Universidade Federal do Amazonas

Faculdade de Tecnologia

Programa de Pós-Graduação em Engenharia Elétrica

BMCLua: Metodologia para Verificação de Códigos Lua Utilizando *Bounded Model Checking*

Francisco de Assis Pereira Januário

Manaus – Amazonas

Abril de 2015

Francisco de Assis Pereira Januário

BMCLua: Metodologia para Verificação de Códigos Lua Utilizando *Bounded Model Checking*

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica. Área de concentração: Automação e Controle.

Orientador: Lucas Carvalho Cordeiro

Co-orientador: Eddie Batista de Lima Filho

Francisco de Assis Pereira Januário

BMCLua: Metodologia para Verificação de Códigos Lua Utilizando *Bounded Model Checking*

Banca Examinadora

Prof. Ph.D. Lucas Carvalho Cordeiro – Presidente e Orientador
Departamento de Eletrônica e Computação
Universidade Federal do Amazonas – UFAM

Prof. Dr. Arilo Claudio Dias Neto
Instituto de Computação
Universidade Federal do Amazonas – UFAM

Prof. Dr. Waldir Sabino da Silva Júnior
Departamento de Eletrônica e Computação
Universidade Federal do Amazonas – UFAM

Manaus – Amazonas
Abril de 2015

À família.

Agradecimentos

Primeiramente agradeço ao Nosso Pai, a quem devo tudo o que sou e a oportunidade de estar aqui na Terra aprendendo e evoluindo.

Agradeço à minha família, pela paciência e compreensão durante o período de estudos do mestrado.

Agradeço aos professores Dr. Lucas Cordeiro e Dr. Eddie Batista, pela orientação, encorajamento e confiança durante todo o mestrado.

Agradeço aos professores Dr. André Cavalcante, Dr.Ing. Vicente Lucena, Dr. Waldir Sabino e Dr. Cícero Ferreira, pelos conselhos e apoio durante as atividades acadêmicas.

Agradeço aos colegas de mestrado pela cooperação e ajuda durante todo o período do mestrado.

Parte dos resultados apresentados neste trabalho foram obtidos através do projeto de pesquisa e formação de recursos humanos, à níveis de graduação e pós-graduação, nas áreas de automação industrial, software para dispositivos móveis e TV digital, patrocinado pela Samsung Eletrônica da Amazônia Ltda, nos termos da Lei Federal brasileira número 8.387/91.

Esta pesquisa também foi apoiado pelas subvenções CNPq 475647/2013-0 e FAPEAM 062.01722/2014.

"Ninguém pode voltar atrás e fazer um novo começo, mas qualquer um pode recomeçar e fazer um novo fim".

Emmanuel

Resumo

O desenvolvimento de programas escritos na linguagem de programação Lua, que é muito utilizada em aplicações para TV digital e jogos, pode gerar erros, *deadlocks*, estouro aritmético e divisão por zero. Este trabalho tem como objetivo propor uma metodologia de verificação para programas escritos na linguagem de programação Lua usando a ferramenta *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC), que representa o estado da arte em verificação de modelos de contexto limitado. O ESBMC é aplicado a programas embarcados ANSI-C/C++ e possui a capacidade de verificar estouro de limites de vetores, divisão por zero e assertivas definidas pelo usuário. A abordagem proposta consiste na tradução de programas escritos em Lua para uma linguagem intermediária, que é posteriormente verificada pelo ESBMC. O tradutor foi desenvolvido com a ferramenta ANTLR (do inglês "ANother Tool for Language Recognition"), que é utilizada na construção de analisadores léxicos e sintáticos, a partir da gramática da linguagem Lua. Este trabalho é motivado pela necessidade de se estender os benefícios da verificação de modelos, baseada nas teorias de satisfatibilidade, a programas escritos na linguagem de programação Lua. Os resultados experimentais mostram que a metodologia proposta pode ser muito eficaz, no que diz respeito à verificação de propriedades (segurança) da linguagem de programação Lua.

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

Abstract

The development of programs written in Lua programming language, which is largely used in applications for digital TV and games, can cause errors, deadlocks, arithmetic overflow, and division by zero. This work aims to propose a methodology for checking programs written in Lua programming language using the Efficient SMT-Based Context-Bounded Model Checker (ESBMC) tool, which represents the state-of-the-art context-bounded model checker. It is used for ANSI-C/C++ programs and has the ability to verify array out-of-bounds, division by zero, and user-defined assertions. The proposed approach consists in translating programs written in Lua to an intermediate language, which are further verified by ESBMC. The translator is developed with the ANTLR (ANother Tool for Language Recognition) tool, which is used for developing the lexer and parser, based on the Lua language grammar. This work is motivated by the need for extending the benefits of bounded model checking, based on satisfiability modulo theories, to programs written in Lua programming language. The experimental results show that the proposed methodology can be very effective, regarding model checking (safety) of Lua programming language properties.

Keywords: lua language, digital TV, model checking.

Índice

Índice de Figuras xii

Índice de Tabelas xiv

Abreviações xv

1 Introdução 1

1.1 Descrição do Problema 2
1.2 Objetivos 3
1.3 Descrição da Solução 4
1.4 Contribuições 5
1.5 Organização da Dissertação 5

2 Fundamentação Teórica 7

2.1 Verificação Formal 7
2.1.1 Verificação de Modelos 9
2.1.2 Modelagem de Sistemas 11
2.1.3 Lógica Proposicional 12
2.1.4 Lógica Temporal 13
2.2 Satisfatibilidade 14
2.2.1 Teorias do Módulo de Satisfatibilidade (SMT) 15
2.3 Linguagem Lua 17
2.3.1 Sintaxe e Estruturas 17
2.3.2 Bibliotecas 19
2.4 ANTLR 21
2.4.1 Tradução 21

2.4.2 Mecanismos de Busca 23
2.5 Resumo 25

3 Trabalhos Relacionados **26**
3.1 CBMC 26
3.2 LLBMC 27
3.3 ESBMC 28
3.4 Java *PathFinder* 29
3.5 Lua *To Cee* 30
3.6 Lua *Checker* 31
3.7 LDT e Lua *Inspect* 32
3.8 Lua AiR 32
3.9 Resumo 33

4 Verificação de Programas Lua com *Bounded Model Checking* **35**
4.1 Metodologia BMCLua 35
4.1.1 Tradução 38
4.1.2 Verificação 53
4.1.3 Interpretação 54
4.2 Avaliação Experimental 54
4.2.1 O Ambiente de Testes 55
4.2.2 Resultados 55
4.2.3 Discussão dos Resultados 57
4.3 Resumo 61

5 Conclusões **63**
5.1 Trabalhos Futuros 64

Referências Bibliográficas **66**

A Publicações **72**
A.1 Referente à Pesquisa 72

B Gramática Lua.g4 **73**

C Estruturas Lua Não Implementadas no Tradutor BMCLua 80

D Biblioteca NCLua Não Implementada no Tradutor BMCLua 84

E Benchmarks em Lua 86

E.1 Bellman-ford.lua 86

E.2 Prim.lua 87

E.3 BubbleSort.lua 88

E.4 SelectionSort.lua 89

E.5 InsertSort.lua 89

E.6 Factorial.lua 90

E.7 Fibonacci.lua 90

Índice de Figuras

1.1 Arquitetura da metodologia BMCLua 4

2.1 Abordagem da técnica de Verificação de Modelos 10

2.2 Árvore computacional a partir de fórmulas CTL 14

2.3 Notação Padrão. 16

2.4 Trecho de código Lua 17

2.5 Declaração de vetor em Lua 18

2.6 Estrutura de registros nas linguagens ANSI-C e Lua 19

2.7 Exemplo da gramática no ANTLR 21

2.8 Fluxo básico de tradução no ANTLR 22

2.9 Classes geradas a partir da AST 22

2.10 Classes de contexto da árvore de análise 23

2.11 Exemplo de busca com o *listener* 24

2.12 Exemplo de busca com o *visitor* 24

3.1 Exemplo de conversão utilizando BMC 27

3.2 Processo de verificação LLBMC 27

3.3 A arquitetura do ESBMC 28

3.4 Trecho do código C gerado pela ferramenta Lua To Cee 30

3.5 A arquitetura do Lua AiR 32

4.1 Fluxo de verificação com BMCLua 36

4.2 Fluxo de tradução com BMCLua 37

4.3 Fluxo de interpretação com BMCLua 37

4.4 Fluxo de tradução do BMCLua 38

4.5 Trecho da gramática Lua 39

4.6 Arquivos gerados a partir da gramática Lua . . . 40
4.7 Árvore AST gerada pelo analisador *parser* . . . 40
4.8 Exemplo de tradução no BMCLua . . . 41
4.9 Exemplo de tradução de variáveis no BMCLua . . . 42
4.10 Árvore AST do código Lua ilustrado na Figura 4.9 . . . 43
4.11 Exemplo de tradução de mudança de valor em variável . . . 43
4.12 Exemplo de tradução de estruturas de controle no BMCLua . . . 44
4.13 Tradução da estrutura *if* . . . 45
4.14 Tradução da estrutura *for* . . . 45
4.15 Exemplo de tradução de *table* no BMCLua . . . 46
4.16 Exemplo de tradução de função no BMCLua . . . 47
4.17 Exemplo de código NCLua para TV digital . . . 48
4.18 Sub-árvore AST da função *drawRect* do código NCLua . . . 49
4.19 Declaração de funções *canvas* em ANSI-C . . . 49
4.20 Sub-árvore AST da variável *event.post* do código NCLua . . . 50
4.21 Definição de estruturas *evt* e *event.post* em ANSI-C . . . 50
4.22 Código da Figura 4.17 traduzido para ANSI-C . . . 51
4.23 Exemplo do resultado de verificação no BMCLua . . . 53
4.24 Exemplo de tradução no BMCLua com informação das linhas no código Lua . . . 54
4.25 Gráfico do desempenho (segundos) para o algoritmo *Bellman-Ford* . . . 58
4.26 Gráfico do desempenho (segundos) para o algoritmo *Prim* . . . 58
4.27 Gráfico do desempenho (segundos) para o algoritmo *Bubblesort* . . . 59
4.28 Gráfico do desempenho (segundos) para o algoritmo *Selectionsort* . . . 59
4.29 Gráfico do desempenho (segundos) para o algoritmo *Factorial* . . . 60
4.30 Gráfico do desempenho (segundos) para o algoritmo *Fibonacci* . . . 60

Índice de Tabelas

1.1 Estruturas Lua suportadas pelo tradutor do BMCLua. 5

2.1 Tabela verdade para a fórmula φ. 15

3.1 Comparativo das estruturas Lua verificadas pelas ferramentas BMCLua e Lua *Checker*. 32
3.2 Tabela comparativa entre os trabalhos relacionados. 34

4.1 Estruturas traduzidas pelo BMCLua 52
4.2 Resultados de desempenho do BMCLua 56
4.3 Estruturas Lua verificadas pelas ferramentas BMCLua e *Lua Checker*. 62

5.1 Estruturas Lua definidas para o BMCLua. 64

C.1 Estruturas faltantes no tradutor do BMCLua. 80
C.2 Funções da biblioteca básica faltantes no tradutor do BMCLua. 81
C.3 Funções da biblioteca de pacotes faltantes no tradutor do BMCLua. 81
C.4 Funções da biblioteca de co-rotinas faltantes no tradutor do BMCLua. 81
C.5 Funções da biblioteca de co-rotinas faltantes no tradutor do BMCLua. 82
C.6 Funções da biblioteca de matemática faltantes no tradutor do BMCLua. 82
C.7 Funções da biblioteca *String* faltantes no tradutor do BMCLua. 82
C.8 Funções da biblioteca de entrada/saída faltantes no tradutor do BMCLua. 83
C.9 Funções da biblioteca OS faltantes no tradutor do BMCLua. 83
C.10 Funções da biblioteca debug faltantes no tradutor do BMCLua. 83

D.1 Funções da biblioteca NCLua faltantes no tradutor do BMCLua. 85

Abreviações

ADT - *Algebraic Data Type*

ANTLR - *ANother Tool for Language Recognition*

AST - *Abstract Symbol Tree*

BMC - *Bounded Model Checking*

BMCLua - *Bounded Model Checking Lua*

BNF - *Backus Naur Form*

CBMC - *C Bounded Model Checker*

CF - *Control-Flow*

CFG - *Control-Flow Graph*

CSE - *Common Subexpression Elimination*

CTL - *Computation Tree Logic*

DFS - *Depth-First Search*

ESBMC - *Efficient SMT-Based Context-Bounded Model Checker*

ILR - *Intermediate Logic Representation*

IDE - *Integrated Development Environment*

JPF - *Java PathFinder*

LALR - *Look–Ahead LR*

LDT - *Lua Development Tools*

LLBMC - *Low Level Bounded Model Checker*

LLVM - *Low Level Virtual Machine*

LT - *Lógica Temporal*

LTL - *Linear Temporal Logic*

NCL - *Nested Context Language*

RD - *Reaching Definitions*

SAT - *SATisfiability*

SMT - *Satisfiability Modulo Theories*

SPIN - *Simple PROMELA INterpreter*

SSA - *Single Static Assignment*

TIC - *Tecnologia da Informação e Comunicação*

VC - *Verification Condition*

VHDL - *VHSIC Hardware Description Language*

VHSIC - *Very High Speed Integrated Circuits*

Capítulo 1

Introdução

Lua é uma linguagem de *script* que foi projetada com a finalidade de expandir aplicações escritas em outras linguagens [1, 2], como NCL [3], C/C++, Java e Ada, entre outras [4]. A linguagem Lua permite, por exemplo, modificar o comportamento de uma aplicação, como em jogos [5], através da descrição de dados por tabelas. Com aplicações que vão desde jogos até programas interativos para a TV digital [6, 7, 8, 9], Lua é utilizada em aplicações críticas [10], que exigem segurança e confiabilidade das informações com garantia no tempo de resposta, e possui estruturas de controle e repetição semelhantes às de linguagens de programação mais conhecidas, como C.

Assim como em outras linguagens de programação, erros podem ocorrer, como *deadlocks*, estouro aritmético e divisão por zero, entre outras violações. Após e durante o desenvolvimento, testes de software devem ser realizados com o objetivo de detectar possíveis erros que podem ocorrer durante a execução do programa. As principais técnicas utilizadas são os testes de unidade, de integração e de sistema (ou funcional). Nos testes de unidade, trechos pequenos do código–fonte são testados para se verificar sua corretude. Nos testes de integração, um conjunto de componentes, que compõem um módulo, são avaliados. Nos testes de sistema, todos os módulos são avaliados, conforme as especificações de requisitos, sem que seja necessário o conhecimento de sua implementação, como no caso dos testes de unidade e de integração. Em qualquer uma das técnicas, é necessário fornecer um conjunto de dados de entrada, de modo a gerar dados de saída que possam ser analisados, com o objetivo de detectar possíveis erros [11].

Apesar das técnicas de engenharia de software empregadas para a realização de testes funcionais serem eficientes, conseguindo detectar em torno de 60% à 90% de erros no código [12],

algumas violações de consistência no código podem passar despercebidas e produzir erros capazes de gerar um alto custo financeiro e perdas de vida. Além disso, muito tempo e esforço são despendidos na fase de testes [13], pois essas técnicas exigem a preparação e a análise dos requisitos a serem testados, além da análise dos resultados dos testes.

Outra técnica utilizada em testes de programas é a verificação por métodos formais [13], que aplicam modelos matemáticos à análise de sistemas complexos, fornecendo uma abordagem eficiente e eficaz para garantir a corretude do programa. Os modelos são baseados em teorias matemáticas, como as Teorias do Módulo de Satisfatibilidade (*Satisfiability Modulo Theories* - SMT). Uma das ferramentas utilizadas na verificação formal de sistemas complexos, em C/C++, é o *Efficient SMT-Based Context-Bounded Model Checker* (ESBMC) [14, 15], cuja arquitetura é baseada nas teorias do módulo de satisfatibilidade. A utilização de ESBMC neste trabalho é devido ao fato de que é uma das ferramentas BMC mais eficientes, tal como indicado nas últimas edições de competição de verificação de software [16, 17, 18]. Outras ferramentas, como CBMC [19] e LLBMC [20], também são baseadas em SMT.

O trabalho é motivado pela necessidade de se poder aplicar os benefícios da verificação de modelos, baseada nas teorias de satisfatibilidade, a programas escritos em Lua. Assim, com a desenvolvimento de uma metodologia que utilize a técnica BMC, será possível detectar violações no código, como o estouro aritmético, que podem ocasionar falhas durante a execução do programa.

1.1 Descrição do Problema

O problema a ser tratado por este trabalho está relacionado à inexistência de uma metodologia que utilize a técnica de verificação de modelos, com o objetivo de garantir a corretude de programas escritos em Lua. Atualmente, as ferramentas disponíveis para testar códigos Lua, como o *plug–in* NCL/Lua [21, 22] para o Ambiente de Desenvolvimento Integrado (IDE, do inglês *Integrated Development Environment*) Eclipse [23], utilizam uma abordagem para a verificação apenas da sintaxe da linguagem. Este tipo de avaliação auxilia os testes de códigos Lua durante o desenvolvimento, mas não detectam violações em tempo de execução.

Os testes para programas Lua em tempo de execução seguem as técnicas tradicionais de engenharia de software para verificação de possíveis erros em código, porém, exigem muito esforço de desenvolvimento. Aplicações interativas para TV digital executam códigos Lua que

podem conter algum tipo erro, como um estouro aritmético, e provocar uma falha capaz de impedir o funcionamento correto de uma programação interativa digital, transmitida por uma emissora de TV [6, 7].

A linguagem Lua possui algumas características que a diferenciam de outras linguagens, como a declaração dinâmica de variáveis, a chamada de funções como tipo de valor e a definição da estrutura *Table* para representar vetores, registros e listas [4]. Assim, tais estruturas precisam de uma tratativa diferenciada, durante a conversão da linguagem Lua para um modelo capaz de ser verificado por algoritmos verificadores, como os solucionadores SMT Z3 [24], ESBMC [14], CBMC [19] e LLBMC [20]. Essa tradução, para uma representação intermediária verificável, é um dos problemas encontrados na definição de uma metodologia de verificação de modelos para a linguagem Lua.

1.2 Objetivos

O objetivo principal deste trabalho é propor uma metodologia de verificação de programas Lua, utilizando as teorias do módulo de satistatibilidade (SMT), que permita que possíveis violações no programa, como *deadlocks*, estouro aritmético e divisão por zero, possam ser detectadas e posteriormente corrigidos pelo desenvolvedor. Esta proposta busca tornar viável a verificação de erros, em programas escritos em Lua.

Os objetivos específicos do trabalho são:

a) Desenvolver uma ferramenta de tradução, que permita converter códigos Lua em ANSI-C.

b) Desenvolver um interpretador para o contraexemplo, gerado pela ferramenta ESBMC, para códigos Lua, conforme a arquitetura mostrada na Figura 1.1 da Seção 1.3.

c) Validar, experimentalmente, a eficácia da verificação de modelos para códigos escritos na linguagem Lua, através de testes padrões utilizando *benchmarks*. Os experimentos medirão o desempenho da ferramenta, comparando com os resultados obtidos a partir dos mesmos *bechmarks* escritos em ANSI-C e verificados diretamente pelo ESBMC. Além disso, a corretude dos resultados obtidos nos experimentos também será observada.

1.3 Descrição da Solução

A metodologia proposta utiliza aspectos teóricos e práticos da verificação de modelos, através da técnica de *Bounded Model Checking* (BMC) [25]. O trabalho ficou concentrado na criação e implementação de um tradutor, que pudesse converter códigos Lua em uma representação intermediária, que por sua vez seria verificada por uma ferramenta BMC. A ferramenta escolhida para a verificação foi o ESBMC, que utiliza o solucionador SMT Z3 [24].

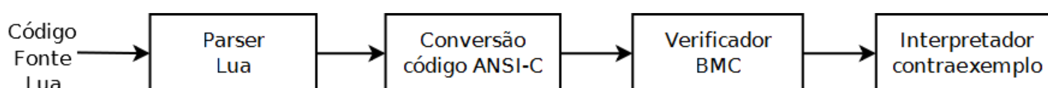


Figura 1.1: Arquitetura da metodologia BMCLua.

Na Figura 1.1, a arquitetura da metodologia BMCLua é apresentada, com a descrição das seguintes etapas:

a) **Analisador Lua** – Foi desenvolvido com o uso da ferramenta ANTLR [26]. A partir do fluxo de entrada de caracteres (*tokens*), ele permite o reconhecimento da estrutura do código-fonte. A base do *parser* é a gramática da linguagem Lua;

b) **Conversão código ANSI-C** – Através de uma interface de saída de texto estruturado, permite gerar código-fonte ANSI-C, a partir do reconhecimento do código-fonte Lua. Essa etapa foi desenvolvida através da ferramenta ANTLR;

c) **Verificador BMC** – É uma ferramenta baseada na técnica de verificação de modelos, que é capaz de realizar a verificação de códigos-fonte ANSI-C gerados pelo tradutor definido na metodologia BMCLua. A ferramenta de verificação utilizada neste trabalho foi o ESBMC, desenvolvida por Cordeiro; Fisher e Marques-Silva [14]. A metodologia BMCLua pode ser modificada, futuramente, para utilizar outros verificadores BMC, cuja linguagem alvo seja o ANSI-C (ex.: CBMC);

d) **Interpretador contraexemplo** – O interpretador, desenvolvido neste trabalho, captura o contraexemplo, gerado pela ferramenta ESBMC, e o traduz em informações sobre o problema encontrado no código Lua.

1.4 Contribuições

Este trabalho apresenta uma abordagem inicial ao problema de validação de códigos Lua, servindo como base para futuras melhorias e possíveis acréscimos à ferramenta BMCLua. Por exemplo, no âmbito da TV digital, este trabalho contribui para a verificação de objetos NCLua, presente em aplicações interativas, que utilizam a API Lua específica para o ambiente do *middleware* Ginga [27]. Com base na literatura disponível, este é um dos primeiros trabalhos que aplica verificação de modelos, baseada na técnica de BMC, em códigos Lua.

O tradutor definido na metodologia proposta converte parte das estruturas da linguagem Lua para ANSI-C, como mostrado na Tabela 1.1. A tradução é realizada a partir da gramática BNF (do inglês, *Backus Naur Form*) [28], que define as regras de sintaxe da linguagem Lua.

Estrutura Lua	Suportada pelo BMCLua
Declaração de variáveis	SIM
Atribuições simples e múltiplas	SIM
Tabelas e estruturas de controle	SIM
Funções	PARCIAL
Biblioteca Padrão	NÃO
Biblioteca NCLua	PARCIAL

Tabela 1.1: Estruturas Lua suportadas pelo tradutor do BMCLua.

Com a metodologia BMCLua é viável, após a tradução do código-fonte Lua para código ANSI-C, a utilização de outros verificadores BMC, além da ferramenta ESBMC, na etapa de verificação.

O interpretador de contraexemplo é outra contribuição deste trabalho, e permite ao desenvolvedor observar os resultados gerados na etapa de verificação, objetivando a análise direta do código-fonte Lua.

1.5 Organização da Dissertação

Este capítulo apresenta uma breve introdução sobre o trabalho, os problemas que motivaram a pesquisa e os objetivos a serem alcançados.

No Capítulo 2, apresenta-se um estudo da técnica de verificação formal e as Teorias do Módulo de Satisfatibilidade (SMT). Uma revisão sobre a ferramenta ESBMC e sua arquitetura é introduzida. Em seguida, um estudo teórico sobre a linguagem Lua e suas estruturas básicas

é fornecido. Fechando esse capítulo, a ferramenta ANTLR é abordada, pois esta foi utilizada para o desenvolvimento do tradutor e também do interpretador da metodologia BMCLua.

No Capítulo 3, um resumo dos trabalhos e ferramentas que utilizam as técnicas de verificação formal é apresentado, abordando especialmente a ferramenta ESBMC. Há também, nesse capítulo, um resumo sobre os trabalhos relacionados à tradução de códigos-fonte Lua para códigos C e verificação de programas Lua.

O Capítulo 4 apresenta a metodologia de verificação de modelos para programas Lua, o desenvolvimento do tradutor de códigos Lua, a verificação do modelo e o desenvolvimento do interpretador do contraexemplo de saída do ESBMC. Na Seção 4.2, os experimentos realizados com padrões de programas Lua (*benchmarks*) e a análise dos resultados obtidos são descritos.

Finalmente, no Capítulo 5, as conclusões e as sugestões para trabalhos futuros são expostas.

Capítulo 2

Fundamentação Teórica

Este capítulo descreve o paradigma da verificação formal, fundamentando os conceitos básicos sobre verificação de modelos. O estudo das teorias do módulo de satisfatibilidade (SMT) e a sua aplicação na verificação de sistemas concorrentes é também apresentado. Em seguida, a sintaxe e as estruturas da linguagem Lua são descritas, mostrando aspectos específicos da linguagem. Ainda na seção que trata da linguagem Lua, a extensão NCLua e a sua aplicação no âmbito da TV digital são abordadas. Por fim, a ferramenta ANTLR é introduzida, focando em aspectos de construção e funcionamento dos analisadores (*lexer* e *parser*) para a tradução de sentenças, tal como a conversão de instruções de uma linguagem de programação para outra.

2.1 Verificação Formal

Nos dias de hoje, sistemas de tecnologia da informação e comunicação (TIC) fazem parte do cotidiano da vida humana e estão presentes em sistemas embarcados (ex.: *smart cards*, telefones móveis, *smart TVs*) e em sistemas críticos, como os sistemas de controle de vôo de aeronaves. Estima-se que 20% dos custos de desenvolvimento em transportes modernos, como carros, trens de alta-velocidade e aviões são dedicados ao desenvolvimento de programas [13].

Atualmente, observa-se amplo uso de sistemas TIC (hardware e software) em aplicações onde falhas são inaceitáveis, como sistemas de controle de tráfego aéreo e instrumentação médica. Por exemplo, em 1996, o foguete Ariane 5 explodiu após 4 segundos do lançamento, devido a uma falha causada por um erro no software de cálculo do movimento do foguete.

Durante o lançamento, uma exceção ocorreu quando um número de ponto flutuante de 64 bits foi convertido em um inteiro de 16 bits (*typecast*) [29]. A falha poderia ter sido evitada, se houvesse uma tratador de exceções no código. Exemplos como este são comuns de acontecer e simplesmente não se pode desligar o sistema, para restaurar a segurança, quando ocorrer um mal funcionamento.

Sistemas embarcados são largamente utilizados em aplicações de tempo real [30], com a finalidade de monitorar e controlar sistemas de engenharia, como reatores nucleares, e que possuem a característica principal de responder à estímulos externos, em um tempo finito e específico. Observa-se também que a complexidade aumenta quando vários sistemas TIC são interligados como componentes, fazendo com que o número de erros cresça exponencialmente com o número de interações de componentes. É importante, portanto, fornecer sistemas TIC sem erros, o que tem sido um desafio e muitas vezes complexo [13].

Existem técnicas que permitem verificar possíveis problemas em programas. As abordagens tradicionais mais conhecidas são: revisão por pares, testes e simulação. Enquanto a revisão por pares é estática, a técnica de teste de software é dinâmica, ou seja, é realizada sobre a execução do programa desenvolvido [13]. Nessa técnica, parte do programa é analisado para se verificar sua corretude. O teste ocorre quando um conjunto de dados de entrada é fornecido, que foi definido a partir de um documento de especificação de requisitos de projeto, para se obter um conjunto de dados de saída para análise. O nível de automação é parcial, pois a análise ainda é realizada por um ser humano. Na prática, não é possível realizar um teste exaustivo, que contemple todos os caminhos de execução do sistema, mas somente é possível realizar testes com um subconjunto de dados de entrada [13]. Através dessa técnica, é possível apenas detectar a presença de erros, mas não a sua ausência.

A simulação é uma técnica muito utilizada para verificar erros em hardware [13]. Nessa técnica, modelos baseados na especificação do projeto são construídos e simulados. Eles permitem descrever o comportamento do hardware, sendo necessário um conjunto de estímulos que definam os caminhos de execução do circuito. A presença de erros é determinada pela comparação entre a saída do simulador e a descrição presente nas especificações.

Apesar da eficiência alcançada com as técnicas descritas acima, a fase de testes em sistemas complexos de software e hardware podem demandar um tempo de execução em torno de até 80% do total do projeto [13]. Com o intuito de reduzir a etapa de testes do projeto e aumentar a confiabilidade nos requisitos do sistema, pesquisas têm sido realizadas para se desenvolver

métodos formais que permitam automatizar o processo de verificação e aumentar a eficácia na determinação de erros em software [13]. Nos métodos formais, modelos matemáticos são construídos e analisados com a meta de determinar a corretude do sistema, que por definição consiste em garantir a confiabilidade das informações geradas pelo sistema, com o máximo de rigor matemático. Entre os métodos formais em destaque, pode-se citar a verificação dedutiva e a verificação de modelos.

O método de verificação dedutiva utiliza axiomas e regras de prova com o intuito de determinar a corretude do sistema [29]. Esse método consome um tempo considerável do processo de verificação, pois exige um conhecimento matemático na análise dedutiva de axiomas. Por isso, esse método é utilizado na verificação de sistemas altamente sensíveis, como protocolos de segurança. A verificação dedutiva possui a vantagem de verificar, de forma automática, sistemas de estados finitos.

Já o método de verificação de modelos utiliza uma estrutura de transição de estados, onde são especificadas as propriedades e as fórmulas matemáticas que descrevem a satisfatibilidade de cada estado do modelo. Esse método complementa outras técnicas, como a técnica de revisão por pares e a verificação dedutiva, permitindo automatizar a verificação de sistemas críticos e complexos.

2.1.1 Verificação de Modelos

A verificação de modelos (*model checking*) é uma técnica formal e automática que permite realizar a verificação de sistemas críticos e concorrentes de estados finitos [29]. É um método formal que permite uma busca exaustiva no espaço de estados do sistema com o intuito de determinar se o modelo satisfaz uma certa propriedade. O número de estados verificados por uma ferramenta atual de verificação de modelos pode chegar até 10^{476} estados possíveis, significando uma verificação de todos os prováveis cenários. Assim, mesmo erros não encontrados por outras técnicas, podem ser descobertos pela verificação de modelos [13].

O modelo do sistema é gerado automaticamente a partir de uma descrição em alguma linguagem de programação, como C/C++ [31, 32], Java [33] ou VHDL (do inglês, *VHSIC Hardware Description Language*) [34] usado em circuitos integrados VHSIC (do inglês, *Very High Speed Integrated Circuits*). Nessa técnica, os estados do sistema são verificados, para se determinar se as propriedades especificadas satisfazem o modelo. Caso seja encontrada uma

violação de alguma propriedade, a ferramenta de verificação fornece um contraexemplo que descreve o caminho de execução, desde o estado inicial do sistema até o estado onde a violação ocorreu. A partir do contraexemplo informado, o projetista pode analisar o código e corrigir o sistema, para evitar a violação, e posteriormente revalidar o projeto, aplicando novamente a verificação de modelos.

Em geral, a técnica da verificação de modelos pode ser descrita em três etapas: modelagem, especificação e verificação [29]. Essa abordagem é ilustrada na Figura 2.1.

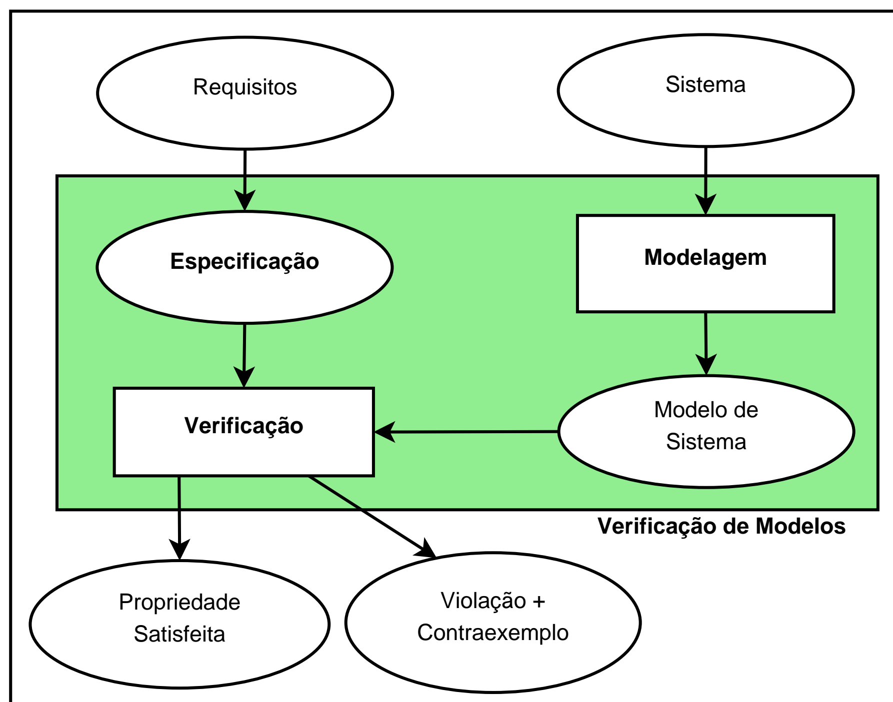


Figura 2.1: Abordagem da técnica de Verificação de Modelos.
Fonte: BAIER (2008, p. 8) – com adaptações.

A etapa de modelagem converte o sistema para o formalismo de um modelo matemático, que é aceito por uma ferramenta de verificação de modelos. Essa tarefa pode ser uma simples compilação ou até mesmo uma abstração do projeto, de forma a eliminar detalhes não relevantes para a verificação, como limites de tempo e memória.

Durante a especificação, as propriedades que deverão satisfazer o modelo do sistema são declaradas. Essa especificação é definida sobre uma lógica formal, sendo mais comum o uso da lógica temporal [13, 29]. Quando o resultado da verificação de modelos informa que o sistema satisfaz a especificação, então existe completude do sistema, ou seja, todas as propriedades do sistema são satisfeitos pelo modelo.

A etapa seguinte é a verificação, onde é realizada, de forma automática, a validação das propriedades especificadas para o modelo [29]. É possível, em uma determinada profundidade, verificar se certa propriedade é satisfeita ou não. Caso a propriedade não seja válida, segundo o modelo, um contraexemplo é fornecido, juntamente com a lista de validações realizadas e a localização da violação. A análise do resultado, em geral, é manual e realizada pelo usuário, que poderá rastrear e corrigir o erro no código, para depois realizar uma nova verificação.

O contraexemplo pode mostrar que o erro ocorreu na modelagem ou na especificação incorreta do sistema. Nesse último, o resultado é denominado falso positivo e o usuário deverá fazer ajustes no modelo, para que a etapa de verificação possa ser realizada com sucesso [29].

2.1.2 Modelagem de Sistemas

A modelagem de sistemas reativos, ou seja, sistemas que interagem com o ambiente, envolve a captura das propriedades que irão estabelecer a corretude do modelo [29]. A primeira característica capturada é seu estado, que é uma descrição instantânea, no tempo, do comportamento do sistema a partir das leituras de suas variáveis. O par de estados anterior e posterior determina a transição de estados no sistema, e o conjunto de transições define um modelo a ser especificado.

Um gráfico de transição de estados pode ser representado por uma estrutura Kripke [29], que é composta por um conjunto de estados, um conjunto de transições entre estados e uma função, que rótula cada estado com um conjunto de propriedades, que são verdadeiras neste estado.

Uma estrutura Kripke pode ser definida sobre um conjunto de proposições, como uma tupla $M = (S,\ S_0,\ R,\ L)$, onde

1. S é um conjunto finito de estados;

2. S_0 é um conjunto inicial de estados, onde $S_0 \subseteq S$;

3. R é a relação de transição, definida sobre um conjunto de pares ordenado de estados $R(s,\ s')$ contido em S ($R \subseteq S \times S$);

4. L é a função que rótula cada estado com um conjunto de proposições verdadeiras para o estado.

A avaliação da função L pode ser realizada por uma lógica proposicional (ver Seção 2.1.3), e é composta por operadores conectivos (*e* $\wedge$, *ou* $\vee$, *negação* $\neg$, implicação $\rightarrow$) e quantificadores de caminho (*universal* $\forall$ e *existencial* $\exists$).

Um estado s pode ser descrito por variáveis pertencentes a um conjunto $V = v_1, ..., v_n$, cujos valores estão contidos no conjunto finito D. Dessa forma, pode-se definir que a função de avaliação L associa cada valor de D a cada valor de V, de modo que a proposição $v = d$ verdadeira para o estado s. Assim, $L(s)$ também pode ser definido como o subconjunto de proposições atômicas e verdadeiras em s.

Dado um conjunto de variáveis do sistema $V = \{v_1,\ v_2,\ v_3\}$ e a avaliação $\langle$ v$_1 \leftarrow 2$, $v_2 \leftarrow 3$, $v_3 \leftarrow 5$ $\rangle$, então pode-se derivar uma fórmula de primeira ordem $(v_1 = 2) \wedge (v_2 = 3) \wedge (v_3 = 5)$, que seja verdadeira para algum estado do modelo.

2.1.3 Lógica Proposicional

A lógica proposicional é uma linguagem capaz de expressar sentenças naturais [35], como a sentença "se p e não q, então r". Tais sentenças são declarativas e podem ser expressas em fórmulas de lógica proposicional.

A lógica proposicional é baseada em proposições ou declarativas atômicas, ou seja, premissas aceitas como verdadeiras e indivisíveis e operadores conectivos, como os operadores de junção *e* ($\wedge$) e *ou* ($\vee$), de *negação* ($\neg$) e *implicação* ($\rightarrow$). A sintaxe utilizada na lógica proposicional é definida pela notação proposicional 2.1.

$$\phi ::= p \mid (\phi) \mid \neg\phi \mid \phi \wedge \phi \mid \quad , \tag{2.1}$$

onde p é uma proposição atômica, que pode ser uma variável ou um valor booleano (*true* ou *false*), e ϕ é uma fórmula proposicional.

Considerando a sentença anterior, "*se p e não q então r*", a tradução para a lógica proposicional pode ser expressa como

$$p \wedge \neg q \rightarrow r$$

As fórmulas definidas pela lógica proposicional são utilizados, por exemplo, em solucionadores SAT (ver Seção 2.2) para verificar a corretude de sistemas, que é um dos objetivos deste trabalho.

2.1.4 Lógica Temporal

A lógica temporal (LT) é uma descrição matemática utilizada para se formular as propriedades de sistemas reativos, que são sistemas caracterizados por um conjunto de estados e transições [36]. Ela é utilizada durante a tarefa de especificação das propriedades, sendo a lógica mais utilizada na verificação de modelos de sistemas concorrentes.

O termo *temporal* da LT está associado ao comportamento do sistema ao longo do tempo, em sentido abstrato [13]. A ordem relativa de eventos é especificada, mas há relação com qualquer instante de tempo do evento.

Existem vários operadores, baseados na lógica proposicional, que permitem especificar propriedades. Por exemplo, pode-se especificar que dois eventos $e1$ e $e2$ nunca poderão ocorrer ao mesmo tempo, no futuro, simplesmente escrevendo a fórmula $\mathbf{AF}\neg(e1 \wedge e2)$.

A lógica temporal pode ser classificada como linear ou ramificada [13], e são usadas estruturas Kripke para se especificar as transições de estados através de gráficos.

Na lógica temporal linear (*Linear Temporal Logic* - LTL), as fórmulas são avaliadas em todos os caminhos de computação [13]. As fórmulas LTL são compostas por proposições atômicas $p \in PA$, sendo PA um conjunto de transição de estados, conectores booleanos como conjunção ($\wedge$) e negação ($\neg$) e dois modais temporais $\mathbf{X}$ (*neXt*) e $\mathbf{U}$ (*Until*). O operador modal $\mathbf{X}$ descreve que o estado atual é verdadeiro, se no próximo estado, a propriedade definida por uma fórmula LTL (φ), também é verdadeira. O modal $\mathbf{U}$, em lógica LTL, considera que o estado atual é verdadeiro para a fórmula $\varphi_1 \bigcup \varphi_2$, se φ_1 e φ_2 são verdadeiros, em todos os estados, até um momento futuro.

As fórmulas LTL (φ) são construídas conforme definido pela regra de sintaxe 2.2

$$\varphi ::= true \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\varphi \mid \varphi_1 \; \mathbf{U} \; \varphi_2 \tag{2.2}$$

As fórmulas definidas, através da lógica de árvore de computação (CTL, do inglês *Computation Tree Logic*), descrevem as propriedades em árvores computacionais [29] e são compostas por quantificadores de caminho e operadores modais. Os quantificadores são dois: universal (**A**), para indicar todos os caminhos computacionais, e existencial (**E**), para indicar que existe algum caminho computacional. Os operadores modais são:

X (neXt) − a propriedade detém no próximo estado no caminho;

F (Future) – a propriedade se mantém em algum estado no caminho;

G (Globally) – a propriedade se mantém em todos os estados no caminho;

U (Until) – é um operador binário, que mantém a propriedade do primeiro estado (φ_1), enquanto o segundo estado (φ_2) mantém a propriedade.

As fórmulas CTL podem ser descritas conforme a regra de sintaxe 2.3

$$\begin{array}{rl}\varphi ::= & true \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid AX\varphi \mid EX\varphi \mid AF\varphi \mid EF\varphi \mid \\ & AG\varphi \mid EG\varphi \mid A[\ \varphi_1 \ \mathbf{U} \ \varphi_2] \mid E[\ \varphi_1 \ \mathbf{U} \ \varphi_2]\end{array} \quad . \tag{2.3}$$

Na Figura 2.2, pode-se observar a representação da semântica CTL, em uma árvore computacional.

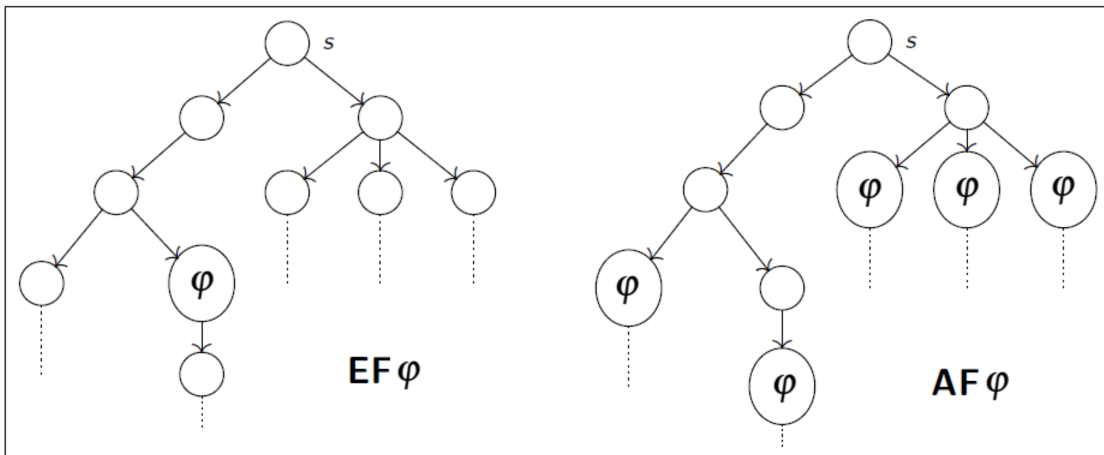


Figura 2.2: Árvore computacional a partir de fórmulas CTL.

2.2 Satisfatibilidade

A partir de uma fórmula proposicional (ϕ) é possível verificar se existem valores que satisfazem as proposições definidas pela lógica proposicional. O problema da satisfatibilidade proposicional ou booleana (SAT) é um processo decisório que retorna uma resposta qualitativa, ou seja, sim ou não, a partir da verificação da fórmula. Formalmente, pode-se definir que uma fórmula em lógica proposicional ϕ é satisfazível se existirem atribuições, de valores verdadeiros, para as proposições atômicas, de tal forma que ϕ seja verdadeiro. Como exemplo, verificar se a fórmula φ (2.4), mostrada abaixo, é satisfazível.

$$\varphi ::= (\phi_1 \vee \phi_2) \wedge \phi_1 \wedge \neg\phi_2 \tag{2.4}$$

Para realizar a verificação de satisfatibilidade da fórmula φ, a tabela verdade (Tabela 2.1) foi montada, onde cada par de valores booleanos para ϕ_1 e ϕ_2, correspondem a um estado do modelo.

Estado	ϕ_1	ϕ_2	φ
1	F	F	F
2	F	V	F
3	V	F	V
4	V	V	F

Tabela 2.1: Tabela verdade para a fórmula φ.

Na Tabela 2.1, observa-se que, para os estados possíveis, apenas no estado 3 a fórmula φ é satisfazível, ou seja, verdadeira. Nos outros estados, a fórmula φ é dita não satisfazível.

Os solucionadores SAT são algoritmos que tomam, como entrada, uma fórmula proposicional (ϕ) e verificam se esta é satisfazível (*sat*) ou não satisfazível (*unsat*). A fórmula é satisfazível quando o solucionador SAT determina valores que tornam todas as proposições verdadeiras, ou seja, que tornem a fórmula verdadeira.

2.2.1 Teorias do Módulo de Satisfatibilidade (SMT)

Nas últimas décadas, mais solucionadores SAT têm sido utilizados na verificação formal de hardware e software. A determinação da satisfatibilidade de expressões lógicas, de primeira ordem, tem sido fundamental para a determinação da corretude de um sistema. No entanto, muitas aplicações precisam determinar a satisfatibilidade que não seja de primeira ordem, mas que seja definida em alguma teoria de fundamentação, cuja interpretação utiliza certos símbolos de predicados e funções [37]. Por exemplo, a fórmula 2.5 utiliza símbolos da aritmética inteira, que não podem ser interpretados apenas pela lógica proposicional.

$$\varphi ::= x < y \wedge \neg(x < y + 0) \tag{2.5}$$

Para que um solucionador SAT possa interpretar tal fórmula, com o objetivo de verificar se é satisfazível ou não, este precisa considerar interpretações compatíveis com uma ou mais teorias de fundamentação. De forma geral, as interpretações devem sempre corresponder a uma teoria de fundamentação (T). O campo de pesquisa relacionado à satisfatibilidade de fórmulas, que tenham respeito a alguma teoria de fundamentação (T), é chamado de Teorias do Módulo de Satisfatibilidade (*Satisfiability Modulo Theories* - SMT). O SMT integra várias teorias, como

aritmética linear e não-linear, funções não interpretadas, *arrays*, vetores de bits e outras teorias de primeira ordem.

Uma teoria é um conjunto de sentenças sobre uma assinatura $\sum$-teoria [38], que representa o conjunto de teorias SMT. Dada uma teoria T, afirma-se que a fórmula ϕ pertence ao módulo T-satisfatibilidade se $T \cup \{\phi\}$ for satisfazível. Além disso, levando-se em consideração a mesma teoria T, uma classe de estruturas e uma fórmula ϕ sem quantificadores, diz-se que ϕ é safistazível, no módulo T, se existir uma estrutura M em T, tal que $M \models \phi$.

A notação padrão para se descrever a sintaxe SMT está descrita na Figura 2.3 [15]. Nessa notação, *Fmt* denota uma expressão booleana, *Trm* são os termos construídos sobre inteiros, reais e vetor de *bit* e *op* identifica os operadores binários. Os conectivos lógicos anotados são: conjunção ($\wedge$), disjunção ($\vee$), ou exclusivo ($\oplus$), implicação ($\Rightarrow$) e equivalência ($\Leftrightarrow$). Os operadores relacionais, identificados por *rel*, e os operadores não lineares ($*, /, rem$) são usados na interpretação de argumentos de vetores de bits, inteiros e reais. Os operadores de deslocamento (« e »), e (&), ou (|), ou exclusivo ($\oplus$), complemento ($\sim$), concatenação (@), $Extract(Trm, i)$, $SignExt(Trm, k)$ e $ZeroExt(Trm, i)$ são utilizados na manipulação de vetores de bits. O operador condicional *ite* avalia uma fórmula booleana *Fml*, selecionando o primeiro argumento se a fórmula for verdadeira; caso contrário, seleciona o segundo argumento. O operador *select* é usado para selecionar o valor armazenado na posição *i* do vetor, e o operador *store* substitui o valor da posição *i* pelo novo valor *v*, dentro do vetor.

$$
\begin{array}{rcl}
Fml & ::= & Fml\ con\ Fml \mid \neg Fml \mid Atom \\
con & ::= & \wedge \mid \vee \mid \oplus \mid \Rightarrow \mid \Leftrightarrow \\
Atom & ::= & Trm\ rel\ Trm \mid Var \mid verdadeiro \mid falso \\
rel & ::= & < \mid \leq \mid > \mid \geq \mid = \mid \neq \\
Trm & ::= & Trm\ op\ Trm \mid \sim Trm \mid Var \mid Const \\
 & & \mid select(Trm, i) \mid store(Trm, i, v) \\
 & & \mid Extract(Trm, i, j) \mid SignExt(Trm, k) \mid ZeroExt(Trm, k) \\
 & & \mid ite(Fml, Trm, Trm) \\
op & ::= & + \mid - \mid * \mid / \mid rem \mid \text{«} \mid \text{»} \mid \& \mid | \mid \oplus \mid @
\end{array}
$$

Figura 2.3: Notação Padrão.

Existem vários solucionadores SMT construídos sobre uma ou várias teorias, para verificar a satisfatibilidade de uma fórmula específica. Como exemplo, pode-se citar o solucionador Z3 [24], da Microsoft.

2.3 Linguagem Lua

Lua é uma linguagem de programação popular no desenvolvimento de jogos e aplicações para TV digital [2]. Na realidade, trata-se de uma linguagem de extensão, que pode ser utilizada por outras linguagens, como C/C++ [39] e NCL [27, 3].

A linguagem de programação Lua é interpretada, sendo que o próprio interpretador foi desenvolvido em ANSI-C, o que a torna compacta, permitindo dessa forma que o tempo de processamento de execução do código seja muito reduzido. Essas características fornecem ao Lua a capacidade de executar em uma vasta gama de dispositivos que vão desde pequenos dispositivos até servidores de rede de alto desempenho [1].

Comparando com outras linguagens de programação, como C++ e Java, o desenvolvimento de códigos Lua demanda um esforço menor devido à facilidade de codificação que a linguagem Lua oferece. Isso a torna muito atrativa para aplicações interativas voltadas à TV digital, que exigem facilidade de programação e resposta em tempo real.

2.3.1 Sintaxe e Estruturas

Para o desenvolvimento deste trabalho, foi realizado um estudo básico sobre a estrutura e a sintaxe da linguagem de programação Lua. Durante esse estudo, constatou-se que, com o intuito de facilitar a codificação com esta linguagem, retiraram-se elementos que tornam outras linguagens mais robustas, como a obrigatoriedade de se definir o tipo de dado da variável durante a sua declaração. No trecho de código Lua da Figura 2.4, 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
```

Figura 2.4: Trecho de código Lua.

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 2.4). 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 a diferentes variáveis, simultaneamente, como pode ser visto na primeira linha do código mostrado na Figura 2.4.

Os tipos de dados existentes na linguagem Lua são:

a) *nil*: significa ausência de valor;

b) *boolean*: valor lógico *true* ou *false*;

c) *number*: apenas tipos numéricos;

d) *string*: cadeia de caracteres;

e) *table*: são vetores associativos;

f) *function*: funções também são definidos como tipos de valores.

Comparando a linguagem Lua a outras linguagens de programação, como C e Java, percebe-se que existem equivalências na sintaxe das estruturas de decisão (*if*) e repetição (*for* e *while*). Além disso, existe uma diferença importante com relação às estruturas de tipo vetor, comum a todas as linguagens de programação. Em Lua, um vetor é um tipo denominado *table*, cujos índices são inteiros que começam em 1, como ilustrado nas linhas 1 a 4 do código mostrado na Figura 2.5. Entretanto, é possível fazer com que o inicio de um vetor comece em 0, através da sintaxe ilustrada na Figura 2.5, 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])
```

Figura 2.5: Declaração de vetor 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 vetores, estruturas de registro. Na Figura 2.6, é possível visualizar uma comparação entre uma estrutura de dados em ANSI-C (*struct*) (2.6a) e o seu código equivalente em Lua (2.6b).

```
1 struct{
2   int dia;
3   int mes;
4   int ano;
5 } data;
6
7 data.dia = 28;
8 data.mes = 2;
9 data.ano = 2013;
```

(a) Código em ANSI-C.

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

(b) Código em Lua.

Figura 2.6: Estrutura de registros nas linguagens ANSI-C e Lua.

Outra estrutura importante na linguagem Lua é a *function*. 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 em um campo de uma *table*, dependendo do que foi utilizado, e ser então 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.

Em Lua, funções podem ser criadas localmente, dentro de outras funções, e depois retornadas ou armazenadas em uma tabela.

As co-rotinas em Lua são funções que executam tarefas concorrentes, de forma similar a *threads*. Sendo funções, as co-rotinas também são valores definidos em variáveis declaradas.

2.3.2 Bibliotecas

A linguagem Lua disponibiliza uma biblioteca–padrão que possui várias funções úteis, como acesso a arquivos externos de entrada e saída [27]. As bibliotecas–padrão são [27, 4]:

a) biblioteca básica: oferece funções essenciais à linguagem Lua, como a função *assert*, utilizada para produzir um erro quando o argumento for falso ou nulo, a função *loadfile*, usada para carregar na memória um trecho de um arquivo, ou as funções que operam sobre co-rotinas;

b) biblioteca de pacotes: oferece funções que permitem a construção de aplicações Lua de forma modular, através da função *module*, que permite criar um módulo, e da função *require*, que carrega o módulo criado;

c) biblioteca *string*: oferece funções que permitem manipular uma cadeira de caracteres;

d) biblioteca de tabelas: oferece funções que permitem manipular vetores e listas, que são declaradas com o tipo *table*;

e) biblioteca matemática: oferece diversas funções matemáticas, como as trigonométricas (sin, cos, tan), exponenciais e logarítmicas (exp, log), de arredondamento (*floor*, *ceil*), para gerar números pseudo-aleatórios (*random*, *randomseed*), assim como a variável *pi*;

f) biblioteca de entrada e saída (E/S): oferece um conjunto de funções para manipular arquivos externos, através do descritor de arquivo *io*, do tipo *table*;

g) biblioteca do sistema operacional: Utilizando a *table os*, oferece funções, como a função *clock* para retornar o tempo da CPU, ou as funções *date* e *time*, que retornam a data e a hora correntes do sistema;

h) biblioteca de depuração: oferece funções, através da *table debug*, que permite construir uma interface de depuração de programas Lua.

Além da biblioteca–padrão, a linguagem Lua disponibiliza a biblioteca NCLua, que é uma biblioteca de uso específico para o ambiente da TV digital [3]. Esta é integrada à linguagem de contexto NCL através de funcionalidades que permitem, ao objeto NCLua, interagir com o documento NCL, como em resposta as teclas do controle remoto. A biblioteca NCLua disponibiliza os seguintes módulos obrigatórios [27]:

a) módulo *ncledit*: permite que um documento NCL seja alterado modificando a forma de sua apresentação (vídeo, áudio, imagem, objeto Lua);

b) módulo *canvas*: permite manipular objetos gráficos na região da tela de TV;

c) módulo *event*: permite a comunicação do objeto NCLua com o documento de apresentação NCL e interfaces externas (controle remoto, dispositivos via *bluetooth*);

d) módulo *settings*: permite acesso às configurações do documento de apresentação NCL;

e) módulo *persistent*: cria uma tabela de variáveis de ambiente e permite exportar para outros objetos NCLua em execução.

2.4 ANTLR

O *ANother Tool for Language Recognition* (ANTLR) [26] é uma ferramenta que permite gerar analisadores léxicos e de sintaxe, possibilitando automatizar a construção de reconhecedores de linguagens. ANTLR é uma ferramenta utilizada para a construção de tradutores e interpretadores de domínio específico.

A ferramenta ANTLR 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, como as instruções de um programa escrito em uma linguagem de programação. A partir da gramática, o ANTLR pode gerar automaticamente classes escritas na linguagem Java que possuem as funcionalidades do analisador léxico (*lexer*) e também do sintático (*parser*).

A Figura 2.7 mostra um exemplo de gramática construída sobre a sintaxe da linguagem alvo, que, no caso deste trabalho, é a linguagem Lua.

```
1  grammar T;
2
3  r : ID '=' INT ';'
4      {System.out.println("int " + $ID.text + "=" + $INT.text + ";")
           ;};
5
6  ID : 'a'..'z'+;
7  INT : '0'..'9'+;
8
9  //ignora espaços em branco
10 WS : (' ' | '\t' | '\n' | '\r') + {$channel=HIDDEN;};
```

Figura 2.7: Exemplo da gramática no ANTLR.

2.4.1 Tradução

O funcionamento básico de tradução, ilustrado na Figura 2.8, é realizado em duas etapas básicas: a análise léxica e a análise sintática.

Na análise léxica, através da classe *lexer* gerada pelo ANTLR, um conjunto de caracteres de entrada (ex.: código-fonte em Lua) produz na saída uma sequência de símbolos, chamados *tokens*. Os *tokens* são segmentos de texto padrões que se repetem em um texto e possuem um significado comum, como o símbolo da barra (/), que separa dia, mês e ano de uma data (ex.: 01/01/2015).

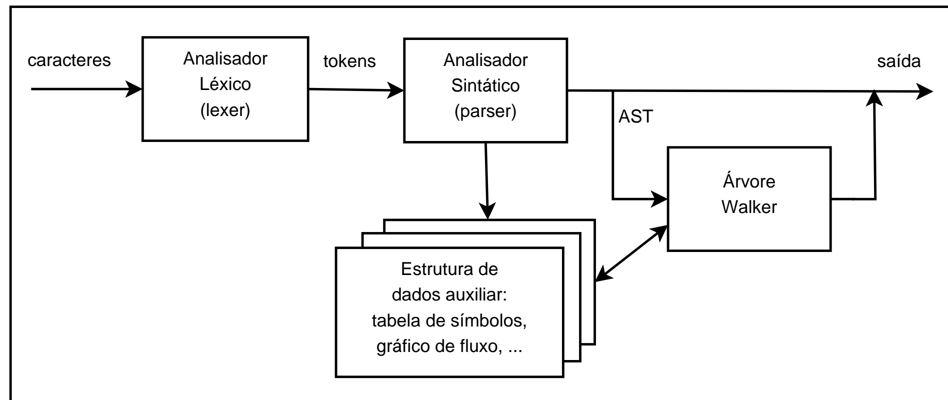


Figura 2.8: Fluxo básico de tradução no ANTLR.

Na fase de 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. É possível também gerar uma árvore AST (*Abstract Symbol Tree*), com a qual a análise e a manipulação de instruções de linguagem são realizadas, através da técnica de caminhamento em profundidade (DFS - *Depth-First Search*) [40].

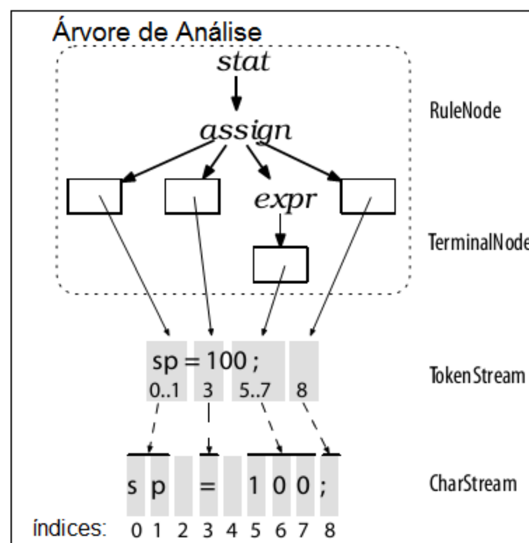


Figura 2.9: Classes geradas a partir da AST.
Fonte: [26]

Para cada etapa descrita acima, o ANTLR utiliza outras classes Java, associadas às classes de analisadores (*lexer* e *parser*) geradas. Considerando-se a instrução *“sp = 100;”*, na Figura 2.9 acima, é possível observar as classes *RuleNode*, *TerminalNode*, *CharStream* e *TokenStream*. As classes *RuleNode* e *TerminalNode* correspondem, respectivamente, aos nós raiz e folha, de uma sub-árvore da AST. A classe *RuleNode* possui os métodos *getChild()* e

getParent() para acessar, respectivamente, os nós pai e filho da sub-árvore. A classe *TokenStream* conecta os analisadores léxico (*lexer*) e sintático (*parser*), através de uma fila encadeada (*pipeline*). Na Figura 2.9, cada nó folha da árvore é um *token* no fluxo de símbolos de entrada. A classe *CharStream* corresponde a uma estrutura de caracteres, associando um índice a cada *token* do fluxo de entrada.

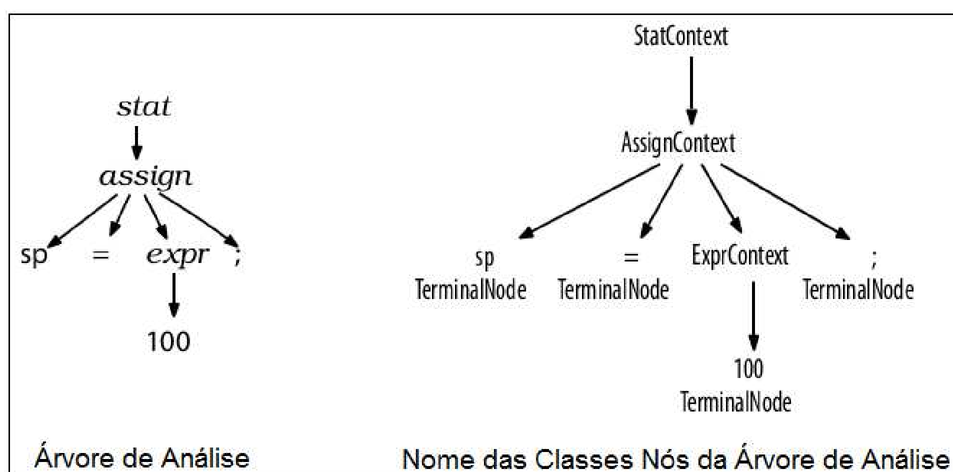


Figura 2.10: Classes de contexto da árvore de análise.
Fonte: [26]

Na Figura 2.10, as classes *StatContext*, *AssignContext* e *ExprContext* são mostradas. Elas são geradas pelo ANTLR, para cada nó da árvore correspondente a cada objeto com função específica, na AST. Essas classes definem os objetos de contexto que registram tudo o que é definido sobre o reconhecimento de uma frase, por uma regra da gramática.

2.4.2 Mecanismos de Busca

A ferramenta ANTLR facilita o desenvolvimento de um tradutor que possa ler o código-fonte de uma linguagem de origem, como de programas escritos em linguagem Lua, e converter para outra linguagem alvo [41], como a linguagem de programação ANSI-C. A tradução de trechos de códigos é realizada pelo ANTLR, através de um mecanismo de busca, que é capaz de responder à eventos desencadeados pelo caminhamento em profundidade, realizado na AST. O ANTLR fornece suporte para dois mecanismos de busca em árvores em tempo de execução: *listener* e *visitor*. O mecanismo de busca *listener* utiliza os métodos *enterContext* e *endContext* das classes de contexto, para cada nó da árvore. Para percorrer a AST e acionar cada evento de um *listener*, o ANTLR fornece a classe *ParseTreeWalker*, conhecida como *walker*. O método

endContext somente é executado quando todos os filhos do nó tiverem sidos percorridos pelo algoritmo DFS.

O mecanismo *listener* executa todos os métodos *enterContext* e *endContext*, de todas as classes da árvore e de forma automática, como é possível observar na Figura 2.11. Assim, não existe controle sobre os métodos (ou eventos) que devem ser executados, o que exige um esforço maior na implementação do tradutor.

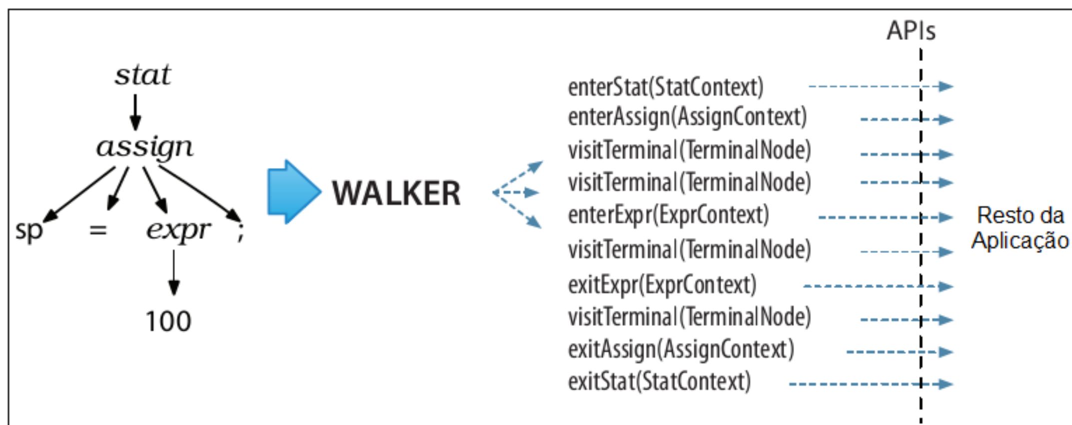


Figura 2.11: Exemplo de busca com o *listener*.
Fonte: [26]

O mecanismo de busca *visitor*, de modo diferente do *listener*, permite controlar os eventos em um caminhamento pela árvore, definindo as regras da gramática que serão visitadas pela DFS. A Figura 2.12 mostra que somente os métodos *visit* são executados. As linhas pontilhadas mostram a busca em profundidade da árvore de análise e a sequência de chamadas entre os métodos *visitor*.

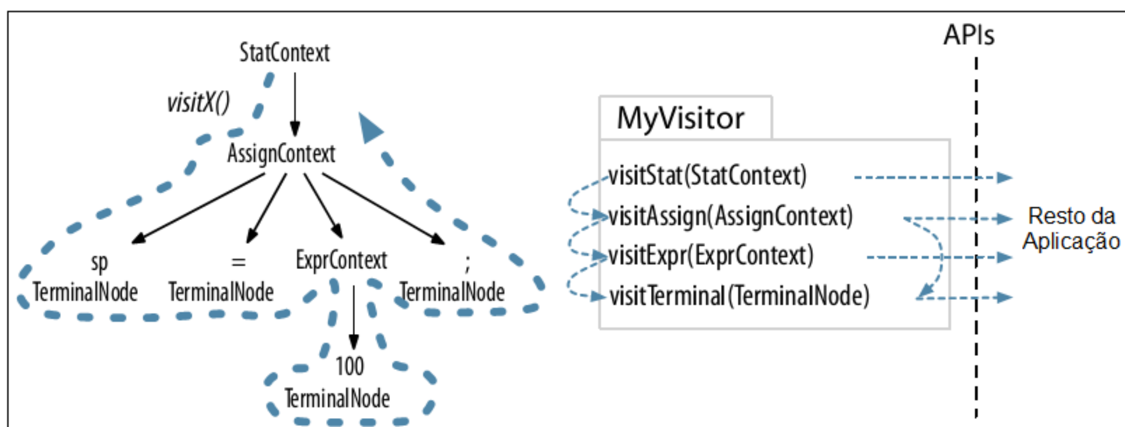


Figura 2.12: Exemplo de busca com o *visitor*.
Fonte: [26]

Com os mecanismos de busca fornecidos pelo ANTLR, através da execução de cada método correspondente, é possível gerar um texto estruturado de saída, através do método *println* da classe *System.out*, da linguagem Java.

2.5 Resumo

Neste capítulo, a fundamentação teórica, necessária ao desenvolvimento deste trabalho foi descrita . Na Seção 2.1, foram apresentados conceitos básicos sobre verificação de modelos e lógicas proposicional e temporal. Na Seção 2.2, foi apresentado o problema da satisfatibilidade, que determinada a verificação de uma fórmula proposicional, através de um processo decisório qualitativo. Dentro dos estudos relacionados à verificação formal, estudo sobre as teorias do módulo de satisfatibilidade (SMT) foi apresentado na Seção 2.2.1, que demonstra a formulação e verificação de propriedades, a partir alguma teoria de fundamentação. Esses estudos foram necessários para definir o desenvolvimento do módulo de verificação da ferramenta BMCLua, que é baseado em solucionadores SMT.

Na Seção 2.3, foram apresentados os aspectos específicos da sintaxe e estruturas da linguagem de programação Lua, que são diferentes de outras linguagens mais tradicionais, como C/C++. Nesta seção, também foram apresentadas as bibliotecas padrão e NCLua, que possuem várias funções úteis para o desenvolvimento de aplicações Lua. As funções da biblioteca NCLua são de uso específico em aplicações interativas, no âmbito da TV digital. O estudo da linguagem Lua permite entender as estruturas compiladas pelo módulo tradutor, definido na metodologia BMCLua.

Finalmente, na Seção 2.4, foi realizado um estudo sobre a ferramenta ANTLR, focando a construção da gramática Lua e os aspectos de construção e funcionamento dos analisadores *lexer* e *parser*. Os analisadores, construídos a partir da gramática Lua, são utilizados para a realização das tarefas de reconhecimento e análise de sentenças, da linguagem Lua. O estudo do ANTLR permite facilitar e formalizar o desenvolvimento do módulo tradutor da ferramenta BMCLua. A simplificação no desenvolvimento do tradutor foi possível através dos mecanismos de busca oferecidos pelo ANTLR.

Capítulo 3

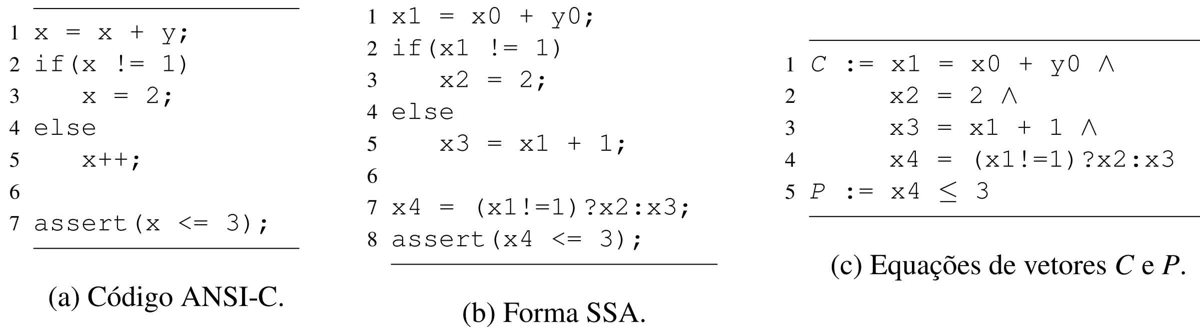
Trabalhos Relacionados

As ferramentas BMC, baseadas nas Teorias dos Módulos de Satisfatibilidade (SMT) [37, 38], são cada vez mais necessárias à verificação de software. Essas ferramentas utilizam modelos bem definidos, capazes de detectar violações como divisão por zero e estouro aritmético, permitindo alta confiabilidade nos testes em projeto de software e hardware. Neste capítulo são apresentadas as ferramentas CBMC, LLBMC e ESBMC, utilizadas para a verificação de códigos escritos na linguagem C/C++. Em seguida é descrita o Java *PathFinder* (JPF), que é uma ferramenta usada na verificação de códigos Java. É apresentada a ferramenta Lua *To Cee*, que permite a tradução de códigos Lua para C, permitindo realizar uma comparação funcional com o módulo *tradutor* desenvolvido para o BMCLua. Finalmente, são apresentadas as ferramentas Lua *Checker*, LDT e Lua *Inspect*, e *Lua AiR*, utilizadas para a verificação de códigos escritos na linguagem Lua.

3.1 CBMC

Clarke *et al.* [19] apresentam a ferramenta *C Bounded Model Checker* (CBMC), utilizada na verificação formal de programas ANSI-C. Essa ferramenta é capaz de verificar propriedades de segurança (*safety*), limites de vetores e assertivas fornecidas pelo usuário. A ferramenta CBMC implementa a técnica de verificação de modelos limitada (*Bounded Model Checking* - BMC). Em BMC, um sistema de transição de estados M é desdobrado até um limite k, para se obter uma fórmula booleana, que é então verificada por um algoritmo SAT.

Conforme o exemplo da Figura 3.1, um programa ANSI-C (Figura 3.1a) é convertido



(a) Código ANSI-C.

(b) Forma SSA.

(c) Equações de vetores C e P.

Figura 3.1: Exemplo de conversão utilizando BMC.
Fonte: [19]

para a forma de atribuição estática única (*Static Single Assignment* - SSA) (Figura 3.1b). A partir da forma SSA, são geradas duas equações de vetores de bits: C (para as restrições) e P (para a propriedades) (Figura 3.1c). A equação C (*Constraints*) define as atribuições e suposições, e a equação P (*Properties*) define as assertivas criadas pelo usuário. Como resultado, o verificador SAT valida a fórmula $C \wedge \neg P$. Se a equação é satisfazível, uma violação da propriedade foi encontrada.

3.2 LLBMC

O verificador *Low Level Bounded Model Checker* (LLBMC), apresentado por Falke *et al.* [20], utiliza a técnica BMC e solucionadores SMT. Diferentemente de outras ferramentas BMC para programas C/C++, o LLBMC realiza a verificação a partir de uma representação intermediária gerada pelo compilador, e não diretamente sobre o código-fonte C. A Figura 3.2 mostra o processo de verificação utilizando a técnica LLBMC.

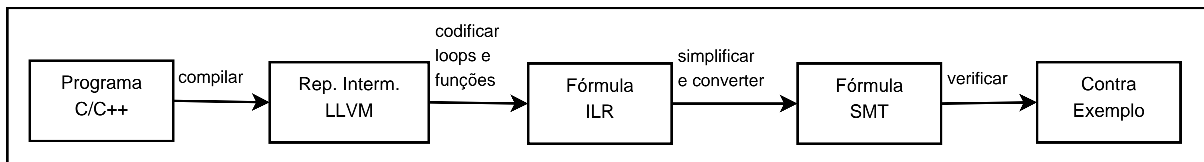


Figura 3.2: Processo de verificação LLBMC.
Fonte: [20]

Conforme observar-se na Figura 3.2, o programa C/C++ deve primeiro ser convertido em uma representação intermediária (*Low Level Virtual Machine Intermediate Representation* – LLVM-IR) [42], através de um compilador. A partir do LLVM IR, laços desdobrados e funções são codificados para uma representação lógica intermediária (*Intermediate Logic Re-*

presentation - ILR). Em seguida, a fórmula ILR é simplificada, convertida em fórmulas SMT e finalmente verificada por um solucionador SMT, como o STP [43].

3.3 ESBMC

Cordeiro *et al.* [14] apresentam a ferramenta ESBMC como uma abordagem eficiente para a verificação de programas embarcados ANSI-C utilizando BMC e solucionadores SMT. Nesse trabalho, os autores 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 CVC3 [44], Boolector [45] e Z3 [24].

O ESBMC é um verificador de modelos baseado nas teorias do módulo da satisfatibilidade (SMT) [14] 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 vetores e outros tipos de violações.

O ESBMC também permite estender essa abordagem de verificação de violação de propriedade a programas complexos, que possuam muitas iterações. Na Figura 3.3, é possível visualizar a arquitetura do ESBMC [14].

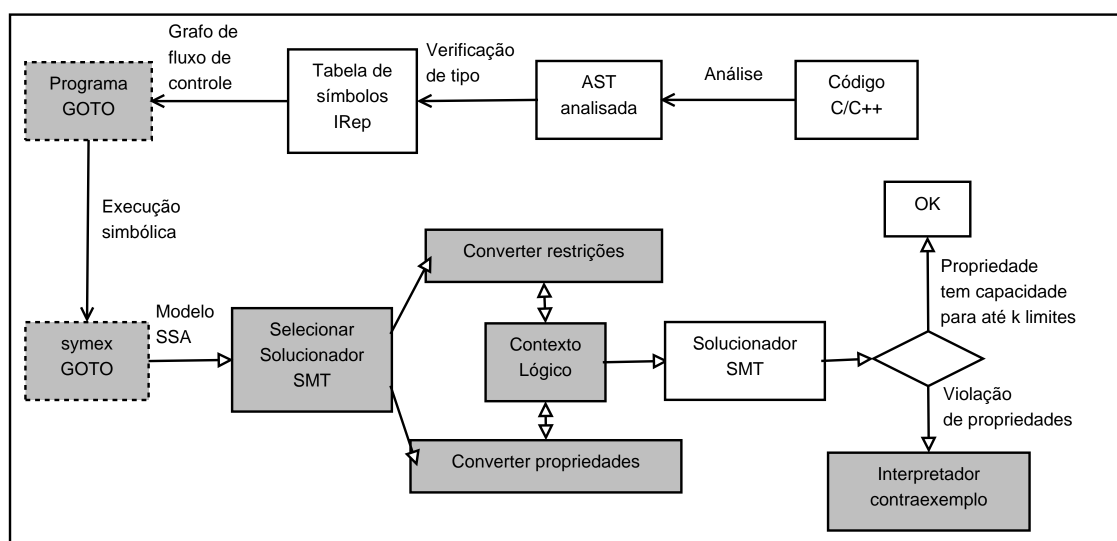


Figura 3.3: A arquitetura do ESBMC.

A ferramenta ESBMC faz uso dos componentes do 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.

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, 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 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, tornando-o ideal para testes eficientes de software embarcado de tempo real.

Para o trabalho desta dissertação, os benefícios oferecidos pela ferramenta ESBMC são estendidos à verificação de programas escritos na linguagem de programação Lua. Na literatura atual, este é um primeiro esforço para a utilização de BMC na verificação de códigos Lua.

3.4 Java *PathFinder*

O Java *PathFinder* (JPF), apresentado por Havelund e Pressbuger [46], traduz programas Java que podem conter assertivas definidas pelo usuário, para o modelo PROMELA [47], que é uma linguagem de modelagem baseada em fórmulas LTL. Em seguida, o modelo é checado pelo *Simple PROMELA Interpreter* - SPIN [48], que é um verificador de modelo de estados finitos, capaz de detectar *deadlocks*, violações e assertivas definidas pelo usuário. O SPIN pode ser automatizado, para determinar se um programa satisfaz ou não uma determinada propriedade.

A abordagem de verificação de programas Java, empregada neste trabalho, foi utilizada como estudo preliminar para a definição da metodologia BMCLua, que consiste na execução de

duas etapas: a tradução da linguagem alvo (ex.: Java ou Lua) para uma linguagem intermediária, como PROMELA ou ANSI-C, e a verificação através de verificadores SAT/SMT, como o SPIN ou ESBMC.

3.5 Lua *To Cee*

A ferramenta Lua *To Cee* [49] converte código Lua em código C com funções da API Lua. O compilador é escrito em Lua, sendo gerada uma árvore de análise AST com meta-dados, permitindo a tradução para o código C desejado.

Na Figura 3.4, é mostrado o resultado da conversão de um código Lua (Figura 3.4a), com apenas uma instrução, para um código C (Figura 3.4b), com esta ferramenta *Lua To Cee*.

```
1 -- Código Lua com apenas 1 instrução de atribuição
2 i = 1
```

(a) Código Lua.

```
1 /* WARNING: This file was automatically generated by lua2c. */
2 #ifdef __cplusplus
3 extern "C" {
4 #endif
5 #include <lua.h>
6 #include <lauxlib.h>
7 #include <lualib.h>
8 #ifdef __cplusplus
9 }
10 /* name: (main) function(...) */
11 static int lcf_main (lua_State * L) {
12   enum { lc_nformalargs = 0 };
13   #ifndef NDEBUG
14   const int lc_nactualargs = lua_gettop(L);
15   #endif
16   #ifndef NDEBUG
17   const int lc_nextra = (lc_nactualargs - lc_nformalargs);
18   #endif
19   /* i = 1 */
20   lua_pushnumber(L,1);
21   lua_setfield(L,LUA_ENVIRONINDEX,"i");
22   assert(lua_gettop(L) - lc_nextra == 0);
23   return 0;
24 }
```

(b) Código em C com API Lua.

Figura 3.4: Trecho do código C gerado pela ferramenta Lua To Cee.

Outras funções, mostradas na Figura 3.4 e que são geradas a partir da API Lua, como *traceback*, *lc_l_message*, *lc_dofile* e *lc_handle_luainit*, são adicionadas ao código C. Para o estudo desta ferramenta, não foi possível compilar e testar o código C, através do verificador ESBMC, pois este apresentou vários erros de vinculação de bibliotecas, durante a compilação.

O *Lua To Cee* não suporta algumas estruturas da linguagem Lua, como co-rotinas. Durante os testes realizado por Manura [49], a ferramenta obteve um desempenho, na conversão do código Lua para C, entre 25% e 75%, conforme informado pelo autor.

Realizando um comparativo com tradutor da metodologia BMCLua, o *Lua To Cee* gera um código muito grande, com diversas funções da API Lua para a linguagem C.

3.6 Lua *Checker*

O Lua *Checker*, apresentado por Smith [50], é uma ferramenta de verificação de *scripts* Lua que permite detectar problemas em variáveis e constantes como:

a) Variáveis não declaradas;

b) Múltiplas declarações de variáveis, com tipos de valores diferentes;

c) Tentativa de alterar o valor das constantes.

A ferramenta Lua *Checker*, que ainda encontra-se em desenvolvimento, contém um analisador LALR (do inglês, *Look-Ahead LR*) [41], compatível com a gramática livre de contexto da linguagem Lua. Esta ferramenta não utiliza a técnica de verificação de modelos limitada (BMC), em códigos Lua, como descrito nesta dissertação.

O uso da ferramenta exige algumas restrições durante a codificação em Lua, como:

a) Todas as variáveis globais precisam ser declaradas antes de serem usados;

b) Variáveis globais acessadas pela *table* _G não são verificadas;

c) As variáveis podem ser declaradas como constante, utilizando a macro do Lua *Checker*, seguida da palavra-chave *const*.

A Tabela 3.1 mostra um comparativo, entre Lua *Checker* e BMCLua, das estruturas da linguagem Lua verificadas por aquelas ferramentas.

Estrutura Lua	BMCLua	Lua Checker
Declaração de variáveis	SIM	SIM
Atribuições simples e múltiplas	SIM	SIM
Tabelas e estruturas de controle	SIM	NÃO
Funções	PARCIAL	NÃO
Biblioteca padrão	NÃO	NÃO
Biblioteca NCLua	PARCIAL	NÃO

Tabela 3.1: Comparativo das estruturas Lua verificadas pelas ferramentas BMCLua e Lua *Checker*.

3.7 LDT e Lua *Inspect*

Ferramentas de desenvolvimento Lua (LDT - *Lua Development Tools*) [51, 52] consiste em um *plug-in* para o IDE Eclipse que realiza a análise estática de códigos Lua, cuja principal aplicação é colocar em destaque (*highlighting*) e refatorar códigos Lua.

Lua *Inspect* é uma ferramenta experimental de análise estática de variáveis [52, 53], que permite analisar e identificar variáveis globais e locais, inferir valores de variáveis, verificar assinaturas de funções e o número de argumentos da função.

Ambas as ferramentas LDT e Lua *Inspect* são baseadas em MetaLua [54], que é uma extensão da linguagem Lua e oferece funcionalidades como compilação AST e análise sintática.

3.8 Lua AiR

O trabalho apresentado por Klint [52] descreve a abordagem do *framework* Lua AiR (do inglês, *Analysis in Rascal*), que é um meta programa Rascal capaz de implementar a análise do código Lua como um *pipeline*. Rascal [55] é uma meta linguagem de programação que permite criar ferramentas de análise de softwares a partir de uma gramática de linguagem. A Figura 3.5 ilustra as etapas do processo de análise de um *script* Lua utilizando o Lua AiR.

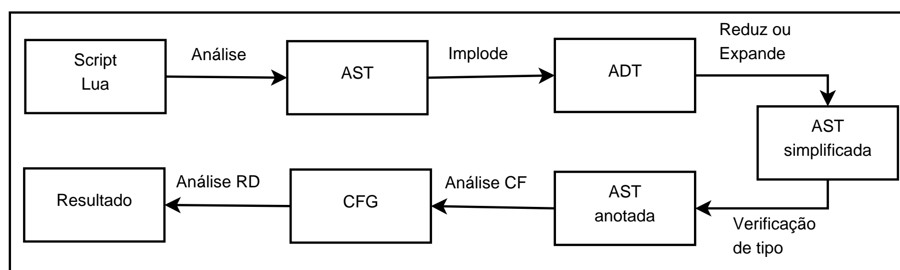


Figura 3.5: A arquitetura do Lua AiR.

A primeira etapa consiste em gerar uma árvore AST a partir da análise do *script* Lua. Em seguida, a árvore AST é implodida em um modelo ADT (do inglês, *Algebraic Data Type*) [56], que consiste em uma definição algébrica de cada elemento da árvore AST. A etapa de verificação é do tipo estática e gera anotações na AST com informações de escopo. A partir da AST é gerado um gráfico de fluxo de controle (CFG) através de uma analisador de controle de fluxo (CF - *Control Flow*). Na última etapa, o CFG é utilizado pelo analisador de alcance de definições (RD - *Reaching Definitions*), gerando as definições de alcance através da computação de ponto fixo. Os resultados fornecidos pela ferramenta são baseados nas anotações da árvore AST.

3.9 Resumo

Neste capítulo, os trabalhos utilizados como base para os estudos realizados nesta dissertação foram apresentados. O trabalho sobre CBMC, apresentado por Clark *et al.* [19], demonstra a capacidade da verificação de modelos, para programas C. Falker *et al.* [20] apresenta o verificador LLBMC, que realiza a verificação de programas C/C++, a partir da linguagem intermediária LLVM. Em seguida, o trabalho de Cordeiro *et al.* [14] foi apresentado, que mostrou a arquitetura e funcionamento do verificador de modelos ESBMC, capaz de verificar programas sequenciais e multi-tarefas. Também foi apresentado o trabalho de Havelund e Pressburger [46], cujo princípio básico foi utilizado como base para o desenvolvimento da ideia principal do módulo tradutor da metodologia BMCLua. Foram descritas as ferramentas LDT e Lua *Inspect*, baseadas em MetaLua e utilizadas na análise estática e refatoração de códigos Lua. É apresentado o trabalho de Klint [52], que descreve o Lua AiR, uma ferramenta baseada em Rascal, que permite análise estática de *script* Lua. O trabalho de Smith [50], que trata da ferramenta Lua Checker para a verificação de códigos Lua, restringe-se as variáveis e constantes. Finalizando este capítulo, foi apresentado o trabalho de Manura [49], que permite a conversão de códigos Lua para códigos C com API Lua.

A Tabela 3.2 mostra um comparativo entre os trabalhos relacionados e o trabalho proposto nesta dissertação. Comparado com as outras ferramentas relacionadas neste capítulo, o trabalho proposto é o primeiro a utilizar a técnica *Bounded Model Checking* para verificar programas Lua.

Trabalho Relacionado	Verificador de Modelos	Linguagem Suportada	Linguagem Intermediária	Baseado em
Clark *et al.* [19]	CBMC	C	Não	Solver SAT
Falker *et al.* [20]	LLBMC	C/C++	LLVM	Solver SMT
Cordeiro *et al.* [14]	ESBMC	C/C++	Não	Solver SMT
Havelund e Pressbuger [46]	JPF	Java	Promela	Solver SAT
Aubry [51]	LDT	Lua	Não	MetaLua
Manura [53]	Lua *Inspect*	Lua	Não	MetaLua
Klint [52]	Lua AiR	Lua	Não	Rascal
Smith [50]	Lua *Checker*	Lua	Não	Solver SAT
Trabalho Proposto	BMCLua	Lua	Ansi-C	Solver SMT

Tabela 3.2: Tabela comparativa entre os trabalhos relacionados.

No desenvolvimento deste trabalho, algumas dificuldades foram encontradas durante a verificação de programas Lua. Diferentemente de outras linguagens, como 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 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*.

Capítulo 4

Verificação de Programas Lua com *Bounded Model Checking*

Neste capítulo, a metodologia BMCLua e cada componente que compõe os módulos de tradução, verificação BMC e interpretação de contraexemplo são apresentados. Os analisadores e as respectivas classes Java, utilizados no tradutor da arquitetura BMCLua, são descritos. Também são listadas as estruturas da linguagem Lua atualmente implementadas no tradutor do BMCLua e as que ainda não estão na ferramenta, para cobrir toda a sintaxe da linguagem Lua. Em seguida, o funcionamento do módulo de verificação com o uso da ferramenta ESBMC é descrito, mostrando um exemplo do processo de rastreamento e a interpretação do contraexemplo, o que pode indicar informações de possíveis violações. Para finalizar, é apresentado, na Seção 4.2 (Avaliação Experimental), o ambiente de testes onde os experimentos foram realizados e os *benchmarks* utilizados, além dos resultados obtidos.

4.1 Metodologia BMCLua

A abordagem da metodologia proposta para a verificação de programas Lua é definida em três etapas básicas: tradução, verificação e interpretação do contraexemplo. A primeira etapa consiste na conversão de um código-fonte escrito em linguagem de programação Lua para a linguagem intermediária (ou modelo) ANSI-C. Esse processo é realizado pelo módulo tradutor, que recebe como entrada o código-fonte Lua e fornece como saída um código-fonte ANSI-C equivalente ao programa Lua.

Na etapa seguinte, o verificador BMC recebe como entrada o código ANSI-C gerado e realiza uma verificação de modelos através da ferramenta ESBMC. Como resultado dessa etapa, será informado ao usuário se a verificação do código foi ou não realizada com sucesso.

A etapa de interpretação consiste em capturar o contraexemplo gerado pelo ESBMC, caso não tenha tido sucesso na verificação, e, através do interpretador do BMCLua, informar qual violação foi detectada e em que trecho do código Lua ocorreu o problema. De posse dessas informações, o usuário poderá corrigir o código-fonte Lua para eliminar o problema detectado e então realizar uma nova verificação do código corrigido.

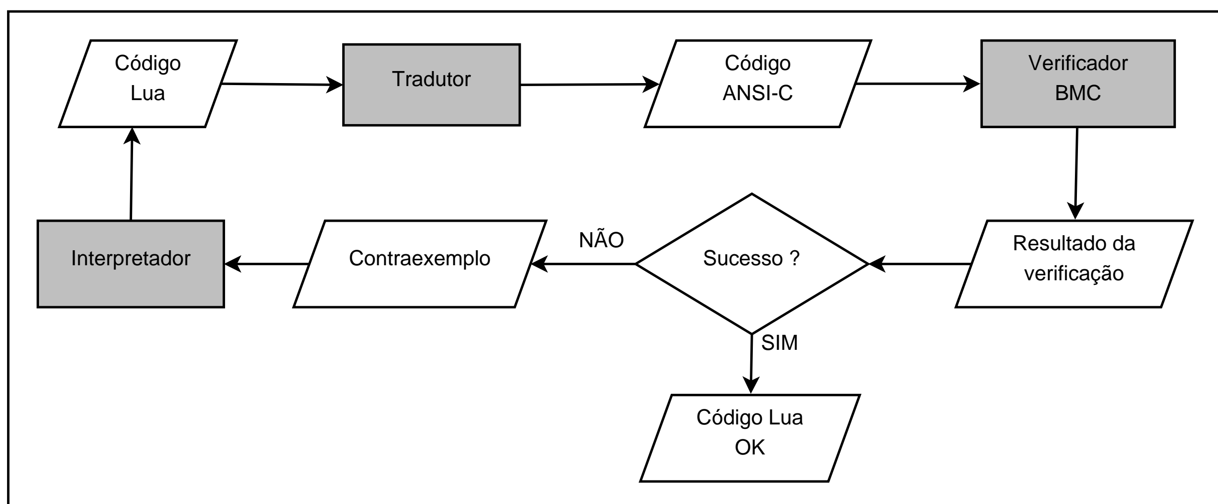


Figura 4.1: Fluxo de verificação com BMCLua.

A Figura 4.1 ilustra o fluxo de verificação definido para a metodologia BMCLua. Pode-se observar o bloco *Tradutor*, que recebe como entrada um fluxo de dados correspondente ao código-fonte Lua e tem como saída, um texto estruturado correspondente ao código-fonte ANSI-C equivalente. A etapa de verificação é representada pelo bloco *Verificador BMC*, que recebe como entrada o código ANSI-C gerado pelo tradutor e fornece, como saída, uma lista de iterações realizadas, representado pelo bloco de saída *Resultado da verificação*. No bloco *Interpretador*, o contraexemplo, gerado pelo verificador BMC é interpretado, mostrando a propriedade violada e a linha no código onde foi detectado o erro.

O tradutor utiliza uma gramática BNF que descreve a sintaxe das estruturas da linguagem Lua. São utilizados dois analisadores, o léxico e o sintático, que avaliam a corretude do código-fonte Lua e geram uma árvore AST, utilizada por uma interface de saída que gera o código-fonte ANSI-C, verificado posteriormente no bloco *Verificador BMC*.

Na Figura 4.2 é apresentado o fluxo de tradução definida na metodologia BMCLua.

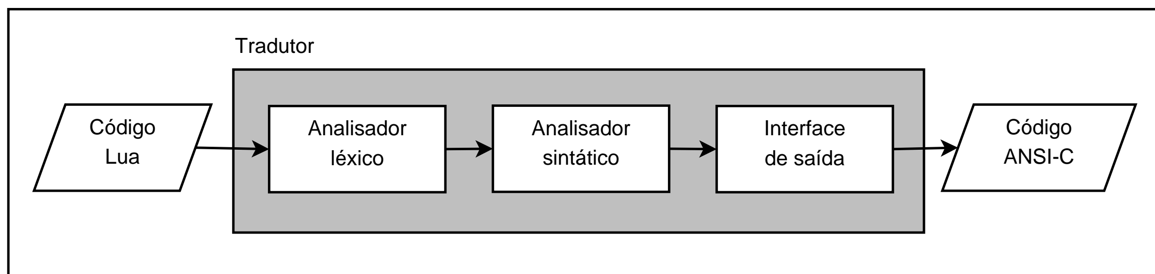


Figura 4.2: Fluxo de tradução com BMCLua.

A ferramenta utilizada no bloco *Verificador BMC*, para este trabalho, é o ESBMC, mas com algumas modificações no bloco *Interpretador*, pode-se utilizar outras ferramentas, como o CBMC. O verificador BMC gera um resultado, que normalmente descreve as iterações realizadas durante a verificação, e caso seja detectada alguma violação no código C, é adicionado no resultado um contraexemplo, mostrando o tipo de violação e em qual linha do código-fonte foi encontrado o erro.

O desenvolvimento do bloco *Interpretador* deve como objetivo facilitar a localização e análise da violação encontrada no código-fonte Lua, a partir do contraexemplo gerado pela ferramenta ESBMC, após a verificação do código-fonte ANSI-C. Esta etapa consiste basicamente em substituir, respectivamente, o nome do arquivo e as linhas do código-fonte C verificado pelo ESBMC para o nome do arquivo e as respectivas linhas do código-fonte Lua, verificado pelo BMCLua.

Na Figura 4.3 é apresentado o fluxo de interpretação, definida na metodologia BMCLua, a partir do resultado do *Verificador BMC*.

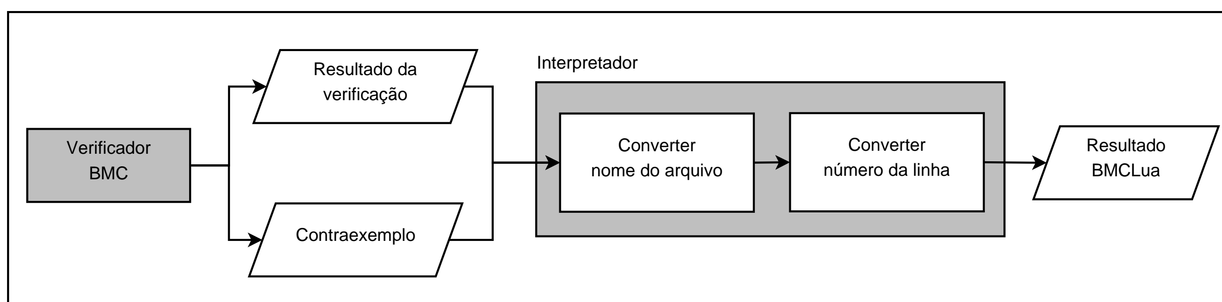


Figura 4.3: Fluxo de interpretação com BMCLua.

O esforço deste trabalho foi concentrado na pesquisa da metodologia adotada para a verificação de códigos Lua e no desenvolvimento dos blocos *Tradutor* e *Interpretador*.

4.1.1 Tradução

Conforme já mencionado na Seção 4.1, o BMCLua realiza a tradução de códigos escritos em Lua para ANSI-C, que são então verificados pela ferramenta ESBMC.

O tradutor foi desenvolvido utilizando a ferramenta ANTLR [26]. O ANTLR permite, a partir de uma gramática de linguagem *Backus Naur Form* - BNF [28], que é amplamente utilizada no desenvolvimento de compiladores, gerar os analisadores léxico e de sintaxe, além de fornecer mecanismos capazes de gerar textos estruturados, que são utilizados, por exemplo, na conversão de sentenças entre linguagens. A escolha da ferramenta ANTLR foi devido à vantagem de poder gerar classes Java para os analisadores, facilitando a integração com as ferramentas desenvolvidas para a metodologia BMCLua.

Na Figura 4.4, o fluxo de tradução é mostrado. Nesse diagrama, podem-se observar as classes Java associadas aos analisadores *lexer* e *parser*.

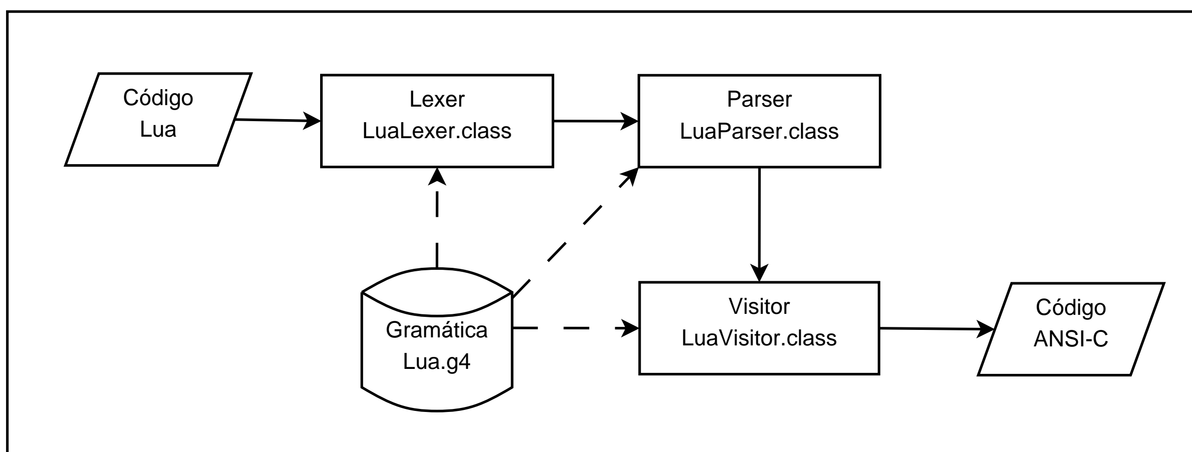


Figura 4.4: Fluxo de tradução do BMCLua.

A gramática Lua (*Lua.g4*) [57], utilizada no desenvolvimento do tradutor, é uma gramática BNF [28] que consiste em um conjunto de regras que descreve toda a sintaxe da linguagem Lua. As classes *LuaLexer*, *LuaParser* e a interface *LuaVisitor* são geradas pelo ANTLR a partir da gramática *Lua.g4* (Apêndice B). A classe *LuaLexer* é construída a partir dos tipos de *tokens*, como INT (inteiros) e FLOAT (número de ponto flutuante), definidos dentro da gramática. A classe *LuaParser* é gerada a partir das regras definidas em *Lua.g4*, como a regra *stat*. A interface *LuaVisitor* é gerada a partir das regras rotuladas, dentro da gramática BNF, cujos eventos precisam ser controlados e modificados, para uso no tradutor.

```
1 grammar Lua;
2
3 chunk : block EOF # blockChunk
4       ;
5
6 block : stat* retstat? # statBlock
7       ;
8
9 stat
10     : ';'
11     | varlist '=' explist # assignMul
12     | functioncall
13     | label
14     | 'break' # breakStat
15     | 'goto' NAME # gotoStat
16     | 'do' block 'end' # doStat
17     | 'while' exp 'do' block 'end' # whileStat
18     | 'repeat' block 'until' exp # repeatStat
19     | 'if' exp 'then' block ('elseif' exp 'then' block)*
20       ('else' block)? 'end' # ifStat
21     | 'for' NAME '=' exp ',' exp (',' exp)? 'do' block 'end' # forStat
22     | 'for' namelist 'in' explist 'do' block 'end' # forInStat
23     | 'function' funcname funcbody # functionStat
24     | 'local' 'function' NAME funcbody
25     | 'local' namelist ('=' explist)? # varLocal
26     ;
27
28 // LEXER
29 INT : [0-9]+ ;
30
31 FLOAT : [0-9]+ '.' [0-9]* [eE] [+-]? Digit+ ;
32
33 COMMENT : '--[[' .*? ']]' ;
34
35 NEWLINE : '\r'? '\n' -> skip ;
```

Figura 4.5: Trecho da gramática Lua.

Na Figura 4.5, mostra-se um trecho da gramática Lua, cujo código completo encontra-se no Apêndice B. A regra *chunk* representa uma parte do código Lua e o início da análise sintática realizada pelo *parser*. Dentro de cada regra *chunk*, existe a regra *block*, que corresponde a 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 vários tipos de instruções 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, dentro de uma instrução da regra *stat*, para formar a estrutura da linguagem Lua.

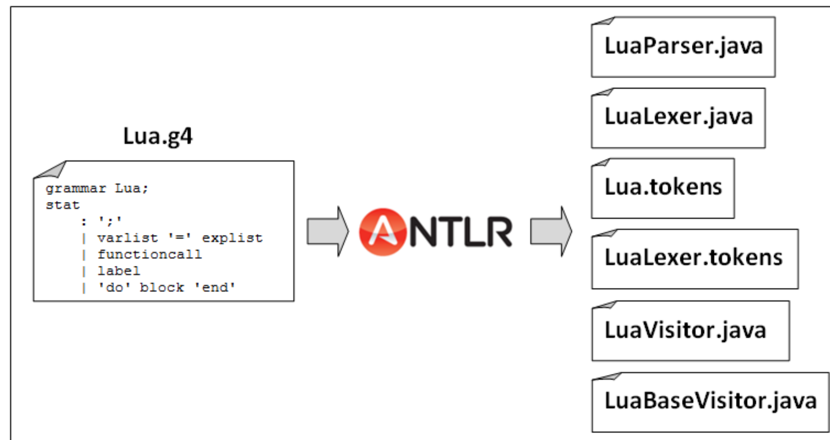
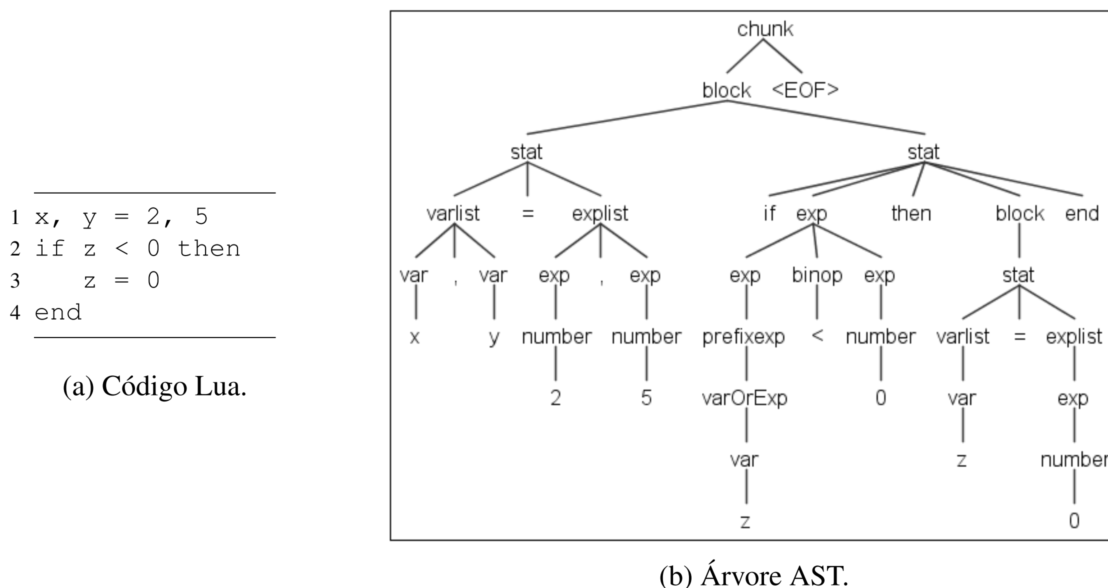


Figura 4.6: Arquivos gerados a partir da gramática Lua.

Na Figura 4.6, estão ilustradas as classes Java e os arquivos de identificação dos *tokens* gerados pelo ANTLR, a partir da gramática Lua. A classe *LuaLexer* é responsável por agrupar os caracteres de código-fonte Lua em grupos de *tokens*. O conjunto de *tokens* gerados pelo *lexer* são reconhecidos e tratados pela classe *LuaParser*. Caso a estrutura do código Lua não corresponda à sintaxe da gramática, o *parser* mostra uma mensagem de erro, informando quais *tokens* estão faltando ou foram definidos incorretamente na estrutura. O *parser* gera uma árvore AST, como ilustrado na Figura 4.7, onde pode-se realizar uma análise em profundidade *Depth-First Search* - DFS [40, 41].



(a) Código Lua.

(b) Árvore AST.

Figura 4.7: Árvore AST gerada pelo analisador *parser*.

Para a construção da árvore, conforme Figura 4.7b, o *parser* utilizou as regras definidas na gramática da linguagem Lua, descrita na Seção 4.1.1. A chamada da regra *chunk* inicia a

análise do código Lua, conforme Figura 4.7a, pelo *parser*, e a regra *block* especifica todo o bloco de código-fonte Lua.

Apesar de existirem quatro linhas de código Lua, existem apenas duas instruções definidas, segundo a gramática Lua. A estrutura de decisão *if ... end* corresponde a uma única instrução. Por isso, na árvore da Figura 4.7, o nó da regra *block* possui apenas dois nós filhos, que correspondem à regra *stat*. Os nós filhos são adicionados conforme as regras da gramática que compõe cada instrução, e cada nó-folha, na árvore AST, corresponde aos *tokens* do fluxo de símbolos de entrada (código-fonte). Por exemplo, a instrução de atribuição na linha 1 do código Lua (Figura 4.7a) é especificada na árvore pelas regras *varlist*, *explist*, *var*, *exp*, *number* e pelos *tokens* *x*, *y*, 2 e 5. A regra *block*, da instrução *if*, chama novamente a regra *stat* para montar na árvore a instrução $\{Z = 0\}$ da linha 3.

O mecanismo de busca do ANTLR, utilizado no tradutor, é o *visitor* [26], 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 à 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 4.8.

```
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 ^ i)
7       end
8    end
9    return ret
10 end
11
12 n = 2
13 y = teste(n)
```

(a) Código Lua.

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

(b) Código traduzido em ANSI-C.

Figura 4.8: Exemplo de tradução no BMCLua.

A linguagem de programação Lua não possui característica de tipagem explícita, ou seja, o tipo não é declarado previamente, mas é definido pelo valor atribuído à variável, em tempo de

execução. Durante a tradução de instruções de atribuição, o tradutor define o tipo da variável, através da avaliação do valor atribuído. Esse processo é muito facilitado pela sintaxe definida na gramática do ANTLR.

Na linguagem Lua, a declaração de variáveis locais é diferente da utilizada para as variáveis globais, devido à adição da palavra chave *local*, precedendo a instrução. Durante a tradução, todas as variáveis, que não são locais, são declaradas, no código ANSI-C, fora da função *main*. As variáveis locais também são declaradas no código ANSI-C gerado pelo BMCLua, dentro do escopo (*main*, *for*, outras estruturas) correspondente à declaração da variável em Lua.

A Figura 4.9 ilustra algumas traduções de variáveis de código Lua para ANSI-C. Em Lua, toda variável do tipo *number* representa um tipo de número real. Não existe a necessidade, na linguagem Lua, de variáveis do tipo inteiro. Na etapa de tradução, os tipos *int* e *double* são descritos na gramática Lua, permitindo a diferenciação dos tipos numéricos. O tipo *nil*, na linguagem Lua, representa uma ausência de valor ou a inexistência da variável. No caso desse tipo, o tradutor simplesmente ignora a instrução de atribuição. Para as variáveis booleanas, o tradutor cria o tipo *bool*, linha 2 da Figura 4.9b, que enumera o valor 0 para *true* e o valor 1 para *false*. Para o tipo *string*, da linguagem Lua, o tradutor declara ponteiros do tipo *char **, acrescentando no final do *string* um caractere "fim de string" (\0).

```
1  -- variáveis global e local
2  -- tipo númerico
3  i = 1
4  j = 2.2324
5
6  -- variáveis NIL
7  -- atribuição múltipla
8  z, w = 3, nil
9
10 -- variável do tipo Booleano
11 var_bool = true
12
13 -- variável do tipo String
14 var_str = "BMCLua"
```

(a) Código Lua.

```
1  #include <string.h>
2  typedef enum {false=0, true=1} bool;
3  int i;
4  double j;
5  int z;
6  bool var_bool;
7  char *var_str;
8  void main(){
9     i = 1;
10    j = 2.2324;
11    z = 3;
12    var_bool = true;
13    var_str = "BMCLua" + '\0';
14 }
```

(b) Código ANSI-C.

Figura 4.9: Exemplo de tradução de variáveis no BMCLua.

Na linguagem Lua, é possível realizar atribuição múltipla, como mostrado na Figura 4.9a. No BMCLua, a tradução de instruções de atribuição múltipla gera um conjunto de instruções

de atribuições simples, com declaração de tipo conforme o valor atribuído.

Para a tradução do código Lua para ANSI-C, foram realizadas análises das estruturas da linguagem Lua, definidas na gramática BNF, através da árvore AST gerada pelo *parser*. A Figura 4.10 ilustra a árvore AST para o código da Figura 4.9.

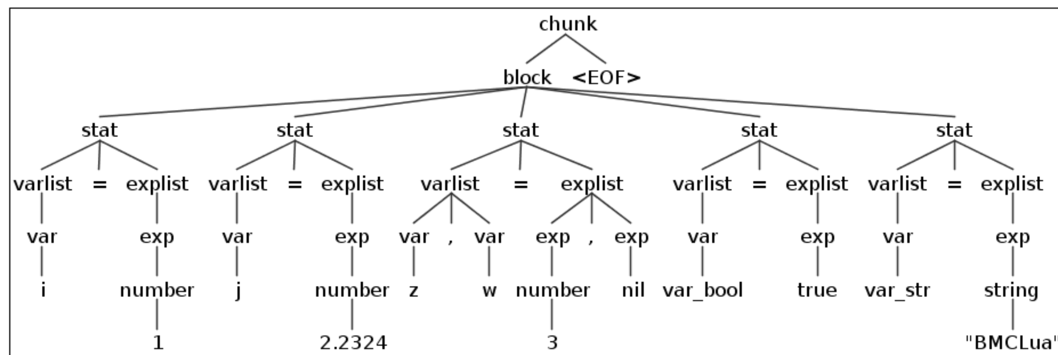


Figura 4.10: Árvore AST do código Lua ilustrado na Figura 4.9.

Todo código Lua, como mostrado na Figura 4.10, começa com a regra *block*, que é um conjunto de regras *stat* correspondendo às instruções da linguagem. No exemplo da Figura 4.9, cada instrução de declaração de variáveis é ilustrada pelo par de regras *varlist* e *explist*, separadas pelo *token* "=". Observando a terceira instrução (*stat*), o nó *varlist* possui três nós filhos, que correspondem as variáveis *z* e *w*, da instrução de atribuição múltipla, e ao *token* ",". Cada expressão *exp* da regra *explist* tem como nó filho o tipo de variável daquela expressão de atribuição, seguida do valor da variável, mostrado no nó folha. A exceção é para o valor *nil*, que não possui tipo definido.

Na linguagem Lua, como as variáveis não possuem tipos declarados, qualquer tipo de valor pode ser atribuído a mesma variável, em qualquer parte do código [4], como ilustrado na Figura 4.11a.

```
1 var = 10
2
3 var = 10.12
4
5 var = "BMCLua"
```

(a) Código Lua.

```
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) Código ANSI-C.

Figura 4.11: Exemplo de tradução de mudança de valor em variável.

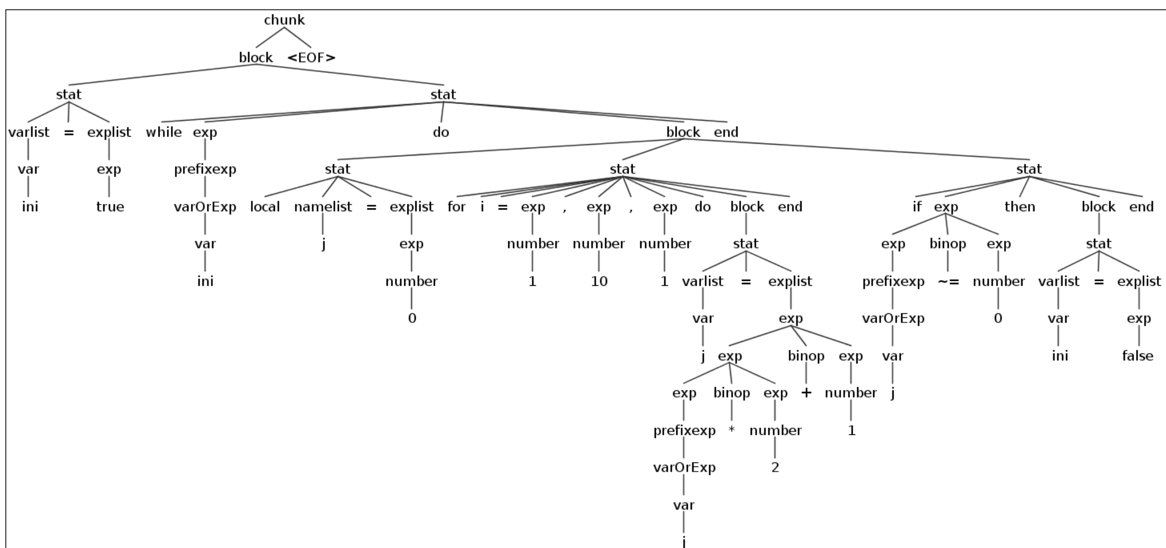
A Figura 4.11 ilustra como é realizada a tradução de instruções, no BMCLua, quando ocorre mudança do tipo de valor para a mesma variável ao longo do código Lua. A metodologia BMCLua registra em uma lista encadeada, durante a etapa de tradução, a variável e o tipo definido na primeira declaração. A partir desse registro, cada nova atribuição para a variável é analisada e, caso um valor de tipo diferente da primeira declaração seja detectado, é criada uma nova variável, de mesmo nome, seguida de uma sequência numérica, como ilustrado para a variável *var* da Figura 4.11. Este procedimento foi definido devido ao fato da linguagem ANSI-C não permitir mudar o tipo do valor atribuída a uma mesma variável.

```
1  ini = true
2  while ini do
3     local j=0
4     for i=1, 10,1 do
5         j = j*2 + 1
6     end
7     if j ~= 0 then
8        ini = false
9     end
10 end
```

(a) Código Lua.

```
1  typedef enum {false=0, true=1} bool;
2  bool ini;
3  init i;
4  void main(){
5     ini = true;
6     while(ini){
7        int j = 0;
8        for(i=1; i<=10; i=i+1){
9           j = j * 2 + 1;
10       }
11       if(j != 0){
12          ini = false;
13       }
14    }
15 }
```

(b) Código ANSI-C.



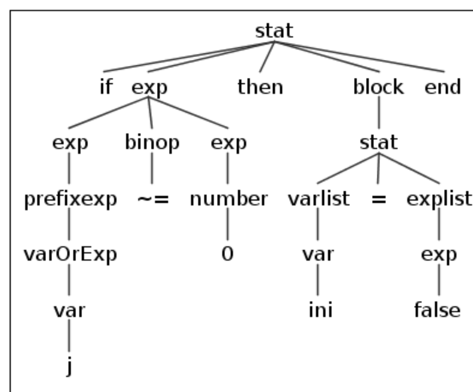
(c) Árvore AST do código Lua.

Figura 4.12: Exemplo de tradução de estruturas de controle no BMCLua.

Na Figura 4.12, são ilustradas as estruturas de controle traduzidas para um código ANSI-

C (Figura 4.12b), a partir de um código escrito na linguagem Lua (Figura 4.12a), e árvore AST completa (Figura 4.12c).

O módulo tradutor realiza a conversão das estruturas de controle *if*, *while*, *for*, *do* e *repeat*. A tradução consiste em reescrever os comandos delimitadores, de cada uma dessas estruturas, com as suas equivalentes em ANSI-C. Conforme é possível observar na Figura 4.12, durante a tradução, o comando "*while exp do block end*" é convertido para "*while (exp) { block }* ".



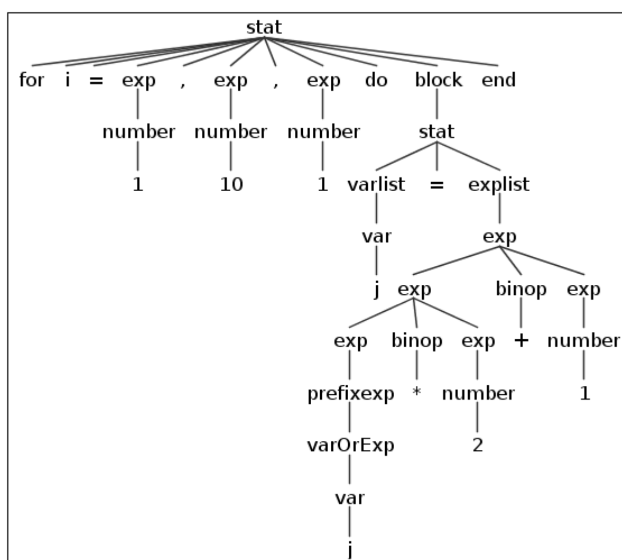
(a) Sub-árvore AST para a estrutura *if* em Lua.

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

(b) Código ANSI-C.

Figura 4.13: Tradução da estrutura *if*.

A tradução da estrutura "*if exp then block end*", ilustrada na sub-árvore AST (Figura 4.13a), é simplificada substituindo as palavras-chaves *then* e *end* por parenteses e chaves. A estrutura *if* terá a sintaxe "*if (exp) { block }* " (Figura 4.13b).



(a) Sub-árvore AST para a estrutura *for*.

```
1 for(i=1; i<=10; i=i+1){
2     j = i * 2 + 1;
3 }
```

(b) Código ANSI-C.

Figura 4.14: Tradução da estrutura *for*.

A estrutura *for*, ilustrada na sub-árvore da Figura 4.14, possui três expressões (*exp*), que correspondem, respectivamente, ao valor inicial da variável de incremento, o valor limite da variável e o valor de passo da variável. Caso este último valor não esteja informado, será traduzido como 1. Na etapa de tradução, a estrutura "*for var=exp, exp, exp do block end*" é convertida para "*for (var=exp, var<=exp, var=var+exp) { block }* " (Figura 4.13b), e a variável de incremento *var* é declarada fora da função *main*, como mostrado na Figura 4.12.

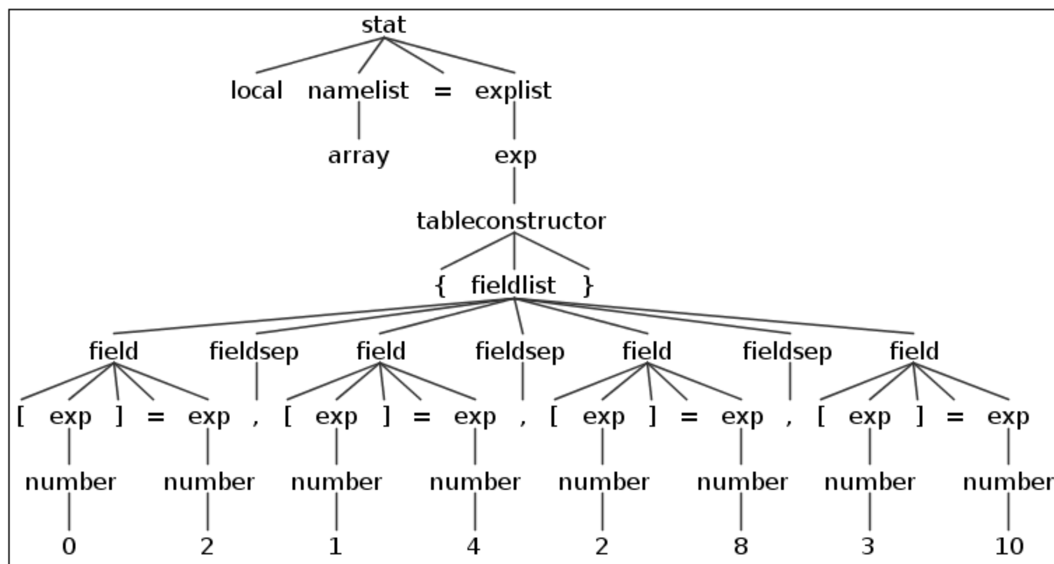
A estrutura de dados *table*, que pode representar vetores, listas e registros, é particularmente difícil de traduzir. Para a versão atual deste trabalho, o tipo construtor *table* foi traduzido somente para uma estrutura de vetor mais simplificada, como ilustrado no exemplo da Figura 4.15, que mostra a tradução de um tipo *table*, escrito em Lua (Figura 4.15a), para um vetor em ANSI-C (Figura 4.15b).

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

(a) Código Lua.

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

(b) Código ANSI-C.



(c) Árvore AST do código Lua.

Figura 4.15: Exemplo de tradução de *table* no BMCLua.

Na árvore AST (Figura 4.15c), da estrutura *table*, que corresponde ao tipo da variável *array*, o nó *tableconstructor* corresponde ao construtor do tipo *table*. Pode-se observar que

cada nó *field* corresponde a um elemento do vetor e o conjunto destes elementos compõe a lista encadeada *fieldlist*. Para cada elemento (*field*), o valor do nó *exp*, entre os *tokens* "[" e "]", define o índice de referência do elemento dentro do vetor. Caso não se defina o índice do elemento, este assumirá uma sequência de números inteiros, começando por 1.

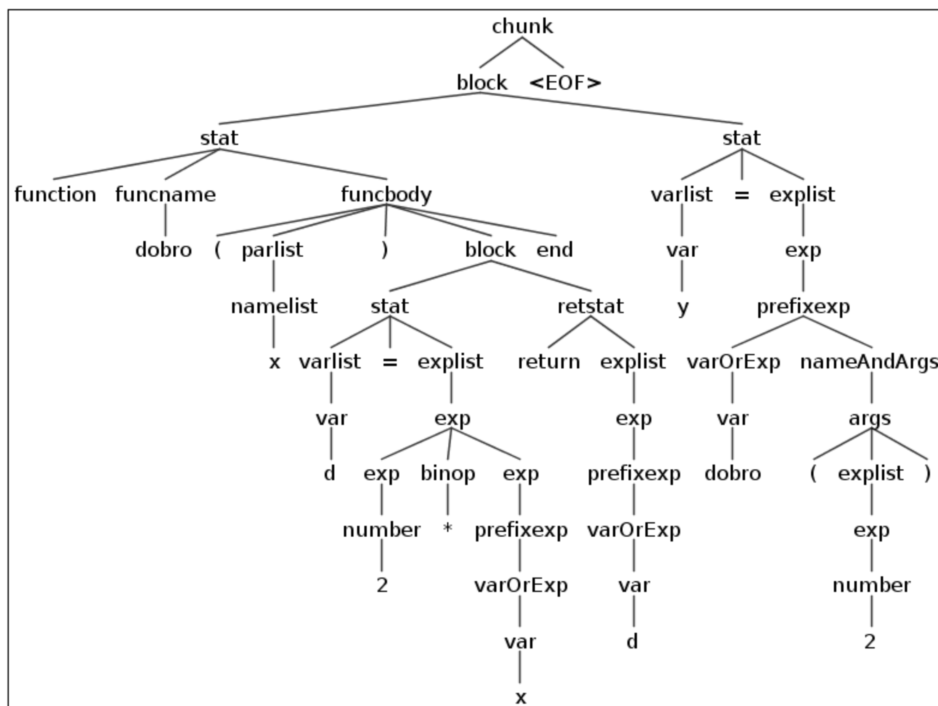
Na versão atual da metodologia BMCLua, o tradutor realiza a conversão de função com retorno de um único valor, pois a linguagem Lua permite múltiplo retorno de valores de função.

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

(a) Código Lua.

```
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) Código ANSI-C.



(c) Árvore AST do código Lua.

Figura 4.16: Exemplo de tradução de função no BMCLua.

Na Figura 4.16, é mostrado um exemplo de tradução de função, da linguagem Lua para ANSI-C. O *parser* do ANTLR, permitiu gerar uma árvore AST (Figura 4.16c), que possibilitou simplificar o desenvolvimento do tradutor para permitir a conversão de funções. A partir do nó *funcbody*, através da regra *parlist*, os parâmetros da função são listados e registrados em uma

lista encadeada, para utilização na tradução da instrução de chamada da função. A partir do nó *retstat*, é possível listar os valores retornados pela função no nó *explist*, após o *token return*.

Na Figura 4.17, é mostrado um exemplo de código NCLua para aplicação em TV digital [3]. Esse código realiza a contagem de vezes que o usuário trocou o ritmo musical, durante a execução de um vídeo. A variável *counter* é incrementado a cada vez que o ritmo foi trocado. Quando terminar a apresentação do vídeo, que corresponde ao valor do evento (*evt.value*) igual ao *string* “FIM”, será apresentado na tela o resultado “Total de ritmos: *x*”

```
1  local counter = 0
2  local dx, dy = canvas:attrsize() -- dimensões do canvas
3  function handler (evt)
4     if evt.class == 'ncl' and evt.type == 'attribution' and
5        evt.name == 'inc' then
6        if evt.value ~= 'FIM' then
7           counter = counter + evt.value
8        else
9           canvas:attrColor ('black')
10          canvas:drawRect ('fill',0,0,dx,dy)
11          canvas:attrColor ('yellow')
12          canvas:attrFont ('vera', 24, 'bold')
13          canvas:drawText (10,10, 'Total de ritmos: ' ... counter)
14          canvas:flush()
15       end
16
17       event.post {
18          class  = 'ncl',
19          type   = 'attribution',
20          name   = 'inc',
21          action = 'stop',
22          value  = counter,
23       }
24
25    end
26 end
27 event.register(handler)
```

Figura 4.17: Exemplo de código NCLua para TV digital.

Para a tradução do código NCLua, nessa versão da metodologia BMCLua, foram analisados as estruturas de funções do módulo *canvas*, utilizado para manipular objetos gráficos na TV digital, e as variáveis *table evt* e *event.post*, que são objetos específicos para aplicações interativas, e que estão associados aos eventos executados na aplicação.

A *table event.post*, mostrada no código da Figura 4.17, pertence ao módulo *event*, da biblioteca NCLua, e que permite a comunicação com documentos NCL [3], e pode responder à eventos externos gerados, por exemplo, pelo controle remoto da TV.

Na Figura 4.18, é mostrada, por exemplo, a sub-árvore associada a função *drawRect*, da biblioteca *canvas*, que permite desenhar um retângulo na tela da TV.

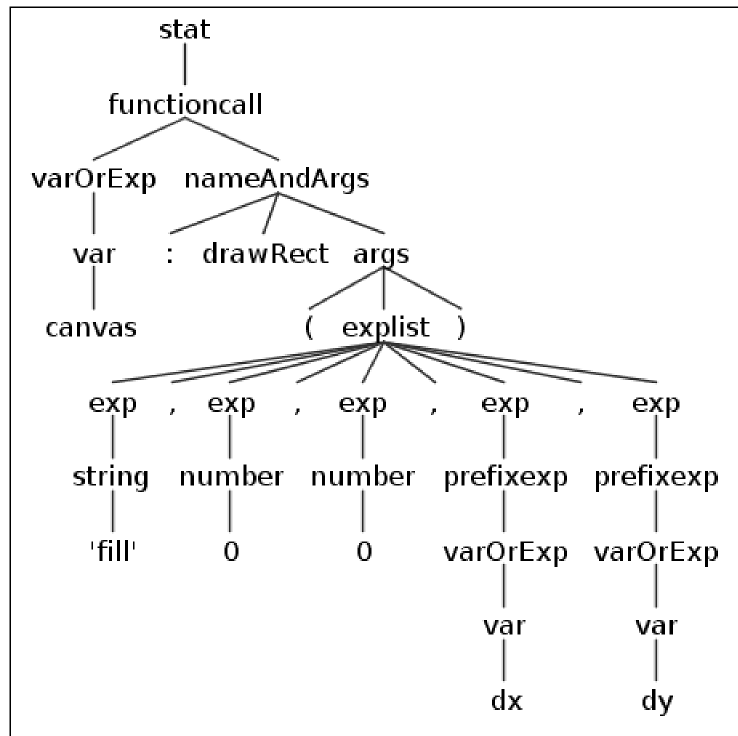


Figura 4.18: Sub-árvore AST da função *drawRect* do código NCLua.

Para cada função definida na variável *canvas*, no código da Figura 4.17, da linha 9 à 14, foi declarada uma função abstrata do tipo *void*, como ilustrado na Figura 4.19.

```
void canvas_attrColor(char *color);
void canvas_drawRect(char *type, int x, int y, int dx, int dy);
void canvas_attrFont(char *type, int size, char *style);
void canvas_drawText(int x, int y, char *text);
void canvas_flush();
```

Figura 4.19: Declaração de funções *canvas* em ANSI-C.

No código da Figura 4.17, existem duas variáveis do tipo *table*, *evt* e *event.post*. A variável, *event.post*, que está associada a um evento na aplicação interativa, finaliza o objeto NCLua, atribuindo o valor de *counter* para a propriedade *inc*, informada no campo *name*.

Para a tradução das variáveis *table*, foi analisada a sub-árvore AST, como exemplo da variável *event.post*, mostrada na Figura 4.20. A partir do nó *tableconstructor*, é possível listar os campos da variável, capturando o nome e do tipo de cada campo. Com estas informações, foi possível traduzir as variáveis *evt* e *event.post*, para estruturas (*struct*) na linguagem C.

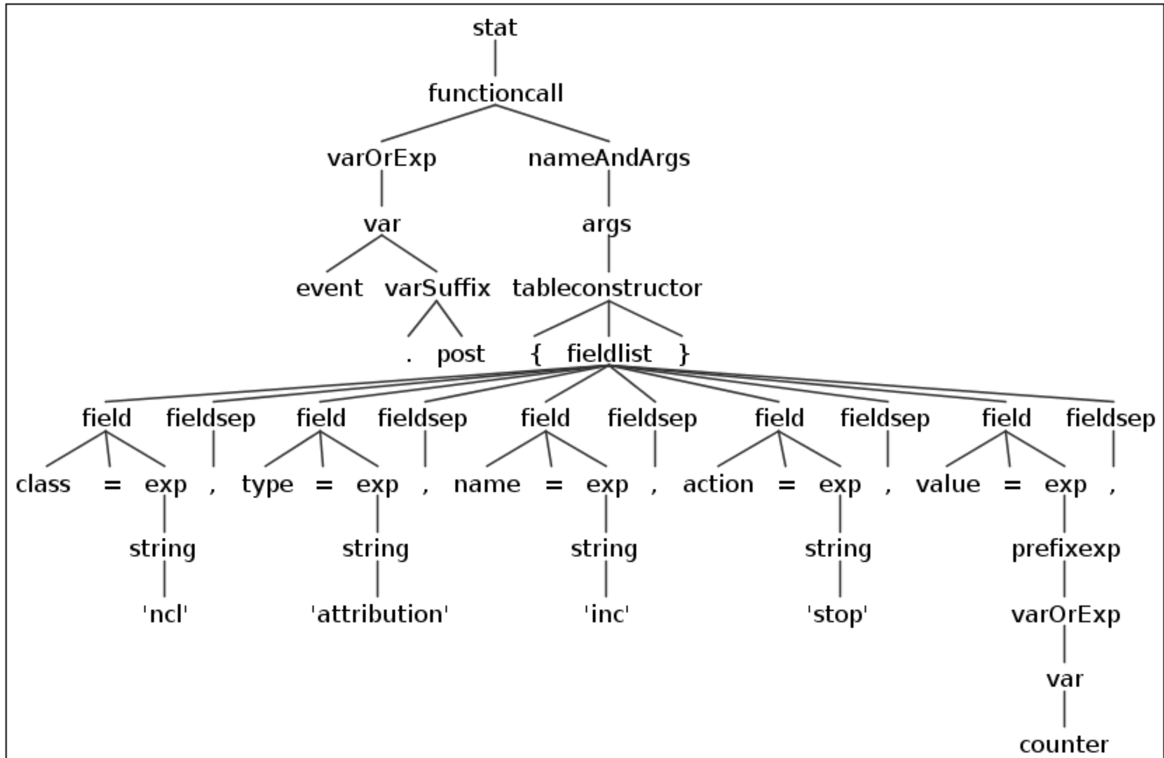


Figura 4.20: Sub-árvore AST variável *event.post* do código NCLua.

Na Figura 4.21, são mostradas as estruturas criadas, na linguagem C, a partir das variáveis *evt* e *event.post*, do código Lua, mostrado na Figura 4.17.

```
struct struct_evt{
  char *class;
  char *type;
  char *name;
  char *value;
  int value2;
};

struct struct_event_post{
  char *class;
  char *type;
  char *name;
  char *action;
  int value;
};
```

Figura 4.21: Definição de estruturas *evt* e *event.post* em ANSI-C.

Para realizar a tradução de códigos Lua, é utilizada a cláusula -nclua, na metodologia BMCLua, para permitir o uso das declarações abstratas das funções da biblioteca NCLua.

A Figura 4.22, mostra o código ANSI-C (Figura 4.22b) gerado pelo BMCLua, precedido

pelo comando da ferramenta (Figura 4.22a), incluindo a cláusula específica para a verificação de aplicações para TV digital.

```
~/bmclua$ java -jar bmclua.jar cod_nclua.lua -nclua
```

(a) Comando BMCLua.

```
1 struct struct_evt{
2   char *class;
3   char *type;
4   char *name;
5   char *value;
6   int value2;
7 };
8 struct struct_event_post{
9   char *class;
10  char *type;
11  char *name;
12  char *action;
13  int value;
14 };
15 int counter;
16 int canvas_attrsize(int pos);
17 int dx;
18 int dy;
19 void canvas_attrColor(char *color);
20 void canvas_drawRect(char *type, int x, int y, int dx, int dy);
21 void canvas_attrFont(char *type, int size, char *style);
22 void canvas_drawText(int x, int y, char *text);
23 void canvas_flush();
24 char *int_str(int number);
25 struct struct_event_post event_post;
26 void event_register(int handler);
27 void handler(struct struct_evt evt){
28    if((evt.class == "ncl")&&(evt.type == "attribution")&&(evt.name == "inc")){
29       if(evt.value != "FIM"){
30          counter = counter + evt.value2;
31       }else{
32          canvas_attrColor("black");
33          canvas_drawRect("fill", 0, 0, dx, dy);
34          canvas_attrColor("yellow");
35          canvas_attrFont("vera", 24, "bold");
36          canvas_drawText(10, 10, strcat("Total de ritmos: ", int_str(counter)));
37          canvas_flush();
38       }
39       event_post.class  = "ncl";
40       event_post.type   = "attribution";
41       event_post.name   = "inc";
42       event_post.action = "stop";
43       event_post.value  = counter;
44    }
45 }
46 void main(){
47    counter = 0;
48    dx = canvas_attrsize(1);
49    dy = canvas_attrsize(2);
50    event_register(0);
51 }
```

(b) Código ANSI-C.

Figura 4.22: Código da Figura 4.17 traduzido para ANSI-C.

O tradutor está em fase de desenvolvimento, com a implementação de variações da sintaxe de algumas estruturas da linguagem Lua, como o construtor *for*. Também falta implementar, no tradutor, a estrutura de co-rotinas, conhecidas como *threads*, e as bibliotecas de funções específicas, como as utilizadas pelas extensão NCLua, como *event* e *persistent*.

A versão atual deste trabalho traduz as estruturas da linguagem Lua listadas na Tabela 4.1.

Estrutura Lua	**Estrutura ANSI-C**
tipos primitivos (*boolean, number*)	*typedef bool, int, float*
string	*char* $*$
operador relacional ($\sim=$)	$!=$
operadores lógicos (*and, or, not*)	$\&\&, \|\|, !$
table	*array*
if .. else .. end	*if( ) { .. } else { .. }*
while .. do .. end	*while( ){ .. }*
for .. do .. end	*for(; ;){ .. }*
repeat .. until	*do{ .. }while(!..);*
do .. end	*{ ... }*
function ***nome*** *(..) .. end*	***tipo nome*** *(..) { .. }*

Tabela 4.1: Estruturas traduzidas pelo BMCLua

Abaixo são listadas as estruturas e comandos da linguagem Lua que faltam ser implementadas no tradutor da ferramenta Lua. Nos Apêndices C e D são detalhadas as estruturas e bibliotecas da linguagem Lua que não estão implementadas na versão atual do BMCLua.

a) *For* genérico (**Ex.: for VARS in ITERATOR do BLOCK end**);

b) Definição de função (**Ex.: f = function (ARGS) BODY end**);

c) Chamada de função (**Ex.: f "hello" para f("hello")**);

d) Chamada de Objeto (**Ex.: x.move(2, -3)**);

e) Declaração das funções da biblioteca Lua (**COROUTINE, METATABLE, PACKAGE, MATH, I/O, DEBUG**).

A corretude das traduções para o código ANSI-C foram realizadas fazendo testes funcionais com o código-fonte gerado, comparando com os resultados obtidos no código-fonte Lua original. Através dessa metodologia foi observado que os resultados gerados, tanto pelo código-fonte Lua como pelo código ANSI-C, eram iguais para o mesmo conjunto de dados de entradas.

4.1.2 Verificação

A etapa de verificação consiste em utilizar uma ferramenta BMC existente (ex.: ESBMC) para verificar o código-fonte ANSI-C, gerado pelo tradutor. Após essa etapa, o resultado gerado pelo verificador BMC é interpretado pelo BMCLua.

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 4.23, um exemplo é mostrado do resultado da verificação de um programa escrito na linguagem Lua. 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.

```
$ 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
```

Figura 4.23: Exemplo do resultado de verificação no BMCLua.

4.1.3 Interpretação

A etapa de interpretação consiste em tratar o texto gerado na saída do ESBMC, e o contraexemplo, em um texto de saída que informe corretamente as linhas verificadas no código Lua, e a linha onde ocorreu a violação informada no contraexemplo. O desenvolvimento dessa etapa foi necessária, pois a ferramenta ESBMC fornece informações sobre o código ANSI-C gerado pelo tradutor do BMCLua, o que não é de interesse do projetista para a verificação de aplicações Lua.

Para o desenvolvimento do interpretador, foi criada uma lista ordenada, dentro do tradutor, para registrar as linhas criadas no código ANSI-C a partir de cada linha, correspondente a uma instrução, do código Lua.

Na Figura 4.24, é ilustrado um código ANSI-C (4.24b), resultado da tradução no BMCLua, de um código Lua (4.24a). Nesse exemplo, pode-se observar, nos comentários, as linhas correspondentes ao código-fonte Lua.

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

(a) Código Lua.

```
~/bmclua$ java -jar bmclua.jar exemplo.lua -show -showln -showlnlua
    -no-check-esbmc

1 - int i;
2 - void main(){
3 -     int array[4] = {2,4,8,10}; // line number in lua code: 1
4 -     for(i=0; i<=3; i=i+1){ // line number in lua code: 2
5 -         print(array[i]); // line number in lua code: 3
6 -     } // line number in lua code: 2
7 - }
```

(b) Resultado da tradução para código ANSI-C.

Figura 4.24: Exemplo de tradução no BMCLua com informação das linhas no código Lua.

4.2 Avaliação Experimental

Para avaliar a eficiência da metodologia BMCLua, experimentos foram realizados, que consistiam em verificar algoritmos padrões, conhecidos como *benchmarks*, usados para testes

de desempenho de software. Foram utilizados os *benchmarks Bellman-Ford*, *Prim*, *BubbleSort*, *SelectionSort*, *Factorial*, *InsertSort* e *Fibonacci*. O algoritmo *Bellman-Ford* é aplicado na solução do problema do caminho mais curto (mínimo), com uso 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. O *Factorial* é um algoritmo recursivo que calcula o fatorial de um número natural [40]. O algoritmo *InsertSort* realiza a ordenação por inserção dos elementos, percorrendo um vetor da esquerda para a direita, ordenando os elementos mais à esquerda. Concluindo a lista dos *benchmarks*, o *Fibonacci* é um algoritmo iterativo que calcula a sequência de Fibonacci. Estes *benchmarks* são os mesmos utilizados na avaliação de desempenho e precisão do verificador ESBMC [14]. Por isso, a avaliação experimental consistiu na verificação de desempenho da metodologia BMCLua, tomando-se como comparação os dados experimentais do trabalho de Cordeiro *et al.* [14]. Os *benchmarks*, escritos em Lua pelo autor, estão disponíveis no Apêndice E.

4.2.1 O Ambiente de Testes

Os experimentos foram realizados na plataforma Linux, em um computador Intel Core i3 2,5 GHz, com 2 GBytes de RAM. Os tempos de processamento dos algoritmos foram medidos em segundos, utilizando-se a classe *ManagementFactory* do pacote *java.lang*, da linguagem Java [33]. Essa classe Java permite mensurar o tempo de CPU, descontando os tempos de E/S e de múltiplos processos (*threads*) que utilizam chaveamento de contexto.

Nos experimentos realizados, durante a etapa de verificação, foi utilizada a ferramenta ESBMC, na versão 1.23, com solucionador Z3 v3.2, configurado para o uso da aritmética *bit-vector*, que permite definir palavras de tamanho fixo para as fórmulas SMT [24].

4.2.2 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 800 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.

Algoritmo	**E**	**L**	**B**	**P**	**TL**	**TE**
Bellman-Ford	5	46	6	1	< 1	< 1
	10	46	11	1	< 1	< 1
	15	46	16	1	< 1	< 1
	20	46	21	1	< 1	< 1
	400	46	401	1	2	1
	800	46	801	1	3	2
Prim	5	70	6	1	< 1	< 1
	6	70	7	1	< 1	< 1
	7	70	8	1	< 1	< 1
	8	70	9	1	< 1	< 1
	200	70	201	1	27	26
	400	70	401	1	114	65
BubbleSort	12	29	13	1	< 1	< 1
	35	29	36	1	5	3
	50	29	51	1	10	6
	70	29	71	1	19	12
	140	29	141	1	85	56
	200	29	201	1	191	123
SelectionSort	12	31	13	1	< 1	< 1
	35	31	36	1	2	1
	50	31	51	1	4	3
	70	31	71	1	7	5
	140	31	141	1	31	23
	200	31	201	1	70	49
Factorial	50	11	51	0	< 1	< 1
	100	11	101	0	< 1	< 1
	150	11	151	0	< 1	< 1
	200	11	201	0	< 1	< 1
	400	11	401	0	1	1
	800	11	801	0	6	3
	2000	11	2001	0	38	26
InsertSort	20	21	21	1	< 1	< 1
	25	21	26	1	< 1	< 1
	50	21	51	1	< 1	< 1
	100	21	101	1	6	5
	200	21	201	1	33	32
	400	21	401	1	220	219
Fibonacci	5000	20	5001	1	1	< 1
	8000	20	8001	1	2	1
	10000	20	10001	1	3	2
	50000	20	50001	1	14	9
	80000	20	80001	1	22	15
	100000	20	100001	1	28	19

Tabela 4.2: Resultados de desempenho do BMCLua

Na Tabela 4.2, os resultados gerados são exibidos, onde **E** identifica o total de elementos do vetor, definido para cada avaliação do mesmo algoritmo. A coluna **L** é o total de linhas de

código Lua, gerado durante a etapa de tradução. A coluna **B** mostra o limite de iterações de *loops* realizadas, que automaticamente corresponderá ao total de linhas de código, em **L**, mais 1. A coluna **P** significa o total de propriedades verificadas, através de assertivas definidas pelo usuário. A coluna **TL** é o tempo de processamento total, em segundos, de verificação do código Lua na metodologia 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 *benchmarks* padrões usam a função *assert*, disponível em ambas as linguagens Lua e ANSI-C, a fim de verificar uma determinada propriedade. Assim, durante o processo de tradução do código-fonte Lua para ANSI-C, a instrução *assert* é convertida diretamente, permitindo que a ferramenta ESBMC verifique uma única propriedade de segurança.

4.2.3 Discussão dos Resultados

Os resultados obtidos demonstram a eficácia da ferramenta BMCLua na verificação de programas Lua utilizando a técnica *Bounded Model Checking*. Os testes realizados se concentraram na corretude da tradução do código-fonte Lua para seu equivalente ANSI-C, na corretude dos resultados obtidos na verificação, e no desempenho do tempo de processamento da metodologia BMCLua, que corresponde ao tempo de processamento da etapa de tradução mais o tempo de processamento da etapa de verificação.

Para que a equivalência entre o código ANSI-C produzido e o código Lua original fosse garantida, foram realizados testes de execução do programa ANSI-C, verificando se este produz o mesmo resultado obtido com o programa Lua, para o mesmo conjunto de entrada. Portanto, para cada arquivo de código traduzido de Lua para ANSI-C, um teste funcional do programa ANSI-C, gerado pelo tradutor, é executado.

Com relação aos resultados de todos os experimentos realizados, o BMCLua não relatou qualquer resultado falso-positivo ou falso-negativo, comprovando a sua regularidade na verificação de programas Lua para os experimentos previstos. Os resultados da verificação para todos os *benchmarks* Lua foram os mesmos mostrados pelos seus equivalentes em ANSI-C.

O desempenho do tempo de processamento da metodologia BMCLua, pode ser observado nos valores obtidos nas colunas **TL** e **TE**, da Tabela 4.2. Os tempos, indicados por "< 1",

correspondem aos valores observados, menores que 1 segundo. Também pode-se observar, que os tempos em **TL** são sempre maiores que os tempos anotados em **TE**, devido ao total do tempo de processamento na metodologia BMCLua, que corresponde a soma dos tempos das etapas de tradução, verificação e interpretação.

Na Figura 4.25, pode-se observar que o desempenho da metodologia BMCLua, para o algoritmo *Bellman-Ford* está bem próximo da ferramenta ESBMC.

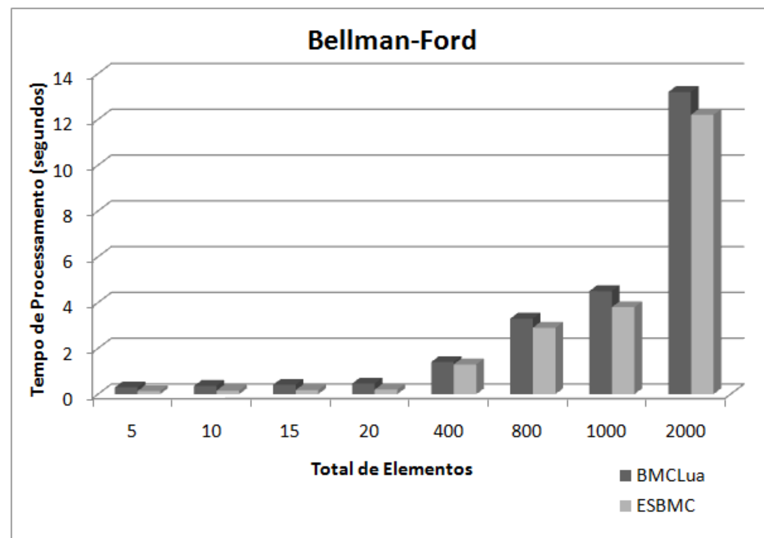


Figura 4.25: Gráfico do desempenho (segundos) para o algoritmo *Bellman-Ford*.

Para o algoritmo *Prim*, da Figura 4.26, existe uma diferença mais acentuada, entre os tempos gerados pelo ESBMC e pelo BMCLua, quando aumenta o número de elementos no vetor para 16.

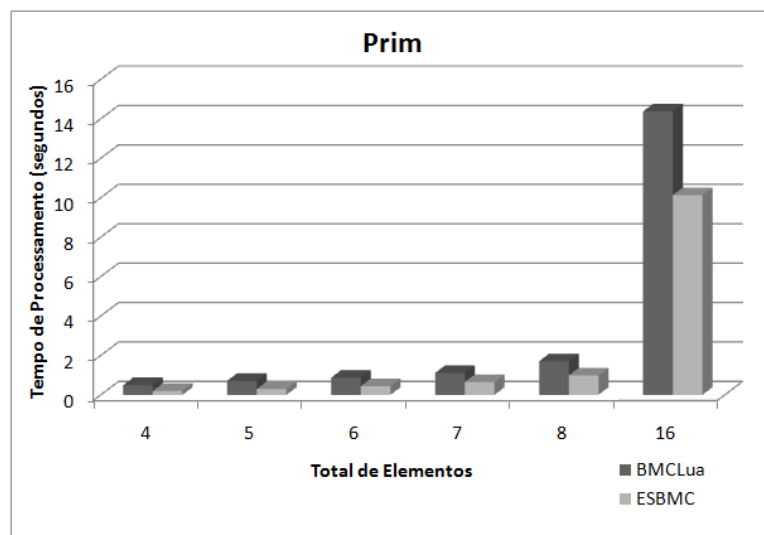


Figura 4.26: Gráfico do desempenho (segundos) para o algoritmo *Prim*.

Nas Figuras 4.27 e 4.28, observa-se que o desempenho da metodologia BMCLua, respectivamente, para os algoritmos *Bubblesort* e *Selectionsort*, está próximo da ferramenta ESBMC, quando comparado os tempos para vetores com menos de 140 elementos.

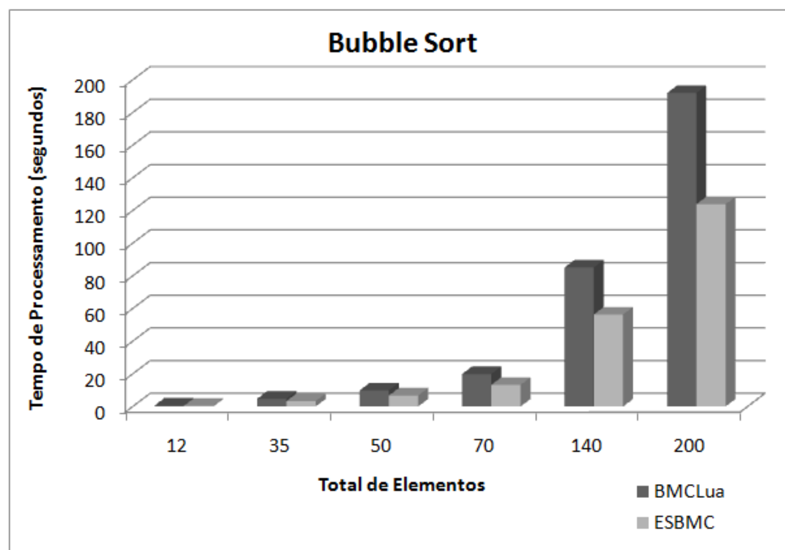


Figura 4.27: Gráfico do desempenho (segundos) para o algoritmo *Bubblesort*.

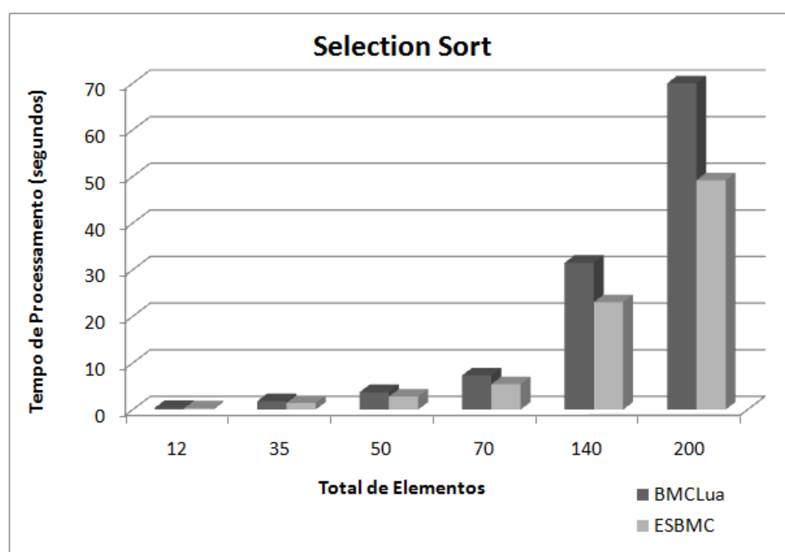


Figura 4.28: Gráfico do desempenho (segundos) para o algoritmo *Selectionsort*.

Nas Figuras 4.26, 4.27 e 4.28, pode-se verificar, para os algoritmos *Prim*, *Bubblesort* e *SelectionSort*, que com o aumento do número de elementos do vetor, o tempo de verificação do BMCLua aumentou muito em relação ao tempo de verificação na ferramenta ESBMC. Esse aumento deve-se aos processos de tradução e interpretação, e podem ser reduzidos aplicando algumas técnicas de otimização de algoritmos, como a propagação de constantes [41].

Nas Figura 4.29 e 4.30, são mostrados, respectivamente, os tempos de processamento dos algoritmos *Factorial* e *Fibonacci*, verificados pela metodologia BMCLua e pela ferramenta ESBMC.

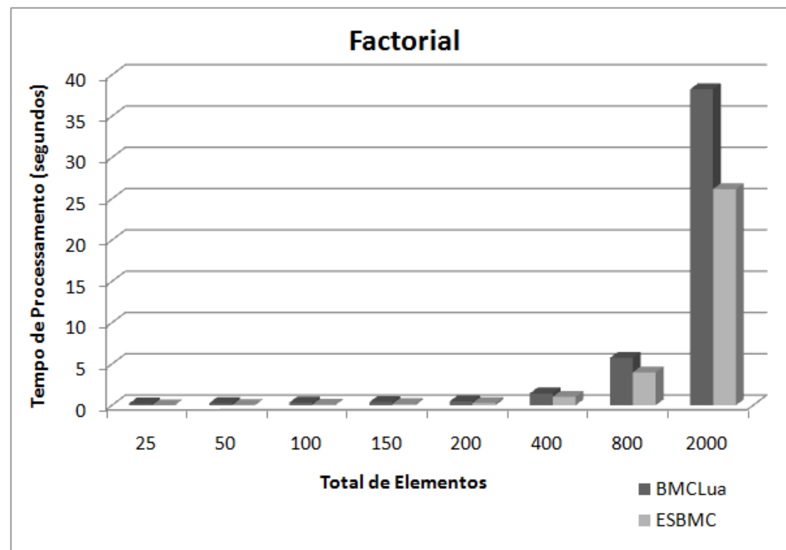


Figura 4.29: Gráfico do desempenho (segundos) para o algoritmo *Factorial*.

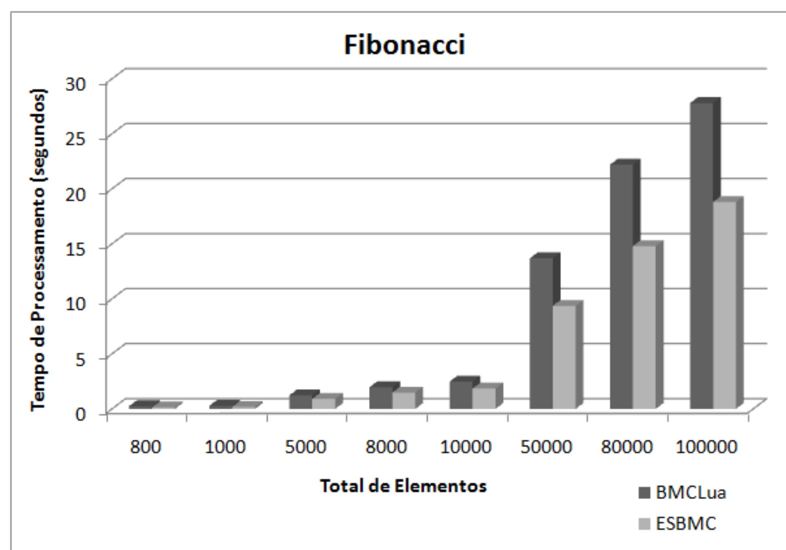


Figura 4.30: Gráfico do desempenho (segundos) para o algoritmo *Fibonacci*.

Como se pode observar nas Figura 4.29 e 4.30, o aumento dos limites de vetores para os algoritmos *Factorial* e *Fibonacci*, também resulta em um aumento do tempo de verificação na metodologia BMCLua.

Comparando as traduções realizadas pela ferramenta Lua *To Cee* e pela metodologia BMCLua, pode-se verificar que o Lua *To Cee* gerou um código com mais linhas de código que

o BMCLua, com diversas funções da API Lua para a linguagem C. O código-fonte ANSI-C, gerado pelo BMCLua, possui um número menor de linhas de código e não utiliza as funções da API Lua, tornando-o mais legível de interpretar e verificar. Além disso, não foi possível realizar a verificação do código C, gerado pelo Lua *To Cee*, utilizando a ferramenta ESBMC, pois muitas funções referenciadas, através de cláusulas *include*, no código C, não estavam disponíveis na biblioteca da ferramenta Lua *To Cee*.

Para a verificação de programas Lua, a ferramenta desenvolvida a partir da metodologia BMCLua, é mais eficiente que a ferramenta *Lua Checker*, que é restrita apenas à verificação de variáveis e constantes. Além das variáveis e constantes, BMCLua também verifica estruturas de controle (ex.: estrutura *for*), atribuições múltiplas e funções.

4.3 Resumo

Neste capítulo, a metodologia BMCLua foi apresentada. A arquitetura da metodologia foi definida para facilitar o desenvolvimento do módulo tradutor e permitir, no módulo de verificação, o uso de mais de um verificador BMC. A escolha da ferramenta ANTLR para o desenvolvimento do tradutor deve-se pela facilidade de gerar automaticamente as classes Java dos analisadores *lexer* e *parser*, e da interface de saída (*visitor*), que gera o código-fonte ANSI-C, simplificando o desenvolvimento das ferramentas, tradutor e interpretador, da metodologia BMCLua.

No módulo de verificação, foi necessário definir um verificador BMC que tivesse a característica de suportar a linguagem C/C++, gerando na saída um contraexemplo, em caso de detecção de violação no código-fonte ANSI-C. Para este trabalho, foi escolhido o verificador ESBMC, que é uma ferramenta eficiente na verificação de códigos C/C++. O ESBMC gera uma lista das iterações realizadas e um contraexemplo quando detectado uma violação no código. A metodologia BMCLua foi definida, para permitir o uso de outros verificadores BMC, como CBMC e LLBMC mencionados no Capítulo 3.

A metodologia BMCLua é o primeiro trabalho que utiliza a técnica de verificação de modelos limitados (BMC), e assim como outras ferramentas (ex.: Lua Checker), verifica programas Lua. Comparado com a ferramenta *Lua Checker*, o BMCLua possui maior abrangência na verificação de estruturas Lua, como mostrado na Tabela 4.3.

Os resultados dos experimentos apresentados na Tabela 4.2, apresentada na Seção 4.2,

Estrutura Lua	**BMCLua**	**Lua Checker**
Declaração de variáveis	SIM	SIM
Atribuições simples e múltiplas	SIM	SIM
Tabelas e estruturas de controle	SIM	NÃO
Funções	PARCIAL	NÃO
Biblioteca NCLua	PARCIAL	NÃO

Tabela 4.3: Estruturas Lua verificadas pelas ferramentas BMCLua e *Lua Checker*.

mostram a eficácia da metodologia BMCLua, para a verificação de programas Lua. Em particular, BMCLua foi capaz de detectar, em todos os *benchmarks*, propriedades relacionadas com a divisão por zero e assertivas especificadas pelo usuário. O tempo de verificação do BMCLua é maior, comparado ao do verificador ESBMC, pois possui mais as etapas de tradução e interpretação, que compõe a metodologia desenvolvida neste trabalho. Melhorias nas ferramentas de tradução e interpretação da metodologia BMCLua através de técnicas de otimização de algoritmos, como eliminação de expressões comuns CSE (do inglês, *Common Subexpression Elimination*) e propagação de constantes [41], poderão reduzir ainda mais esta diferença.

Capítulo 5

Conclusões

O trabalho realizado até o momento alcançou o objetivo esperado, que consistia na definição de uma metodologia para a verificação de códigos Lua e 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. A integração dos analisadores *lexer* e *parser*, gerados pelo ANTLR, tornou eficiente a tradução do código-fonte Lua para ANSI-C e facilitou o desenvolvimento da ferramenta, através do uso de uma gramática BNF.

Os resultados dos experimentos comprovaram a eficácia do tradutor em converter, com corretude, código-fonte Lua em código-fonte ANSI-C, mesmo que o desempenho, na etapa de verificação, tenha sido reduzido, devido ao aumento do tempo de processamento total de verificação de programas Lua. Todos os *benchmarks* verificados apresentaram tempo de verificação no BMCLua maior, quando comparado à ferramenta ESBMC. Isso pode ser minimizado através da otimização dos módulos tradutor e interpretador, da metodologia BMCLua. Algumas técnicas de otimização de algoritmos, como a eliminação de expressões comuns CSE (do inglês, *Common Subexpression Elimination*) e propagação de constantes [41, 58], serão utilizados para otimizar o código ANSI-C gerado pelo tradutor, permitindo reduzir o tempo de verificação pelas ferramentas BMC.

Como a ferramenta BMCLua, baseada na metodologia descrita neste trabalho, está em desenvolvimento, novas estruturas do Lua serão implementadas no tradutor, com o objetivo de cobrir toda a sintaxe da linguagem. As principais estruturas do Lua já estão implementadas na

ferramenta BMCLua, apesar de algumas dessas estruturas possuírem variações de sintaxe, que ainda devem ser programadas no tradutor. Por exemplo, a estrutura de controle *for* pode ser determinística no limite de laços ou utilizar o objeto *iterator*, para percorrer uma lista dinâmica. Para o desenvolvimento atual falta, implementar a tradução da sintaxe do controle *for* para listas dinâmicas.

Também será implementado a tradução da biblioteca de extensão NCLua, para poder contemplar códigos Lua de aplicações interativas para TV digital.

A Tabela 5.1 mostra as estruturas suportadas pela metodologia BMCLua.

Estrutura Lua	**Suportada pelo BMCLua**
Tipos primitivos (*boolean, number, string*)	SIM
Conversão de tipos	SIM
Operadores relacional e lógicos	SIM
Operadores unários	SIM
Tabelas	SIM
Atribuições simples e múltiplas	SIM
Estruturas de controle	SIM
Definição de função	PARCIAL
Chamada de função e objeto	PARCIAL
Declaração das funções da biblioteca Lua	NÃO
NCLua	PARCIAL

Tabela 5.1: Estruturas Lua definidas para o BMCLua.

5.1 Trabalhos Futuros

Para trabalhos futuros, serão implementadas estruturas da linguagem Lua ainda não suportadas pelo BMCLua: conversão de tipos, chamadas de função e objeto, declarações de funções das bibliotecas Lua e NCLua. Algumas técnicas de otimização de algoritmos, como eliminação de expressões comuns CSE e propagação de constantes [41], serão utilizados para otimizar o código ANSI-C gerado pelo tradutor, permitindo reduzir o tempo de verificação pelas ferramentas BMC.

A próxima etapa de desenvolvimento é incorporar a API SMT-LIB [59] à metodologia BMCLua, que permitirá incrementar o número de solucionadores SMT (ex.: Z3). Como a API SMT-LIB foi desenvolvida em Java, será possível integrá-la ao pacote de bibliotecas do

BMCLua, permitindo ao pesquisador focar sua atenção nas implementações das estruturas Lua para o uso dos solucionadores SMT.

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. Essa integração possibilitará minimizar o tempo de desenvolvimento da aplicação.

Referências Bibliográficas

[1] KURT, J.; BROWN, A. *Beginning Lua Programming*. 1. ed. Indiana, Estados Unidos: Wisley Publishing, Inc., 2007. 644 p.

[2] BRANDÃO, R. et al. Extended Features for the Ginga-NCL Environment: Introducing the LuaTV API. In: *IEEE Conference Publications*. Zurich: Proceedings of 19th International Conference on Computer Communications and Networks, 2010. p. 1–6.

[3] SOARES, L.; BARBOSA, S. *Programando em NCL 3.0: Desenvolvimento de Aplicações para Middleware Ginga, TV digital e Web*. 1. ed. São Paulo, Brasil: Editora Campus, 2009. 341 p.

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

[5] BARBOZA, D. C.; CLUA, E. W. G. Ginga game: A framework for game development for the interactive digital television. In: . Los Alamitos, CA, USA: IEEE Computer Society, 2009. p. 162–167. ISBN 978-0-7695-3963-8.

[6] MEGRICH, A. *Televisão digital: princípios e técnicas*. 1. ed. São Paulo, Brasil: Érica, 2009. 336 p.

[7] ALENCAR, M. *Televisão digital*. 1. ed. São Paulo, Brasil: Érica, 2007. 352 p.

[8] LUCENA JR., V. F. de et al. Designing an extension API for bridging Ginga iDTV applications and home services. *IEEE Transactions on Consumer Electronics*, IEEE, v. 58, p. 1077–1085, ago. 2012.

[9] SALVIATO, T. P. et al. Framework for context-aware applications on the brazilian digital tv. In: *The 4th International Conference on Ubi-media Computing (U-Media 2011)*. São Paulo, Brazil: [s.n.], 2011. p. 112–117.

[10] KOPETZ, H. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. 2. ed. University of Virginia, Virginia, USA: Series Editor, 2011.

[11] KOSCIANSKI, A.; SOARES, M. *Qualidade de Software*. 2. ed. São Paulo: Novatec, 2007.

[12] SOMMERVILE, I. *Software Engineering*. 9. ed. Estados Unidos: Pearson Education, Inc., 2011. 790 p.

[13] BAIER, C.; KATOEN, J.-P. *Principles of Model Checking*. 1. ed. Estados Unidos: The MIT Press, 2008. 984 p.

[14] CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transaction of Software Engineering*, v. 38, n. 6, p. 957–974, 2012.

[15] CORDEIRO, L. *SMT-Based Bounded Model Checking of Multi-threaded Software in Embedded Systems*. Southampton, Reino Unido: University of Southampton, 2011. 197 p.

[16] CORDEIRO, L. et al. Context-bounded model checking with esbmc 1.17 - (competition contribution). In: *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)*. Tallinn, Estonia: [s.n.], 2012. v. 7214, p. 534–537.

[17] MORSE, J. et al. Handling unbounded loops with esbmc 1.20 - (competition contribution). In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*. Roma, Itália: [s.n.], 2013. v. 7795, p. 619–622.

[18] MORSE, J. et al. Esbmc 1.22 - (competition contribution). In: *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*. Grenoble, França: [s.n.], 2014. v. 8413, p. 405–407.

[19] CLARKE, E.; KROENING, D.; LERDA, F. A tool for checking ANSI-C programs. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. [S.l.]: Springer, 2004. (Lecture Notes in Computer Science, v. 2988), p. 168–176. ISBN 3-540-21299-X.

[20] FALKE, S.; MERZ, F.; SINZ, C. LLBMC: Improved Bounded Model Checking of C Programs Using LLVM. In: *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.]: Springer Berlin Heidelberg, 2013. v. 7795, p. 623–626.

[21] LAWS(Laboratory of Advanced Web Systems) – UFMA. *NCL Eclipse*. 2014. Disponível em: <http://laws.deinf.ufma.br/ncleclipse/pt-br:start>. Acesso em: 16 março 2015.

[22] AZEVEDO, R.; TEIXEIRA, M.; NETO, C. S. *NCL Eclipse: Ambiente Integrado para o Desenvolvimento de Aplicações para TV Digital Interativa em Nested Context Language*. 2009. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/sbrc/2009/087.pdf>. Acesso em: 16 março 2015.

[23] INTERNATIONAL BUSINESS MACHINES CORP. *Eclipse platform technical overview*. 2006. Disponível em: <http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf>. Acesso em: 16 março 2015.

[24] MOURA, L.; BJØRNER, N. Z3: An efficient SMT solver. In: *Lecture Notes in Computer Science*. Budapeste, Hungria: Springer, 2008. v. 4963, p. 337–340.

[25] BIERE, A. et al. Bounded model checking. In: *Handbook of Satisfiability*. [S.l.]: IOS Press, 2009. p. 457–481.

[26] PARR, T. *The Definitive ANTLR 4 Reference*. 1. ed. Texas, Estados Unidos: The Pragmatic Bookshelf, 2013. 328 p.

[27] 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, 2007.

[28] KNUTH, D. Backus normal form versus backus naur form. In: *Selected Papers on Computer Languages*. [S.l.]: CSLI-Center for Study of Language and Information, 2011. p. 96–97.

[29] CLARKE, E.; GRUMBERG, O.; PELED, D. *Model Checking*. 1. ed. Estados Unidos: MIT Press, 1999. 314 p.

[30] BURNS, A.; WELLINGS, A. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. 3. ed. Estados Unidos: Addison Wesley, 2001. 784 p.

[31] HERBERT, S. *C Completo e Total*. 3. ed. São Paulo, Brasil: Makron Books, 1996. 827 p.

[32] STROUSTRUP, B. *The C++ Programming Language*. 3. ed. Estados Unidos: ATT, 1997. 1040 p.

[33] DEITEL, P.; DEITEL, H. *Java: How to Program*. 8. ed. Estados Unidos: Prentice Hall, 2009. 1184 p.

[34] PEDRONI, V. *Circuit Design with VHDL*. 1. ed. Estados Unidos: MIT Press, 2004. 363 p.

[35] HUTH, M.; RAYN, M. *Logic in Computer Science: Modelling and Reasoning about Systems*. 2. ed. Estados Unidos: Cambridge Univerty Press, 2004. 440 p.

[36] PNUELI, A. The temporal logic of programs. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1977. (SFCS '77, v. 33), p. 46–57.

[37] BARRETT, C. et al. Satisfiability modulo theories. In: *Handbook of Satisfiability*. [S.l.]: IOS Press, 2009. v. 185, p. 825–885.

[38] MOURA, L.; BJØRNER, N. *Satisfiability Modulo Theories: An Appetizer*. 2009. Disponível em: <http://leodemoura.github.io/files/sbmf09.pdf>. Acesso em: 16 março 2015.

[39] HIISCHI, A. Traveling Light, the Lua Way. In: . [S.l.]: IEEE Computer Society, 2007. p. 31–38.

[40] CORMEN, T. et al. *Algoritmos: teoria e prática*. 6. ed. Rio de Janeiro: Editora Campus, 2002. 916 p.

[41] AHO, A. et al. *Compilers: Principles, Techniques and Tools*. 2. ed. Estados Unidos: Addison Wesley, 2007. 796 p.

[42] LATTNER, C.; ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. Washington, DC, USA: IEEE Computer Society, 2004.

[43] DILL, D. L.; GANESH, V. A decision procedure for bit-vectors and arrays. In: . [S.l.]: Springer, 2007. (Lecture Notes in Computer Science, v. 4590), p. 519–531.

[44] STUMP, A.; BARRETT, C. W.; DILL, D. L. CVC: A cooperating validity checker. In: *Proceedings of the 14th International Conference on Computer Aided Verification*. [S.l.]: Springer-Verlag, 2002. (Lecture Notes in Computer Science, v. 2404), p. 500–504. Copenhagen, Denmark.

[45] BRUMMAYER, R.; BIERE, A. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: *Lecture Notes in Computer Science (LNCS)*. Austria: Springer, 2009. v. 5505, p. 174–177.

[46] HAVELUND, K.; PRESSBURGER, T. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, Springer-Verlag, v. 2, p. 366–381, 2000. ISSN 1433-2779.

[47] HOLZMANN, G.; BOSNACKI, D. The design of a multicore extension of the spin model checker. In: *IEEE Transactions on Software Engineering*. Estados Unidos: [s.n.], 2007. v. 33, p. 659–674.

[48] HOLZMANN, G. Software model checking with spin. In: *Advances in Computers*. Estados Unidos: [s.n.], 2005. v. 65, p. 78–109.

[49] MANURA, D. *Lua To Cee*. 2012. Disponível em: <http://lua-users.org/wiki/LuaToCee>. Acesso em: 16 março 2015.

[50] SMITH, R. *Lua-Checker: Check Lua source code for various errors.* 2008. Disponível em: <https://code.google.com/p/lua-checker/>. Acesso em: 16 março 2015.

[51] AUBRY, M.; BERNARD, S. *Lua development tools*. 2014. Disponível em: <https://wiki.eclipse.org/LDT>. Acesso em: 16 março 2015.

[52] KLINT, P.; ROOSENDAAL, L.; ROZEN, R. van. Game developers need lua air: static analysis of lua using interface models. In: *Proceedings of the 11th International Conference on Entertainment Computing*. Bremen, Alemanha: [s.n.], 2012. p. 530–535.

[53] MANURA, D. *Lua Inspect*. 2012. Disponível em: <http://lua-users.org/wiki/LuaInspect>. Acesso em: 16 março 2015.

[54] FLEUTOT, F.; TRATT, L. Contrasting compile-time meta-programming in metalua and converge. In: *Proceedings of the International on Workshop on Dynamic Languages and Applications*. Berlim, Alemanha: [s.n.], 2007.

[55] HILLS, M.; KLINT, P.; VINJU, J. J. Program analysis scenarios in rascal. In: *Proceedings of the 9th International Workshop on Rewriting Logic and Its Applications (WRLA 2012)*. [S.l.]: Springer, 2012. (Lecture Notes in Computer Science). Invited Paper.

[56] HARTEL, P.; MULLER, H. *Simple algebraic data types for C*. 2010. Disponível em: <http://eprints.eemcs.utwente.nl/17771/01/adt8.pdf>. Acesso em: 16 março 2015.

[57] SAKAMOTO, K. *Grammars written for ANTLR v4*. 2013. Disponível em: <https://github.com/antlr/grammars-v4/tree/master/lua>. Acesso em: 16 março 2015.

[58] PLATZER, A. *Lecture Notes on Basic Optimizations*. 2014. Disponível em: <http://symbolaris.com/course/Compilers11/14-opt.pdf>. Acesso em: 16 março 2015.

[59] BARRETT, C.; STUMP, A.; TINELLI, C. *The SMT-LIB Standard: Version 2.0*. 1. ed. The University of Iowa: Department of Computer Science, 2010.

Apêndice A

Publicações

A.1 Referente à Pesquisa

- **Francisco Januário**, Lucas Cordeiro e Eddie Filho. *Verificação de Códigos Lua Utilizando BMCLua*. **XXXI Simpósio Brasileiro de Telecomunicações - SBrT2013**, Fortaleza, 2013.
  DOI: 10.14209/sbrt.2013.92

- **Francisco Januário**, Lucas Cordeiro, Vicente Lucena e Eddie Filho. *BMCLua: Verification of Lua Programs in Digital TV Interactive Applications*. **2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE 2014)**, Tokyo, 2014. IEEE, p. 707–708.
  DOI: 10.1109/GCCE.2014.7031344

- **Francisco Januário**, Lucas Cordeiro, Vicente Lucena e Eddie Filho. *BMCLua: Verificação de Programas Lua em Aplicações Interativas de TV Digital*. **IV Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC 2014)**, Manaus, 2014.
  Disponível em: <http://sbesc.lisha.ufsc.br/sbesc2014/dl181>

Apêndice B

Gramática Lua.g4

```
1  /*
2  BSD License
3
4  Copyright (c) 2013, Kazunori Sakamoto
5  All rights reserved.
6
7  Redistribution and use in source and binary forms, with or without
8  modification, are permitted provided that the following conditions
9  are met:
10
11 1. Redistributions of source code must retain the above copyright
12    notice, this list of conditions and the following disclaimer.
13 2. Redistributions in binary form must reproduce the above copyright
14    notice, this list of conditions and the following disclaimer in the
15    documentation and/or other materials provided with the distribution.
16 3. Neither the NAME of Rainer Schuster nor the NAMEs of its contributors
17    may be used to endorse or promote products derived from this software
18    without specific prior written permission.
19
20 THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
21 "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
22 LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
23 A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
24 HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
25 SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
26 LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
27 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
28 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
29 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
30 OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
31
```

```
32 This grammar file derived from:
33
34     Lua 5.2 Reference Manual
35     http://www.lua.org/manual/5.2/manual.html
36
37     Lua 5.1 grammar written by Nicolai Mainiero
38     http://www.antlr3.org/grammar/1178608849736/Lua.g
39
40 I tested my grammar with Test suite for Lua 5.2 (http://www.lua.org/tests/5.2/)
41 */
42
43 grammar Lua;
44
45 @lexer::members {
46     public static final int WHITESPACE = 1;
47 }
48
49 chunk
50     : block EOF # blockChunk
51     ;
52
53 block
54     : stat* retstat? # statBlock
55     ;
56
57 stat
58     : ';' # l_1
59     | varlist '=' explist # assignMul
60     | functioncall # l_2
61     | label # l_3
62     | 'break' # breakStat
63     | 'goto' NAME # gotoStat
64     | 'do' block 'end' # doStat
65     | 'while' exp 'do' block 'end' # whileStat
66     | 'repeat' block 'until' exp # repeatStat
67     | 'if' exp 'then' block ('elseif' exp 'then' block)* ('else' block)? 'end' #
          ifStat
68     | 'for' NAME '=' exp ',' exp (',' exp)? 'do' block 'end' # forStat
69     | 'for' namelist 'in' explist 'do' block 'end' # forInStat
70     | 'function' funcname funcbody # functionStat
71     | 'local' 'function' NAME funcbody # l_13
72     | 'local' namelist ('=' explist)? # varLocal
73     ;
74
75 retstat
```

```
76      : 'return' explist? ';'? # returnStat
77      ;
78
79 label
80      : '::' NAME '::'
81      ;
82
83 funcname
84      : NAME ('.' NAME)* (':' NAME)?
85      ;
86
87 varlist
88      : var (',' var)*
89      ;
90
91 namelist
92      : NAME (',' NAME)*
93      ;
94
95 explist
96      : exp (',' exp)*
97      ;
98
99 exp
100     : 'nil'
101     | 'false'
102     | 'true'
103     | number
104     | string
105     | '...'
106     | functiondef
107     | prefixexp
108     | tableconstructor
109     | exp binop exp
110     | unop exp
111     ;
112
113 var
114     : (NAME | '(' exp ')' varSuffix) varSuffix*
115     ;
116
117 prefixexp
118     : varOrExp nameAndArgs*
119     ;
120
```

```
121 functioncall
122     : varOrExp nameAndArgs+ # funcCallStat
123     ;
124
125 varOrExp
126     : var | '(' exp ')'
127     ;
128
129 nameAndArgs
130     : (':' NAME)? args
131     ;
132
133 varSuffix
134     : nameAndArgs* ('[' exp ']' | '.' NAME)
135     ;
136
137 args
138     : '(' explist? ')' | tableconstructor | string
139     ;
140
141 functiondef
142     : 'function' funcbody
143     ;
144
145 funcbody
146     : '(' parlist? ')' block 'end'
147     ;
148
149 parlist
150     : namelist (',' '...')? | '...'
151     ;
152
153 tableconstructor
154     : '{' fieldlist? '}' # ruleTable
155     ;
156
157 fieldlist
158     : field (fieldsep field)* fieldsep?
159     ;
160
161 field
162     : '[' exp ']' '=' exp
163     | NAME '=' exp
164     | exp
165     ;
```

```
166
167 fieldsep
168     : ',' | ';'
169     ;
170
171 binop
172     : '+' | '-' | '*' | '/' | '^' | '%' | '..' | '<'
173     | '<=' | '>' | '>=' | '==' | '~=' | 'and' | 'or'
174     ;
175
176 unop
177     : '-' | 'not' | '#'
178     ;
179
180 number
181     : INT | HEX | FLOAT | HEX_FLOAT
182     ;
183
184 string
185     : NORMALSTRING | CHARSTRING | LONGSTRING
186     ;
187
188 // LEXER
189
190 NAME
191     : [a-zA-Z_][a-zA-Z_0-9]*
192     ;
193
194 NORMALSTRING
195     : '"' ( EscapeSequence | ~('\\'|'"') )* '"'
196     ;
197
198 CHARSTRING
199     : '\'' ( EscapeSequence | ~('\''|'\\') )* '\''
200     ;
201
202 LONGSTRING
203     : '[' NESTED_STR ']'
204     ;
205
206 fragment
207 NESTED_STR
208     : '=' NESTED_STR '='
209     | '[' .*? ']'
210     ;
```

```
211
212 INT
213     : Digit+
214     ;
215
216 HEX
217     : '0' [xX] HexDigit+
218     ;
219
220 FLOAT
221     : Digit+ '.' Digit* ExponentPart?
222     | '.' Digit+ ExponentPart?
223     | Digit+ ExponentPart
224     ;
225
226 HEX_FLOAT
227     : '0' [xX] HexDigit+ '.' HexDigit* HexExponentPart?
228     | '0' [xX] '.' HexDigit+ HexExponentPart?
229     | '0' [xX] HexDigit+ HexExponentPart
230     ;
231
232 fragment
233 ExponentPart
234     : [eE] [+-]? Digit+
235     ;
236
237 fragment
238 HexExponentPart
239     : [pP] [+-]? Digit+
240     ;
241
242 fragment
243 EscapeSequence
244     : '\\' [abfnrtvz"'\\]
245     | DecimalEscape
246     | HexEscape
247     ;
248
249 fragment
250 DecimalEscape
251     : '\\' Digit
252     | '\\' Digit Digit
253     | '\\' [0-2] Digit Digit
254     ;
255
```

```
256 fragment
257 HexEscape
258     : '\\' 'x' HexDigit HexDigit
259     ;
260
261 fragment
262 Digit
263     : [0-9]
264     ;
265
266 fragment
267 HexDigit
268     : [0-9a-fA-F]
269     ;
270
271 COMMENT
272     : '--[[' .*? ']]'-> channel(HIDDEN)
273     ;
274
275 LINE_COMMENT
276     : '--' '['? (~('['|'\n'|'\r') ~('\n'|'\r')*)? ('\n'|'\r')* -> skip
277     ;
278
279 WS
280     : [ \t\u000C]+ -> channel(WHITESPACE)
281     ;
282
283 NEWLINE
284     : '\r'? '\n' -> skip
285     ;
```

Apêndice C

Estruturas Lua Não Implementadas no Tradutor BMCLua

Este apêndice lista as estruturas e funções da biblioteca padrão da linguagem Lua que não estão implementadas no tradutor da ferramenta BMCLua.

A Tabela C.1 lista as estruturas da da linguagem Lua que faltam ser implementadas no tradutor do BMCLua.

Estrutura Lua	Sintaxe em Implementação
Estrutura de dados *Table*	Tabela Hash (Ex.: t={x=5, y=10})
	Mista (Ex.: t={x=5, y=10; “yes”, “no”})
	Tabelas Aninhadas (Ex.: t={msg=“choice”,{“yes”,“no”,“?”}})
Instrução de atribuição	Troca de Valores (Ex.: a, b = b, a)
Estrutura de controle *for*	*for VARS in ITERATOR do BLOCK end*
Definição de função	*f = function (ARGS) BODY [return VALUES] end*
	function t.NAME (ARGS) BODY [return VALUES] end
	function obj:NAME (ARGS) BODY [return VALUES] end
Chamada de função	Abreviado (Ex.: *f “hello”*)
	Abreviado (Ex.: *f {x=3,y=4}*)
	Chamada de função associada a tabela (Ex.: *t.f(x)*)
	Chamada de objeto (Ex.: *x.move(2,-3)*)

Tabela C.1: Estruturas faltantes no tradutor do BMCLua.

As Tabelas de C.2 à C.10, listam as funções da biblioteca padrão que faltam ser implementadas no tradutor do BMCLua. As funções são apenas declaradas no código ANSI-C gerado pelo tradutor, não sendo necessária a implementação do código.

Biblioteca Básica	**Funções**
Operadores de Metatable	setmetatable(t,mt); getmetatable(t) rawget(t,i); rawset(t,l,v); rawequal(t1,t2)
Variáveis Globais e de Ambiente	getfenv([f]); setfenv(f,t)
Carregando e Executando	require(pkgname); dofile(filename) load(func [,chunkname]); loadfile(filename) loadstring(s [,name]); pcall(f [,args]); xpcall(f, h)
Saída e FeedBack Erro	print(args); error(msg [,n]); assert(v [,msg])
Informação e Conversão	select(index,...); type(x); tostring(x) tonumber(x, [,b]); unpack(t)
ITERATOR do *for*	ipairs(t); pairs(t); next(t [,inx])
Coletor de Lixo	collectorgarbage(opt [,arg])

Tabela C.2: Funções da biblioteca básica faltantes no tradutor do BMCLua.

Biblioteca de Pacotes	**Funções**
Package	module(name, ...); package.loadlib(lib, func) package.path; package.cpath, package.loaded package.preload; package.seeall(module)

Tabela C.3: Funções da biblioteca de pacotes faltantes no tradutor do BMCLua.

Biblioteca de Co-rotinas	**Funções**
Coroutine	coroutine.create(f); coroutine.resume(co, args) coroutine.yield(args); coroutine.status(co) coroutine.running(); coroutine.wrap(f)

Tabela C.4: Funções da biblioteca de co-rotinas faltantes no tradutor do BMCLua.

Biblioteca de Tabelas	**Funções**
Table	table.insert(t,[l,]v); table.remove(t [,i]); table.maxn(t) table.sort(t [,cf]); table.concat(t [,s [,l [,j]]])

Tabela C.5: Funções da biblioteca de co-rotinas faltantes no tradutor do BMCLua.

Biblioteca Matemática (MATH)	**Funções**
Operações básicas	math.abs(x); math.mod(x,y); math.floor(x) math.ceil(x); math.min(x,y); math.max(x)
Exponencial e logaritmicos	math.sqrt(x); math.pow(x,y); math.exp(x) math.log(x); math.log10(x,y)
Trigonometria	math.deg(a); math.rad(a); math.pi; math.sin(a) math.cos(a); math.tan(a); math.asin(x) math.acos(x); math.atan(x); math.atan2(x,y)
Divisor por potência de 2	math.frexp(x); math.ldexp(x)
Números pseudo-randomicos	math.random([n [,m]]); math.randomseed(n)

Tabela C.6: Funções da biblioteca de matemática faltantes no tradutor do BMCLua.

Biblioteca String	**Funções**
Operações básicas	string.len(s); string.sub(s, l [,j]) string.rep(s, n); string.upper(s); string.lower(s)
Códigos de caracteres	string.byte(s [,i]); string.char(args)
Storage	string.dump(f)
Formatação de String	string.formats(s [,args])
Pesquisa e substituição	string.find(s, p [,l [,d]]); string.gmatch(s, p) string.gsub(s, p, r [,n]); string.match(s, p [,i])

Tabela C.7: Funções da biblioteca *String* faltantes no tradutor do BMCLua.

Biblioteca de Entrada/Saída (I/O)	**Funções**
Completo I/O	io.open(fn [,m]); file:close() file:read(formats); file:write(values) file:lines; file:seek([p][,of])
Simples I/O	io.input([file]); io.output([file]); io.close([file]) io.read(formats); io.lines([fn]); io.flush()
Arquivos Padrões e Funções Úteis	io.stdin; io.stdout; io.stderr io.popen([prog [,mode]]); io.type(x); io.tmpfile()

Tabela C.8: Funções da biblioteca de entrada/saída faltantes no tradutor do BMCLua.

Biblioteca do Sistema Operaciona (OS)	**Funções**
Interação com o Sistema	os.execute(cmd); os.exit([code]); os.getenv(var) os.setlocale(s [,c]); os.remove(fn) os.rename(of, nf); os.tmpname()
Data e Hora	os.clock(); os.time([tt]) os.date([fmt [,t]]); os.difftime(t2, t1)

Tabela C.9: Funções da biblioteca OS faltantes no tradutor do BMCLua.

Biblioteca Debug	**Funções**
Funções básicas	debug.debug(); debug.getinfo(f [,w]) debug.getlocal(n,i); debug.getupvalue(f,i) debug.traceback([msg]); debug.setlocal(n,l,v) debug.setvalue(f,l,v); debug.sethook([h, m [,n]])

Tabela C.10: Funções da biblioteca debug faltantes no tradutor do BMCLua.

Apêndice D

Biblioteca NCLua Não Implementada no Tradutor BMCLua

A Tabela D.1 lista os módulos da biblioteca NCLua [27] que não estão implementadas no tradutor BMCLua e que são utilizadas no desenvolvimento de aplicações interativas para a TV digital.

Biblioteca NCLua	Funções
Módulo *ncledit*	ncledit.openBase; ncledit.activateBase ncledit.deactivateBase; ncledit.saveBase ncledit.closeBase; ncledit.addDocument ncledit.removeDocument; ncledit.startDocument ncledit.stopDocument; ncledit.pauseDocument ncledit.resumeDocument; ncledit.addRegion ncledit.removeRegion; ncledit.addRegionBase ncledit.removeRegionBase; ncledit.addRule ncledit.removeRule; ncledit.addRuleBase ncledit.removeRuleBase; ncledit.addConnector ncledit.removeConnector; ncledit.addConnectorBase ncledit.removeConnectorBase; ncledit.addDescriptor ncledit.removeDescriptor; ncledit.addDescriptorSwitch ncledit.removeDescriptorSwitch; ncledit.addDescriptorBase

Biblioteca NCLua	**Funções**
Módulo *ncledit*	ncledit.removeDescriptorBase ncledit.addTransition ncledit.removeTransition ncledit.addTransitionBase ncledit.removeTransitionBase ncledit.addImportBase ncledit.removeImportBase ncledit.addImportDocumentBase ncledit.removeImportDocumentBase ncledit.addImportNCL ncledit.removeImportNCL; ncledit.addNode ncledit.removeNode; ncledit.addInterface ncledit.addLink; ncledit.removeLink ncledit.setPropertyValue
Módulo *canvas*	canvas:new; canvas:attrSize; canvas:attrClip canvas:drawLine; canvas:drawPolygon; canvas:drawEllipse canvas:compose; canvas:pixel; canvas:measureText
Módulo *event*	event.post; event.timer event.register; event.unregister; event.uptimer
Módulo *settings*	A *table* settings.system A *table* settings.user A *table* settings.default A *table* settings.service
Módulo *persistent*	A *table* persistent.service A *table* persistent.channel A *table* persistent.shared

Tabela D.1: Funções da biblioteca NCLua faltantes no tradutor do BMCLua.

Apêndice E

Benchmarks em Lua

E.1 Bellman-ford.lua

```
1 INFINITY = 899
2 nodecount = 5
3 edgecount = 5
4 source = 0
5 local Source = {[0]=0,[1]=0,[2]=0,[3]=3,[4]=4}
6 local Dest = {[0]=1,[1]=0,[2]=0,[3]=1,[4]=1}
7 local Weight = {[0]=0,[1]=1,[2]=2,[3]=3,[4]=4}
8 local distance = {[0]=0,[1]=0,[2]=0,[3]=0,[4]=0}
9 x, y, i, j = 0, 0, 0, 0
10
11 for i = 0, nodecount-1 do
12     if i == source then
13        distance[i] = 0
14     else
15        distance[i] = INFINITY
16     end
17 end
18 for i = 0, nodecount-1 do
19     for j = 0, edgecount-1 do
20         x = Dest[j]
21         y = Source[j]
22         if distance[x] > (distance[y] + Weight[j]) then
23            distance[x] = distance[y] + Weight[j]
24         end
25     end
26 end
27 for i = 0, edgecount-1 do
28     x = Dest[i]
```

```
29     y = Source[i]
30     if distance[x] > (distance[y] + Weight[i]) then
31        return
32     end
33 end
34 for i = 0, nodecount-1 do
35     assert(distance[i]>=0)
36 end
```

E.2 Prim.lua

```
1 INFINITY = 899
2 local Source = {[0]=3,[1]=2,[2]=1,[3]=0,[4]=0}
3 local Dest = {[0]=0,[1]=3,[2]=2,[3]=1,[4]=3}
4 local Weight = {[0]=4,[1]=3,[2]=2,[3]=1,[4]=4}
5 nodecount = 5
6 edgecount = 5
7 local ResultNodes = {[0]=0,[1]=0,[2]=0,[3]=0,[4]=0}
8 local ResultEdges = {[0]=0,[1]=0,[2]=0,[3]=0}
9 i, j, k, k_1, h, sourceflag, destflag, min = 0, 0, 0, 0, 0, 0, 0, 0
10 local visited = {[0]=0,[1]=0,[2]=0,[3]=0,[4]=0}
11 ResultNodes[0] = 0
12
13 for i = 1, nodecount-1 do
14     ResultNodes[i] = INFINITY
15 end
16 for i = 0, nodecount-2 do
17     ResultEdges[i] = INFINITY
18 end
19
20 k = 0
21 while k ~= nodecount-1 do
22    min = 0
23    for i = 0, edgecount-1 do
24        visited[i] = Weight[i]
25    end
26    for k_1 = 0, edgecount-1 do
27        for h = 0, edgecount-1 do
28            if visited[h] < visited[min] then
29                min = h
30            end
31        end
32        sourceflag = 0
```

```
33        for j = 0, nodecount-1 do
34            if Source[min] == ResultNodes[j] then
35                sourceflag = 1
36            end
37        end
38        destflag = 1
39        for j = 0, nodecount-1 do
40            if  Dest[min] == ResultNodes[j] then
41                destflag = 0
42            end
43        end
44        if sourceflag == 0 and destflag == 0 then
45            visited[min] = 889
46        end
47    end
48    ResultEdges[k] = min
49    ResultNodes[k+1] = Dest[min]
50    Weight[min] = INFINITY
51    k = k + 1
52 end
53 assert(k==4)
```

E.3 BubbleSort.lua

```
1 i, j = 0, 0
2 n = 12
3 -- array com n + 1 elementos
4 local array = {0,0,0,0,0,0,0,0,0,0,0,0,0}
5
6 for i=1, n do
7     array[i] = n-i
8 end
9 temp = 0
10 k = 0
11 for j=1, n do
12   for k=1, n-j do
13     if array[k] > array[k+1] then
14        temp = array[k]
15        array[k] = array[k+1]
16        array[k+1] = temp
17     end
18   end
19 end
```

```
20 for i=1, n do
21     assert(array[i] == i-1)
22 end
```

E.4 SelectionSort.lua

```
1 i, j = 0, 0
2 n = 12
3 -- array com n + 1 elementos
4 local array = {0,0,0,0,0,0,0,0,0,0,0,0,0}
5
6 for i = 1, n do
7     array[i] = n-i+1
8 end
9 min = 0
10 temp = 0
11 for j = 1, n do
12     min = j
13     for i = j+1, n do
14         if array[i] < array[min] then
15             min = i
16         end
17      end
18      temp = array[j]
19      array[j] = array[min]
20      array[min] = temp
21 end
22 for i = 1, n do
23     assert(array[i] == i)
24 end
```

E.5 InsertSort.lua

```
1 local a = {[0]=0,[1]=10,[2]=9,[3]=8,[4]=7,[5]=6,[6]=5,[7]=4,[8]=3,[9]=2}
2 tot = 9
3 i = 2
4 temp = 0
5 while i <= tot do
6    j = i
7    while a[j] < a[j-1] do
8       temp = a[j]
```

```
 9         a[j] = a[j-1]
10         a[j-1] = temp
11         j = j - 1
12     end
13     i = i + 1
14 end
```

E.6 Factorial.lua

```
1 fac = 1
2 N = 2000
3 while N ~= 1 do
4    fac = fac * N
5    N = N - 1
6 end
```

E.7 Fibonacci.lua

```
 1 a = 5000
 2 i, temp = 2, 0
 3 Fnew, Fold = 1, 0
 4
 5 while i <= a do
 6   temp = Fnew
 7   Fnew = Fnew + Fold
 8   Fold = temp
 9   i = i + 1
10 end
11
12 assert(i == a + 1)
```