, e J. Ullman. *Compilers: Principles, Techniques and Tools*. 2. ed. Addison Wesley, 2007. 796 p.

[23] D. Knuth. *Backus Normal Form versus Backus Naur Form*. Selected Papers on Computer Languages (CSLI-Center) for Study of Language and Information, pp. 96–97, 2011.

[24] K. Sakamoto. *Grammars written for ANTLR v4*. Disponível em: <https://github.com/antlr/grammars-v4/tree/master/lua>. Acesso em: 31 dezembro 2013.

[25] P. Deitel e H. Deitel, *Java: How to Program*. 8. ed. Prentice Hall, 2009. 1184 p.

[26] C. Barrett, A. Stump e T. Cesare, *The SMT-LIB Standard: Version 2.0*. Department of Computer Science–The University of Iowa, 2010.