Verificação de Códigos Lua Utilizando BMCLua

Francisco de A. P. Januário, Lucas Cordeiro e Eddie B. L. Filho

***Resumo*— O presente artigo descreve uma abordagem de verificação de possíveis defeitos em códigos Lua, através da ferramenta *Bounded Model Checker*. Tal abordagem traduz código escrito em Lua para ANSI-C e o avalia através do *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC), que é um verificador de modelos de contexto limitado para códigos embarcados ANSI-C/C++ e possui a capacidade de verificar estouro de limites de vetores, divisão por zero e assertivas definidas pelo usuário. Este trabalho é motivado pela necessidade de se estender os benefícios da verificação de modelos, baseada nas teorias de satisfatibilidade, para códigos escritos na linguagem Lua. Os resultados apresentados, neste artigo, mostram a viabilidade da verificação de códigos Lua através da ferramenta ESBMC.**

Palavras-Chave*— *TV Digital, Verificação de modelos, Lua.

***Abstract*— The present paper describes an approach to check for potential defects on Lua codes, through the *Bounded Model Checker* tool. Such an approach translates code written in Lua to ANSI-C, which is validated by a contex-bounded model checker limited to embedded ANSI-C/C++ codes, that is, the *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC). It has the ability to check for array bounds, division by zero, and user-defined assertions. This work is motivated by the need for extending the benefits of bounded model checking based on satisfiability modulo theories, for code written in Lua. The results presented in this paper show the checking feasibility of Lua codes, through the ESBMC tool.**

Keywords*— *Digital TV, Model Checking, Lua.

I. Introdução

Neste artigo, descreve-se uma abordagem para a verificação de códigos Lua, baseada no *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC) [1], através da ferramenta *Bounded Model Checking - Lua* (BMCLua). Lua é uma linguagem de *script* poderosa e leve, que foi projetada para se estender aplicações [2]. O ESBMC, por sua vez, é um verificador de modelos para *software* ANSI-C/C++ embarcado. A linguagem ANSI-C, no contexto deste trabalho, passa a ser considerada como uma linguagem de modelo de verificação para o tradutor BMCLua, pois permite gerar um código capaz de ser verificado pelo ESBMC.

O BMCLua traduz um código Lua para ANSI-C (modelo), que é então verificado pelo ESBMC. Este permite identificar *deadlocks*, estouro aritmético e divisão por zero, dentre outras violações. Além disso, os códigos escritos em Lua podem conter *asserts*, que são similares aos empregados em ANSI-C.

Dado que a linguagem Lua é muito utilizada em aplicações imperativas e configuráveis, como, por exemplo, jogos, aplicações interativas para TV digital e aplicações em tempo real, esta também necessita de formas eficientes de verificação de defeitos, que surgem no processo de desenvolvimento.

Francisco de A. P. Januário, Lucas Cordeiro e Eddie B. L. Filho, Universidade Federal do Amazonas - UFAM, Av. Gen. Rodrigo Octávio Jordão Ramos, 3000, Manaus - AM, 69077- 000, Brasil. E-mails: franciscojanuario@ufam.edu.br, lucascordeiro@ufam.edu.br, eddie@ctpim.org.br.

A principal contribuição deste trabalho é fornecer uma abordagem inicial ao problema de validação de códigos Lua, servindo como base para melhorias e possíveis acréscimos à ferramenta BMCLua e ao verificador de modelos ESBMC. Com base na literatura disponível, este é o primeiro trabalho que aplica verificação de modelos para código Lua.

O artigo está organizado como descrito a seguir. Na seção II, apresenta-se um resumo sobre a linguagem Lua e suas estruturas básicas. Após isso, a seção III introduz o verificador ESBMC. Na seção IV, descreve-se a ferramenta BMCLua e sua arquitetura de tradução, com exemplos de códigos Lua traduzidos para a “linguagem de modelagem” ANSI-C, que são depois validados pelo verificador ESBMC. Finalmente, na seção V, alguns experimentos reais, com o BMCLua, são apresentados, e a seção VI expõe as conclusões e as sugestões para trabalhos futuros.

II. A linguagem Lua

Lua é uma linguagem de programação muito popular no desenvolvimento de jogos e aplicações para TV digital [3]. Na realidade, ela é uma linguagem de extensão que pode ser utilizada por outras linguagens, como C/C++ [4] e NCL [5], [6].

A linguagem de programação 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 [2].

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

A. Sintaxe e Estruturas

Para o desenvolvimento da ferramenta BMCLua, foi necessário um estudo básico sobre a estrutura e a sintaxe da linguagem de programação Lua. Durante o estudo, constatou-se que, com o intuito de facilitar a codificação com esta linguagem, retiraram-se elementos que tornam outras linguagens mais robustas, como, por exemplo, a obrigatoriedade de se definir o tipo de dado de variável, durante a sua declaração. No trecho de código Lua da Figura 1, observam-se algumas instruções que demonstram a forma de sintaxe da linguagem.

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 1). Assim, a mesma variável pode armazenar dados diferentes em momentos distintos, durante a execução do programa. Outra característica da linguagem Lua é a múltipla

```
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. 1. Trecho de código Lua.

atribuição de variáveis, onde vários valores são atribuídos a diferentes variáveis, ao mesmo tempo, como pode ser visto na na Figura 1, na primeira linha do código ilustrado.

Na Tabela I, os tipos de dados existentes na linguagem Lua são listados. Vale ressaltar que, ao se comparar Lua a outras linguagens, percebe-se que existem poucas variações para as estruturas de decisão e repetição.

TABELA I

TIPOS DE VÁRIAVEIS EM LUA

Tipo de variável	Descrição
nil	Ausência de valor
boolean	Valor lógico *true* ou *false*
number	Apenas tipos numéricos
string	Cadeia de caracteres
table	São *array's* associativos
function	Funções são valores
thread	Identifica corotina em execução
userdata	Tipo mapeado por ANSI-C

Além disso, existe uma diferença importante com relação ao tipo *array*, comum a todas as linguagens de programação. Em Lua, um *array* é um tipo chamado *Table*, cujos índices são inteiros que começam em 1, como ilustrado do código da Figura 2, nas linhas 1 a 4. Entretanto, é possível fazer com que o inicio de um *array* seja em 0, através da sintaxe ilustrada na Figura 2, linha 6.

```
1 array = {"A", "B", "C"}
2
3 -- Mostra o valor A do indice 1
4 print(array[1])
5
6 array2 = {[0]="A", [1]="B", [2]="C"}
7
8 -- Mostra o valor A do indice 0
9 print(array2[0])
```

Fig. 2. Declaração de *array* em Lua.

O tipo de variável *table* é o único mecanismo para a estruturação de dados em Lua e pode ser utilizado para se representar, além de *arrays*, estruturas de registro. Na Figura 3, é possível visualizar uma comparação entre uma estrutura de dados em ANSI-C (*struct*) e o seu código equivalente em Lua.

III. O VERIFICADOR ESBMC

O ESBMC é um verificador de modelos, baseado nas teorias do módulo da satisfatibilidade (*Satisfiability Modulo Theories* - SMT) [1], para códigos embarcados ANSI-C/C++. Através do ESBMC, é possível realizar a validação de programas sequenciais ou multi-tarefas (*multi-thread*) e também verificar *deadlocks*, estouro aritmético, divisão por zero, limites de *array* e outros tipos de violações.

Código em ANSI-C

```
struct{
   int dia;
   int mes;
   int ano;
} data;

data.dia = 28;
data.mes = 2;
data.ano = 2013
```

Código em Lua

```
data = {["dia"]=28,
        ["mes"]=2,
        ["ano"]=2013}
```

Fig. 3. Estrutura de registros em Lua.

A ferramenta ESBMC faz uso dos componentes do *C Bounded Model Checker* (CBMC), que é um verificador de modelos que utiliza solucionadores de *satisfiability* (SAT). O ESBMC é capaz de modelar, em um sistema de transição de estados, o programa a ser analisado. 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*. Com base em um sistema de transição de estados $M = (S, T, S_0)$, uma propriedade ϕ a ser analisada e um limite de iterações k, é possível verificar ϕ, construindo-se valores de contraexemplo para a validação da propriedade.

O ESBMC também permite estender essa abordagem de verificação de violação de propriedade para programas complexos, que possuam muitas iterações. Na Figura 4, é possível visualizar a arquitetura do ESBMC [1].

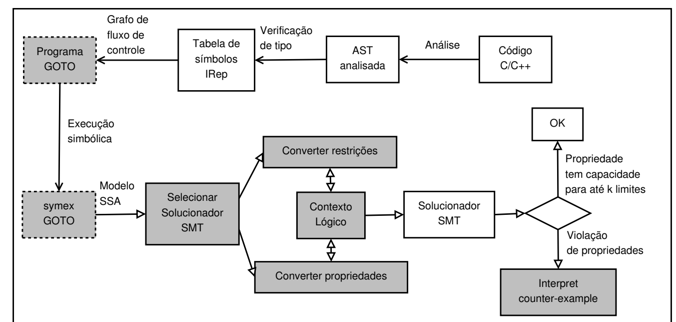

Fig. 4. A arquitetura do ESBMC.

Com essa abordagem, é possível gerar condições de verificação (*Verification Condition* - VC) 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 atende a capacidade do limite de iterações k.

O processo de verificação do ESBMC é automatizado, o que o torna ideal para testes eficientes de *software* embarcado de tempo real.

IV. O BMCLua

Este artigo visa comprovar a eficiência da verificação formal de códigos escritos em Lua, através do verificador de modelos ESBMC. A escolha desse verificador foi devida à sua eficiência em verificar defeitos em códigos ANSI-C/C++, como demonstrado nas duas últimas competições internacionais em verificação de software [7], [8].

A abordagem empregada neste trabalho é baseada no que foi desenvolvido por Klaus e Pressbuger [9], cujo princípio foi traduzir códigos escritos em Java para a linguagem de modelagem PROMELA [12], [13], que eram posteriormente checados pelo verificador de modelo de estados finitos *Simple PROMELA Interpreter* (SPIN). Em resumo, adotou-se a linguagem ANSI-C, como linguagem de modelagem para o tradutor, e o ESBMC, como verificador de modelos.

Como resultado, a ferramenta BMCLua foi desenvolvida, utilizando-se a linguagem Java. Na verdade, criou-se um pequeno ambiente integrado de desenvolvimento (*Integrated Development Environment* - IDE), que é capaz de codificar, executar e verificar códigos escritos em Lua. Vale ressaltar que o nome BMCLua é uma referência a *Bounded Model Checking - Lua*, ou seja, um verificador de modelos para Lua.

Com o objetivo de verificar programas Lua, o BMCLua primeiro realiza a tradução para a linguagem de modelagem ANSI-C, que em seguida é verificado pelo ESBMC. Na Figura 5, ilustra-se o fluxo de verificação com o BMCLua.

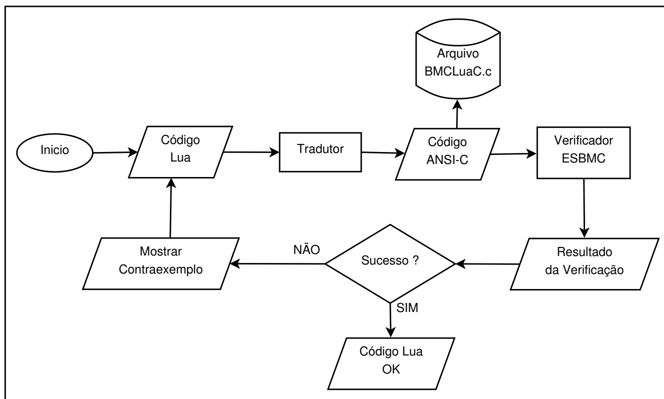

Fig. 5. Fluxo de verificação com BMCLua.

A. Tradução

Conforme já mencionado, o BMCLua realiza a tradução de códigos escritos em Lua para ANSI-C, que são então verificados pelo ESBMC.

O projeto atual contempla a tradução de apenas um subconjunto do Lua. Entretanto, não é difícil, em trabalhos futuros, estender-se o que foi desenvolvido para as demais funcionalidades da linguagem. Na Tabela II, listam-se os comandos e estruturas possíveis de tradução para ANSI-C.

A tradução dos comandos e das estruturas informadas na Tabela II consiste, basicamente, na substituição de palavras-chaves Lua por equivalentes na sintaxe ANSI-C.

A tradução de variáveis sem tipo definido, por sua vez, é realizada considerando-se o valor atribuído à mesma. Na Figura 6, o fluxo de tradução de variáveis é visualizado.

TABELA II
COMANDOS E ESTRUTURAS TRADUZIDAS

Comando/Estrutura Lua	**Comando/Estrutura ANSI-C**
comando *print*	*printf( );*
comando *break*	*break;*
comando *return*	*return;*
if .. else .. end	*if( ) { .. } else { .. }*
while .. do .. end	*while( ){ .. }*
for .. do .. end	*for(; ;){ .. }*
repeat .. until	*do{ .. }while(!..);*
do .. end	{ ... }

É importante salientar que a linguagem Lua é sensível aos caracteres maiúsculos e minúsculos (*case sensitive*), assim como na linguagem ANSI-C. Na Figura 7, ilustra-se um exemplo de tradução de código Lua para ANSI-C, realizada pelo BMCLua.

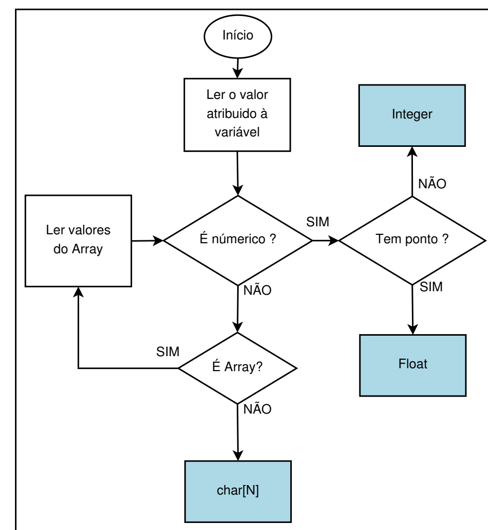

Fig. 6. Fluxo de tradução de variável.

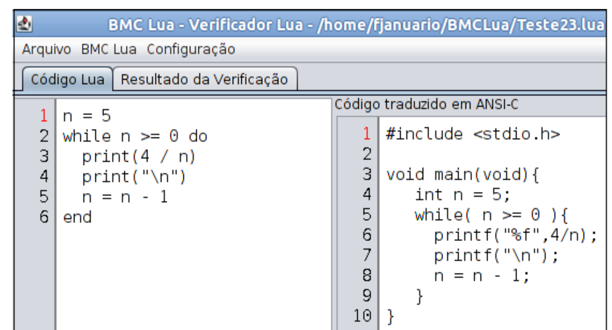

Fig. 7. Um exemplo de tradução no BMCLua.

Apesar do exemplo da Figura 7 mostrar um caso de conversão praticamente direta, existem algumas diferenças entre as linguagens Lua e ANSI-C que dificultam a tradução automática, como, por exemplo, o índice de um *array*. Na linguagem C/C++, o primeiro índice do *array* é zero, porém, na linguagem Lua, este valor é 1. Caso o *array* seja referenciado dentro de uma estrutura de repetição, como, por exemplo, o *for*, pode acontecer um erro de "índice fora de faixa". De modo a se contornar essa situação, a versão atual do BMCLua

utiliza modificadores, que são variáveis interpretadas apenas pela ferramenta de tradução e não influenciam na execução do código Lua.

No código da Figura 8, existe o modificador *__BMC_AttrForVal*, que faz a substituição do primeiro termo da estrutura *for* por *i=0*, e também o modificador *__BMC_AttrForRel*, que faz a substituição do operador relacional do segundo termo, da mesma estrutura, por $<$. Na Tabela III, os modificadores existentes no BMCLua são listados.

```
for i = 1, nodecount do

    __BMC_AttrForVal="i=0"
    __BMC_ModyForRel="<"

    if i == source then
       distance[i] = 0
    else
       distance[i] = INFINITY
    end
end
```

Fig. 8. Código Lua com modificadores BMCLua.

TABELA III

MODIFICADORES DO BMCLUA

Modificador	**Propósito**
__BMC_ArrayNullDimension	Declara um *array* nulo de dimensão **n**.
__BMC_AttrValArray	Modifica os valores dos elementos de um *array*.
__BMC_AttrVal	Modifica o valor de uma variável.
__BMC_AttrForVal	Modifica o valor inicial da variável da estrutura *for*.
__BMC_ModyForRel	Modifica o operador de relacionamento da estrutura *for*.
__BMC_ModyForLim	Modifica o limite de iteração da estrutura *for*.

B. Verificação

O passo seguinte, após a tradução, é a verificação do código ANSI-C gerado, através do verificador ESBMC, o que produz resultados como o mostrado na Figura 9. O contraexemplo é resultado da verificação do código apresentado na Figura 7.

C. Limitações

Dado que o presente projeto consiste em uma abordagem inicial, que visa o desenvolvimento futuro de uma ferramenta mais completa, existem algumas limitações impostas para verificação de códigos Lua.

O *IDE* BMCLua, no presente trabalho, traduz, para a linguagem ANSI-C, somente as estruturas e comandos descritos na Tabela II e *arrays* unidimensionais com apenas um tipo, que pode ser *int*, *float* ou *char*. Além disso, no que diz respeito à estrutura *for*, considera-se apenas a tradução do bloco "***for*** variável = valor_inicial, limite ***do*** .. ***end***".

Apesar da linguagem Lua trabalhar com tarefas concorrentes e co-rotinas, também conhecidas como fluxos de execução

```
file /home/fjanuario/BMCLua/BMCLuaC.c: Parsing
Converting
Type-checking BMCLuaC
Generating GOTO Program
Pointer Analysis
Adding Pointer Checks
Starting Bounded Model Checking
Unwinding loop 1 iteration 1 file /home/fjanuario/BMCLua/BMCLua
Unwinding loop 1 iteration 2 file /home/fjanuario/BMCLua/BMCLua
Unwinding loop 1 iteration 3 file /home/fjanuario/BMCLua/BMCLua
Unwinding loop 1 iteration 4 file /home/fjanuario/BMCLua/BMCLua
Unwinding loop 1 iteration 5 file /home/fjanuario/BMCLua/BMCLua
Unwinding loop 1 iteration 6 file /home/fjanuario/BMCLua/BMCLua
size of program expression: 36 assignments
Generated 6 VCC(s), 1 remaining after simplification
Encoding remaining VCC(s) using integer/real arithmetic
Solving with SMT Solver Z3 v3.2
Runtime decision procedure: 0.001s
Building error trace
Violated property:
  file /home/fjanuario/BMCLua/BMCLuaC.c line 15 function main
  division by zero
  n != 0

VERIFICATION FAILED
```

Fig. 9. Resultado de verificação ESBMC pelo BMCLua.

(*threads*), tais estruturas não foram traduzidas na versão atual do BMCLua.

D. Contribuição

O BMCLua traz como principal contribuição a verificação de *bugs*, como por exemplo, estouro aritmético e de limite de vetores, divisão por zero, dentre outras violações que estão relacionadas a linguagem de programação.

Em muitas aplicações interativas para a TV digital, *scripts* Lua usados com objetos NCL, podem apresentar defeitos quando em execução no ambiente do *middleware* Ginga [5]. Nessa situação, encontrar esses defeitos através da verificação de códigos Lua, antes da transmissão para os receptores de TV digital, poderá evitar que os *scripts* deixem de funcionar corretamente tornando a aplicação inoperante em algumas ou em todas as suas funcionalidades.

V. 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. Concluindo a lista dos *benchmarks*, os algoritmos *Bubblesort* e *SelectionSort* ordenam objetos, através da permutação iterativa de elementos adjacentes, que estão inicialmente desordenados [11]. Estes *benchmarks* são os mesmos utilizados na avaliação de desempenho e precisão do verificador ESBMC [1]. Em resumo, a avaliação experimental consistiu na verificação de desempenho da ferramenta BMCLua, tomando-se como comparação os dados experimentais do trabalho de Cordeiro et al. [1].

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 o método *currentTimeMillis* da classe *System*, da linguagem Java [14].

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 *arrays* 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 *array*, com base no número de iterações.

Na Tabela IV, os resultados gerados são exibidos, onde **E** identifica o total de elementos do *array*, **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.

TABELA IV
RESULTADOS DE DESEMPENHO DO BMCLUA

Algoritmo	E	L	B	P	TL	TE
Bellman-Ford	5	40	6	1	< 1	< 1
	10	40	11	1	< 1	< 1
	15	40	16	1	< 1	< 1
	20	40	21	1	< 1	< 1
Prim	4	61	5	1	< 1	< 1
	5	61	6	1	< 1	< 1
	6	61	7	1	< 1	< 1
	7	61	8	1	< 1	< 1
	8	61	9	1	< 1	< 1
BubbleSort	12	28	13	1	< 1	< 1
	35	28	36	1	2	2
	50	28	51	1	5	5
	70	28	71	1	10	10
	140	28	141	1	56	52
	200	28	201	1	203	163
SelectionSort	12	31	13	1	< 1	< 1
	35	31	36	1	1	1
	50	31	51	1	2	2
	70	31	71	1	5	4
	140	31	141	1	39	25
	200	31	201	1	175	89

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 IV. 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.

VI. CONCLUSÃO E TRABALHOS FUTUROS

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. Os resultados dos experimentos realizados com o BMCLua comprovaram o desempenho da ferramenta, observando-se as limitações de tradução de códigos Lua, determinadas nesse trabalho.

A versão atual do tradutor BMCLua não atende aos formalismos gramaticais como um analisador sintático (*parser*) e um analisador léxico (*lexer*). Atualmente, o tradutor está sendo modificado para atender a todo o conjunto de instruções e estruturas da linguagem Lua, utilizando-se como base o *framework Another Tool for Language Recognition* (**ANTLR**) [10], que permitirá criar um tradutor gramatical formal para a sintaxe completa da linguagem Lua. Essa nova versão irá eliminar o uso dos modificadores de variáveis que dificultam o uso da ferramenta.

A próxima etapa de desenvolvimento é utilizar a gramática do ANTLR para traduzir o código Lua diretamente para programas *goto* utilizando a representação intermediária (IRep) do ESBMC. Além disso, 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. Também se buscará adequar o BMCLua ao *middleware* Ginga, que é a plataforma onde são interpretadas as aplicações interativas, permitindo verificar o código Lua utilizado na aplicação.

REFERÊNCIAS

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

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

[3] R. Brandão, G. Filho, C. Batista, L. Soares, *Extended Features for the Ginga-NCL Environment: Introducing the LuaTV API*, In ICCCN, pp. 1–6, 2010.

[4] A. Hiischi, *Traveling Light, the Lua Way*, In IEEE Software, vol. 24, pp. 31-38, 2007.

[5] ABNT (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS). NBR 15606-2:2007: televisão digital terrestre: codificação de dados e especificações de transmissão para radiodifusão digital: parte 2 - Ginga-NCL para receptores fixos e móveis: linguagem de aplicação XML para codificação de aplicações. Rio de Janeiro: ABNT, 2007.

[6] L. Soares, S. Barboza, *Programando em NCL 3.0: desenvolvimento de aplicações para middleware Ginga, TV digital e Web*, Editora Campus, 1 ed., 2009.

[7] L. Cordeiro, J. Morse, D. Nicole, and B. Fischer, *Context-bounded model checking with ESBMC 1.17*, In TACAS, LNCS 7214., pp. 533-536, 2012.

[8] J. Morse, L. Cordeiro, D. Nicole, and B. Fischer, *Handling unbounded loops with ESBMC 1.20*, In TACAS, LNCS 7795, pp. 621-624, 2013.

[9] K. Havelund e T. Pressburger, *Model Checking Java Programs using Java PathFinder*. California: NASA Ames Research Center, 1999.

[10] T. Parr, *The Definitive ANTLR Reference - Building Domain-Specific Languages*. North Carolina: The Pragmatic Bookshelf, 2007.

[11] T.Cormen, *Algoritmos: teoria e prática*. Rio de Janeiro: Campus, 2002.

[12] Gerard J. Holzmann, *Software model checking with SPIN*. In Advances in Computers, v. 65, pp. 78–109, 2005.

[13] G. Holzmann and D. Bosnacki, *The Design of a Multicore Extension of the SPIN Model Checker*. In IEEE Trans. Software Eng., v. 33, n.10, pp. 659–674, 2007.

[14] H. Deitel and P. Deitel, *Java: How to Program*, Pearson, 8th Edition, 2009.