BMCLua : A Translator for Model Checking Lua Programs

Felipe R. Monteiro
Federal University of Amazonas
Manaus, Brazil
felipemonteiro@ufam.edu.br

Francisco A. P. Januário
Federal University of Amazonas
Manaus, Brazil
fapj@yahoo.com.br

Lucas C. Cordeiro
University of Oxford
Oxford, UK
lucas.cordeiro@cs.ox.ac.uk

Eddie B. de Lima Filho
Samsung Electronics Ltda.
Manaus, Brazil
eddie_batista@yahoo.com.br

ABSTRACT

Lua is a programming language designed as scripting language, which is fast, lightweight, and suitable for embedded applications. Due to its features, Lua is widely used in the development of games and interactive applications for digital TV. However, during the development phase of such applications, some errors may be introduced, such as deadlock, arithmetic overflow, and division by zero. This paper describes a novel verification approach for software written in Lua, using as backend the Efficient SMT-Based Context-Bounded Model Checker (ESBMC). Such an approach, called bounded model checking - Lua (BMCLua), consists in translating Lua programs into ANSI-C source code, which is then verified with ESBMC. Experimental results show that the proposed verification methodology is effective and efficient, when verifying safety properties in Lua programs. The performed experiments have shown that BMCLua produces an ANSI-C code that is more efficient for verification, when compared with other existing approaches. To the best of our knowledge, this work is the first that applies bounded model checking to the verification of Lua programs.

1. INTRODUCTION

Lua is a powerful and lightweight language, which was designed to extend functionalities of other programming languages [30], such as NCL [51], C/C++ [52], Java [49], and Ada [7]. It has been used in several consumer electronics applications, ranging from games for interactive digital TV to critical applications [6]. Lua is also the scripting language used in Ginga [17], which is a middleware standard developed for the Japanese-Brazilian Digital TV System [48].

As in other languages, errors may also occur in Lua programs, such as deadlock, arithmetic overflow, and division by zero, among other types of violations. During (and after) the software development phase, tests are carried out with the goal of detecting possible errors, which can occur during program execution. One possible alternative for testing is verification with formal methods [4], which applies mathematical models to the analysis of complex systems. Indeed, even a program with a single input integer has 2^{32} possible input values, which makes manual test a laborious task. Therefore, as an alternative, formal verification techniques could be applied in order to provide an efficient verification with significant time reduction during program validation. Furthermore, the related mathematical models are based on mathematical theories, such as Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT). One of the most popular tools used in formal verification of ANSI-C/C++ code is the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) [14], whose architecture is based on bounded model checking (BMC) and SMT. Indeed, the use of ESBMC, in this work, is due to the fact that it is one of the most efficient BMC tools to deal with bit-vector programs, as indicated in the last editions of the competition on software verification [15], [38], [39]. Nonetheless, there are other tools, like the C bounded model checker (CBMC) [13] and the low-level bounded model checker (LLBMC) [35], which are based on SAT and SMT, respectively.

The motivation for the present research is the need for developing an (efficient) program translator, in order to model check Lua programs based on satisfiability modulo theories. This is highly desirable, given that errors in Lua scripts may cause problems during program execution and even result in system reset, depending on the underlying platform. Particularly, this kind of test is very suitable for consumer electronics (CE) and helps avoid software refactoring and recall of CE devices.

As a result, it is possible to detect errors, probably caused by code violations, such as arithmetic overflow, which might lead to failures. Experimental results, regarding the proposed approach, show that BMCLua produces an ANSI-C code that is more efficient for verification than that provided by another program translator, Lua to Cee [34], which allows the conversion of Lua to ANSI-C. Indeed, Lua to Cee produces an extensive ANSI-C code, with many dependencies and function calls to the Lua API, which increases unnecessarily the complexity of the verification process. Experimental results also show that the performance of the bounded model checking - Lua (BMCLua) tool is comparable to that of ESBMC using standard benchmarks from literature [14].

1.1 Contributions

Given the current knowledge in software verification, this paper marks the first application of BMC to Lua programs. It is worth noting that the present paper extends a previously published work, in order to support a wide range of Lua constructs and syntax [28]. Indeed, the BMCLua version described and evaluated here has been significantly improved and uses the most recent stable versions of the chosen BMC tool and SMT solver. In brief, we make four major contributions:

i. we extend benefits of SMT-based context-bounded model checking for Lua programs, to detect more failures than other existing approaches, while keeping lower rates of false results; although SMT-based context-bounded model checking is not a novel technique, we have not seen in the literature its application to verify Lua programs.

ii. this work marks the first application of formal verification through model checking to Lua-based scripting programs.

iii. we provide an effective and efficient tool implementation (BMCLua) to support the checking of several Lua programs. BMCLua tool and all benchmarks used during the evaluation process are available at `https://sites.google.com/site/bmclua/`.

iv. we provide an extensive experimental evaluation of our approach against Lua to Cee [34] using well-known Lua benchmarks. Such evaluation has shown that our present approach outperforms the existing ones with respect to the number of correct results.

1.2 Organization of this work

The present work is organised as follows. Firstly, fundamental aspects regarding the Lua programming language and its basic structures and syntax, is provided. Next, section 2.2 presents the ESBMC tool, highlighting its main features. Next, the related work is discussed in section 2.2.1. Then, in section 2.3, the ANTLR tool is tackled, given that it is used here for the development of the BMCLua translator, as described in section 3. Experimental results are presented in section 5 and consider widely known benchmarks, which were previously used for the core BMC tool [14]. Finally, section 6 draws conclusions and discusses future research topics.

2. BACKGROUND

This section describes the basic concepts for understanding Lua programming language and the techniques used to perform formal verification of Lua programs.

2.1 Lua Programming Language

Lua is widely used for the development of games and interactive digital TV applications [18]. Indeed, it is an extension language that can be used by other languages, such as ANSI-C/C++ [27] and NCL [1, 51], which is one of the reasons of its popularity. Lua is interpreted, although it always precompiles the input source code to an intermediate form, which is then run. Besides, its interpreter was developed in ANSI-C, which makes it compact, fast, and able to run in a wide collection of devices, ranging from small systems up to high-performance network servers [30].

Lua is powerful, due in part to its set of available libraries, which allows functionality extension, both through Lua code and external libraries written in ANSI-C [30]. This feature favours its integration with other programming languages, such as ANSI-C/C++, Ada, and Java [33]. Besides, it can be easily ported to any computing environment, mainframes, and other processors. In contrast to many programming languages, programs written in Lua are fast to be executed and its ability to be integrated into many other languages makes it very attractive for the development of interactive applications focused on digital TV [6], [18], [48].

During the development of this work, some difficulties were identified, when verifying Lua programs. In particular, Lua is not strongly typed, that is, it is not required to associate a data type during variable declaration; the variable can bind to objects of different types during the execution of the program, which thus makes it hard to model the variable word-length and then check its related properties (*e.g.*, arithmetic overflow). Furthermore, as the same variable can receive different data types, the translation procedure becomes particularly complex. The *table* type also incurs a certain level of complexity, because it is used to create other structures, in the same way as *struct* in ANSI-C. Moreover, functions are particularly difficult to translate, because they can be used as objects or elements of tables.

2.2 ESBMC

Cordeiro, Fischer, and Marques-Silva [14] presented the ESBMC tool as an efficient approach for verifying ANSI-C embedded programs, which is based on bounded model checking and SMT solvers. Indeed, ESBMC is able to verify properties on single- and multi-thread programs such as deadlock, arithmetic overflow, division by zero, array bounds, among other types of violations.

In summary, ESBMC is able to model an input program from a state-transition system. Thus, consider a transition system with states defined as $M = (S, T, S_0)$, where $S_0 \subseteq S$, $T \subseteq S \times S$, S is a set of states, S_0 is a set of initial states, and T is a transition relation. Hence, given a transition system M, a property ϕ, and a limit k, ESBMC unfolds the system k times and transforms the result into a verification condition (VC) ψ, such that ψ is satisfiable if, and only if, ϕ has a counterexample of length less than or equal to k.

One can note that, with this approach, it is possible to generate VCs to check for properties and to perform an analysis on the program's control-flow graph (CFG), which is generated by ESBMC, in order to determine the best solver for a certain class of VCs and also to simplify the formula-deployment procedure. In general, ESBMC converts an ANSI-C/C++ program into a *GOTO* program, that is, it converts expressions, such as *switch* and *while*, into *GOTO* instructions, which are then symbolically simulated by the *GOTO* symex. Thus, a single static assignment (SSA) model is generated, with static-value assignments to properties and, based on them, two sets of quantifier-free formula are created: C for the constraints and P for the properties. The two sets of formula are then verified by an appropriate SMT solver thus, if there is a property violation, a counterexample interpretation is performed and the identified error is reported; otherwise, the related property is valid, up to k iterations.

The verification process of ESBMC is completely automatic, making it ideal for efficient testing of real-time embedded software, even in automated environments. It is worth noting that ESBMC has also been successfully extended to perform formal verification of C++ programs [47], Qt-based applications [36, 23, 37], and CUDA programs [44, 45, 46]. However, this work distinguishes from the rest, since it is the first to propose a translation procedure of the source code, which enables the use of not only ESBMC as verifier, but also others model checkers that support verification of ANSI-C code.

2.2.1 Related Work on Model Checking Lua

The work about CBMC, which was presented by Clarke, Kroening, and Lerda [13], showed the suitability of model checking regarding ANSI-C programs. Its main difference regarding ESBMC lies on the backend, that is, instead of using a SAT solver, ESBMC uses different background theories and pass verification conditions to a SMT solver [14]. It is worth noting that CBMC performs SAT based verification over its internal GOTO intermediate representation, and not directly on ANSI-C itself, as well as, ESBMC. Falke, Merz, and Sinz [35] also introduced LLBMC, which performs verification in C/C++ programs, using the Low-Level Virtual Machine (LLVM) intermediate language. The work about Java PathFinder (JFP), presented by Havelund and Pressburger [24], is also worth noting, due to the fact that it was the inspiration for the development of the translator module present in BMCLua, once it performs a translation of Java programs into Promela in order to apply model checking to Java programs [25].

Regarding Lua code verification, Lua checker [31] stands out as

a tool for analysing Lua code and is restricted to variables and constants. Lua Development Tools (LDT) [31] consist in a simple plug-in for static analysis, whose main applications are syntax highlighting and refactoring. Lua Inspect [31], in turn, infers values. Both LDT and Lua inspect are based on Metalua [21], which is an extension to Lua and provides features as AST compilation and syntax parsing. Lua Analysis in Rascal (Lua AiR) [31] performs code analysis as a pipeline, where the input Lua script is parsed (with a module generated by Rascal, through a Lua Grammar) into a parse tree, which is matched to an AST. The latter is used for type checking and is annotated with scope information. It is worth noting that a CFG is generated from the mentioned AST, which is used for fixed-point computations.

Model Checker	Supported Language	Intermediate Language	Based on
CBMC	C	-	SAT solver
LLBMC	C/C++	LLVM	SMT solver
JPF	Java	PROMELA	SAT solver
Lua Checker	Lua	-	-
LDT	Lua	-	Metalua
Lua Inspect	Lua	-	Metalua
Lua AiR	Lua	-	Rascal
BMCLua	Lua	ANSI-C	SMT solver

Table 1: Comparison regarding the presented related work.

In addition, regarding code translators, there is also a tool called Lua To Cee [34], which allows the conversion of Lua code into ANSI-C code, using the Lua API. Unfortunately, Lua To Cee produces an extensive ANSI-C code with many dependencies to Lua API, as aforementioned, which unnecessarily increases the complexity of the verification process. Table 1 shows a simple comparison among the mentioned work and the present proposal. When comparing with the previously mentioned tools, one can note that the proposed work is the first to use BMC techniques for checking Lua programs. Besides, regarding approaches specific to Lua, BMCLua incurs no extension to the base language and employs an intermediate representation, which is related to the original one, that is, ANSI-C. Indeed, Lua is implemented in ANSI-C.

```
1  grammar Lua;
2
3  chunk : block EOF # blockChunk
4        ;
5  block : stat* retstat? # statBlock
6        ;
7  stat
8      : ';'
9      | varlist '=' explist # assignMul
10     | functioncall
11     | label
12     | 'break' # breakStat
13     | 'goto' NAME # gotoStat
14     | 'do' block 'end' # doStat
15     | 'while' exp 'do' block 'end' # whileStat
16     | 'repeat' block 'until' exp # repeatStat
17     | 'if' exp 'then' block
18       ('elseif' exp 'then' block)*
19       ('else' block)? 'end' # ifStat
20     | 'for' NAME '=' exp ',' exp (',' exp)? 'do' block
21       'end' # forStat
22     | 'for' namelist 'in' explist 'do' block
23       'end' # forInStat
24     | 'function' funcname funcbody # functionStat
25     | 'local' 'function' NAME funcbody
26     | 'local' namelist ('=' explist)? # varLocal
27     ;
28
29 // LEXER
30 INT : [0-9]+ ;
31 FLOAT : [0-9]+ '.' [0-9]* [eE] [+-]? Digit+ ;
32 COMMENT : '--[[' .*? ']]' ;
33 NEWLINE : '\r'? '\n' -> skip ;
```

Figure 1: Grammar fragment illustrating some Lua features.

2.3 ANTLR

Another Tool for Language Recognition (ANTLR) [43] is a tool used to generate lexicon and syntax analyzers, which is able to automate the construction of language recognizers and allows the creation of grammars for specific language syntaxes. From a grammar, as shown in Fig. 1, ANTLR can automatically generate Java classes, which implement the necessary features of lexical (*lexer*) and syntactic (*parser*) analyzers.

2.3.1 Translation

The basic translation operation in ANTLR, shown in Fig. 2, is performed in two steps: the lexical analysis and the syntactic analysis.

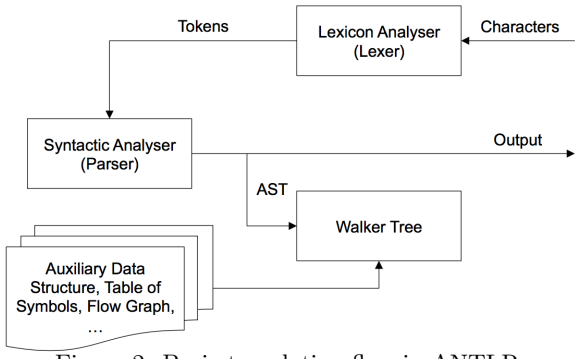


Figure 2: Basic translation flow in ANTLR.

In the lexical analysis stage, using the class *lexer* created by ANTLR, a set of input characters (Lua code) produces a sequence of symbols, called tokens. Those tokens are standard text segments repeated in a text, which have a common meaning, as the slash symbol separating day, month and year, in a date.

In the syntactic analysis (or parsing) stage, the class *parser*, also generated by ANTLR, from a specific grammar, uses the token flow produced by the class *lexer*, in order to generate an output response, according to the code syntax. Thus, it is possible to generate an abstract syntax tree (AST), which is used to perform analysis and manipulation of language instructions, by means of depth-first search (DFS) [16]. The classes associated with AST, which was generated by the parser, are shown in Fig. 3.

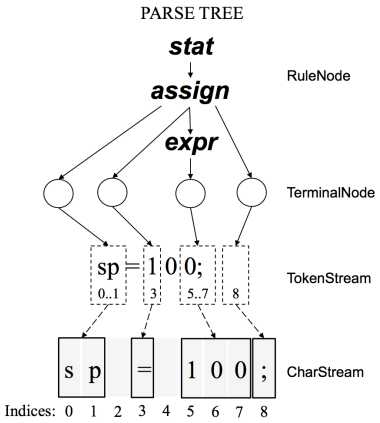


Figure 3: Classes generated from an AST.

In order to perform the translation of code fragments, ANTLR uses search mechanisms able to respond to events, which are triggered by the in-depth parsing performed in the related AST. In addition, a search mechanism called *visitor* [43], which was used in

this work and is shown in Fig. 4, allows the control of DFS events, enabling the generation of a structured text output by means of the method *println* in the class *System.out*, present in Java.

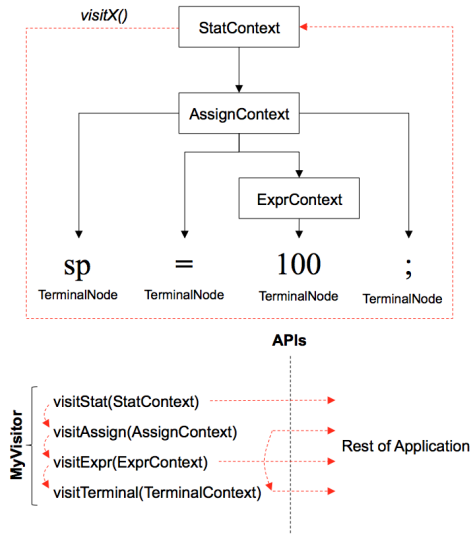


Figure 4: Example of a search procedure using *visitor*.

3. RESEARCH DESIGN AND METHODOLOGY

BMCLua is a translator suitable for model checking Lua programs. The current version of BMCLua is modular, structured, and the mentioned translator module was developed with ANTLR, which allows automated generation of the modules *lexer* and *parser*, from a formal *backus naur form* (BNF) grammar [32]. In fact, BNF is widely used in the development of compilers. In addition, this approach simplifies and standardizes the translator development.

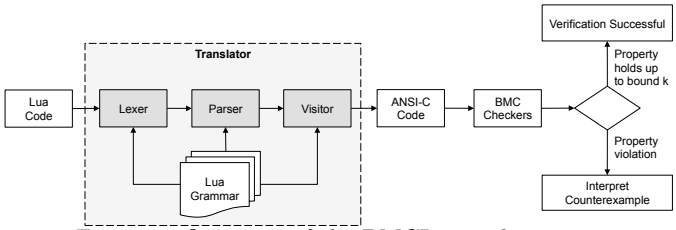


Figure 5: Overview of the BMCLua architecture.

Fig. 5 shows the current BMCLua architecture, which is based on BNF grammars. Indeed, BMCLua translator is based on the *Lua Grammar* component, which describes the valid rules for the Lua syntax. From this grammar, the classes *lexer* and *parser* are built. Moreover, the *visitor* output interface [43] allows the generation of ANSI-C code from Lua code. Finally, BMC checkers, such as ESBMC, CBMC, and LLBMC represent the model-verification tools, which are able to check the ANSI-C code generated by the translator. As aforementioned, in order to assess the effectiveness of our approach, we have applied ESBMC in our experimental results, due to its performance highlighted in the last edition of the International Competition on Software Verification (SV-COMP 2016) [9].

Fig. 6 shows an illustrative example of the translation, verification, and interpretation steps, which are all automatically performed by BMCLua, in order to model check Lua programs. Note that the Lua code is translated into an ANSI-C code, which usually adds more code lines to the translated program. Note further

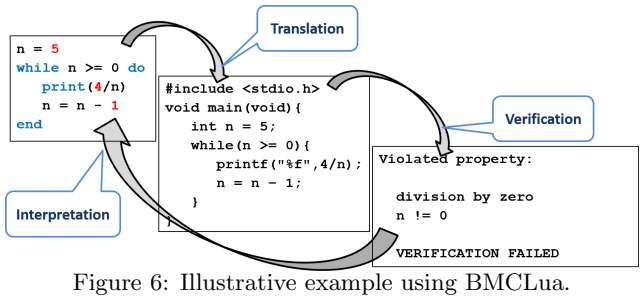


Figure 6: Illustrative example using BMCLua.

that the counterexample informs the code line and the violation in the original Lua program.

4. METHODS

In order to accomplish formal verification of Lua programs, via BMC, it is necessary to use an intermediate representation of the original source code. Thus, the BMCLua translator converts a Lua program into an equivalent ANSI-C code. Later, that same ANSI-C code will be automatically verified by ESBMC. Such an approach allows the use of different model checkers (*e.g.*, CPAChecker [20], CBMC [13] and LLBMC [35]) as back-ends for the BMCLua translator.

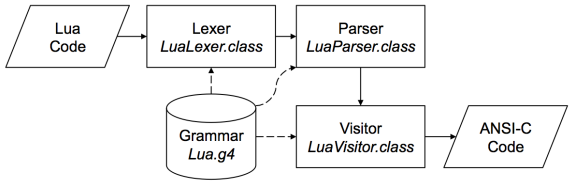


Figure 7: BMCLua translation flow.

The BMCLua translator was developed with the ANTLR tool [43], which allows the generation of Java classes for analysers. Indeed, such an approach favours integration with BMCLua. Fig. 7 presents the BMCLua translation flow. In this diagram, one can note the Java classes associated to modules *lexer* and *parser*, which were built from a BNF grammar.

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

Figure 8: Notation example for the *Lua grammar* component.

Fig. 8 shows an example of the *Lua grammar* notation, which consists of a set of rules describing the Lua syntax. In this example, a rule *block* represents a block of executable Lua instructions. In addition, the respective rule can contain one or many instructions, whose syntax is specified by rule *stat*. However, a rule *stat*, defined in *Lua grammar*, specifies the syntax of a Lua language instruction. For instance, the syntax of a *while* structure is specified as “*while*” *exp* “*do*” *block* “*end*”, where *while*, *do* and *end* are key-words that define the repetition structure, and *exp* and *block* represent grammar rules. The rule *block* is recursively called, in order to form the Lua language structure (see Fig. 8).

The *visitor* [43] mechanism from ANTLR allows the generation of a structured output corresponding to the associated ANSI-C code. Indeed, *LuaVisitor* uses the AST structure for generating

an output text, with ANSI-C syntax and functionally equivalent to the input Lua code. A translation example is shown in Fig. 9. In the current version, for instance, the translation of multiple assignments results in a set of simple assignment instructions, in ANSI-C, with a type declaration according to the assigned value. Table 2 shows what is either fully or partially supported by the BMCLua tool. It is worth noting that partially supported structures are under development.

```
1  function teste(N)
2    i = 0
3    ret = 0
4    if N > 1 then
5      for i = 1, N do
6        ret = ret + (N^1)
7      end
8    end
9    return ret
10 end
11
12 n = 2
13 y = teste(n)
```

(a) Lua code

```
1  #include<stdio.h>
2  #include<assert.h>
3  #include <math.h>
4
5  int teste (int N){
6    int i = 0;
7    int ret = 0;
8    if(N > 1){
9      for(i = 1; i <= N; i++){
10       ret = ret + pow(N,1);
11     }
12   }
13   return ret;
14 }
15
16 void main(){
17   int n = 2;
18   y = teste(n);
19 }
```

(b) ANSI-C code

Figure 9: Translation example using BMCLua.

Lua Structure	Supported
Primitive types (*e.g.*, numbers)	Yes
Relational and logical operators	Yes
Unary operators	Yes
Simple and multiple assignments	Yes
Control structures	Partial
Function definition	Partial

Table 2: Lua syntactic components supported by BMCLua.

Besides the partially implemented syntax, BMCLua translator does not convert library functions yet: *coroutine*, *metatable*, *package*, *math*, *i/o*, and *debug*. Actually, the function declaration used in Lua is kept, when converting to ANSI-C, without implementing the function code, given that the associated call is not verified by the BMC tool. In a future research, ANSI-C equivalent functions will be implemented, in such a way that function code, syntax, and parameters are completely verified.

The equivalency between the produced ANSI-C code and the original Lua code is guaranteed through black-box testing techniques: if the execution of the ANSI-C program produces the same result obtained with the Lua script, for the same input set. Therefore, for each code file translated from Lua to ANSI-C, a functional check of the ANSI-C program, generated by the translator, is performed, as described in previous work [29].

4.1 Translated Structures

In order to translate a Lua program into an ANSI-C program, an analysis of Lua language structures is performed by means of the AST tree generated by the parser, which is defined in terms of the BNF grammar. An example of a typical AST tree is shown in Fig. 10.

As already mentioned, the Lua language is not strongly typed [30]. Indeed, users do not need to assign any type to any variable, *i.e.*, values of different types can be assigned to the same variable, as shown in Fig. 11a. With the goal to overcome this, during the translation procedure, BMCLua creates new variables, in the ANSI-C code, if the value of a new type is assigned to the same variable in the Lua code, as shown in Fig. 11b.

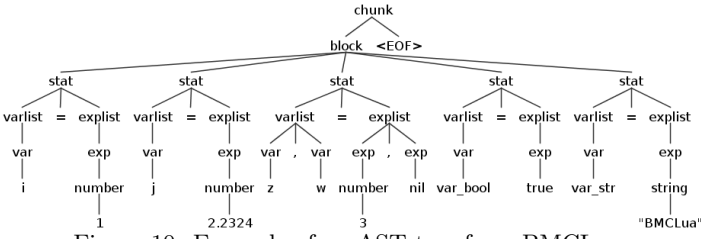


Figure 10: Example of an AST tree from BMCLua.

```
1 var = 10
2 var = 10.12
3 var = ''BMCLua''
```

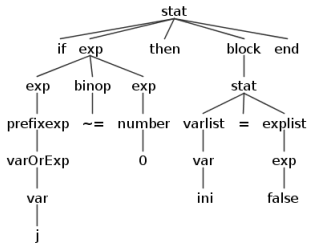
(a) Lua code

```
1  #include <string.h>
2  int var;
3  double var1;
4  char* var2;
5  void main(){
6      var = 10;
7      var1 = 10.12;
8      var2 = ''BMCLua''+'\0';
9  }
```

(b) ANSI-C code

Figure 11: Example of the Lua code translation.

The translator is able to convert common control structures, such as *if*, *while*, *for*, *do*, and *repeat*. Such a translation takes place during the modification of the respective command delimiters of each structure, for its equivalent in ANSI-C. For instance, the translation of the structure "*if exp then block end*", in which the respective AST tree is shown in Fig. 12a, is simplified by replacing keywords *then* and *end* by parentheses and brackets. This way, the translation procedure results in an *if* structure, where syntax corresponds to "*if (exp)* { *block* }", as shown in Fig. 12b.



(a) AST tree correspondent to an *if* structure

```
1 if(j != 0){
2     ini = false;
3 }
```

(b) ANSI-C code

Figure 12: Example of the Lua code translation, which contains an *if* structure.

For BMCLua, the *table* structure, which can represent vectors, lists, and records is translated into a more simplified *vector* structure. Fig. 13a shows a Lua program, which implements a *table* structure, and Fig. 13b shows the resulting translation, as a vector structure in ANSI-C.

```
1 local array={
2   [0]=2,
3   [1]=4,
4   [2]=8,
5   [3]=10
6 }
7 for i = 0, 3 do
8     print(array[i])
9 end
```

(a) Lua code

```
1  int i;
2  void main(){
3      int array[4] = {
4      2,
5      4,
6      8,
7      10};
8      for(i=0; i<=3; i=i+1){
9          print(array[i]);
10     }
11 }
```

(b) ANSI-C code

Figure 13: Example of the Lua code translation, which contains a *table* structure.

In the example shown in Fig. 14, the *table* structure corresponds to an array type and the *tableconstructor* node corresponds to the structure declared in the source code shown in Fig. 13a.

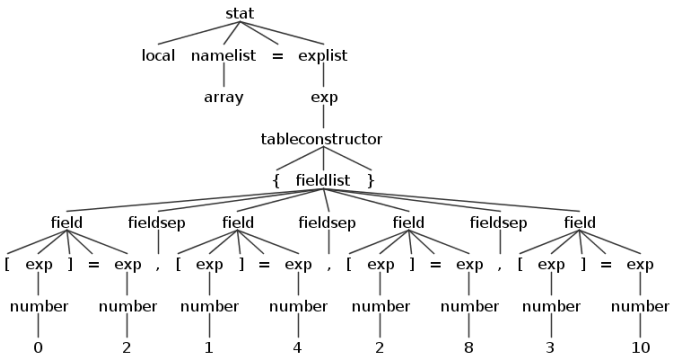


Figure 14: AST tree related to the *array* structure from Fig. 13a.

In addition to the mentioned structures, BMCLua also translates *function* structures. However, while the Lua language allows a certain function to return multiple values, BMCLua supports only functions that return single values. A translation of the *function* structure is shown in Fig.15. From the ANTLR parser, an AST tree is generated (see Fig.14), which allows the conversion of functions by the translator. By means of the *parlist* rule, from the *funcbody* node, the parameters of the respective function are stored in a linked list, in order to be used in the function call translation process. Furthermore, all values, returned by the function, can be listed in the *explist* node.

```
1 function dobro(x)
2     d = 2 * x
3     return d
4 end
5 y = dobro(2)
```

(a) Lua code

```
1 double d;
2 double dobro(int x){
3     d = 2 * x;
4     return d;
5 }
6 void main(){
7     y = dobro(2);
8 }
```

(b) ANSI-C code

Figure 15: Lua code translation example, which contains a *function* structure.

```
$ java -jar bmclua.jar teste.lua
file teste.lua: Parsing
Converting
Type-checking teste
Generating GOTO Program
GOTO program creation time: 0.073s
GOTO program processing time: 0.001s
Starting Bounded Model Checking
Unwinding loop 0 iteration 1 file teste.lua line 2 function main
Unwinding loop 0 iteration 2 file teste.lua line 2 function main
Unwinding loop 0 iteration 3 file teste.lua line 2 function main
Unwinding loop 0 iteration 4 file teste.lua line 2 function main
Unwinding loop 0 iteration 5 file teste.lua line 2 function main
Unwinding loop 0 iteration 6 file teste.lua line 2 function main
size of program expression: 30 assignments
Generated 6 VCC(s), 1 remaining after simplification
Encoding remaining VCC(s) using bit-vector arithmetic
Encoding to solver time: 0.000s
Solving with SMT Solver Z3 v4.0
Runtime decision procedure: 0.106s
Building error trace

Counterexample:

Violated property:
  file teste.lua line 3 function main
  assertion
  m != 0

VERIFICATION FAILED
```

```
1  n, m = 0, 5
2  while m >= 0 do
3      n = 4 / m
4      m = m - 1
5  end
```

Figure 16: Example of verification result using BMCLua.

4.2 Verification

The next stage of BMCLua is the verification of ANSI-C code generated by the translator. The verification module, which uses a BMC tool, is responsible for checking ANSI-C code and generating a counterexample, if a violation is found (*e.g.*, division by zero). In this particular case, if a violation is found, then BMCLua shows a list of states that lead to the violation, and also the code line where it was detected.

An example of verification result produced by BMCLua is shown in Fig. 16. As can be seen, in code line 3, a violation occurred due to a division by zero. Indeed, such a problem was detected and reported as a violated property, showing that the variable m must be different than zero and reporting the code line number, where the violation was found. Such verification process was carried out as described by Cordeiro *et al.* [14].

4.3 Interpretation

From the verification process of an ANSI-C code, ESBMC generates an output text that is processed in the interpretation stage. At this point, based on this output, BMCLua identifies the correct line and the error type in the respective Lua source code. Such identification is accomplished through a sorted list, with Lua code line numbers. Besides, its respective correspondence w.r.t the ANSI-C source code is kept in the translator. As an illustration of this feature, Fig. 17 shows ANSI-C code produced by the BMCLua translator, where each ANSI-C code line contains a comment that informs the respective line, in the Lua source code. In fact, the source code in Fig.17 corresponds to the translation of what is shown in Fig.13a.

```
1  ~/bmclua java -jar bmclua.jar exemplo.lua
2      -show -showln
3      -showlnlua -no-check-esbmc
4
5  1 - int i;
6  2 - void main(){
7  3 -     int array[4] = {2,4,8,10}; // lua code: 1
8  4 -     for(i=0; i<=3; i=i+1){ // lua code: 2
9  5 -         print(array[i]); // lua code: 3
10 6 -     } // lua code: 2
11 7 - }
```

Figure 17: Translation result with information of correspondent lines in Lua code.

5. RESULTS AND DISCUSSION

An experimental evaluation was carried out, with the goal to evaluate the efficiency and effectiveness of the BMCLua tool[1]. Such experiments consist in verifying standard well-known algorithms (*i.e.*, benchmarks) used to test software performance. Indeed, the primary purposes of such experiments are (1) to evaluate the trustworthy of the translation process, (2) to evaluate the BMCLua's performance against ESBMC, in order to check how much the translation procedure affects the time and correctness of the verification process, and (3) to compare BMCLua results against a state-of-the-art Lua translator, which in this case is Lua to Cee tool. We have used an evaluation approach similar to Alessandro *et al.* [3], however, in the present research, the *Bellman-Ford*, *Prim*, *BublleSort*, *SelectionSort*, *Factorial*, and *Fibonacci* algorithms were also used. The *Bellman-Ford* algorithm [26, 5] is commonly applied to the solution of the shortest path problem, with application in network routers, in order to determine the best route for data packages. As *Bellman-Ford*, the *Prim* algorithm [5], [12] is graph search algorithm for minimum spanning trees, whose goal is to find the shortest connection. The *BubbleSort* [16] and the *SelectionSort* [50] algorithms sort objects by means of iterative permutation of adjacent elements, which are initially disordered. The *InsertSort* algorithm [50], in turn, builds a sorted array, one element at a time, by scanning the input array

[1]Available at https://sites.google.com/site/bmclua/

from left to right. The *Factorial* is a recursive algorithm, which computes the factorial of a certain natural number. Concluding the list, the *Fibonacci* is an iterative algorithm for computing the *Fibonacci* sequence. Such benchmarks are the same used to evaluate performance and accuracy regarding ESBMC [14].

5.1 Environment

The mentioned experiments were performed on a desktop platform, with a clock of 2.5 GHz and 2 GB of RAM. The execution time of each algorithm was measured in seconds, using the class *ManagementFactory* from package *java.lang* [19]. Indeed, it allows the measurement of CPU allocation times, deducting I/O intervals and other shared tasks performed during program execution.

5.2 Results

Different loop limits were tested, for each algorithm in the benchmark set, in order to perform an appropriate evaluation of BMCLua. For instance, regarding the *Bellman-Ford* algorithm, array bounds ranging from 5 to 800 elements were used. Thus, it was possible to estimate the associated processing time due to the increase in the number of elements per array, based on the number of iterations.

In Table 3, the obtained results are shown. L is the number of Lua code lines, B shows the upper limit for loop iterations, that is, the number of array elements plus 1, TE is the elapsed processing time, in seconds, for verifying the associated ANSI-C code, using ESBMC, and TL is the elapsed processing time, in seconds, for verifying the input Lua code, using BMCLua. TL also considers the translation time (Lua to ANSI-C), in addition to the verification time of the resulting source code. This way, it is clear that TL will always be greater than TE. Indeed, TL and TE are used as performance-comparison metrics for the translated ANSI-C source code and the original benchmark code written in ANSI-C, respectively. For each experiment, one single property is verified.

The standard benchmarks use the *assert* function, available in both Lua and ANSI-C/C++, in order to verify a certain property. Thus, during the translation process from Lua to ANSI-C, the *assert* instruction is directly converted, so that the verification engine is able to check for a single safety property. In addition, all benchmarks underwent a black-box test procedure known as functional testing, in order to evaluate the trustworthy of the translation procedure. Thus, the same inputs were manually applied to the Lua programs and their (respective) translated ANSI-C programs, and we then checked their outputs equivalency. As a result, such test ensures that all benchmarks were successfully translated by BMCLua tool.

Values regarding columns TL and TE, for the *Bellman-Ford* and *InsertSort* algorithms, reveal that the proposed methodology presents a verification time comparable to that provided by ESBMC, that is, the elapsed time for verifying the input Lua code is similar to that of ESBMC, for an equivalent counterpart in ANSI-C. However, one can also note that, for the other benchmarks, the elapsed verification times presented by BMCLua are much higher than the ones presented by ESBMC, if the largest bound limits are considered (number of array elements).

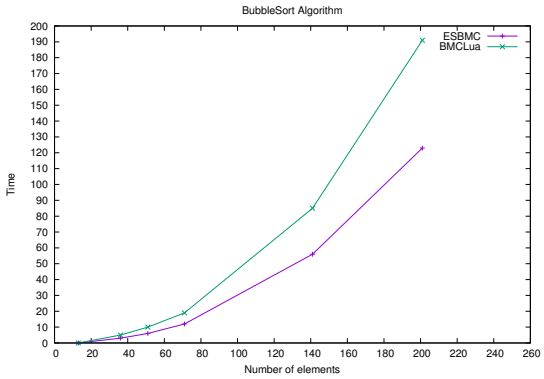
For instance, when the number of elements is increased, the *Prim* algorithm executions results in a wide difference between the verification time presented by both verifiers. This occurs because the conversion from Lua to ANSI-C involves many more variables than the original benchmark, in ANSI-C. This is a common problem in automated code conversion, since the resulting code is not properly optimised [2]. In the future, some algorithm-optimisation techniques, such as common sub-expression elimination and constant propagation [2], will be applied, in order to optimize the ANSI-C code generated by the translator. Such an approach has the potential to reduce the verification time presented by the BMC tools. Nevertheless, the associated verification times are still very similar, when small element numbers are considered and, for the initial range, they are even lower than one second.

Algorithm	L	B	TE	TL
Bellman-Ford	46	6	< 1	< 1
	46	11	< 1	< 1
	46	16	< 1	< 1
	46	21	< 1	< 1
	46	401	1	2
	46	801	2	3
Prim	70	5	< 1	< 1
	70	6	< 1	< 1
	70	7	< 1	< 1
	70	8	< 1	< 1
	70	9	< 1	< 1
	70	201	26	27
	70	401	65	114
BubbleSort	29	13	< 1	< 1
	29	36	3	5
	29	51	6	10
	29	71	12	19
	29	141	56	85
	29	201	123	191
SelectionSort	31	13	< 1	< 1
	31	36	1	2
	31	51	3	4
	31	71	5	7
	31	141	23	31
	31	201	49	70
Factorial	11	51	< 1	< 1
	11	101	< 1	< 1
	11	151	< 1	< 1
	11	201	< 1	< 1
	11	401	1	1
	11	801	3	6
	11	2001	26	38
Fibonacci	20	5001	< 1	1
	20	8001	1	2
	20	10001	2	3
	20	50001	9	14
	20	80001	15	22
	20	100001	19	28
InsertSort	21	21	< 1	< 1
	21	26	< 1	< 1
	21	51	< 1	< 1
	21	101	5	6
	21	201	32	33
	21	401	219	220

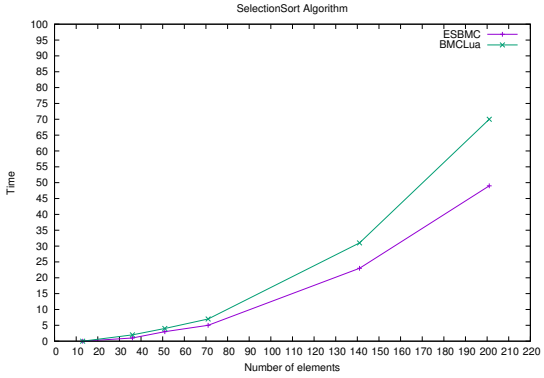
Table 3: Experimental results for the BMCLua tool.

As already mentioned, the verification time presented by BMCLua will always be higher than what is provided by ESBMC alone, due to the translation and also the interpretation (regarding the counterexample) procedures. All tests, for each benchmark, were performed with the same code, which can be seen in column L. This was done in order to plot consistent curves w.r.t. processing

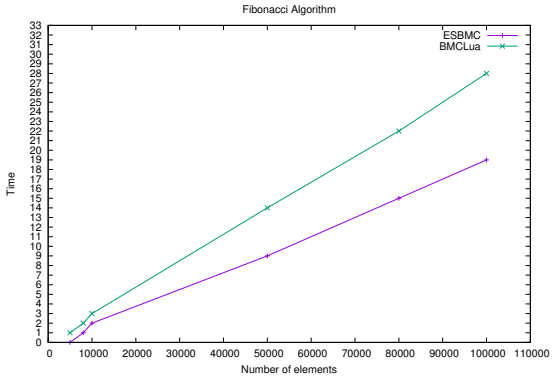
time and number of array elements, which can be seen Fig. 18.



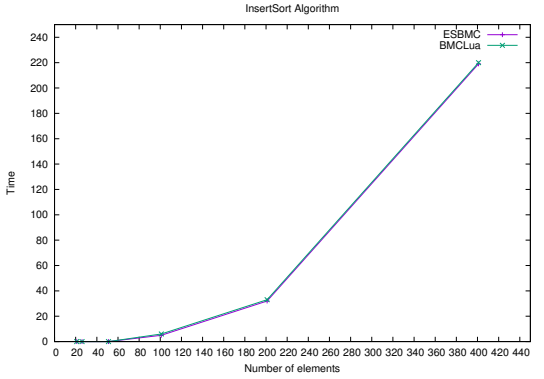
(a) BubbleSort algorithm



(b) SelectionSort algorithm



(c) Fibonacci algorithm



(d) InsertSort algorithm

Figure 18: Performance (in seconds) of the verification process with ESBMC and BMCLua, for the respective algorithms.

Indeed, one can notice that the *Fibonacci* algorithm presents a processing time that increases almost linearly with the number of elements. However, the *BublleSort*, *SelectionSort*, and *InsertSort* algorithms present an exponential tendency. This way, in general, the increase in processing time will be much higher with a larger number of elements, which naturally leads to research effort regarding the mentioned optimization techniques [2].

Finally, it is worth noting that, in all experiments, BMCLua did not report any false-positive or false-negative results, which shows its correctness regarding the verification of Lua programs.

5.3 Comparison with Lua To Cee

Comparing the translation results provided by Lua To Cee and BMCLua, it was noticed that Lua to Cee generates ANSI-C code with much more lines, including many functions from the Lua API (*e.g.*, *lua_pushnumber*, *lua_setfield*, *lc_nextra*) than the one output by BMCLua. Indeed, ANSI-C source code generated by BMCLua does not use any additional function from the Lua API, which makes it more legible and, consequently, simplifies the verification process. Furthermore, the verification of ANSI-C code generated by Lua To Cee was not possible using ESBMC (or any other BMC tool), due to the fact that many functions in the generated ANSI-C source code belong to specific libraries that are not available in the library set of such model checkers.

Table 4 shows a quantitative comparison regarding code lines produced by BMCLua and Lua To Cee. E indicates the upper limit for loop iterations, that is, the number of array elements plus 1, LB denotes the number of Lua code lines produced by BMCLua, and LC denotes the number of Lua code lines produced by Lua To Cee.

Algorithm	E	LB	LC
Bellman-Ford	6	46	596
	21	46	776
	801	46	10136
Prim	5	70	741
	8	70	795
	401	70	5475
BubbleSort	13	29	419
	71	29	531
	201	29	798
SelectionSort	13	31	418
	71	31	537
	201	31	803
Factorial	51	11	245
	401	11	245
	2001	11	245
Fibonacci	5001	20	288
	50001	20	288
	100001	20	288
InsertSort	21	21	399
	101	21	639
	401	21	1539

Table 4: Comparative results of code translation performed by BMCLua and Lua To Cee.

The benchmarks presented in Table 3 are used for performing translation comparisons. For the benchmarks *Bellman-Ford*, *Prim*, *BubbleSort*, *SelectionSort*, and *InsertSort*, the number of code lines produced by Lua To Cee increase significantly with the num-

ber of array elements, except for the benchmarks *Factorial* and *Fibonacci*, which do not use arrays (the produced ANSI-C code lines remain constant). However, note that, in all cases, the number of code lines produced by Lua To Cee is much higher than what is output by BMCLua. In conclusion, BMCLua produces a much more feasible ANSI-C code for formal verification.

6. CONCLUSIONS

The present work reached the planned goal, which consisted in developing a tool able to translate Lua code into ANSI-C and verify that through ESBMC. At the time this paper was written, it was the first reference regarding the use of BMC techniques for verifying Lua programs, by providing a complete and comprehensive methodology. Besides, this work is one of the few that tackle the verification of a language that is not strongly typed.

The experimental results confirmed the effectiveness of the proposed architecture, despite its performance reduction due to the increase in verification processing times, when Lua programs with many elements (in the associated structures) are used. In all tests cases regarding the adopted benchmarks, 55.81% presented a verification time 66.44% greater than that provided by ESBMC. However, this can be minimised with optimisations in the BMCLua translation module [2]. Additionally, BMCLua was able to detect all property violations in the verified benchmarks, without reporting any false-positive or false-negative results, since the k upper bound is known for those benchmarks. The performed experimental results also show that BMCLua produces more efficient and effective ANSI-C code for software verification, when compared with Lua To Cee.

It is worth noting that the present work can help increase robustness regarding consumer electronics devices, such as games, set-top boxes, and mobile devices, which are implemented using the Lua programming language. Indeed, it can be integrated into current software development processes, with the potential to reduce errors in Lua programs. As a consequence, the development phase itself can be shortened, given that many potential problems would be found earlier.

The main goal of the present research is to reach a high coverage degree regarding the Lua language syntax. Thus, one possible future work is to provide support to type conversion, function call, and declaration of functions present in Lua and NCLua libraries. Furthermore, the number of SMT-solver options can be increased (*e.g.*, Z3), for instance, with the incorporation of the SMT-LIB API [8] to the proposed BMCLua framework. Finally, the BMCLua tool could be integrated into existing development environments, through plug-ins that would allow developers to validate Lua code on the fly.

Acknowledgements

Part of the results presented in this paper were obtained through a research and human resources qualification project, for under- and post-graduate levels, in the areas of industrial automation, mobile devices software, and Digital TV. The respective project was sponsored by Samsung Eletrônica da Amazônia Ltda., under the terms of Brazilian Federal Law number 8.387/91. This research was also supported by CNPq 475647/2013-0 and FAPEAM 062.01722/2014 grants.

7. REFERENCES

[1] ABNT NBR 15606-2: "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"; (2009).

[2] Aho, Alfred V. and Sethi, Ravi and Ullman, Jeffrey D.: "Compilers: Principles, Techniques, and Tools"; Addison-Wesley, Boston / MA (1986).

[3] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania.: "Bounded model checking of software using SMT solvers instead of SAT solvers"; Int. J. Softw. Tools Technol. Transf. 11, 1 (January 2009), 69-83.

[4] Baier, C. and Katoen, J.: "Principles of Model Checking (Representation and Mind Series)"; The MIT Press. (2008).

[5] Bannister, M. J. and Eppstein, D.: "Randomised Speedup of the Bellman-Ford Algorithm"; CoRR, Cornell, 1111.5414, (2011); available at: `http://dblp.uni-trier.de/db/journals/corr/corr1111.html#abs-1111-5414`

[6] Barbosa, D. C. and Clua, E.: "Ginga Game: A framework for game development for the interactive digital television"; Proc. of the 6^{th} Brazilian Symposium on Games and Digital Entertainment (2009), 162–167.

[7] Barnes, J.: "Programming in Ada"; Cambridge University Press. (2014).

[8] Barrett, C. and Stump, A. and Tinelli, C.: "The Satisfiability Modulo Theories Library (SMT-LIB)"; (2010); available at: `www.SMT-LIB.org`

[9] Beyer, Dirk.: "Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)"; Springer Berlin Heidelberg; Chechik, Marsha and Raskin, Jean-François (2016), 887–904.

[10] Biere, A.: "Bounded Model Checking", in Handbook of Satisfiability; IOS Press. (2009), 457–481.

[11] Brummayer, R. and Biere, A.: "Boolector: An efficient SMT solver for bit-vectors and arrays"; Lect. Notes Comp. Sci. 5505, Springer (2009), 174-177.

[12] Cheriton, D. and Tarjan, R. E.: "Finding minimum spanning trees"; SIAM Journal on Computing, SIAM, 5, 4 (1976), 724–742.

[13] Clarke, E. and Kroening, D. and Lerda, F.: "A tool for checking ANSI-C programs"; Lect. Notes Comp. Sci. 2988, Springer (2004), 168–176.

[14] Cordeiro, L. C. and Fischer, B. and Marques-Silva, J.: "SMT-based bounded model checking for embedded ANSI-C software"; IEEE Trans. Software Eng., IEEE, 38, 4 (2012), 957–974.

[15] Cordeiro, L. C. and Morse, J. and Nicole, D. and Fischer, B.: "Context-Bounded Model Checking with ESBMC 1.17 - (Competition Contribution)"; Lect. Notes Comp. Sci. 7214, Springer (2012), 534–537.

[16] Cormen, T. H. and Stein, C. and Rivest, R. L. and Leiserson, C. E.: "Introduction to Algorithms"; McGraw-Hill Higher Education (2001).

[17] de Lucena Jr., V. F. and Viana, N. S. and Maia, O. B. and Chaves Filho, J. E. and da Silva Junior, W. S.: "Designing an extension API for bridging Ginga iDTV applications and home services"; IEEE Trans. Consumer Electronics, v.58, n.2 (2012), 1077–1085.

[18] de Melo Brandão, R. R. and de Souza Filho, G. L. and Batista, C. E. C. F. and Gomes Soares, L. F.: "Extended features for the Ginga-NCL environment: introducing the LuaTV API"; Proc. of the 19^{th} International Conference on Computer Communications and Networks, IEEE (2010), 1–6.

[19] Deitel, H. M. and Deite, P. J.: "Java: How to Program"; Prentice Hall: Upper Saddle River (2010), 315–336.

[20] Dirk Beyer, M. Erkan Keremoglu.: "CPAchecker: A Tool for Configurable Software Verification"; CAV (2011), 184–190.

[21] Fleutot, F. and L. Tratt, L.: "Contrasting compile-time meta-programming in Metalua and Converge"; Proc. of the 3^{rd} Workshop on Dynamic Languages and Applications, ACM (2007), 1–10.

[22] Friedman, D. P. and Wand, M.: "Essentials of Programming Languages, 3rd Edition"; The MIT Press. (2008), 55–65.

[23] Garcia M. P., Monteiro F. R., Cordeiro L. C., de Lima Filho E. B. *ESBMC^{QtOM}: A Bounded Model Checking Tool to Verify Qt Applications*. SPIN 2016; 97–103.

[24] Havelund, K. and Pressburger, T.: "Model checking JAVA programs using JAVA PathFinder"; International Journal on Software Tools for Technology Transfer, Springer, 2, 4 (2000), 366–381.

[25] Havelund, K and Skakkebæk, J U.: "Applying Model Checking in Java Verification"; In Proceedings of the 5^{th} and 6^{th} International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking, Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink (Eds.). Springer-Verlag, London, UK (1999), 216–231.

[26] Heineman, G. and Pollice, G. and Selkow, S.: "Algorithms in a Nutshell"; O'Reilly Media, Inc. (2008), 160–164.

[27] Hiischi, A.: "Traveling Light, the Lua way"; IEEE Software, IEEE, 24, 5 (2007), 31–38.

[28] Januário, F. A. P. and Cordeiro, L. C. and de Lucena Jr., V. F. and de Lima Filho, E. B.: "BMCLua: verification of Lua programs in digital TV interactive applications"; Proc. of the 3^{rd} Global Conference on Consumer Electronics, IEEE (2014), 707–708.

[29] Januário, F. A. P., Cordeiro, L. C., Lima Filho, E. B. ; Lucena Jr., V. F.: "BMCLua: Verificação de Programas Lua em Aplicações Interativas de TV Digital"; Proc. of the 4^{th} Simpósio Brasileiro de Engenharia de Sistemas Computacionais, SBESC (2014), 1–6.

[30] Jung, K. and Brown, A.: "Beginning Lua Programming"; Wiley (2007), 35–42.

[31] Klint, P. and Roosendaal, L. and van Rozen, R.: "Game developers need Lua air: static analysis of Lua using interface models"; Proc. of the 11^{th} International Conference on Entertainment Computing, Springer (2012), Berlin / Heidelberg, 530–535.

[32] Knuth, D. E.: "Backus Normal Form vs. Backus Naur Form"; Commun. ACM, ACM, 7, 12 (Dec 1964), New York / NY, 735–736.

[33] Lerusalimschy, R.: "Programming in Lua, Second Edition"; Lua.Org (2006).

[34] Manura, D.: "Lua To Cee"; (2012); available at: `http://lua-users.org/wiki/LuaToCee`

[35] Falke, S. and Merz, F. and Sinz, C.: "LLBMC: Improved bounded model checking of C programs using LLVM"; Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems; Notes Comp. Sci. 7795, Springer (2013), 623–626.

[36] Monteiro F. R., Cordeiro L. C., de Lima Filho E. B. *Bounded Model Checking of C++ Programs Based on the Qt Framework*. GCCE 2015; 179–447.

[37] Monteiro F. R., Garcia M. P., Cordeiro L. C., de Lima Filho E. B. *Bounded Model Checking of C++ Programs based on the Qt Cross-Platform Framework,* Softw Test Verif Reliab. 27 (2017) e1632.

[38] Morse, J. and Cordeiro, L. C. and Nicole, D. and Fischer, B.: "Handling Unbounded Loops with ESBMC 1.20 - (Competition Contribution)"; Lect. Notes Comp. Sci. 7795, Springer (2013), 619–622.

[39] Morse, J. and Ramalho, M. and Cordeiro, L. C. and Nicole, D. and Fischer, B.: "Handling Unbounded Loops with ESBMC 1.22 - (Competition Contribution)"; Lect. Notes Comp. Sci. 8413, Springer (2014), 405–407.

[40] Moura, L. M. and Bjørner, N.: "Z3: An Efficient SMT Solver"; Lect. Notes Comp. Sci. 4963, Springer (2008), 337–340.

[41] Moura, L. M. and Bjørner, N.: "Satisfiability modulo theories: an appetiser"; Proc. of the 12^{th} Brazilian Symposium on Formal Methods, Springer (2009), 23–36.

[42] Moura, L. M. and Bjørner, N.: "Satisfiability modulo theories: introduction and applications"; Commun. ACM, ACM, 54, 9 (2011), 69–77.

[43] Parr, T.: "The Definitive ANTLR Reference: Building Domain-Specific Languages"; Pragmatic Bookshelf (2007).

[44] Pereira P, Albuquerque H, Marques H, Silva I, Carvalho C, Santos V, Ferreira R, Cordeiro L. *Verificação de Kernels em Programas CUDA usando Bounded Model Checking.* WSCAD-SSC 2015; 24–35.

[45] Pereira P, Albuquerque H, Marques H, Silva I, Carvalho C, Santos V, Ferreira R, Cordeiro L. *Verifying CUDA Programs using SMT-Based Context-Bounded Model Checking.* SAC SVT track 2016; 1648–1653.

[46] Pereira P., Albuquerque H., Silva I., Marques H., Monteiro F. R., Ferreira R., Cordeiro L. C., *SMT-Based Context-Bounded Model Checking for CUDA Programs,* Concurrency Computat.: Pract. Exper. (2016)

[47] Ramalho M, Freitas M, Sousa F, Marques H, Cordeiro L, Fischer B. *SMT-Based Bounded Model Checking of C++ Programs*. ECBS 2013; 147–156.

[48] Salviato, T. P. and Costa, P. D. and Filho, J. G. P. and Vale, I. M.: "Framework for Context-Aware Applications on the Brazilian Digital TV"; Proc. of the 4^{th} International Conference on Ubi-Media Computing, IEEE (2011), 112–117.

[49] Schildt, H.: "Java: The Complete Reference (Complete Reference Series)"; Oracle Press. (2014).

[50] Sedgewick, R.: "Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, Third Edition"; Addison-Wesley (1998), 273–274.

[51] Soares, L. F. G. and Rodrigues, R. F. and Moreno, M. F.: "Ginga-NCL: The Declarative Environment of the Brazilian Digital TV System"; Journal of the Brazilian Computer Society, Springer, 12, 4 (2007), 37–46.

[52] Stroustrup, B.: "The C++ Programming Language - Special Edition"; Addison Wesley (2007).

[53] Stump, A. and Barrett, C. W. and Dill, D. L.: "CVC: a cooperating validity checker"; Proc. of the 14^{th} International Conference on Computer-Aided Verification, Springer-Verlag (2002), 500–504.