UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Iverson Luís Pereira

Uma Abordagem para Tradução de uma Linguagem de Programação de Robôs para um Modelo Formal

Recife

Agosto, 2018

Iverson Luís Pereira

Uma Abordagem para Tradução de uma Linguagem de Programação de Robôs para um Modelo Formal

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal Rural de Pernambuco – UFRPE

Departamento de Computação

Curso de Bacharelado em Ciência da Computação

Orientador: Professor Sidney de Carvalho Nogueira

Recife

Agosto, 2018

Dedico este trabalho aos meus amados pais, Rosânia e Ivanildo, e ao meu grande irmão, Igor. Os meus maiores presentes.

Agradecimentos

Agradeço à minha mãe, Rosânia, por sempre me apoiar e me incentivar nas horas mais difíceis. Obrigado por sua dedicação à família. Você é o meu maior tesouro, a minha maior fortaleza. Te amo, minha rainha.

Agradeço ao meu pai, Ivanildo, que tanto lutou e possibilitou a nossa família uma vida melhor na cidade grande. Onde, pude ter a oportunidade de ir à escola e mais tarde à Universidade. Tenho muito orgulho de você, pai. Te amo!

Agradeço ao meu grande e único irmão, Igor, a quem tenho muito carinho, e sua esposa Naiany por sempre estarem ao meu lado.

Agradeço aos meus avós, tias e primos(as), que de alguma forma contribuíram em minha jornada e compreenderam minha ausência pelo tempo dedicado aos estudos.

Aos amigos da vida pela força e torcida para que tudo desse certo. Especialmente, Artur, Igor Souza, Kamila, Mariana e Osmar. Vocês são os irmãos que a vida me presenteou.

Ao meu orientador, Sidney Nogueira, pela dedicação, pela paciência, por acreditar e motivar nosso trabalho, pelo apoio constante e pela atenção durante todo o período de construção deste trabalho. À quem tenho grande admiração e respeito. Meu muito obrigado!

Ao meu tutor, Mike, e ao grupo PET-Ciranda da Ciência, por me incentivar nas atividades ensino, pesquisa e extensão, atividades essenciais durante minha trajetória acadêmica.

Meus sinceros agradecimentos a Universidade Federal Rural de Pernambuco e a todos os professores do curso de Bacharelado em Ciência da Computação. Aos meus amigos e companheiros de curso, em especial, Ana Juriti, Bruno Marques, Carlos Magnum, Danielly Queiroz, Jeremias Leite e Ricardo Luna. Algumas das pessoas mais inteligentes e fascinantes que tive a oportunidade de conviver.

Agradeço ao Núcleo de Tecnologia da Informação - NTI/UFPE que me deu a oportunidade de aplicar na prática todo o conhecimento que adquiri durante a minha graduação, além de aprender com colegas de trabalho incríveis.

Finalmente, agradeço à Deus por ter me dado forças e me fazer querer continuar. Sou extremamente grato à Ele, por ter me proporcionado momentos gloriosos com pessoas essenciais à minha vida e conquistado grandes oportunidades.

“Isn’t it splendid to think of all the things there are to find out about? It just makes me feel glad to be alive – it’s such an interesting world. It wouldn’t be half so interesting if we know all about everything, would it? There’d be no scope for imagination then, would there?”

(L.M. Montgomery, Anne of Green Gables)

Resumo

O interesse por ambientes de programação de robôs virtuais para fins educacionais tem crescido nos últimos anos. Estes ambientes são uma alternativa para o uso de robôs reais que possuem um alto custo de aquisição. No entanto, não existem ambientes gratuitos que oferecem mecanismos automatizados para verificação dos programas de robôs virtuais, o que impossibilita que alunos e professores tenham um *feedback* rápido e automático sobre o funcionamento dos programas. Este trabalho propõe uma abordagem de verificação automática de programas de robôs virtuais escritos na linguagem educacional ROBO. Desenvolvemos um compilador que lê programas escritos em ROBO e traduz os programas para uma notação formal chamada CSP (*Communicating Sequential Processes*), que é a entrada para uma ferramenta de verificação automática de modelos chamada FDR (*Failures-Divergences Refinement*). As fases da compilação foram implementadas usando a plataforma Spoofax, onde definimos a gramática da linguagem ROBO e especificamos regras de tradução de ROBO para CSP. Este trabalho remove uma limitação da nossa abordagem anterior de verificação que não permite a análise de programas ROBO contendo variáveis e procedimentos. Uma importante contribuição deste trabalho é a extensão da abordagem de verificação para permitir a análise automática de programas ROBO com variáveis e procedimentos. A extensão consiste na modificação da gramática do compilador pela inclusão de variáveis e procedimentos e na inclusão de novas regras de tradução que definem a semântica formal para os elementos adicionados na gramática. O trabalho propõe uma ferramenta que torna transparente o processo de tradução de ROBO para CSP e a verificação automática usando FDR. Validamos a abordagem utilizando a ferramenta proposta para verificar o comportamento de um programa ROBO com variáveis e procedimentos.

Palavras-chave: Engenharia de Software, Verificação de Software, Métodos Formais, Verificador de Modelos, CSP, Tradução Automática, Spoofax.

Abstract

There is an increasing interest in virtual robot programming environments for educational purposes in recent years. These environments are an alternative to the use of real robots, which have a high acquisition value. Automatic verification of robot programs is a demand of students and teachers that expect to have fast and automatic feedback about the correctness of robot programs. However, no free software provides an automatic verification of virtual robot programs. This work proposes an approach for the automatic verification of virtual robot programs authored in the educational language called ROBO. We propose a compiler that reads programs written in ROBO and translates its source code into a formal notation called CSP (Communicating Sequential Processes), which is the input to a model checking tool called FDR (Failures-Divergences Refinement). The compiler was implemented using the facilities of the Spoofax framework, which is used to define a parser for the ROBO language and a set of translation rules from ROBO to CSP. This work removes a limitation of our previous verification approach that does not perform the verification of ROBO programs containing variables and procedures. A significant contribution is the extension of the verification approach to allow the automatic analysis of ROBO programs with variables and procedures. The extension consists of the modification of the compiler grammar by the inclusion of variables and procedures and the inclusion of translation rules that define the formal semantics for the elements added into the grammar. Moreover, the work proposes a tool that makes transparent the translation process from ROBO to CSP and the automatic verification using FDR. We validate the approach using the proposed tool to verify the behavior of a ROBO program with variables and procedures.

Keywords: Software Engineering, Software Verification, Formal Methods, Model Checking, CSP, Automatic Translation, Spoofax.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
ATerm	*Annotated Term Format*
BNF	*Backus-Naur Form*
CSP	*Communicating Sequential Processes*
CTL	*Computation Tree Logic*
DSL	*Domain-Specific Language*
FDR	*Failures-Divergences Refinement*
GUI	*Graphical User Interface*
LTL	*Linear Temporal Logic*
SGLR	*Scannerless Generalized LR*
SVA	*Shared Variable Programming*
TCTL	Lógica de Árvore de Cálculo Temporizado

Sumário

1 Introdução

O interesse pela robótica educacional vem aumentando nas últimas décadas, uma vez que a mesma traz benefícios em todos os níveis da educação, seja no ensino de crianças, adolescentes ou adultos (ALIMISIS, 2013). Robótica tem sido utilizada para ajudar no aprendizado da matemática, ciências ou engenharia através de atividades práticas que envolvem a programação de robôs (BENITTI, 2012). A programação de robôs educacionais oferece apoio para o ensino e aprendizagem de programação, principalmente para aqueles que estão iniciando a construção do pensamento computacional, com o propósito de usar o raciocínio lógico para estruturar soluções coerentes para problemas complexos (BOMBASAR et al., 2015).

Ambientes de programação de robôs são geralmente atrativos e lúdicos com a intenção de despertar o fascínio e a curiosidade dos estudantes pela programação (SILVA et al., 2014). Segundo a reportagem Pernambuco (2018), o uso de ambientes de robótica nas escolas públicas do estado de Pernambuco tem aumentado o interesse dos estudantes pelas disciplinas de matemática e física, além de aumentar a criatividade, a sociabilidade, a concentração e o senso de coletividade.

O ambiente de programação LEGO MindStorms[1] é um exemplo de aplicação com essa finalidade; apresenta uma interface onde o aluno pode desenvolver o programa que controla o robô antes de executar o programa no robô. Um problema no uso da robótica educacional é o alto custo para adquirir equipamentos, como por exemplo, o robô MindStorms da LEGO. Uma alternativa para o uso de robôs é utilizar ambientes de robótica educacional baseados em simulação como o RoboMind[2]. Estes ambientes permitem a simulação passo a passo do robô na tela do computador, a partir da execução dos comandos do programa, ocasionando em uma melhor compreensão para o aluno (LESSA et al., 2015).

Um problema dos ambientes de simulação de robôs é que estes não oferecem um mecanismo de verificação automática das soluções propostas por estudantes, ou seja, os programas escritos pelos alunos não são verificados quanto a sua corretude para o problema proposto, de modo que estudante e professor possam obter *feedback* automático sobre o funcionamento dos programas. O atual mecanismo de verificação em ambientes de simulação de robôs virtuais ocorre através da observação dos passos do robô, o que pode tornar a tarefa demorada e trabalhosa. Além de onerosa, a verificação pode ser complexa. Por isso, métodos de verificação automática são tão importantes para determinar a corretude de um programa em diferentes perspectivas

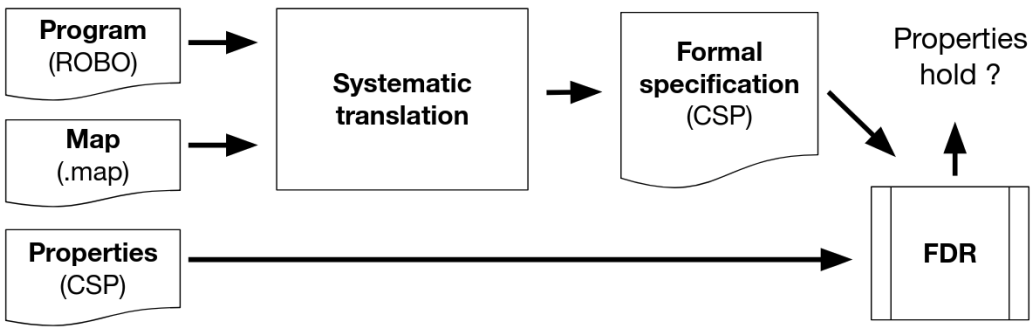
[1] https://www.lego.com/en-us/mindstorms
[2] http://robomind.net/

de maneira automatizada e rápida (DUARTE, 2011).

Não existem ambientes gratuitos de programação de robôs que oferecem a estudantes e professores um mecanismo de análise automática dos programas. O único ambiente de verificação para robôs é o Robomind Academy[3]. Este ambiente além de ser pago, só permite a verificação dos mapas já cadastrados no sistema, que estão associados a desafios de programação fixados; não é possível realizar a verificação automática dos programas levando em consideração diferentes mapas e soluções para um determinado problema. Exemplos de problemas são: identificar se o robô encontra a saída de um labirinto; se o robô encontra um objeto no mapa; ou simplesmente se o robô termina sua execução (não executa um laço infinito).

Em um trabalho anterior (NOGUEIRA et al., 2016) propomos a sistematização do processo de tradução dos programas escritos em ROBO para um modelo formal. Naquele trabalho é utilizado a verificação de modelos para automatizar a verificação de programas de robôs virtuais no ambiente RoboMind. Esta abordagem de verificação se dá por meio de algumas etapas como mostrado na Figura 1. É usado como entrada o programa escrito na linguagem ROBO e o mapa no qual o programa será executado. Um processo de tradução automática produz como saída a especificação formal do programa escrita na linguagem da álgebra de processos CSP (*Communicatting Sequential Processes*) (CLEAVELAND; ROSCOE; SMOLKA, 2018). Através de um processo automático também é obtida a representação formal do mapa. A especificação formal do mapa e do robô são entradas do verificador de modelos FDR (GIBSON-ROBINSON et al., 2014) que é utilizado para verificar as propriedades do robô. Como exemplo de propriedade, utilizamos a técnica de verificação de *deadlock* para verificar se um programa termina sua execução.

Figura 1 – Visão geral da abordagem de verificação automática



Fonte – (NOGUEIRA et al., 2016)

A maior complexidade da abordagem ilustrada na Figura 1 é a tradução da notação ROBO para a notação formal de CSP. Esta tradução é definida a partir de regras

[3] https://www.robomindacademy.com/

de mapeamento de cada elemento da linguagem ROBO para o seu equivalente em CSP; estas regras fazem parte de um compilador que automatiza a aplicação das regras. As etapas internas do compilador são: (1) criação de um *parser* para a sintaxe da linguagem ROBO e (2) a geração do modelo CSP a partir da árvore sintática do programa usando regras de mapeamento. O *framework* Spoofax (KATS; VISSER, 2010) foi utilizado para facilitar a implementação das etapas internas do compilador.

Atualmente, o compilador de ROBO para CSP não considera vários elementos importantes da linguagem como variáveis e procedimentos. Além disto, não existe uma integração do compilador com o verificador de modelos FDR. Também não existe uma interface que permita ao usuário utilizar a abordagem da Figura 1 de forma transparente.

O presente trabalho estende a gramática do compilador para aceitar programas ROBO com variáveis e procedimentos. A semântica das variáveis e procedimentos foi definida através de regras de mapeamento dos elementos sintáticos de ROBO para elementos na notação de CSP. Em complemento à extensão do compilador, uma ferramenta é proposta para tornar transparente o processo de tradução usando Spoofax e a verificação automática realizada por FDR. A interface da ferramenta permite carregar um programa e os mapas para realizar a verificação automática. Internamente, a ferramenta utiliza APIs de Spoofax e de FDR para automatizar o processo de tradução e verificação. Além da interface gráfica, o código da ferramenta possui um componente que encapsula os serviços de tradução e verificação, e pode ser utilizado para a implementação de outras ferramentas de verificação de programas ROBO. Por último, um exemplo de problema ROBO é usado como estudo de caso para validar a implementação da ferramenta e a abordagem de verificação.

O próximo capítulo apresenta os aspectos teóricos utilizados no desenvolvimento deste trabalho. O Capítulo 3 introduz a principal contribuição deste trabalho que é a extensão da abordagem para a verificação de programas com variáveis e procedimentos. No Capítulo 4, propomos uma ferramenta com interface gráfica que é usada para a verificação dos programas. Ainda neste capítulo, é proposto um exemplo de programa que é usado na validação da abordagem proposta. Por fim, o Capítulo 5 apresenta as considerações finais, trabalhos relacionados e trabalhos futuros.

2 Fundamentação Teórica

O presente capítulo apresenta os principais conceitos utilizados neste trabalho. A Seção 2.1 introduz o ambiente RoboMind e a linguagem ROBO que é utilizada para programar os robôs virtuais dentro deste ambiente. A Seção 2.2 explana sobre a importância da verificação automática através da Verificação de Modelos e as principais técnicas utilizadas. A Seção 2.3 introduz a notação de CSP utilizada na representação formal de programas ROBO, além de explicar como FDR é utilizado para verificação automática. Por fim, na Seção 2.4 é explicado como ocorre o processo de compilação através da plataforma Spoofax.

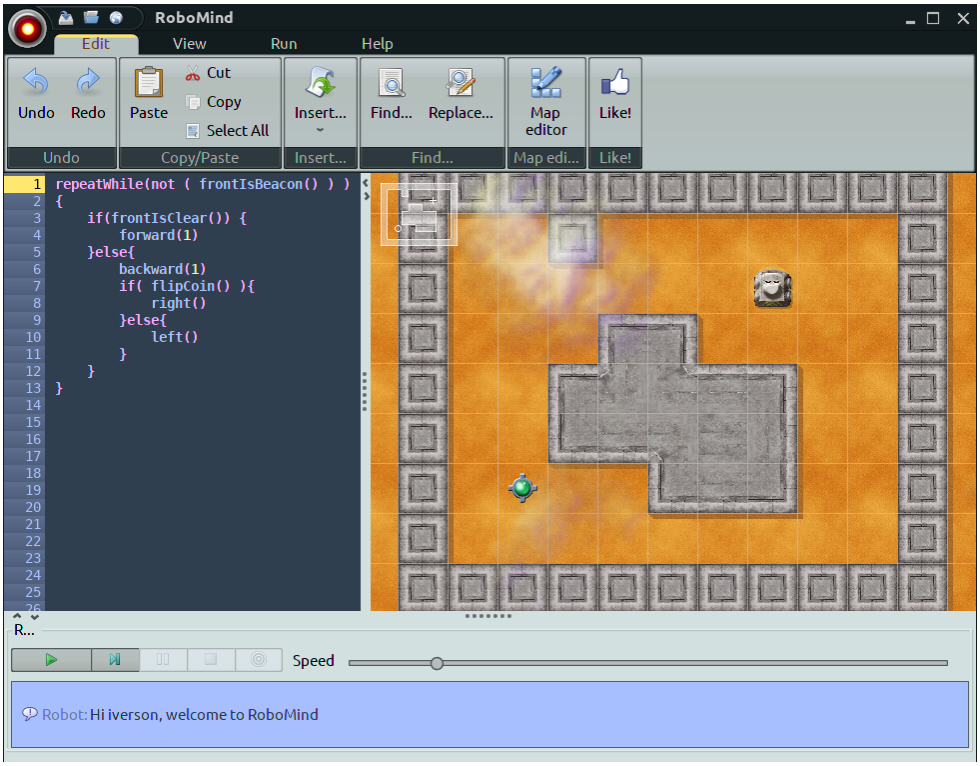
2.1 Ambiente RoboMind

RoboMind é um ambiente de programação de robôs virtuais para o ensino e aprendizagem de robótica. A Figura 2 ilustra a versão *desktop* deste ambiente que possui uma interface com um espaço para a escrita do programa que controla o robô virtual e um outro espaço onde o aluno pode acompanhar a execução (simulação) do robô em um mapa. Nesse ambiente, os programas são escritos na linguagem ROBO, uma linguagem educacional desenvolvida para a programação de robôs que oferece os principais comandos de programação: estruturas condicionais, estruturas de repetição, procedimentos e declaração de variáveis. O lado esquerdo da Figura 2 mostra um exemplo de programa ROBO que movimenta o robô enquanto o objeto (`beacon`) não é detectado à sua frente. Neste programa, se não há nenhum obstáculo à sua frente, o robô avança uma célula (comando `forward`), caso contrário o robô recua uma célula (comando `backward`) e muda sua orientação para a direita (comando `right`).

A seguir, listamos as principais construções sintáticas da linguagem ROBO.

- Funções booleanas predefinidas para detecção de obstáculos e objetos ao redor do robô, por exemplo, a função `frontIsObstacle` retorna verdadeiro quando há um obstáculo na frente do robô (retorna falso no caso contrário), e a função `frontIsClear` retorna verdadeiro quando a frente do robô está livre de quaisquer objetos (retorna falso no caso contrário).
- Comandos predefinidos para movimentar o robô, como por exemplo o comando `forward(n)` que movimenta o robô para frente `n` células no mapa, e o comando `backward(n)` que movimenta o robô para trás `n` células.

Figura 2 – RoboMind, ambiente de programação de robôs virtuais



Fonte – O autor

- Comandos para a mudar a orientação do robô, como por exemplo o comando `right` que é usado para alterar a orientação do robô para a direita, e o comando `left` que é usado para alterar a orientação do robô para esquerda.
- Estruturas condicionais e de repetição, como por exemplo, `if-else`, `repeat(n)` e `repeatWhile`.
- Possui o procedimento `show(v)` que exibe o valor de uma expressão inteira ou booleana `v` recebida como parâmetro.
- Permite definir variáveis globais que armazenam valores inteiros ou booleanos.
- Permite que o usuário defina procedimentos parametrizados e não parametrizados.

Um mapa no RoboMind por padrão é representado textualmente por um conjunto de letras para representar cada elemento do mapa. A Figura 3 mostra a representação textual do mapa ilustrado na figura anterior. O caractere `@` representa o robô, as caixas são denotadas pela letra `Q` e as paredes pelos demais caracteres.

Na literatura não há trabalhos que utilizem técnicas de verificação automática para programas objetivando a simulação de robôs educacionais. O que não garante

Figura 3 – Representação textual para um mapa do RoboMind

```
map:
CHHHHHHHHD
GMFFFFFFJI
GI  Q Q GI
GI@    GI
GI  QQQ GI
GLHHHHHHKI
BFFFFFFFFE
```

Fonte – O autor

que os programas foram escritos corretamente no contexto para o qual foram programados. Os próprios ambientes de programação não provêm essa automaticidade, além da verificação sintática de código, por este motivo o aluno acaba verificando manualmente os programas. Por meio de técnicas automáticas é possível verificar diferentes propriedades de um programa e provar formalmente que estas propriedades são satisfeitas, como a ausência de *deadlock* e outras propriedades específicas de um programa (MIYAZAWA et al., 2017). Uma dessas técnicas é a Verificação de Modelos, descrita na seção seguinte.

2.2 Verificação de Modelos

O crescimento e a complexidade dos sistemas de software possibilitam que falhas sejam empregadas durante o desenvolvimento, o que pode acarretar em prejuízos financeiros e desperdício. Quando o software consiste em um sistema crítico as perdas podem ser irreparáveis, como exemplo, perda de vidas humanas (CLARKE; WING, 1996). Por essa razão, a Engenharia de Software vem propondo técnicas para auxiliar o processo de desenvolvimento de software, objetivando a construção de sistemas confiáveis para as mais diversas finalidades. Métodos formais (SILVA, 2012) consistem em métodos da engenharia de software baseados em princípios matemáticos e apoiados por linguagens formais. Métodos formais são métodos rigorosos para a verificação de propriedades de sistemas, de modo a garantir que comportamentos indesejáveis não venham a acontecer (BHATT et al., 2017).

Uma especificação formal é a base para vários métodos formais; consiste na descrição de um sistema e suas propriedades em uma linguagem formal. Essa especificação é a base para a verificação consistente das propriedades de um sistema (GANNON; PURTILO; ZELKOWITZ, 2001; SILVA, 2012). A partir da especificação é possível explorar os possíveis estados alcançados por um sistema considerando um conjunto de entradas e saídas, afirmando, assim, as propriedades antes mesmo que o sistema seja implementado.

Na literatura de métodos formais, existem duas principais técnicas de verificação, a Verificação de Modelos (*Model Checking*) e a Verificação Dedutiva (*Deductive Verification*). A verificação formal tem sido gradualmente utilizada como um importante passo para a modelagem de sistemas críticos. Verificação de Modelos (*Model Checking*) é uma das técnicas mais usadas dentro dos métodos formais; tem sido muito utilizada principalmente na verificação de sistemas concorrentes (ZHAO; ROZIER, 2014). Como estes verificadores proveem técnicas automáticas, sua utilização se torna muito efetiva para sistemas dessa finalidade.

Apesar de pouco utilizada no contexto educacional, a verificação automática de programas pode ser utilizada para obter *feedback* automático sobre o funcionamento dos programas.

Verificação de modelos consiste em utilizar como entradas para uma ferramenta de verificação de modelo (*Model Checker*) um modelo do sistema, que pode ser expressado usando álgebra de processos ou até notação UML (*Unified Modeling Language*), como também a especificação das propriedades que o sistema deve satisfazer (FEIGN, 2005). A Verificação de Modelos automaticamente analisa exaustivamente se a especificação de um sistema satisfaz as propriedades definidas. Essa análise pode ser por meio de Lógica Temporal Linear (*Linear Temporal Logic* - LTL) ou Lógica de Árvore de Computação (*Computation Tree Logic* - CTL) (ZHAO; ROZIER, 2014). Como saída da verificação automática, temos o resultado positivo, se as propriedades são satisfeitas pelo modelo, ou resultado negativo, se as propriedades não são satisfeitas. Quando o resultado é negativo, o verificador de modelos mostra um cenário de execução onde o sistema não satisfaz alguma das propriedades, isso é chamado de contraexemplo. Uma desvantagem da Verificação de Modelos é que a técnica só consegue analisar sistemas com número finito e relativamente pequeno de estados.

A técnica Verificação Dedutiva, diferentemente da Verificação de Modelos, requer maior conhecimento e experiência do usuário, entretanto, considera todos os estados possíveis do sistema, mesmo que sejam infinitos através de ferramentas que utilizam provadores de teoremas, podendo ser manuais ou automáticos (FEIGN, 2005). Neste trabalho usamos Verificação de Modelos escritos na notação formal CSP para a verificação automática de programas escritos na linguagem ROBO.

2.3 Modelo CSP para programas ROBO

Communicatting Sequential Processes (CSP) é uma álgebra de processos usada na especificação formal de sistemas concorrentes e distribuídos (CLEAVELAND; ROSCOE; SMOLKA, 2018). Essa notação é composta por processos e eventos que são utilizados para a especificação de um sistema. Eventos são uma abstração para ocor-

rência de ações do sistema ou do usuário como, por exemplo, os comandos de movimentação do robô podem ser modelados como eventos. Processos definem a ordem de ocorrência dos eventos (ROSCOE, 2010). Na sintaxe de CSP, `P = e1 -> ... -> en -> Q` representa um processo `P` que comunica uma sequência de eventos e_i, separados pelo operador de prefixo de CSP (`->`) e se comporta como o processo `Q` após comunicar o último evento. A concorrência de processos pode ser representada em diferentes formas em CSP, como por exemplo através do operador de `P [| A |] Q`, onde os processos `P` e `Q` são colocados em paralelo sincronizados com os eventos contidos em `A`, esse operador é chamado de paralelismo generalizado (*Generalised Parallel*). Outro operador é o de intercalação (*Interleave*), que é dado por `P ||| Q`, onde os processos `P` e `Q` são executados em paralelo sem qualquer sincronização.

Os comandos e o ambiente de RoboMind foram modelados usando o modelo CSP proposto em (NOGUEIRA et al., 2016). O modelo CSP proposto permite representar os estados possíveis de um programa ROBO, considerando a execução dos comandos a partir da posição inicial do robô em um mapa. Neste modelo, a posição do robô, sua orientação e a posição do *beacon* são variáveis que são modificadas pela execução dos comandos contidos no programa do robô.

A Figura 4 mostra uma parte da especificação CSP usada para modelar um programa ROBO. A notação de CSP não possui variáveis. Uma forma de modelar variáveis em CSP é criando processos que funcionam como uma memória, e canais que são usados para ler e atualizar as variáveis mantidas no processo memória. Na linha 4 estão definidos os canais (`channel`) `get` e `set`. O primeiro canal é uma abstração para a consulta de uma variável. O segundo representa uma atualização de uma variável. O tipo destes canais é `VarType`, expressado na linha 2. Este tipo contém o nome das variáveis do programa e as respectivas faixas de valores para as variáveis. A ferramenta que é utilizada para verificação (FDR) precisa que todos os valores a serem analisados tenham limite superior e inferior. Por exemplo, a variável `X` representa a coluna da posição atual do robô, os valores desta variável correspondem ao conjunto `TX`, cujos valores variam de 0 até a quantidade máxima de colunas de um mapa. A variável `Y` representa a linha da posição atual do robô, a variável `ORIENTATION` a orientação atual do robô. As variáveis `BX` e `BY` representam a posição do objeto *beacon* (quando presente no mapa). Cada variável é guardada em uma célula de memória. A linha 6 apresenta o processo `Mcel(v,val)` que simula uma célula da memória, onde `v` é a variável e `val` o seu valor. Esse processo pode comunicar dois eventos: `get!v!val`, para comunicar o nome e o valor da variável; e `set!v?val_`, para exibir o nome e receber um novo valor para a variável. Na linha 9, há o processo `Memory(binding)` com a finalidade de colocar as células de memória em paralelo. Este paralelismo é representado pelos símbolos `|||`, onde para cada item `(v, val)` do conjunto `binding` cria-se uma instância de uma célula (`binding @ Mcel(v, val)`). Isto é, cada elemento do conjunto `INIT` é criado

um processo `Mcel` de modo intercalado a outro elemento, ou seja, seria equivalente a `Mcel(X, startX) ||| ... ||| Mcel(BY, startBY)`.

A inicialização da memória ocorre através do processo `MEMORY` passando o conjunto `INIT` para o processo `Memory`, linha 13 da Figura 4. Na especificação proposta só é possível consultar e atualizar os valores da posição (x,y) do robô (`(X, startX), (Y, startY)`), sua orientação (`ORIENTATION, NORTH_`) e a posição (x,y) de um determinado objeto (`(BX, startBX), (BY, startBY)`). Os valores `startX` e `startY` são obtidos através da leitura da especificação do mapa em CSP. O mesmo vale para `startBX` e `startBY`, que representa a posição inicial do objeto *beacon* no mapa. Quando uma especificação de um programa ROBO é gerada em CSP, é criado um processo chamado `COMMANDS`, que representa o programa que controla o robô. Diante disso, as linhas 15 e 16 representam os programas ROBO quando executados. O processo `PROGRAM_DEBUG` tem o objetivo de sincronizar os eventos `get` e `set` entre os processos `COMMANDS` e `MEMORY`. Isto é, todos os valores de consulta e atualização de variáveis que ocorrerem em `COMMANDS` serão os mesmos em `MEMORY`. Esses dois processos são colocados em paralelo usando o operador de paralelismo generalizado, no qual o processo `COMMANDS` é posto em paralelo com o processo `MEMORY` forçando a sincronização nos eventos `get` e `set`. O processo `PROGRAM` é o processo onde é possível observar os eventos que representam o comportamento de um programa ROBO. Este processo tem o comportamento de `PROGRAM_DEBUG` escondendo os eventos de controle. Para esconder os eventos de controle `get`, `set` e `coin` é usado o operador de *hiding* (barra invertida), conforme destacado na linha 16 da Figura 4. O uso desse operador é importante, pois, quando os *traces* são gerados eventos internos do modelo não são exibidos, portanto, apenas eventos que correspondem aos comandos em ROBO são mostrados.

O processo `COMMANDS`, como dito anteriormente, representa um programa ROBO escrito na área à da Figura 4. Este processo é obtido como resultado da tradução automática de ROBO para CSP. A Figura 5 apresenta um fragmento da tradução realizada para o programa ROBO ilustrado na Figura 14. Os comandos de robô são adicionados sequencialmente para comunicar alguns eventos em CSP para simular ocorrências dos comandos em ROBO. Na Figura 5, o comando `repeatWhile` (linha 18 da Figura 14) é representado pelas linhas de 5 a 26, onde `let` é utilizado para modelar outros processos em um escopo específico; esses processos são chamados após a palavra reservada `within`, no caso é chamado o processo interno `WHILE` para simular um laço enquanto a função `frontIsClear` retornar verdadeiro. Essa tradução foi realizada utilizando a abordagem automática proposta por este trabalho que será explicada no Capítulo 3.

A tradução atual do mapa ROBO é realizada automaticamente para a notação CSP. A Figura 6 mostra o resultado da tradução realizada para o mapa ilustrado pela

Figura 4 – Especificação da memória em CSP

```
1  ...
2  datatype VarType = X.TX | Y.TY | ORIENTATION.TO | BX.TBX | BY.
       TBY
3
4  channel get,set : VarType
5
6  Mcel(v,val) = get!v!val -> Mcel(v,val)
7                [] set!v?val_ -> Mcel(v,val_)
8
9  Memory(binding) = ||| (v,val) : binding @ Mcel(v,val)
10
11 INIT = { (X, startX), (Y, startY), (ORIENTATION, NORTH_), (BX,
       startBX), (BY, startBY) }
12
13 MEMORY = Memory(INIT)
14 ...
15 PROGRAM_DEBUG = COMMANDS [|{|get,set|}|] MEMORY
16 PROGRAM = PROGRAM_DEBUG \ {|get,set,coin|}
```

Fonte – (NOGUEIRA et al., 2016)

Figura 2 que é armazenado em `RAW_MAP` em forma de sequência. Cada elemento da sequência representa uma linha do mapa. O valor `Obs` é usado para representar os obstáculos do mapa, como paredes ou caixas; `Empty` é usado para as células vazias; e `Start` representa a posição inicial do robô.

No modelo CSP existem processos que especificam os comandos da linguagem ROBO. A Figura 7 destaca alguns desses processos. Como exemplo, o processo `FORWARD` (linha 19) é equivalente ao comando `forward` em ROBO. Esse processo comunica o evento `get!ORIENTATION?o` que busca na memória a orientação do robô e em sequência modifica a posição do robô por meio do processo `MOVE_STEPS(n, o)`, onde `n` é a quantidade de movimentos e `o` a direção onde o movimento deve acontecer. Na linha 5, tem um trecho de `MOVE_STEPS`, onde inicialmente existe uma sequência de eventos `get` que é usada para recuperar da memória o valor das variáveis. Em seguida, o processo verifica, de acordo com a sua orientação, se o espaço à sua frente está livre de obstáculos. Caso não haja obstáculos, a posição do robô é atualizada em um passo e o comportamento do processo é realizar **n-1** passos. A Figura 7 mostra o comportamento do processo quando a orientação é norte, representada pela constante `NORTH_`. Se o norte do robô está livre de obstáculos, isto é, a avaliação da função `frontIsClear(x,y,o,bx,by)` é verdadeira, significa que é possível mover o robô. Neste caso, é comunicado o evento `forward.1` (declarado na linha 2) que corresponde ao movimento do robô de andar para a frente. Em sequência, é atualizado o valor da variável `Y` pelo evento `set!Y(y-1)` e o processo se comporta recursivamente como o processo `MOVE_STEPS(n-1, o)`, que é executado até o caso base, onde o valor de

Figura 5 – Especificação em CSP de um programa ROBO

```
1  ...
2  COMMANDS =
3      RIGHT
4      ;
5      (let
6         WHILE =
7              get.X?x ->
8              get.Y?y ->
9              get.ORIENTATION?o ->
10             get.BX?bx ->
11             get.BY?by ->
12             if (frontIsClear(x,y,o,bx,by)) then (
13               get.lookLeft?lookLeftVar ->
14               countBoxesProc(lookLeftVar)
15               ;
16               FORWARD(1)
17               ;
18               SKIP
19               ;
20               WHILE
21             ) else (
22               SKIP
23             )
24       within
25         WHILE
26     )
27     ;
28     get.lookLeft?lookLeftVar ->
29     countBoxesProc(lookLeftVar)
30     ;
31     get.countBoxes?countBoxesVar ->
32     showInt!(countBoxesVar) ->
33     SKIP
```

Fonte – O autor

`n` é zero (linha 4). Em CSP existe um processo chamado `SKIP` para indicar que um processo terminou com sucesso.

Ainda na Figura 7, apresenta o processo `RIGHT` equivalente ao comando `right` em ROBO. Este processo inicialmente comunica o evento `right` (declarado na linha 20), em seguida lê a orientação atual do robô (`get!ORIENTATION?o`) e atualiza a orientação do robô para direita (`set!ORIENTATION!(o+1)%4`). Por fim, o processo termina com sucesso `SKIP`. A orientação do robô é representada pelos inteiros 0 até 3 que representam respectivamente norte, leste, sul e oeste.

2.3.1 Verificador de Modelos FDR

Uma especificação CSP pode ser verificada através de um verificador de modelos. Existem várias ferramentas que podem ser utilizadas para este fim, as mais conhe-

Figura 6 – Especificação em CSP de um mapa ROBO

```
RAW_MAP = <
<Obs,Obs,Obs,Obs,Obs,Obs,Obs,Obs,Obs,Obs>,
<Obs,Empty,Empty,Obs,Empty,Empty,Empty,Empty,Empty,Empty,Obs>,
<Obs,Empty,Empty,Empty,Empty,Empty,Empty,Start,Empty,Empty,Obs
   >,
<Obs,Empty,Empty,Empty,Obs,Obs,Empty,Empty,Empty,Empty,Obs>,
<Obs,Empty,Empty,Obs,Obs,Obs,Obs,Obs,Empty,Empty,Obs>,
<Obs,Empty,Empty,Obs,Obs,Obs,Obs,Obs,Empty,Empty,Obs>,
<Obs,Empty,Beacon,Empty,Empty,Obs,Obs,Obs,Empty,Empty,Obs>,
<Obs,Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,Obs
   >,
<Obs,Obs,Obs,Obs,Obs,Obs,Obs,Obs,Obs,Obs,Obs>
>
```

Fonte – O autor

Figura 7 – Especificação em CSP de comandos de movimentação e orientação

```
...
channel forward : {1}

MOVE_STEPS(0,o) = SKIP
MOVE_STEPS(n,o) =
    get!X?x ->
    get!Y?y ->
    get!BX?bx ->
    get!BY?by ->
    if(o == NORTH_) then (
        if(frontIsClear(x,y,o,bx,by)) then
          forward!1 -> set.Y!(y - 1) -> MOVE_STEPS(n-1, o)
        else
          forward!1 -> MOVE_STEPS(0, o)
    )
    ...
...
FORWARD(n) = get!ORIENTATION?o -> MOVE_STEPS(n,o)
...
channel right

RIGHT = right -> get!ORIENTATION?o -> set!ORIENTATION!(o+1)%4
   ->  SKIP
...
```

Fonte – (NOGUEIRA et al., 2016)

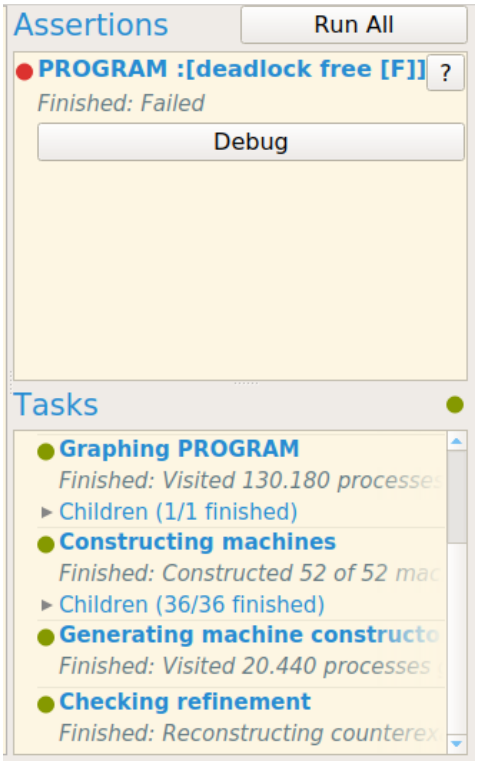
cidas são FDR[4], ProB[5] e PAT[6]. Destas, a ferramenta FDR (*Failures Divergence Refinement*) é a mais madura e utilizada; sua versão atual é FDR4 (GIBSON-ROBINSON et al., 2014). Por este motivo, este trabalho adota FDR4.

[4] https://www.cs.ox.ac.uk/projects/fdr/
[5] https://www3.hhu.de/stups/prob
[6] http://pat.comp.nus.edu.sg/

O verificador de modelos FDR, como citado, é o verificador de refinamento mais difundido para a álgebra de processo CSP. Este verificador utiliza a lista de processos CSP da especificação, e é capaz de verificar se os processos refinam uns aos outros de acordo com os modelos disponíveis na ferramenta: *traces*; falhas (*failures*); e falhas e divergências (*failures-divergences-models*). Além disso, é capaz de verificar outras propriedades: se o modelo está livre de *deadlock* (*deadlock-free*); se está livre de *livelock* (*livelock-free*) e verificação de determinismo (GIBSON-ROBINSON et al., 2014).

A Figura 8 exibe um exemplo de verificação de uma propriedade em FDR. Na interface de FDR, as propriedades aparecem na área *Assertions*, que mostra uma lista de propriedades definidas a serem analisadas por FDR. Nesse caso, a propriedade analisada é se o processo `PROGRAM` está livre de *deadlock*. Em CSP, esta propriedade é escrita como `PROGRAM :[deadlock free [F]]`. O resultado da verificação desta propriedade para o processo `PROGRAM`, que representa o programa da Figura 14, é falso (*Failed*), o que significa que o programa não é livre de *deadlock*, uma vez que o laço do programa termina. Caso o programa nunca terminasse o laço (quando não encontra o *beacon*), o resultado da verificação seria verdadeiro. Neste caso, o programa estaria livre de *deadlock*.

Figura 8 – Exemplo de verificação da propriedade *deadlock-free*

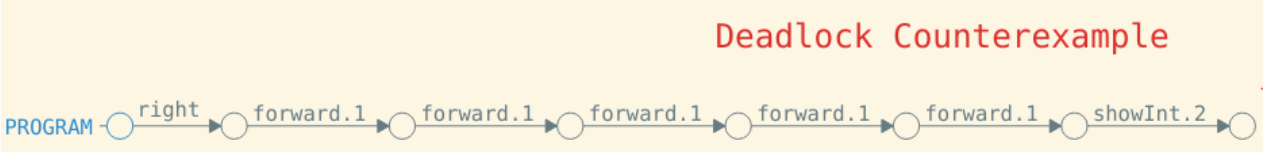


Fonte – O autor

Como o resultado dessa verificação falhou, FDR exibe na interface um con-

traexemplo indicando o ponto no qual a propriedade é violada. A Figura 9 apresenta o contraexemplo da verificação da propriedade `:[deadlock free [F]]` para o processo `PROGRAM`. Onde é possível ver a sequência de eventos comunicados no processo `COMMANDS` que leva o processo do seu estado inicial até o comportamento de *deadlock*. Analisando os *traces*, sabemos que esse processo comunicou o evento `right` inicialmente, seguido por cinco eventos `forward.1` e finalizou comunicando `showInt.2`, este mostra a quantidade de caixas à esquerda do robô, equivalente ao comando `show` em ROBO. Comparando com o programa ROBO descrito pela Figura 14 é possível notar que o último comando que o robô realiza é `show(counter)`.

Figura 9 – Contraexemplo gerado pelo FDR



Fonte – O autor

Além da interface gráfica, FDR possui uma API (*Application Programming Interface*) que permite o desenvolvimento de sistemas que usam os serviços de FDR.

2.4 Plataforma Spoofax

Para criar um compilador da linguagem ROBO para CSP é necessário um mecanismo que ajude nesse processo, pois o desenvolvimento através ferramentas é possível utilizar funções para definição sintática e funções para auxiliar na transformação, assim facilitando o desenvolvimento. Por isso, neste trabalho vamos utilizar a Plataforma Spoofax.

Spoofax[7] é uma plataforma para desenvolvimento textual de linguagens de programação. Esta plataforma permite o desenvolvimento através de um ambiente interativo usando metalinguagens (*meta-languages)* para definição de linguagem declarativa de alto nível. Além disso, possui geradores de código que produzem analisadores (*parses*), checagem de tipo, compiladores, interpretadores e outras ferramentas. A vantagem de usar Spoofax é o foco na essência da definição da linguagem, ignorando detalhes irrelevantes da implementação (KATS; VISSER, 2010).

Um projeto em Spoofax possui uma arquitetura muito bem estruturada, seguindo alguns princípios. Primeiro, separação de interesses, por exemplo, separa definição de sintaxe da definição da semântica estática. Segundo, não repete aspectos da linguagem em diferentes implementações, objetivando a geração de diferentes artefatos a

[7] http://www.metaborg.org/

partir de um único código. Por último, a definição de linguagem declarativa ocorre de modo independente, onde cada tipo de implementação possui sua própria linguagem correspondente. Por exemplo, para definição de sintaxe utiliza o formalismo SDF3, já para a parte de transformação, utiliza-se a metalinguagem Stratego (KATS; VISSER, 2010). Spoofax ainda provê outras metalinguagens, como por exemplo NaBL2, utilizada para a checagem semântica da linguagem, o que inclui checagem de nomes e análise de tipos.

Para o escopo desse projeto, utilizamos SDF3 e Stratego, tendo em vista que a checagem de nomes e tipos da linguagem ROBO é realizada durante o desenvolvimento do programa dentro do ambiente RoboMind.

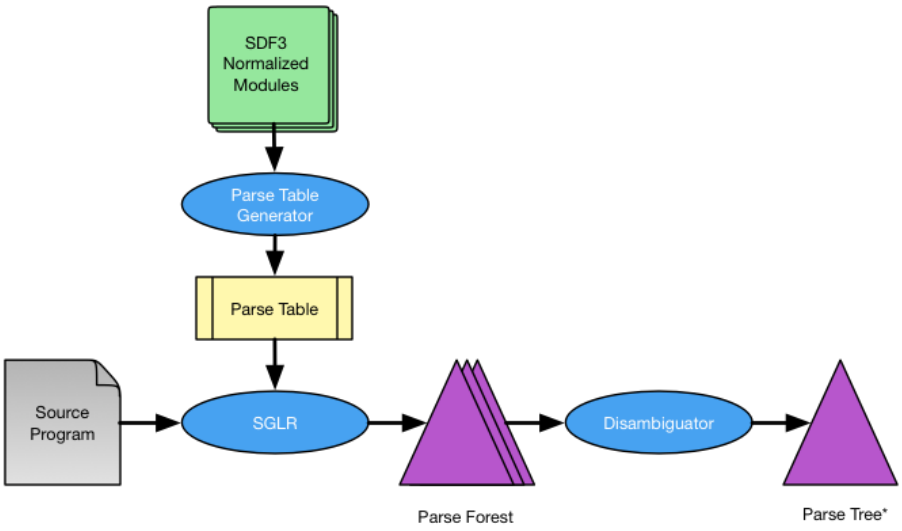
Assim como o verificador de modelos FDR, a plataforma Spoofax dispõe de uma API que pode ser utilizada para diferentes objetivos, como por exemplo, o desenvolvimento de ferramentas integradas a um compilador gerado pela plataforma.

2.4.1 Definição Sintática com SDF3

A definição sintática com Spoofax ocorre através da metalinguagem SDF3. Esta é um formalismo que propicia a definição da sintaxe para linguagens de programação e Linguagem de Domínio Específico (*Domain-Specific Language* - DSL). Na Figura 10 é mostrado o fluxograma de como ocorre a definição sintática com essa plataforma, desde a definição da gramática até a geração da árvore sintática. O primeiro passo é a definição dos módulos em SDF3, onde ocorre toda a definição da gramática da linguagem. Como saída dos módulos normalizados em SDF3, é gerada uma tabela de análise (*Parse Table*) que é utilizado como entrada no *Scannerless Generalized LR* (SGLR), responsável por produzir a Árvore de Sintaxe Abstrata (*Abstract Syntax Tree* - AST). Após a geração de uma AST, o SDF3 conta com um mecanismo que remove todas as ambiguidades, gerando uma única AST (*Parse Tree*) de um programa em uma linguagem específica (*Source Program*).

A Figura 11 mostra como é a estrutura de um ambiente de programação utilizando essa metalinguagem. A parte mais à esquerda da figura mostra um exemplo da gramática de uma linguagem escrita no formato SDF3 (*Example.sdf3*), uma sintaxe similar ao formalismo BNF (*Backus-Naur Form*). Esse arquivo define um módulo (`module`) chamado `Example` que importa o módulo `Common` e define a gramática a partir de quatro seções diferentes. A seção `context-free start-symbols` indica o símbolo inicial quando ocorre uma análise dos termos, neste caso o termo `Start`. A seção `context-free syntax` descreve em alto nível a estrutura sintática das sentenças em uma linguagem, contendo uma lista de produções. Nesta seção há dois termos `Stmt` e `Expr`. O termo `Stmt` define as produções: `Stmt.If` que representa uma estrutura sintática para uma condição; e `Stmt.Assign` representa a estrutura sintática para uma

Figura 10 – Processo de definição sintática do Spoofax



Fonte – (METABORG, 2018)

atribuição. O termo `Expr` define algumas produções para representar a estrutura sintática para alguns tipos de expressões. A seção `context-free priorities` é utilizado para remover ambiguidades através da definição de prioridades entre as produções. Por último, a seção `lexical syntax` descreve a parte léxica da linguagem, ou seja, como cada produção deve ser representada e também a definição de palavras reservadas da linguagem. O lado direito da Figura 11 mostra três diferentes arquivos. O quadrante superior é um editor com *syntax highlighting* que é produzido automaticamente pelo ambiente a partir da definição da gramática SDF3. Além da marcação da sintaxe, este editor possui *auto complete* dos termos da linguagem. O quadro do meio mostra um exemplo de código aceito pela gramática. O último quadro (*Example.aterm*) mostra a árvore sintática do programa *Example.tes*.

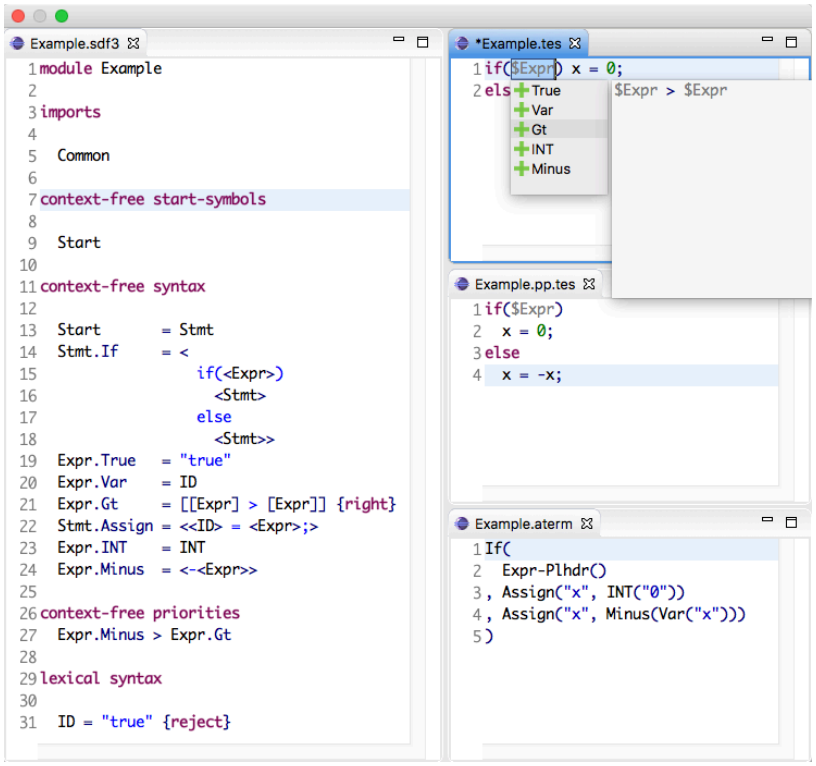
A definição sintática com Spoofax, mais especificamente usando a metalinguagem SDF3 produz uma árvore sintática não ambígua que é essencial para a etapa seguinte do processo de compilação, a geração de código. A transformação ocorre através da metalinguagem Stratego.

2.4.2 Transformação com Stratego

Este projeto usa Stratego para traduzir automaticamente o código ROBO em CSP com finalidade de analisar automaticamente a corretude dos programas ROBO. Stratego é usada para definir a regra de transformação de cada elemento sintático de ROBO para a respectiva representação em CSP.

Stratego é uma metalinguagem para a definição de transformações aplicadas à uma AST utilizando o conceito de reescrita de termos (*term rewriting*). Essas transfor-

Figura 11 – Definição sintática com SDF3



Fonte – (METABORG, 2018)

mações ocorrem em decorrência da definição de regras e estratégias. Os elementos da AST são representados por termos, como mostrado no arquivo *Example.aterm* da Figura 11. Esses termos estão na notação chamada de Formato de Termo Anotado (*Annotated Term Format* - ATerm), uma notação que facilita a manipulação da árvore pelo Stratego (METABORG, 2018). Por exemplo, em ROBO o comando `forward(n)` é denotado em *ATerm* por `Instr(FORWARD(Var("n")))`.

Uma regra em Stratego é definida como uma transformação aplicada em um ou mais termos. Uma regra é dada por `L : p1 -> p2`, onde `L` é o nome da regra, `p1` é o termo que deve casar com a regra e `p2` o termo reescrito. Como alternativa, a saída de uma regra pode ser textual, ao invés de um termo reescrito. Esta abordagem é chamada interpolação de *string* (KATS; VISSER, 2010). Uma regra desse tipo é dada por `L : T(p) -> $[[p]]`, no qual `L` é o nome da regra, `T(p)` o termo que deve casar com a regra, e o texto na linguagem destino é gerado com o conteúdo entre `$[` e `]` utilizando o valor de `p`. Além disso, é possível aplicar outras regras a partir de `p` utilizando a palavra reservada `with` no fim da regra. As regras definidas em (NOGUEIRA et al., 2016) para mapear a AST de programas ROBO para CSP utilizam interpolação de *strings*. As *strings* produzidas como resultado da regra correspondem ao modelo CSP para o programa ROBO representado pela AST de entrada. Isso é demonstrado em mais detalhes no Capítulo 3.

O Stratego possui algumas funções que podem ser utilizadas durante uma transformação. Por exemplo, há algumas funções que são usadas para filtrar termos em uma AST, ou seja, apenas termos de interesse são aplicados em uma transformação. É o caso da função `filter` que retorna uma lista com o conjunto de termos filtrados, esse filtro é aplicado apenas aos termos que estão no primeiro nível da árvore. Outra função importante é `collect-all`, que é responsável por recolher todos os termos em uma AST que casem com o termo analisado, essa busca ocorre em todos os níveis da árvore.

3 Tradução de ROBO para CSP

Este capítulo descreve uma importante contribuição deste trabalho que é o processo de tradução automática de ROBO para CSP utilizando o *framework* Spoofax. Este processo inclui a definição da sintaxe da linguagem ROBO até a definição das regras de transformação de programas ROBO para a sua representação CSP correspondente. O presente trabalho estende a abordagem de tradução atual por permitir a tradução de variáveis e procedimentos no programa ROBO para CSP. A apresentação do processo é ilustrada através de um exemplo de programa ROBO.

3.1 Definição da Sintaxe

No trabalho anterior (NOGUEIRA et al., 2016), como já mencionado, propomos um compilador que contempla a tradução de programas ROBO sem variáveis e procedimentos para CSP. Esta seção mostra a extensão da gramática introduzida em (NOGUEIRA et al., 2016) para permitir que programas ROBO com variáveis e procedimentos possam ser traduzidos para CSP.

Antes de estender a gramática SDF3 introduzida em (NOGUEIRA et al., 2016), foi preciso reescrever algumas de suas produções sintáticas para simplificar a AST obtida como resultado do *parsing*. Esta simplificação foi fundamental para a criação das regras Stratego que lidam com variáveis e procedimentos. A gramática definida em (NOGUEIRA et al., 2016) resulta em uma AST onde o programa é uma sequência que possui a estrutura recursiva (`Instr`, `Sequence`) semelhante a uma árvore, o que impossibilitava o uso de funções do *framework* Spoofax como filtros que trabalham com sequencias. Para facilitar o mapeamento da AST para CSP, foi necessário reconstruir partes da gramática de modo que a AST gerada apresentasse um formato de lista, o que torna possível o uso das funções nativas do Spoofax. A Figura 12 mostra um trecho da gramática definida no trabalho anterior no formato SDF3.

Figura 12 – Gramática escrita em forma de árvore

```
1 context-free syntax
2
3     Start.Program = Sequence
4
5     Sequence.Empty = []
6     Sequence.Sequence = [[Instr][Sequence]]
```

Fonte – O autor

A Figura 13 mostra um fragmento da gramática após a reescrita. Na parte superior da figura, são importados três módulos que contém definições da gramática: o módulo `Commom` que contém toda a parte léxica da linguagem, como por exemplo, palavras reservadas; o módulo `ExpressionsBoolean` que possui todas as definições da gramática para expressões booleanas; o módulo `ExpressionsMath` que contém a sintaxe de expressões aritméticas; e o módulo `Robo2CSP` que importa os demais módulos e define a sintaxe da linguagem ROBO. Nesta gramática, o que antes era `Sequence` tornou-se `Statement` seguido pelo operador * para representar zero ou mais ocorrências. Dessa forma, uma ocorrência de `Statement` é adicionada ao lado de outra `Statement` e forma uma sequência. A gramática completa e os módulos auxiliares podem ser encontrados no Apêndice A.

Figura 13 – Gramática proposta para ROBO

```
module Robo2CSP

imports

    Common
    ExpressionsBoolean
    ExpressionsMath

context-free start-symbols

    Start

context-free syntax

    Start.Program = <<Statement*>>

    Statement.Instr = <<Instr>>
    Statement.Declaration = <<Declaration>>

    Declaration.Variable = <<Identifier> = <Expr>>
    Declaration.Procedure = <procedure <Identifier> {
                                <Statement*>
                            }>
    Declaration.ProcParam = <procedure <Identifier> <Params> {
                                <Statement*>
                            }>

    Params.Params = <(<{ Identifier ", "}*>)>
    Identifier.ID = <<ID>>

    Expr.ExpBool = <<ExprBoolean>>
    Expr.ExpMath = <<ExprMath>>

    Instr.ProcCall = <<Identifier> <TypeParams>>
    TypeParams.ExprParams = <(<{ Expr ", "}*>)>
```

Fonte – O autor

A partir da gramática reescrita foram acrescentadas produções sintáticas que correspondem a variáveis e procedimentos. A primeira parte da extensão consistiu na adição do termo `Declaration` (linha 18 na Figura 13) que possui três tipos: `Variable`, `Procedure` e `ProcParam` (linhas 20, 21 e 24, respectivamente). Portanto, um programa consiste em uma lista de comandos (`Statement`) que podem ser do tipo `Instr` ou `Declaration`. O tipo `Instr` contém todas as instruções básicas da linguagem ROBO, ou seja, comandos de movimentação, pintura de mapa, captura de objetos, estruturas condicionais e de repetição. Já o tipo `Declaration` define a sintaxe para declaração de variáveis e de procedimentos.

A produção `Statement.Declaration` define três alternativas para uma declaração. A primeira representa variáveis (`Declaration.Variable`), a segunda procedimentos não parametrizados (`Declaration.Procedure`), e por fim, procedimentos parametrizados (`Declaration.ProcParam`). O corpo da produção para variáveis `<<Identifier> = <Expr>>` define uma declaração do tipo `Variable` composta do identificador da variável (`Identifier`), seguido por um símbolo de igual e terminado por uma expressão que pode ser do tipo booleana ou aritmética (linhas 31 e 32 da Figura 13, respectivamente). Esta mesma produção é usada para uma chamada de variável, o que difere uma da outra é a aplicação de regras de tradução distintas. Um procedimento não parametrizado (`<procedure <Identifier>{ <Statement*> }>`) é composto pela palavra reservada `procedure`, seguida do identificador do procedimento, que por sua vez é seguido por zero ou mais `Statement` dentre dos símbolos de abre e fecha chaves. Vale salientar que na linguagem ROBO um procedimento não parametrizado também pode ser escrito com abre e fecha parênteses. Dessa forma, também é possível representar um procedimento não parametrizado através da produção `Declaration.ProcParam` quando a lista de parâmetros for vazia. Um procedimento parametrizado, após o seu identificador, inclui a produção o tipo `Params` que inclui zero ou mais identificadores separados por vírgula dentro dos símbolos de abre e fecha parênteses dada pela produção `Params.Params` (linha 28).

A linguagem ROBO permite definir procedimentos que retornam valores (funções). Uma limitação da gramática atual é que a mesma não aceita programas ROBO com funções definidas pelo usuário. Entretanto isto não é uma limitação importante, uma vez que pode-se definir um procedimento que utiliza uma variável global para armazenar o resultado produzido pelo procedimento como forma de suprir a ausência de uma função.

Além da declaração de variáveis e procedimentos, a gramática foi ampliada para incluir a definição de produções para chamada de procedimento. Esta produção está indicada na linha 34 da Figura 13. Uma chamada de procedimento nada mais é do que um subtipo de `Instr`, chamado de `ProcCall`. O corpo desta produção é composto

por um identificador seguido pelos parâmetros. A produção `ExprParams` define a lista de parâmetros de uma chamada de procedimento como expressões separadas por vírgula e inseridas dentre de parênteses.

Com a definição da gramática da linguagem ROBO em SDF3 torna-se possível a geração da árvore para representar os programas ROBO. Lembrando que as etapas para a geração da tabela de símbolos (*Parse Table*) e da remoção de ambiguidades ocorre de modo implícito pelo *framework*.

Usamos o programa ROBO mostrado na Figura 14 para ilustrar o *parsing* de um programa ROBO que contém duas variáveis e um procedimento. Esta figura mostra um programa ROBO que resolve o problema da contagem de caixas. Este programa é uma solução simplificada para o problema “Contando Caixas” proposto em (BENITTI et al., 2009). O objetivo do programa é fazer com que o robô seja capaz de contar caixas dispostas na primeira ou na última linha de um mapa com três linhas. Tal programa possui duas variáveis globais: `counter` que armazena a quantidade de caixas encontradas; e `lookLeft` que indica qual a linha que o robô deve contar as caixas. O código também possui um procedimento parametrizado chamado `countBoxes` com o parâmetro chamado `side`, cujo valor indica o lado em que o robô vai contar as caixas. Se o valor de `side` é igual a 1, o robô conta as caixas no seu lado esquerdo (primeira linha do mapa); quando o parâmetro possui outros valores o robô conta as caixas do lado direito (segunda linha do mapa). O primeiro comando do programa do robô é o comando `right` (indicado na linha 16), o qual altera sua orientação em 90 graus para a direita. O próximo passo é a execução de um laço pelo comando `repeatWhile`, o laço ocorre enquanto não houver quaisquer objetos ou paredes na célula à frente do robô. Esse laço é responsável por chamar o procedimento `countBoxes` passando o valor da variável `lookLeft`, que neste exemplo tem valor 1, ou seja, contará as caixas da linha esquerda, seguindo de um `forward` que move o robô para frente em uma unidade a cada execução do laço. Ao sair dessa estrutura de repetição, o procedimento será executado mais uma vez, com o objetivo de verificar possíveis caixas na última posição do robô e por fim a quantidade de caixas é exibida por meio do comando `show` exibindo o valor armazenado na variável `counter` contendo a quantidade de caixas na linha de interesse. A execução deste programa faz o robô percorrer em linha reta contando as caixas à sua esquerda e parar na posição final, quando o programa mostra o resultado 2 (quantidade de caixas encontradas na primeira linha). Caso a variável `lookLeft` seja inicializada com o valor 0, o programa contaria as caixas encontradas à direita do robô, portanto o valor impresso pelo programa seria 3.

A Figura 15 mostra um possível mapa utilizado usado como entrada para o programa que conta as caixas. Nele é possível ver um robô e algumas caixas: duas caixas na linha superior e três na linha inferior. Na figura foi adicionado um “X” para

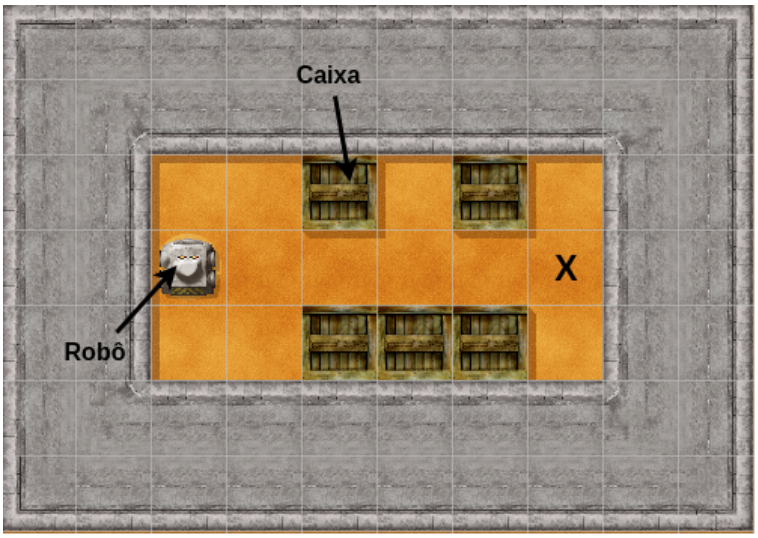
Figura 14 – Programa escrito em ROBO

```
1  counter = 0
2  lookLeft = 1
3
4  procedure countBoxes(side){
5      if(side == 1){
6          if(leftIsObstacle){
7              counter = counter + 1
8          }
9      } else {
10         if(rightIsObstacle){
11             counter = counter + 1
12         }
13     }
14 }
15
16 right
17
18 repeatWhile(frontIsClear){
19     countBoxes(lookLeft)
20     forward(1)
21 }
22
23 countBoxes(lookLeft)
24 show(counter)
```

Fonte – O autor

indicar a posição final que o robô deverá parar após a execução do programa.

Figura 15 – Exemplo de Mapa usado no RoboMind



Fonte – O autor

Considerando a gramática apresentada, a Figura 16 corresponde a árvore sintática resultante do *parsing* (parte dela foi omitida, versão completa disponível no Apêndice B) do programa ROBO introduzido na Figura 14. A raiz da árvore corresponde

ao elemento `Program` que contém uma lista com todos os comandos (`Statement`) do programa. No programa, a primeira linha possui a declaração da variável `counter`. Na AST essa declaração é representada pelo primeiro elemento da lista (linha 2): `Declaration(Variable(ID("counter")...)`. O mesmo vale para a variável `lookLeft`. Isto é, uma declaração de variável com um identificador e uma expressão matemática. A declaração do procedimento `countBoxes` é representada na AST pelo termo `Declaration(ID("countBoxes), ...)` (linhas 4 a 8). O comando `right` na árvore é dado por `Instr(RIGHT())`. O comando para executar um laço (`repeatWhile`) é equivalente a `Instr(RPTWLE(...))` (linhas 11 a 20), onde o primeiro elemento de `RPTWLE` é `FROISCLR` para representar o comando `frontIsClear`, e o segundo é uma representação de `forward` (`Instr(FORWARD(...)`). Em uma chamada do procedimento `countBoxes`, sua representação na árvore é denotada por `Instr(ProcCall(...))` (linhas 21 a 23), ou seja, um identificador, e uma lista de parâmetros no termo `ExprParams`. Por fim, temos o comando `show` que é representado por `Instr(SHOW(...))`.

Figura 16 – AST de um programa ROBO

```
1  Program(
2    [ Declaration(Variable(ID("counter"), ExpMath(Num("0"))))
3    , Declaration(Variable(ID("lookLeft"), ExpMath(Num("1"))))
4    , Declaration(
5          ProcParam(ID("countBoxes")
6                  , Params([ID("side")])
7                  , [...]
8          )
9      )
10   , Instr(RIGHT())
11   , Instr(
12       RPTWLE(
13         FROISCLR()
14       , [ Instr(
15             ProcCall(ID("countBoxes"), ExprParams([ExpMath(Var
     ("lookLeft"))]))
16           )
17         , Instr(FORWARD(Num("1")))
18         ]
19       )
20     )
21   , Instr(
22       ProcCall(ID("countBoxes"), ExprParams([ExpMath(Var("
     lookLeft"))]))
23     )
24   , Instr(SHOW(ExpMath(Var("counter"))))
25   ]
26 )
```

Fonte – O autor

Uma limitação atual da abordagem é que a análise semântica de programas ROBO não é realizada pela versão atual do compilador. Portanto, espera-se que o

usuário escreva os programas no ambiente RoboMind, que revela erros semânticos no programa, antes de verificar o programa com a abordagem proposta. Quando o programa ROBO é escrito fora de RoboMind é possível que o programa tenha erros semânticos, neste caso, a atual abordagem deixa a cargo do compilador de FDR apontar problemas semânticos da especificação CSP.

As alterações na gramática descritas acima permitem representar sintaticamente programas escritos em ROBO com variáveis e procedimentos e gerar sua árvore sintática. Esta árvore é a entrada para obter a representação formal para o programa em CSP que é apresentada na próxima seção.

3.2 Transformação com Stratego

Conforme mostrado na seção anterior, a gramática SDF3 foi estendida para aceitar variáveis e procedimentos definidos pelo usuário. Esta seção introduz regras de tradução que mapeiam os elementos da AST que representam variáveis e procedimentos para os respectivos elementos em CSP.

A Seção 2.3 mostrou o modelo CSP que representa um programa ROBO que armazena a posição e orientação do robô em um processo que faz o papel de memória. A seguir, mostramos como Stratego foi utilizado para coletar na AST as variáveis definidas pelo usuário e incluí-las no processo que representa a memória. Além das variáveis, Stratego foi utilizado para coletar da AST do programa os procedimentos definidos pelo usuário e produzir a representação dos procedimentos em CSP.

A Figura 17 mostra parte da especificação formal gerada automaticamente, como resultado da aplicação das regras que serão introduzidas a seguir, quando a entrada das regras é a AST mostrada na Figura 16. No Apêndice D.1 está disposta a especificação formal completa dessa tradução. Na Figura 17 podemos notar que as variáveis `counter` e `lookLeft` são representadas pelas constantes `counterConst` e `lookLeftConst` na especificação CSP. Também foi gerado o tipo de dados para cada variável (linha 5). Além de gerar o conjunto `INIT` para a inicialização do processo `MEMORY` com os valores iniciais de cada variável (linha 6). O procedimento `countBoxes` virou o processo `countBoxesProc` e o seu parâmetro `side` foi chamado de `sideParam`, isso ocorreu porque quando um processo parametrizado for chamado o valor do parâmetro é armazenado na memória (`set.side!(sideParam)`), possibilitando sua atualização como uma variável local do procedimento. A linha 36 mostra uma chamada do processo `countBoxesProc` com o parâmetro `lookLeftVar` que foi obtido na linha anterior por meio do evento `get.lookLeft?lookLeftVar`, que busca o valor da variável na memória. O mesmo vale para o evento `showInt`, que é equivalente ao comando `show` em ROBO. Para valores booleanos a instrução `show` corresponde ao evento `showBool`

em CSP. Esta figura omite a definição das células da memória e outros processos que foram apresentados na Seção 2.3 e fazem parte do modelo CSP para um program ROBO.

Figura 17 – Especificação CSP gerada a partir de um programa ROBO

```
1  counterConst = 0
2  lookLeftConst = 0
3
4  nametype INTVALUES = {0..10}
5  datatype VarType = side.INTVALUES | counter.INTVALUES |
       lookLeft.INTVALUES |  X.TX | Y.TY | ORIENTATION.TO | BX.TBX
       | BY.TBY
6  INIT = { (side,0), (counter,0), (lookLeft,0), (X, startX), (Y,
       startY), (ORIENTATION, NORTH_), (BX, startBX), (BY, startBY)
        }
7
8  countBoxesProc(sideParam) =
9      set.side!(sideParam) ->
10     (let
11         IFELSE =
12             ...
13             get.side?sideVar ->
14             if((sideVar == 1)) then (
15                 (let
16                     ...
17                     if(leftIsObstacle(x,y,o)) then (
18                     get.counter?counterVar ->
19                     member((counterVar + 1), INTVALUES) & set.
       counter!((counterVar + 1)) ->
20                     ...
21                 ...
22             )
23             ...
24     within
25       IFELSE
26     )
27
28 COMMANDS =
29     RIGHT
30     ;
31     (let
32         WHILE =
33             ...
34     ...
35     ;
36     get.lookLeft?lookLeftVar ->
37     countBoxesProc(lookLeftVar)
38     ;
39     get.counter?counterVar ->
40     showInt!(counterVar) ->
41     SKIP
```

Fonte – O autor

A Figura 18 mostra a regra `main-to-csp` (linha 3) que transforma `Program(T1)`,

que corresponde a raiz da AST, em uma especificação CSP. O termo `T1` corresponde ao restante da árvore sintática do programa. Esta regra é a primeira regra a ser executada por Spoofax durante o processo de tradução. O CSP gerado possui uma estrutura fixa como a definição de constantes (linha 5), o domínio das variáveis definidas pelo usuário (linha 6), a definição de variáveis (linha 7), a inicialização da memória (linha 8), procedimentos (linha 10) e os comandos do programa (linha 12). Dentro desta estrutura fixa existem elementos que aparecem entre parêntesis e são substituídas por fragmentos de especificação CSP obtidos pela aplicação de outras regras, que usam como entrada o termo `T1`. Por exemplo, o elemento `[proc']` (linha 10) corresponde ao modelo CSP para os procedimentos definidos pelo usuário. Os elementos entre parêntesis são definidos nas linhas 16 a 22 da Figura 18. Explicamos cada um destes elementos.

Figura 18 – Regra inicial para um programa ROBO

```
1  rules
2
3  main-to-csp:
4    Program(T1) ->
5    $[[vars']
6    nametype INTVALUES = {0..10}
7    datatype VarType = [paramsType'][varsType'] X.TX | Y.TY |
      ORIENTATION.TO | BX.TBX | BY.TBY
8    INIT = { [paramsInit'][varsInit'] (X, startX), (Y, startY), (
      ORIENTATION, NORTH_), (BX, startBX), (BY, startBY) }
9
10   [proc']
11
12   COMMANDS =
13     [instr']
14   ]
15   with
16     vars'       := <var-analyze-const> <get-vars> T1;
17     paramsType' := <param-analyze-type> <get-ids> <get-params>
      T1;
18     varsType'   := <var-analyze-type> <get-vars> T1;
19     paramsInit' := <param-analyze-init> <get-ids> <get-params>
      T1;
20     varsInit'   := <var-analyze-init> <get-vars> T1;
21     proc'       := <statement-definition-decl> <union>(<get-
      procs> T1, <get-procs-param> T1);
22     instr'      := <statement-definition> <get-commands> T1
```

Fonte – O autor

O elemento `vars'` é obtido pela aplicação da regra `<get-vars>` em `T1`, essa regra tem como objetivo obter uma lista, através da aplicação de filtro na árvore, com o conjunto de *ATerms* que casam com o termo `Declaration(Variable(_,_))`. A definição desta regra está na linha 6 da Figura 19. Após isso, uma lista com os termos que correspondem a declaração de variáveis são a entrada da regra mais à es-

querda, `<var-analyze-const>`, que gera como saída as variáveis declaradas no programa ROBO em forma de constantes CSP. A Figura 17 ilustra nas linhas 1 e 2 as variáveis `counter` e `lookLeft` como constantes. A Figura 20 mostra a definição das regras responsáveis pela geração das constantes. A regra `<var-analyze-const>` é aplicada de modo recursivo, muito similar às funções de linguagens funcionais, no qual uma função é aplicada na cabeça (*head*) enquanto a mesma função é aplicada na calda (*tail*) até chegar no caso base, que neste é caso é uma lista vazia, como está indicado na linha 3 da Figura 20. A análise ocorre em cima do termo `Declaration(var)`, onde a regra `<write-variable-const>` é aplicada em `var` para escrever o CSP correspondente ao tipo da variável. Por isso, há três declarações dessa regra, a primeira para escrever um texto vazio em casos de expressões matemáticas, já que as constantes não recebem expressões (exemplo, a + b); a segunda para escrever o texto `[name]Const = [v]`, ou seja, o nome da variável (`name`) e seu respectivo valor inteiro (`v`); por último, para expressões booleanas, onde uma outra regra, `<to-csp-e>`, é aplicada para escrever o valor booleano (*true* ou *false*), dependendo do valor do termo em `exp'`. Padrões dessa regra podem ser conferidas no Apêndice C.1. A transformação das variáveis de ROBO como constantes no modelo CSP é necessária quando a expressão que define o valor inicial de uma variável usa o valor de outras variáveis. Por exemplo, na atribuição `a = b + 1` a variável `a` é inicializada como o valor da variável `b` acrescido de uma unidade. Em CSP não é possível utilizar o nome da variável diretamente em `INIT`, pois seria necessário um evento de leitura `get` para consultar o valor das variáveis, isto só pode ser realizado dentro de processos. Portanto, uma constante para cada valor inicial da variável resolve este problema, uma vez que pode ser utilizada em `[n]Const`, onde `n` é o identificador da variável. Usando o exemplo `a = b + 1`, `INIT` é inicializado com o par `(a, (bConst + 1))` que define o valor inicial para a variável `a`.

Figura 19 – Conjunto de regras auxiliares

```
1 rules
2
3 get-procs-param = filter(?Declaration(ProcParam(_,_,_)))
4 get-procs = filter(?Declaration(Procedure(_,_)))
5 get-commands = filter(?Instr(_))
6 get-vars = filter(?Declaration(Variable(_,_)))
7 get-vars-exp = collect-all(?Var(_))
8 get-params = collect-all(?Params(_))
9 get-ids    = collect-all(?ID(_))
```

Fonte – O autor

O conjunto `INTVALUES` na linha 6 da Figura 18 define os valores que as variáveis inteiras definidas pelo usuário podem apresentar durante a execução do programa. Este conjunto é necessário pois FDR necessita de conjuntos finitos de valores. Quando o conjunto numérico é muito grande, a verificação poder ter o desempenho comprome-

Figura 20 – Regras para geração de constantes em CSP

```
1  rules
2
3  var-analyze-const: [] -> []
4  var-analyze-const: [Declaration(var) | es] -> [<write-variable-
       const> var, "\n" | <var-analyze-const> es ]
5
6  write-variable-const: Variable(ID(name), ExpMath(s)) -> $[]
7
8  write-variable-const:
9    Variable(ID(name), ExpMath(Num(v))) ->
10     $[[name]Const = [v]]
11
12 write-variable-const:
13   Variable(ID(name), ExpBool(exp)) ->
14     $[[name]Const = [exp']]
15     with
16       exp' := <to-csp-e> exp
```

Fonte – O autor

tido devido a grande quantidade de estados que o programa possui. A definição deste conjunto deve ser adaptada manualmente para os valores sejam adequados para os programas que serão analisados. Isto é uma das limitações atuais da abordagem.

Foi explicado no Capítulo 2, que em CSP existe um processo (`MEMORY`) que armazena informações de estado do robô e de objetos no mapa. Usamos este mesmo processo para armazenar as variáveis definidas pelo usuário, em adição às variáveis da posição e a orientação do robô. Na linha 7 da Figura 18, o tipo das variáveis é definido por `VarType`, que corresponde a junção dos tipos dos parâmetros (`paramsType'`) e das variáveis (`varsType'`) encontrados nos programas ROBO. Uma limitação disto é que variáveis globais e locais (aos procedimentos) são colocadas no mesmo escopo. Desde que as variáveis tenham nomes diferentes, não acontece conflito de nomes na memória.

Para a obtenção de `paramsType'`, a primeira regra aplicada é `<get-params>` (linha 8 da Figura 19). Esta regra usa a função nativa do Stratego chamada de `collect-all`, cujo objetivo é recolher todos os termos que são alcançados a partir da raiz de uma AST. Portanto, o retorno desta regra é uma lista contendo todos os termos `Params(_)` do programa ROBO. Os parâmetros são a entrada para a regra `<get-ids>`, que recolhe todos os identificadores (`ID(_)`) dos parâmetros. Os identificadores são a entrada para a regra `<param-analyze-type>`, que está detalhada na Figura 21. Esta última regra gera os tipos dos parâmetros no formato de CSP. Essa regra é aplicada também de modo recursivo, a recursão ocorre na linha 4 na Figura 21, onde o primeiro elemento da lista `ID(n)` é analisado e destinado à regra `<write-param-type>` que é aplicada em `n`, enquanto para o restante da lista (`es`) a regra principal é aplicada recursivamente. A

regra `write-param-type` escreve a notação CSP correspondente ao tipo do parâmetro, o conteúdo que está entre os símbolos cifrão e colchetes é convertido em texto e `[n]` é substituído pelo nome do parâmetro analisado. Para exemplificar, o termo `ID("side")`, visto na linha 6 da Figura 16, é escrito em CSP como `side.INTVALUES`. Uma limitação da tradução atual é que ela funciona apenas para parâmetros do tipo inteiro, pois não há informação sobre o tipo do parâmetro na árvore.

Figura 21 – Regras para os tipos de dados de parâmetros em notação CSP

```
1 rules
2
3 param-analyze-type: [] -> []
4 param-analyze-type: [ID(n) | es] ->  [ <write-param-type> n | <
     param-analyze-type> es ]
5
6 write-param-type: n -> $[[n].INTVALUES | ]
```

Fonte – O autor

As regras usadas para obter `varsType'` são semelhantes àquelas usadas para obter `paramsType'`. A diferença está em como ocorre a escrita de CSP, uma vez que as variáveis ROBO possuem dois tipos de dados em suas expressões: `Bool`, para expressões com valores booleanos e `INTVALUES`, para expressões com valores inteiros. A regra `<write-variable-type>`, como mostra nas linhas 6 e 9 da Figura 22, aparece duas vezes, a primeira para termos que possuem expressões aritméticas e a segunda para os termos com expressões booleanas. Por exemplo, na linha 5 da Figura 17 é ilustrado a adição dos tipos das variáveis `side`, `counter` e `lookLeft`.

Uma limitação atual é que o compilador (por não realizar análise semântica) considera que as expressões que definem o valor inicial das variáveis são do tipo inteiro. Entretanto as regras para construção do CSP já estão prontas para quando a análise semântica for realizada, neste caso, variáveis com valores lógicos serão mapeadas para o tipo correto.

Os elementos `paramsInit'` e `varsInit'` que aparecem na Figura 18 (linha 8) correspondem ao valor inicial dos parâmetros de procedimento e variáveis definidas pelo usuário. Estes valores são colocados na constante `INIT` que é utilizada para definir o estado inicial da memória. A transformação ocorre de modo semelhante as regras para os tipos de variáveis e parâmetros. Na Figura 23 está definida a regra `<param-analyze-init>` para adicionar a essa constante os parâmetros dos procedimentos. Essa regra percorre recursivamente a lista de termos aplicando a regra `<write-param-init>` no termo analisado. Ou seja, todos os parâmetros são definidos em CSP como `([n], 0)`, onde `n` é o identificador do parâmetro. Essa abordagem foi proposta para garantir que um parâmetro possa ser atualizado durante a execução de

Figura 22 – Regras para os tipos de dados de variáveis em notação CSP

```
1  rules
2
3  var-analyze-type: [] -> []
4  var-analyze-type: [Declaration(var) | es] -> [ <write-variable-
       type> var, " | " | <var-analyze-type> es ]
5
6  write-variable-type:
7    Variable(ID(name), ExpMath(exp)) -> $[[name].INTVALUES]
8
9  write-variable-type:
10   Variable(ID(name), ExpBool(exp)) -> $[[name].Bool]
```

Fonte – O autor

um procedimento. Por exemplo, na Figura 17 mostra o parâmetro *side* como uma variável em `INIT` (linha 6). Na Figura 24 estão dispostas as regras para inicialização das variáveis. A regra `<var-analyze-init>` é aplicada recursivamente nos termos, onde a regra `<write-variable-init>` é aplicada ao valor de `var` do termo `Declaration(var)`. Essa regra gera o CSP para as variáveis no formato `([name].[exp'])`, onde `name` é o identificador da variável e `exp'` a expressão que pode ser booleana ou aritmética, por isso há duas definições de `<write-variable-init>`. Na Figura 17, a aplicação dessa regra está ilustrada na linha 6 (variáveis `counter` e `lookLeft`).

Figura 23 – Regras para os tipos de dados de variáveis em notação CSP

```
1  rules
2
3  param-analyze-init: [] -> []
4  param-analyze-init: [ID(n)| es] ->  [ <write-param-init> n | <
       param-analyze-init> es ]
5
6  write-param-init: n -> $[([n],0), ]
```

Fonte – O autor

O elemento `proc'` que aparece na linha 10 da Figura 18 corresponde a especificação CSP dos procedimentos. Como existem dois tipos de procedimentos na linguagem ROBO, os parametrizados e não parametrizados, é necessário aplicar individualmente as regras `get-procs` e `get-procs-param` (definidas nas linhas 3 e 4 da Figura 19) em `T1` para recolher todos os termos relacionados aos procedimentos e depois combiná-los em uma única lista. Na linha 21 da Figura 18 é utilizada a regra `<union>` nativa do *framework*, cujo objetivo é unir duas listas de termos. A lista resultante é a entrada da regra `<statement-definition-decl>`, exposta na Figura 25 que produz o CSP dos procedimentos.

A regra `<statement-definition-decl>`, definida nas linhas 3 e 4 da Figura

Figura 24 – Regras para os tipos de dados de variáveis em notação CSP

```
1  rules
2
3  var-analyze-init: [] -> []
4  var-analyze-init: [Declaration(var) | es] -> [<write-variable-
       init> var, "," | <var-analyze-init> es ]
5
6  write-variable-init:
7    Variable(ID(name), ExpMath(exp)) ->
8      $[([name],[exp'])]
9      with
10       exp' := <to-csp-e> exp
11
12 write-variable-init:
13   Variable(ID(name), ExpBool(exp)) ->
14     $[([name],[exp'])]
15     with
16       exp' := <to-csp-e> exp
```

Fonte – O autor

Figura 25 – Regras para a geração de CSP dos procedimentos

```
1  rules
2
3  statement-definition-decl: [] -> []
4  statement-definition-decl: [Declaration(s) | ps] -> [<to-csp> s
       | <statement-definition-decl> ps]
5
6  to-csp:
7    ProcParam(ID(name), Params(params), procedureBody) ->
8    $[[name]Proc([params']) =
9      [paramVar']
10     [procedureBody']
11   ]
12   with
13    paramVar'       := <put-param-mem> params;
14    procedureBody'  := <statement-definition> procedureBody;
15    params'         := <get-param-names> params
```

Fonte – O autor

25, aplica a regra `<to-csp>` para cada declaração (`Declaration(s)`) encontrada na lista recebida como entrada. Algumas definições de `<to-csp>` podem ser consultadas no Apêndice C.2. Quando a declaração casa com o padrão `ProcParam(ID(name), Params(params), procedureBody` é executada a regra na linha 6 da Figura 25 que gera a especificação CSP para o procedimento. Sendo `name` o nome do procedimento; `params` a lista de parâmetros; e `procedureBody` as instruções dentro do corpo do procedimento. Na especificação CSP, o nome do processo que representa o procedimento é definido como o nome do procedimento em ROBO concatenado com a pa-

lavra `Proc`, a Figura 17 ilustra através do procedimento `coutBoxesProc` (linha 8). O nome do processo é seguido pelos parâmetros (`params'`), o símbolo de igual, o elemento `paramVar'` e `procedureBody'`. Como os parâmetros são colocados na memória, o CSP do elemento `paramVar'` corresponde à atualização do valor do parâmetro na memória usando o valor recebido como parâmetro usando o canal `set`. O CSP deste elemento é obtido aplicando a regra `<put-param-mem>` definida na Figura 26. A representação de um parâmetro como uma variável permite que o parâmetro seja utilizado com uma variável local dentro do procedimento, que pode ser tanto lida como atualizada pelo programa. Vemos na Figura 14 linha 9 um exemplo de aplicação desta regra. O parâmetro `side` no procedimento `countBoxes` é representado no CSP pelo parâmetro `sideParam`, cujo valor é usado para atualizar a variável `side` na memória através do evento `set.side!(sideParam)`.

Figura 26 – Regras que adicionam os valores dos parâmetros na memória em uma chamada de procedimento

```
1 rules
2
3 put-param-mem: [] -> []
4 put-param-mem: [ID(n) | ps] -> ["set.",n,"!(",n,"Param) -> \n"
     | <put-param-mem> ps ]
```

Fonte – O autor

Depois de obter o CSP de `paramVar'`, ocorre a geração de código para o corpo do procedimento. Na Figura 25, o elemento `procedureBody'` é obtido aplicando a regra `<statement-definition>` que é explicitada na Figura 27. Essa regra aplica recursivamente a regra `<to-csp>` para todos os termos que representam comandos dentro do corpo do procedimento; quando não existem mais comandos para traduzir é escrito o processo `SKIP`, que é um processo CSP que termina com sucesso.

Figura 27 – Regras que aplica `to-csp` para cada instrução

```
1 rules
2
3 statement-definition: [] -> [$[SKIP]]
4 statement-definition: [s | ps] -> [<to-csp> s  | <statement-
     definition> ps]
```

Fonte – O autor

A regra `<to-csp>` possui múltiplos padrões introduzidos em (NOGUEIRA et al., 2016). Para cada termo que representa uma construção da linguagem ROBO existe uma regra `<to-csp>` que gera o CSP correspondente. Adicionamos mais dois padrões para a regra `<to-csp>` para gerar o CSP que corresponde aos termos `ProcCall` e `SHOW`

da gramática. Na Figura 28 são mostrados estes padrões. O primeiro (linha 3) gera o CSP para a chamada do procedimento. O segundo (linha 13) gera o CSP do comando que exibe valores inteiros no *console*. Por concisão, omitimos o padrão que gera CSP para o comando que exibe valores booleanos.

Detalhamos a regra que gera o CSP para uma chamada de procedimento. O CSP gerado para uma chamada de procedimento obtém os valores das variáveis que são passadas como parâmetro para a chamada, conforme mostrado na linha 5 da Figura 28, que é representado pelo elemento `vars'`. Para obter o CSP deste elemento é aplicada a regra `<get-vars-exp>` que tem o objetivo de recolher todas as variáveis presentes na expressão (`ExprParams()`). Em seguida, é aplicada a regra `<put-get-var-exp-analyze>`, cujo objetivo é colocar todas as variáveis no formato de CSP, como mostra na Figura 30. Na linha 6 ocorre a escrita do texto CSP pela regra `<put-get-var-exp>`, por exemplo, quando a variável `lookLeft` em ROBO é passada como parâmetro para uma chamada de procedimento sua leitura é escrita como a comunicação do evento `get.lookLeft?lookLeftVar` no modelo formal à linha anterior a chamada do processo `countBoxesProc(lookLeftVar)` (linhas 36 e 37 da Figura 17). Já para escrever o nome dos parâmetros na chamada do procedimento é aplicada a regra `<get-param-names-call>` nos termos de `params`. Lembrando que em uma chamada de procedimento é possível passar qualquer expressão inteira como parâmetro, portanto a regra está preparada para isto. A Figura 29 mostra as regras auxiliares para obter o CSP da chamada do procedimento.

Figura 28 – Exemplos da regra `to-csp`

```
1  rules
2
3  to-csp:
4    ProcCall(ID(name), ExprParams(params)) ->
5    $[[vars']
6    [name]Proc([params'])
7    ;
8    ]
9    with
10     vars'   := <put-get-var-exp-analyze> <get-vars-exp> params;
11     params' := <get-param-names-call> params
12
13 to-csp:
14     SHOW(ExpMath(e)) ->
15     $[[vars']
16     showInt!([e']) ->]
17     with
18       vars' := <put-get-var-exp-analyze> <get-vars-exp> e;
19       e'    := <to-csp-e> e
```

Fonte – O autor

Figura 29 – Regras para gerar parâmetros em uma chamada de procedimento

```
1  rules
2
3  get-param-names-call: ExpMath(es) -> <get-param-names-call> es
4  get-param-names-call: [] -> []
5  get-param-names-call: [ExpMath(exp)] -> [exp']
6    with exp' := <to-csp-e> exp
7  get-param-names-call: [ExpMath(exp) | es] -> [exp', "," | <get-
       param-names-call> es ]
8    with exp' := <to-csp-e> exp
9
10 get-param-names-call: ExpBool(es) -> <get-param-names-call> es
11 get-param-names-call: [] -> []
12 get-param-names-call: [ExpBool(exp)] -> [exp']
13   with exp' := <to-csp-e> exp
14 get-param-names-call: [ExpBool(exp) | es] -> [exp', "," | <get-
       param-names-call> es ]
15   with exp' := <to-csp-e> exp
```

Fonte – O autor

Figura 30 – Regras para buscar e atualizar valores das variáveis

```
1  rules
2
3  put-get-var-exp-analyze: [] -> []
4  put-get-var-exp-analyze: [var | es] -> [<put-get-var-exp> var |
       <put-get-var-exp-analyze> es ]
5
6  put-get-var-exp:
7    Var(name) -> $[get.[name]?[name]Var ->
8    ]
9
10 to-csp:
11   Variable(ID(n),ExpMath(e)) ->
12   $[get.[n]?[n]Var ->
13   [var']
14   member([e'], INTVALUES) & set.[n]!([e']) -> ]
15   with
16     var' := <put-get-var-exp-analyze> <get-vars-exp> e;
17     e' := <to-csp-e> e
```

Fonte – O autor

Outra instância de `<to-csp>` foi definida para atualização de variáveis, indicada pela linha 10 na Figura 30. Na linha 12 é possível notar quando o CSP é gerado com intuito de escrever uma consulta do valor da própria variável, enquanto as demais variáveis são representadas pelo elemento `var'`, na linha 13. Na Figura 17, linhas 18 e 19, mostra um exemplo com a variável `counter`, onde ela é utilizada na expressão `counter + 1`, dessa forma uma consulta é realizada (`get.counter?counterVar`) antes da atribuição. Essa abordagem é necessária porque uma variável pode ser atualizada por ela

mesma, além de uma combinação de expressões com diferentes variáveis e valores. O CSP da atualização do valor dela na memória ocorre na linha 14. Para fins práticos, considerando a expressão `counter = counter + 1`, ela seria escrita em CSP como `member((countBoxesVar + 1), INTVALUES) & set.countBoxes!((countBoxesVar + 1))`. Isso quer dizer que primeiro é verificado se o novo valor da variável está dentro do limite inferior e superior para que de fato seja atualizada na comunicação do evento `set`. A verificação que ocorre com a função `member` é para garantir que o novo valor da variável pertence ao conjunto `INTVALUES`, caso contrário o evento `set` não é comunicado, e consequentemente a variável não será atualizada, tal fato é uma limitação dessa abordagem.

A parte final da tradução corresponde ao elemento `instr'` (Figura 18 linha 13), que representa o CSP dos comandos do programa que aparecem fora dos procedimentos, isto é, os eventos que são adicionados dentro do processo `COMMANDS`.

A tradução é contemplada com as regras de compilação iniciais apresentadas em (NOGUEIRA et al., 2016) para as instruções básicas do robô, agora também inclui uma extensão do compilador com a adição de regras para programas que possuem declarações de procedimentos e variáveis e chamadas de procedimentos e atualizações de variáveis, além de outras melhorias.

4 Implementação e Validação

Este capítulo apresenta uma ferramenta para a verificação de programas ROBO e um estudo de caso usado na validação da abordagem. A Seção 4.1 apresentada uma ferramenta com interface gráfica que integra as etapas da compilação apresentadas no capítulo anterior com o verificador de modelos FDR, e apresenta para o usuário o resultado da verificação automática. A Seção 4.2 apresenta um estudo de caso usado na validação da abordagem. O estudo de caso usa a ferramenta proposta para verificar um programa ROBO completo contendo variáveis e procedimentos.

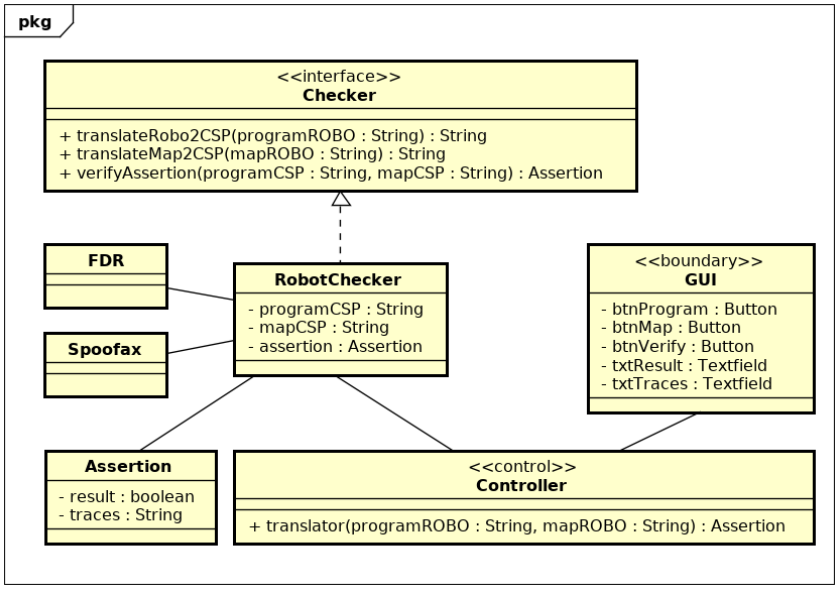
4.1 Ferramenta

Esta seção descreve uma ferramenta desenvolvida na linguagem de programação Java[8] com a finalidade de verificar automaticamente programas ROBO. Internamente, a ferramenta está estruturada conforme o Diagrama de Classes disposto na Figura 31. A interface (`Checker`) possui três métodos: `translateRobo2CSP`, que recebe como entrada uma *string* do programa escrito na linguagem ROBO e gera uma *string* como saída; `translateMap2CSP` que recebe uma *string* com a representação textual de um mapa do ambiente RoboMind e produz uma *string*; e `verifyAssertion` que recebe como entrada o programa e o mapa na notação CSP e gera o objeto `Assertion`. A Classe `RobotChecker` implementa a Interface `Checker`. Essa classe possui alguns atributos que são responsáveis por armazenar informações importantes da tradução e da verificação. O atributo `programCSP` é utilizado para armazenar uma *string* com a especificação formal CSP que representa o programa ROBO obtido como retorno do método `translateRobo2CSP`. A implementação deste método utiliza os serviços providos pelo *framework* `Spoofax`. O atributo `mapCSP` armazena o texto que representa o CSP do mapa obtido como retorno do método `translateMap2CSP`. O método `verifyAssertion` chama métodos da API de FDR (representada pela classe `FDR`) passando como entrada as representações CSP para o programa e para o mapa gerado. O atributo `assertion` (instância da Classe `Assertion`) armazena no campo `result` o resultado da verificação realizado por FDR (se passou ou falhou) e no campo `traces` o contra-exemplo quando o resultado da verificação de FDR falha. Para tornar transparente a tradução com Spoofax e a verificação com FDR para o usuário, desenvolvemos uma camada de apresentação que é representada pela Classe `GUI`. Destacamos alguns elementos desta classe. O atributo `btnProgram` representa um botão com a finalidade de buscar o programa ROBO fornecido pelo usuário. O atributo `btnMap` representa um bo-

[8] https://java.com/download

tão para o usuário fornecer os mapas do RoboMind como entrada. O atributo `btnVerify` tem o intuito de iniciar o processo de tradução e posteriormente a verificação por meio da chamada do método `translator` da Classe `Controller` que recebe como entrada o programa e mapas do usuário e retorna o objeto `Assertion`. Os atributos `txtResult` e `txtTraces` são utilizados para exibir na tela da ferramenta o resultado da verificação e os *traces* (contraexemplos) gerados pela verificação, respectivamente.

Figura 31 – Diagrama de Classes do protótipo



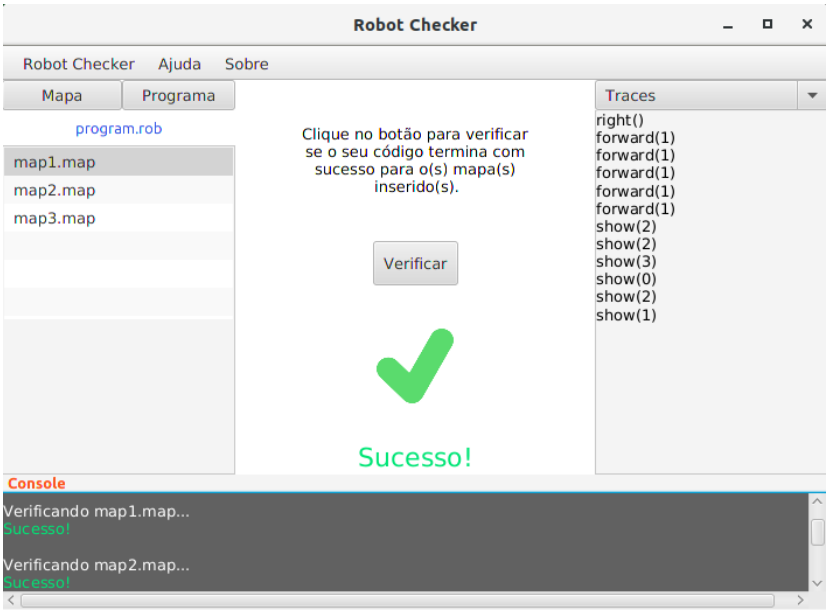
Fonte – O autor

A Figura 32 mostra a Interface Gráfica (*Graphical User Interface* - GUI) da ferramenta que implementa o diagrama mostrado anteriormente. A interface conta com três botões: “Mapa” onde o usuário escolhe um ou mais mapas do RoboMind que são listados logo abaixo deste botão; “Programa” onde o usuário seleciona o arquivo do programa ROBO (o nome do arquivo é colocado abaixo dos botões); e no meio da tela está o botão “Verificar“ que executa a tradução e a verificação. No lado direito da figura são listados os *traces* (contraexemplos) gerados pela verificação. Na parte inferior da figura existe um *console* para que o usuário acompanhe o andamento da verificação automática. Ao clicar no nome de um mapa, a ferramenta exibe o contraexemplo para a propriedade que foi verificada (caso a verificação falhe).

A propriedade verificada pela ferramenta atualmente é se o programa ROBO possui loop infinito, isto é, se está livre de *deadlock* (ver Capítulo 2). Caso não possua, um contraexemplo é apresentado, que representa as ações realizadas pelo robô em um dado mapa até a terminação do programa. A ferramenta está preparada para verificar e apresentar o resultado de qualquer propriedade CSP baseada em refinamento de traces que seja colocada no arquivo que contém a especificação formal CSP. Em (NO-

GUEIRA et al., 2016) são propostas várias propriedades que podem ser especificadas usando refinamento de traces de CSP, como por exemplo, saber se o robô alcança alguma posição do mapa ou se o robô segue uma sequência de ações predefinidas.

Figura 32 – Interface gráfica da ferramenta



Fonte – O autor

4.2 Estudo de caso

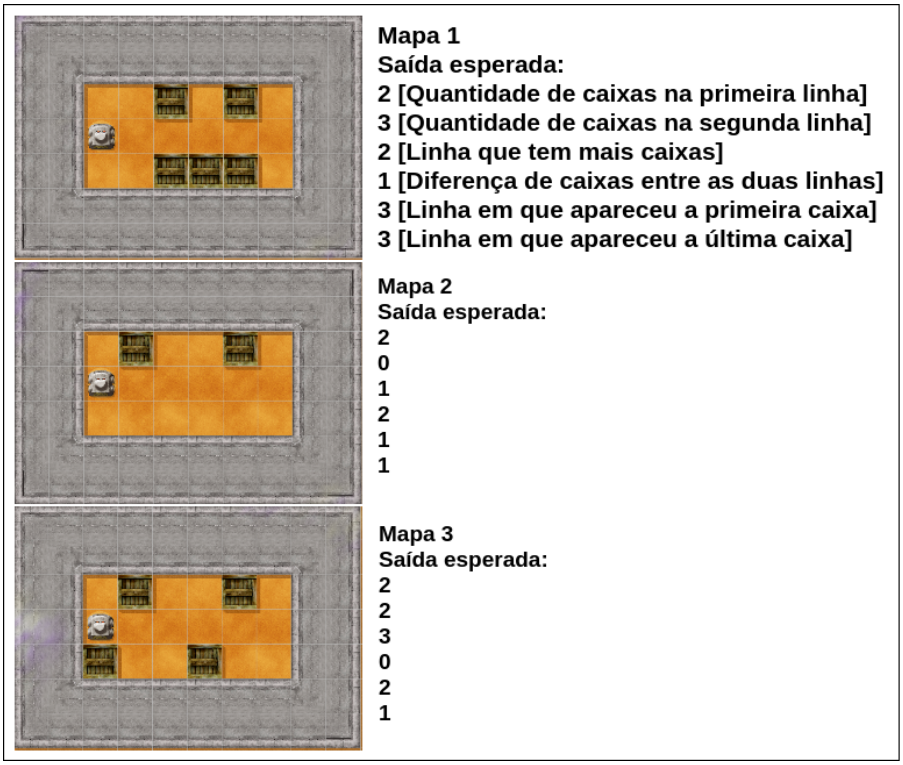
No Capítulo 3 foi apresentado parcialmente o exemplo (Contando Caixas), que é apresentado na íntegra nesta seção e utilizado como estudo de caso para validar o processo de tradução e verificação automática.

O problema Contando Caixas consiste em um desafio proposto para alunos do ensino superior em (BENITTI, 2012) que possui um grau de dificuldade maior do que os outros problemas propostos pelo mesmo autor. A solução deste problema exige uma quantidade maior de linhas de código e exige a utilização de procedimentos e variáveis. Por este motivo, foi escolhido como estudo de caso.

A validação do problema Contando Caixas consiste em três mapas de 6 colunas e 3 linhas que possuem caixas distribuídas aleatoriamente na primeira e última linhas. Em todos os mapas, o robô inicia na primeira coluna da segunda linha e tem o objetivo de andar até a última coluna e parar. Após parar o robô, o programa deve mostra seis valores inteiros através do comando `show`. Os dois primeiros valores representam a quantidade de caixas encontradas na primeira e última linhas, respectivamente. O terceiro valor mostrado indica quais das linhas possui mais caixas: o valor 1 indica que a primeira linha possui mais caixas, o valor 2 que a última linha tem mais caixas, e o

valor 3 que as linhas possuem a mesma quantidade de caixas. O quarto valor exibido é a diferença de caixas entre as duas linhas. O quinto valor indica em qual linha apareceu a primeira caixa: 1 se apareceu na primeira linha, 2 se apareceu na última linha; e 3 se apareceu em ambas as linhas. O sexto e último valor indica em qual linha apareceu a última caixa: 1 se apareceu na primeira linha, 2 se apareceu na última linha; e 3 se apareceu em ambas as linhas. Neste estudo de caso o programa ROBO foi verificado considerando os mapas apresentados na Figura 33. Nesta figura, no lado direito dos mapas mostra os valores esperados como saída pelo programa.

Figura 33 – Mapas do problema “Contando Caixas” e as saídas esperadas



Fonte – (BENITTI et al., 2009)

Como solução candidata para o problema de contar caixas desenvolvemos o programa ROBO apresentado no Código 4.1. Nas linhas 1 a 12 do código, estão declaradas as variáveis. As variáveis `countFirstLine` e `countLastLine` armazenam a quantidade de caixas encontradas pelo robô na primeira e última linhas, nesta ordem. As variáveis `findFirstBoxLeft` e`findFirstBoxRight` apontam quando a primeira caixa é encontrada na primeira e última linhas, respectivamente. As variáveis `firstBoxLeft` e `firstBoxRight` são utilizadas para armazenar a linha que aparece a primeira caixa. Já as variáveis `lastBoxLeft` e `lastBoxRight` armazenam a linha (primeira ou última) que aparece a última caixa. A quantidade de vezes que a instrução `forward` é executada é armazenada na variável `countForward`. Foram implementados alguns procedimentos para cada valor. O procedimento `countBoxes` (linhas 14 a 38) conta a quantidade

de caixas à direita ou à esquerda do robô dependendo do valor do parâmetro `side`: valor 1 conta caixas a esquerda (primeira linha) e qualquer valor diferente de 1 conta caixas a direita (última linha). Para `side` igual a 1, esse procedimento incrementa em 1 a variável `countFirstLine` se à esquerda do robô tem obstáculo, isto é, se possui uma caixa. Em seguida, verifica se a variável `findFirstBoxLeft` é igual a 0, o que significa que ainda não encontrou a primeira caixa, e atualiza o valor de `firstBoxLeft` para a quantidade de comandos `forward` executados (`countForward`), assim, atualizando `findFirstBoxLeft` para o valor 1, indicando que a caixa da primeira linha foi encontrada. Essa explicação é análoga para `side` diferente de 1, o que significa que a verificação ocorre na última linha (à direita do robô). O procedimento `showsMoreBoxes` (linha 40 a 51) exibe no *console* do RoboMind a linha com mais caixas e a diferença de caixas entre a primeira e última linha. O procedimento `getBoxFirstLine` (linhas 53 a 61) exibe qual das linhas há uma ocorrência da primeira caixa. Já o procedimento `getboxLastLine` (linhas 63 a 71) exibe o valor correspondente à linha no qual a última caixa aparece. As instruções que iniciam o programa estão dispostas nas linhas 73 a 88. O primeiro comando é `right`, que altera a orientação do robô para a direção leste. O comando seguinte é o laço `repeatWhile` que executa os comandos em seu corpo enquanto não existe um obstáculo à frente do robô. No escopo desse laço, há duas chamadas para o procedimento `countBoxes`, os valores 1 e 2 são passados no parâmetro para verificar à esquerda e à direita do robô, respectivamente. A próxima instrução é um `forward` para movimentar o robô uma célula à frente e em seguida incrementar o valor de `countForward` em 1. Ao sair do laço, o programa executa novamente o procedimento `countBoxes` para verificar na célula onde o robô parou, se há ocorrência de caixas na primeira e última linhas. As próximas instruções são para exibir as saídas do programa no *console* do ambiente RoboMind: `show(countFirstLine)` e `show(countLastLine)` exibem o primeiro e segundo valor; a chamada do procedimento `showsMoreBoxes` exibe o terceiro e quarto valor; a chamada do procedimento `getBoxFirstLine` exibe o quinto valor; e `getBoxLastLine` exibe o sexto valor.

Código 4.1 – Solução proposta em ROBO para o problema Contando Caixas

```
1  countFirstLine = 0
2  countLastLine = 0
3
4  findFirstBoxLeft = 0
5  findFirstBoxRight = 0
6
7  firstBoxLeft = 10
8  firstBoxRight = 10
9  lastBoxLeft = 0
10 lastBoxRight = 0
11
12 countForward = 0
13
14 procedure countBoxes(side){
15     if(side == 1){
```

```
16          if(leftIsObstacle){
17              countFirstLine = countFirstLine + 1
18
19              if(findFirstBoxLeft == 0){
20                  firstBoxLeft = countForward
21                  findFirstBoxLeft = 1
22              }
23
24              lastBoxLeft = countForward
25          }
26      } else{
27          if(rightIsObstacle){
28              countLastLine = countLastLine + 1
29
30              if(findFirstBoxRight == 0){
31                  firstBoxRight = countForward
32                  findFirstBoxRight = 1
33              }
34
35              lastBoxRight = countForward
36          }
37      }
38  }
39
40  procedure showsMoreBoxes(){
41      if(countFirstLine > countLastLine){
42          show(1)
43          show(countFirstLine - countLastLine)
44      } else if(countFirstLine < countLastLine){
45          show(2)
46          show(countLastLine - countFirstLine)
47      } else{
48          show(3)
49          show(countLastLine - countFirstLine)
50      }
51  }
52
53  procedure getBoxFirstLine(){
54      if(firstBoxLeft < firstBoxRight){
55          show(1)
56      } else if(firstBoxLeft > firstBoxRight){
57          show(2)
58      } else {
59          show(3)
60      }
61  }
62
63  procedure getBoxLastLine(){
64      if(lastBoxLeft > lastBoxRight){
65          show(1)
66      } else if(lastBoxLeft < lastBoxRight){
67          show(2)
68      } else {
69          show(3)
70      }
71  }
72
73  right
```

```
74
75 repeatWhile(frontIsClear){
76     countBoxes(1)
77     countBoxes(2)
78     forward(1)
79     countForward = countForward + 1
80 }
81
82 countBoxes(1)
83 countBoxes(2)
84 show(countFirstLine)
85 show(countLastLine)
86 showsMoreBoxes()
87 getBoxFirstLine()
88 getBoxLastLine()
```

Fonte – O autor

Para validarmos a ferramenta implementada, utilizamos como entrada os três mapas apresentados na Figura 33 e o programa mostrado no Código 4.1. Expomos em tabelas as entradas e saídas para cada dos métodos da classe `RobotChecker`. Na Tabela 1, `program.robo` é a entrada para o método `translateRobo2CSP` e como saída é gerado a especificação formal em CSP (`program.csp`). No método `translateMap2CSP`, map1.map é o arquivo de entrada, e map1.csp é a saída gerada pela tradução do mapa na notação de CSP. No método `verifyAssertion`, a notação CSP do mapa e do programa são entradas para a verificação (`program.csp` e `map1.csp`). Na implementação deste método é criado o arquivo `test1_program_map1.csp` que é responsável por incluir os arquivos CSP utilizados na verificação. Na Figura 34 é mostrado o conteúdo do arquivo `test1_program_map1.csp` que é dado como entrada para FDR realizar a verificação. Neste arquivo, as linhas 1 a 3 incluem as especificações do modelo, do mapa e do programa usando o comando `include` de FDR. A inclusão em um arquivo CSP é feita através do comando `include`. Por exemplo, o comando `include 'program.csp'` copia o conteúdo do arquivo `program.csp` no local onde o comando é utilizado. Na linha 5 está disposto a propriedade a ser verificada por FDR (`assert PROGRAM :[deadlock free [F]]`) que corresponde a verificação se o programa está livre de *deadlock*. As saídas do método `verifyAssertion` são o resultado da verificação da propriedade (`result1`) e os traces gerados (`traces1`). As explicações mencionadas são análogas para a Tabela 2 e para a Tabela 3.

Utilizando a ferramenta apresentada na Figura 32, selecionamos os três mapas apresentados na Figura 33 e o programa apresentado no Código 4.1 como entrada. Executamos a verificação e obtivemos os resultados apresentados na Tabela 4. Para os três mapas o resultado da propriedade `assert PROGRAM :[deadlock free [F]]` falhou, o que significa que para cada um dos mapas o programa ROBO possui *deadlock*, portanto, o programa termina sua execução em todos os mapas. Os contraexemplos

Figura 34 – Exemplo de entrada para a verificação no FDR

```
1 include "robomodel.csp"
2 include "map1.csp"
3 include "program.csp"
4
5 assert PROGRAM :[deadlock free [F]]
```

Fonte – O autor

Tabela 1 Entradas e saídas para o mapa 1

Map 1		
method	**input**	**output**
translateRobo2CSP	program.robo	program.csp
translateMap2CSP	map1.map	map1.csp
verifyAssertion	program.csp, map1.csp, test1_program_map1.csp	result1, traces1

Fonte O autor

Tabela 2 Entradas e saídas para o mapa 2

Map 2		
method	**input**	**output**
translateRobo2CSP	program.robo	program.csp
translateMap2CSP	map2.map	map2.csp
verifyAssertion	program.csp, map2.csp, test2_program_map2.csp	result2, traces2

Fonte O autor

gerados (*traces*) para os três mapas possuem os seis primeiros eventos iguais. Inicialmente, possuem o evento `right`, pois a primeira instrução do programa é virar a direita. Em seguida, realizam um laço que avança a posição do robô enquanto não há uma parede na frente do robô. Como resultado, o trace apresenta cinco vezes o comando `forward`. Os eventos que diferem entre os mapas são os eventos (`show`). Analisando o valor dos eventos `show`, pode-se confirmar que os mapas apresentaram as mesmas saídas apresentadas na Figura 33. As saídas produzidas pela verificação foram comparadas com as saídas apresentadas no *console* do ambiente RoboMind após concluir a execução do mesmo programa. Com esta comparação pode-se confirmar que o resultado da simulação em RoboMind foi idêntico ao resultado mostrado na verificação. O que mostra que o modelo para o programa analisado captura a mesma semântica da simulação de RoboMind.

Tabela 3 Entradas e saídas para o mapa 3

Map 3		
method	**input**	**output**
translateRobo2CSP	program.robo	program.csp
translateMap2CSP	map3.map	map3.csp
verifyAssertion	program.csp, map3.csp, test3_program_map3.csp	result3, traces3

Fonte O autor

Tabela 4 Resultado da verificação usando a ferramenta proposta

Map 1	Map 2	Map 3
assert PROGRAM :[deadlock free [F]]	assert PROGRAM :[deadlock free [F]]	assert PROGRAM :[deadlock free [F]]
Result: Failed	Result: Failed	Result: Failed
Counterexample - Traces:	Counterexample - Traces:	Counterexample - Traces:
right()	right()	right()
forward(1)	forward(1)	forward(1)
forward(1)	forward(1)	forward(1)
forward(1)	forward(1)	forward(1)
forward(1)	forward(1)	forward(1)
forward(1)	forward(1)	forward(1)
show(2)	show(2)	show(2)
show(3)	show(0)	show(2)
show(2)	show(1)	show(3)
show(1)	show(2)	show(0)
show(3)	show(1)	show(2)
show(3)	show(1)	show(1)

Fonte O autor

Coletamos o tempo utilizado pela ferramenta para realizar a verificação do estudo de caso. A verificação levou aproximadamente 2 segundos para cada mapa (o que já inclui o tempo para tradução e verificação), totalizando 6 segundos para os três mapas. Portanto, tendo em vista os resultados no estudo de caso, apesar das suas limitações, a abordagem proposta é capaz de verificar o comportamento de programas ROBO com variáveis e procedimentos para casos de programas escritos utilizando aspectos similares ao programa do estudo de caso em um tempo aceitável.

Este tempo é inferior ao tempo que seria necessário para carregar os mapas e RoboMind, iniciar a simulação para cada mapa e analisar o resultado de cada um deles. Além disto, a ferramenta possibilita mudar o programa que será verificado mantendo os mesmos mapas, isto ajuda muito na verificação de diferentes programas para o mesmo conjunto de mapas.

Outra vantagem da ferramenta em relação a verificação manual é que a ferramenta identifica quando um programa entra em um laço infinito (está livre de *deadlock*). Apenas observando a simulação não é possível afirmar que o programa para quando a simulação demora muito tempo executando.

Para validar a ferramenta com um programa que não termina utilizamos um outro exemplo. O problema chamado *findBeacon*, consiste na busca pelo objeto *beacon*

no mapa. O programa ROBO para esse problema está destacado na Figura 35. Os comandos do robô estão dentro do laço (`repeatWhile`). Esse laço é executado até que o *beacon* não esteja na frente do robô (`frontIsBeacon`), enquanto o mesmo se move para frente se não há obstáculos, caso contrário move para trás e toma uma decisão aleatória (`flipCoin`) para alterar sua orientação à direita (`right`) ou à esquerda (`left`). O comando `flipCoin` de ROBO retorna um valor booleano de forma aleatória. Os mapas de entrada para esse programa estão ilustrados na Figura 36. A diferença entre os eles está na posição do objeto (*beacon*).

Figura 35 – Programa ROBO para o problema *findBeacon*

```
1  repeatWhile(not ( frontIsBeacon() ) )
2  {
3      if(frontIsClear()) {
4          forward(1)
5      }else{
6          backward(1)
7          if( flipCoin() ){
8              right()
9          }else{
10             left()
11         }
12     }
13 }
```

Fonte – O autor

Figura 36 – Mapas para o problema *findBeacon*

Fonte – O autor

Selecionamos os dois mapas da Figura 36 e o programa da Figura 35 e executamos a verificação utilizando a ferramenta. O resultado obtido da execução está apresentado na Tabela 5. Para o Mapa 1 (esquerda da Figura 36) a propriedade `assert PROGRAM :[deadlock free [F]]` falhou, ou seja, o programa possui *deadlock*, isto é, o programa termina sua execução e o contraexemplo é gerado. Já para o Mapa 2 (direita da Figura 36), a propriedade passou no teste. Portanto, o programa ROBO

para o Mapa 2 não possui *deadlock*. Diante disso, nenhum contraexemplo é gerado. A especificação CSP do programa gerada pela abordagem pode ser consultada no Apêndice D.3

Validamos que o comportamento verificado pela ferramenta foi o mesmo que pode ser observado usando a ferramenta RoboMind. Ao utilizar o Mapa 2 em RoboMind a simulação nunca termina, sendo o usuário obrigado a interromper a simulação. Isto acontece, pois, o programa não é capaz de encontrar o objeto no Mapa 2, portanto não termina a execução do laço.

Tabela 5 Resultado da verificação para o problema *findBeacon*

Map 1	Map 2
assert PROGRAM :[deadlock free [F]] Result: Failed Counterexample - Traces: forward(1) backward(1) left() forward(1) forward(1) forward(1) forward(1) forward(1) forward(1) backward(1) left() forward(1) forward(1) forward(1)	assert PROGRAM :[deadlock free [F]] Result: Passed Counterexample - Traces: null

Fonte O autor

Portanto, validamos a abordagem de tradução e verificação automática por meio de dois exemplos. O primeiro para programas ROBO com procedimentos e variáveis, no qual a verificação encontrou contraexemplos para todos os mapas verificados. No segundo exemplo validamos a abordagem por meio de um programa que termina com sucesso em um mapa e para outro mapa sua execução nunca termina. Sendo assim, a ferramenta proposta consegue encapsular a abordagem de tradução e a verificação automática como esperado.

5 Conclusão

Uma importante contribuição deste trabalho é a extensão da atual abordagem de verificação automática de programas ROBO para considerar programas com procedimentos e variáveis utilizando as facilidades do *framework* Spoofax. Para isto foi preciso reformular e estender a gramática SDF3 utilizada para fazer o *parsing* de programas ROBO. Foram incluídos na gramática elementos sintáticos para declaração de variáveis, procedimentos, atualização de variáveis e chamada de procedimentos. Em complemento à extensão da gramática, definimos a semântica em CSP dos elementos sintáticos relacionados a variáveis e procedimentos. A semântica foi definida através da inclusão de regras de transformação escritas em Stratego usadas na tradução dos elementos sintáticos adicionados na gramática para os respectivos elementos em CSP. Outra importante contribuição é uma ferramenta com interface gráfica implementada em Java onde o usuário pode selecionar um programa e um conjunto de mapas para verificar se o programa realiza as ações esperadas nos mapas. A ferramenta torna transparente para os usuários o processo de tradução de ROBO para CSP e a verificação do modelo CSP usando FDR. Em acréscimo à interface, as classes da ferramenta podem ser reusadas para implementação de outras ferramentas, uma vez que encapsulam as principais funções de tradução e verificação que são necessárias para implementação de ferramentas de verificação para ROBO. Validamos a abordagem utilizando a ferramenta para verificar o comportamento de um programa ROBO com variáveis e procedimentos.

Uma das principais limitações da abordagem atual é que a mesma não considera toda a sintaxe da linguagem ROBO, como exemplo, não é possível verificar programas que contém procedimentos que retornam valores. Outra importante limitação é que é preciso definir a faixa de valor para as variáveis. Por último, uma das limitações é que a ferramenta está verificando apenas um tipo de propriedade. Apesar das limitações, o estudo de caso mostra como a ferramenta pode ajudar a aumentar a produtividade na verificação de programas em diferentes mapas.

5.1 Trabalhos Relacionados

O único trabalho encontrado na literatura que realiza verificação de programas de robôs educacionais de forma totalmente automática é (NOGUEIRA et al., 2016). Porém encontramos outro trabalho que usa verificação automática no contexto educacional. O artigo (ROSCOE; HOPKINS, 2007) propõe SVA (*Shared Variable Programming*), que é uma ferramenta para o aprendizado de programação concorrente usando uma

linguagem educacional; esta ferramenta integra um compilador da linguagem educacional para um modelo CSP com o verificador de modelos FDR. SVA permite analisar propriedades de programas concorrentes como, por exemplo, se os programas entram ao mesmo tempo em uma região crítica. Além disso, a ferramenta possui uma interface gráfica que apresenta os resultados das verificações retornados por FDR de uma forma amigável para o usuário. Esse trabalho mostra que é possível utilizar o FDR e CSP para a criação de ferramentas de verificação automática de linguagens de domínio específico.

Em (OLIVEIRA et al., 2017), é proposto um protótipo de um ambiente para avaliação automática de robôs virtuais. Esse trabalho é uma continuação do trabalho (NOGUEIRA et al., 2016) com foco no projeto e prototipação da interface gráfica de um ambiente para avaliação automática de robôs virtuais. O ambiente proposto objetiva a avaliação de forma automática e *feedback* sobre o funcionamento dos programas escritos em ROBO, através de uma interface gráfica que lembra um sistema de julgamento online (*Online Judgment System*) utilizados nas maratonas de programação. A implementação deste ambiente depende diretamente de um dos produtos deste trabalho, que é a extensão do compilador de ROBO para CSP e sua integração com a ferramenta FDR.

Fora do contexto educacional, existem trabalhos que usam verificação de robôs como os trabalhos (SILVA, 2012), (WEBSTER et al., 2014), (ARAUJO; MOTA; NOGUEIRA, 2017) e (MIYAZAWA et al., 2017).

O trabalho (SILVA, 2012) apresenta um método para a verificação automática durante a simulação de futebol de robôs; o trabalho considera a especificação formal e a verificação de planos de um time de robôs simulados. A simulação ocorre de modo que vários robôs são executados ao mesmo tempo em busca de uma solução conjunta (sistema multiagente). Naquele trabalho é utilizado o verificador de modelos UPPAAL[9] para analisar algumas propriedades, dentre elas a verificação da ausência de *deadlock* e *livelock* levando em consideração requisitos de tempo, que é um fator essencial na verificação da simulação de futebol de robôs. Em UPPALL, o modelo do time de robôs é descrito na forma de autômatos temporais (*timed automata*) e as propriedades especificadas na linguagem chamada TCTL (Lógica de Árvore de Cálculo Temporizado). Uma diferença deste trabalho relacionado para o nosso é que a verificação automática em (SILVA, 2012) é aplicada no domínio de sistemas multiagentes e a notação utilizada na especificação são autômatos temporais. O trabalho que propomos esconde do usuário a representação formal do programa do robô a ser verificado.

Webster et al. (2014) propõe a verificação formal de robôs para assistência pessoal. Esses robôs estão presentes nas casas das pessoas e as ajudam em suas tarefas

[9] http://www.uppaal.org/

diárias. Este trabalho usa o popular verificador de modelos SPIN[10] para garantir que os robôs não causem danos às pessoas ou se coloquem em situações inesperadas. O modelo do robô é escrito na notação chamada Brahms[11] (linguagem para modelar processos com multiagentes) que é traduzida para a linguagem PROMELA[12]. Esta última, é a notação de entrada para o verificador SPIN analisar propriedades do robô, como por exemplo, se o robô vai se mover para cozinha quando o usuário enviar o pedido para o robô. O processo de verificação em Webster et al. (2014) é similar a nossa proposta, uma vez que a linguagem para modelagem do robô é traduzida para uma especificação formal através de um compilador. Uma diferença é que o trabalho relacionado trata de uma simulação voltada para o mundo real, enquanto a proposta deste trabalho é para verificação de robôs virtuais.

O trabalho (ARAUJO; MOTA; NOGUEIRA, 2017) propõe uma abordagem para realizar análises em robôs de limpeza (*cleaning robots*) utilizando Verificação de Modelos Probabilísticos (*Probabilistic Model Checking*) através da linguagem e ferramenta PRISM[13]. Nessa abordagem os algoritmos de movimentação do robô são escritos utilizando uma DSL proposta pelos autores, que é traduzida automaticamente para a notação formal PRISM. A notação formal é utilizada como entrada para a ferramenta PRISM que verifica se o algoritmo satisfaz fórmulas temporais probabilísticas, que medem o consumo energético e o tempo que um robô leva para concluir a tarefa de limpeza em um ambiente predefinido. Tanto o trabalho (ARAUJO; MOTA; NOGUEIRA, 2017) como o nosso trabalho geram uma especificação formal partir de uma descrição do programa do robô em uma DSL. Entretanto, enquanto a DSL proposta em (ARAUJO; MOTA; NOGUEIRA, 2017) objetiva a especificação de algoritmos de robôs para limpeza, a linguagem ROBO que é o foco do nosso trabalho objetiva a especificação de robôs educacionais.

O trabalho Miyazawa et al. (2017) propõe a notação gráfica chamada RoboChart que é usada para modelagem de software controlador de robôs. A edição de modelos RoboChart é apoiada pela ferramenta RoboTool[14], que possui um editor gráfico para a notação RoboChart e gera automaticamente modelos CSP a partir dos modelos descritos em RoboChart. A ferramenta é integrada com o verificador de modelos FDR que verifica de forma transparente as propriedades dos modelos RoboChart. A abordagem proposta em (MIYAZAWA et al., 2017) possui uma ideia similar ao nosso trabalho, pois realiza tradução de uma notação de entrada para modelos CSP que são verificados automática usando FDR. Entretanto, as linguagens RoboChart e ROBO possuem objetivos bastante distintos. Enquanto ROBO é uma linguagem textual para

[10] http://spinroot.com/spin/whatispin.html
[11] http://brahms.sourceforge.net/docs/What%20is%20BRAHMS.html
[12] http://spinroot.com/spin/Man/promela.html
[13] http://www.prismmodelchecker.org/
[14] www.cs.york.ac.uk/circus/RoboCalc

fins educacionais, a notação RoboChart é visual e tem foco na especificação de controladores de robôs.

Outro trabalho relacionado é (LEINO, 2008) que propõe Boogie, uma linguagem intermediária desenvolvida para gerar condições e propriedades para a verificação de linguagens de programação de propósito geral, como exemplo, as linguagens C e Spec#. A proposta em (LEINO, 2008) é traduzir automaticamente o código fonte de programas (ex: na linguagem C) para a linguagem Boogie, que permite a análise automática de propriedades do programa através do verificador (também) chamado Boogie. Enquanto (LEINO, 2008) usa Boogie como linguagem intermediária para representar programa escritos em linguagens de programação de propósito geral, nosso trabalho usa a notação formal de CSP para representar programas escritos na linguagem educativa ROBO. Além disto, as propriedades que podem ser verificadas com Boogie e CSP são diferentes. Enquanto Boogie foca na verificação de invariantes do programa, CSP foca na verificação de refinamentos.

Portanto, este trabalho de pesquisa traz várias contribuições para o contexto educacional, uma vez que oferece para estudantes e professores que utilizam ambientes de programação de robôs educacionais um mecanismo de verificação automática para a produção de *feedback* automático sobre a corretude de programas na linguagem ROBO.

5.2 Trabalhos Futuros

A abordagem proposta precisa ser validada com outros exemplos além do exemplo do estudo de caso. Um trabalho futuro consiste em usar outros exemplos para validar a abordagem, além de realizar estudos de caso com potenciais usuários.

Uma das melhorias que pode ser realizada está na criação de regras de tradução que na especificação formal seja proposta uma abordagem que simule uma memória local, dessa forma, mantém apenas as variáveis e parâmetros no escopo de um procedimento. Assim, descartando a adição destes à memória global da especificação, como foi proposto neste trabalho.

Outra melhoria está relacionada com a implementação do compilador para realizar a checagem semântica de ROBO, como por exemplo, checagem de nomes e tipos. Atualmente, a avaliação semântica de ROBO não é realizada pelo compilador, pois espera-se que os programas tenham a semântica validada no ambiente RoboMind antes da tradução para CSP.

Nossa abordagem proposta não considera todos os elementos sintáticos de ROBO, então um importante trabalho futuro seria considerar todos os elementos da

linguagem para a verificação, como exemplo, procedimentos com retorno de valores.

Um trabalho futuro consiste em encontrar representações formais mais eficientes. A atual representação em CSP emula a memória como um processo que é colocado em paralelo com os comandos do programa. À medida em que o tamanho do mapa aumenta e o domínio das variáveis, esta forma de modelagem cria uma grande explosão no número de estados do programa durante a análise. A investigação de abordagens mais compactas no número de estados precisa ser estudada.

Outro trabalho futuro é permitir a especificação das propriedades esperadas para o programa através de uma notação amigável que possa ser traduzida automaticamente para CSP; dispensando assim o conhecimento de CSP.

Referências

ALIMISIS, D. Educational robotics: Open questions and new challenges. v. 6, p. 63–71, 01 2013. Citado na página 13.

ARAUJO, R. P. d.; MOTA, A. C.; NOGUEIRA, S. d. C. Probabilistic analysis applied to cleaning robots. In: *2017 IEEE International Conference on Information Reuse and Integration (IRI)*. [S.l.: s.n.], 2017. p. 275–282. Citado 2 vezes nas páginas 61 e 62.

BENITTI, F. B. V. Exploring the educational potential of robotics in schools: A systematic review. *Computers & Education*, Elsevier BV, v. 58, n. 3, p. 978–988, apr 2012. Disponível em: <https://doi.org/10.1016/j.compedu.2011.10.006>. Citado 2 vezes nas páginas 13 e 51.

BENITTI, F. B. V. et al. Experimentação com robótica educativa no ensino médio: ambiente, atividades e resultados. *XXIX Congresso da Sociedade Brasileira de Computação*, p. 1811–1820, 2009. Citado 2 vezes nas páginas 34 e 52.

BHATT, D. et al. Opportunities and challenges for formal methods tools in the certification of avionics software. In: *2017 IEEE Aerospace Conference*. [S.l.: s.n.], 2017. p. 1–20. Citado na página 18.

BOMBASAR, J. et al. Ferramentas para o ensino-aprendizagem do pensamento computacional: onde está alan turing? In: *Anais do XXVI Simpósio Brasileiro de Informática na Educação (SBIE 2015)*. Sociedade Brasileira de Computação - SBC, 2015. Disponível em: <https://doi.org/10.5753/cbie.sbie.2015.81>. Citado na página 13.

CLARKE, E. M.; WING, J. M. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 28, n. 4, p. 626–643, dez. 1996. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/242223.242257>. Citado na página 18.

CLEAVELAND, R.; ROSCOE, A. W.; SMOLKA, S. A. Process algebra and model checking. In: *Handbook of Model Checking*. Springer International Publishing, 2018. p. 1149–1195. Disponível em: <https://doi.org/10.1007/978-3-319-10575-8_32>. Citado 2 vezes nas páginas 14 e 19.

DUARTE, L. Integrating software testing and model checking through model extraction. 09 2011. Citado na página 14.

FEIGN, B. Model checking. *University College London*, 2005. Citado na página 19.

GANNON, J. D.; PURTILO, J. M.; ZELKOWITZ, M. Software specification: A comparison of formal methods. 03 2001. Citado na página 18.

GIBSON-ROBINSON, T. et al. Fdr3 — a modern refinement checker for csp. In: ÁBRAHÁM, E.; HAVELUND, K. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. p. 187–201. ISBN 978-3-642-54862-8. Citado 3 vezes nas páginas 14, 24 e 25.

KATS, L. C.; VISSER, E. The spoofax language workbench: Rules for declarative specification of languages and ides. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. New York, NY, USA: ACM, 2010. (OOPSLA '10), p. 444–463. ISBN 978-1-4503-0203-6. Disponível em: <http://doi.acm.org/10.1145/1869459.1869497>. Citado 4 vezes nas páginas 15, 26, 27 e 29.

LEINO, R. This is boogie 2. In: . Microsoft Research, 2008. Disponível em: <https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/>. Citado na página 63.

LESSA, V. et al. Programação de computadores e robótica educativa na escola: tendências evidenciadas nas produções do workshop de informática na escola. In: *Anais do XXI Workshop de Informática na Escola (WIE 2015)*. Sociedade Brasileira de Computação - SBC, 2015. Disponível em: <https://doi.org/10.5753/cbie.wie.2015.92>. Citado na página 13.

METABORG. *Spoofax*. 2018. Disponível em: <http://www.metaborg.org/en/latest/index.html>. Citado 2 vezes nas páginas 28 e 29.

MIYAZAWA, A. et al. Automatic property checking of robotic applications. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. [S.l.: s.n.], 2017. p. 3869–3876. ISSN 2153-0866. Citado 3 vezes nas páginas 18, 61 e 62.

NOGUEIRA, S. et al. An approach for verifying educational robots. In: RIBEIRO, L.; LECOMTE, T. (Ed.). *Formal Methods: Foundations and Applications*. Cham: Springer International Publishing, 2016. p. 59–77. ISBN 978-3-319-49815-7. Citado 11 vezes nas páginas 14, 20, 22, 24, 29, 31, 45, 48, 51, 60 e 61.

OLIVEIRA, E. et al. Ambiente para avaliação automática de robôs virtuais: uma forma de apoio à aprendizagem de robótica. *Anais II Congresso sobre Tecnologias na Educação (Ctrl+E 2017)*, p. 590–596, 2017. UFPB. Paraíba – Brasil. Citado na página 61.

PERNAMBUCO, F. de. *Pernambuco na onda da Robótica*. 2018. Disponível em: <https://www.folhape.com.br/noticias/noticias/dez-anos-de-educacao-em-pernambuco/2016/10/10/NWS,1832,70,512,NOTICIAS,2190-PERNAMBUCO-ONDA-ROBOTICA.aspx>. Citado na página 13.

ROSCOE, A. W. Parallel operators. In: *Texts in Computer Science*. Springer London, 2010. p. 45–66. Disponível em: <https://doi.org/10.1007/978-1-84882-258-0_3>. Citado na página 20.

ROSCOE, A. W.; HOPKINS, D. SVA, a tool for analysing shared-variable programms. In: *Proceedings of AVoCS 2007*. [s.n.], 2007. p. 177–183. To appear. Disponível em: <http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/119.pdf>. Citado na página 60.

SILVA, E. et al. Análise de ferramentas para o ensino de computação na educação básica. *XXXIV Congresso da Sociedade Brasileira de Computação (CSBC)*, 2014. Citado na página 13.

SILVA, R. C. B. A. D. Verificação formal de planos para agentes autônomos e multiagentes: Um estudo de caso aplicado ao futebol de robôs. *Universidade Federal da Bahia*, 2012. Dissertação de Mestrado. Citado na página 18.

SILVA, R. C. B. A. da. Verificação formal de planos para agentes autônomos e sistemas multiagentes: um estudo de caso aplicado ao futebol de robôs. feb 2012. Dissertação de Mestrado. Disponível em: <http://repositorio.ufba.br/ri/handle/ri/21342>. Citado na página 61.

WEBSTER, M. et al. *Formal Verification of an Autonomous Personal Robotic Assistant*. 2014. Disponível em: <https://www.aaai.org/ocs/index.php/SSS/SSS14/paper/view/7734>. Citado 2 vezes nas páginas 61 e 62.

ZHAO, Y.; ROZIER, K. Y. Formal specification and verification of a coordination protocol for an automated air traffic control system. *Science of Computer Programming*, v. 96, p. 337 – 353, 2014. ISSN 0167-6423. Special Issue on Automated Verification of Critical Systems (AVoCS 2012). Disponível em: <http://www.sciencedirect.com/science/article/pii/S016764231400166X>. Citado na página 19.

A Gramática em SDF3

Código A.1 – Módulo Common

```
module Common

lexical syntax

  ID              = [a-zA-Z] [a-zA-Z0-9]*
  INT             = "-"? [0-9]+
  STRING          = "\"" StringChar* "\""
  StringChar      = ~[\"\n]
  StringChar      = "\\\""
  StringChar      = BackSlashChar
  BackSlashChar   = "\\"
  LAYOUT          = [\ \t\n\r]
  CommentChar     = [\*]
  LAYOUT          = "/*" InsideComment* "*/"
  InsideComment   = ~[\*]
  InsideComment   = CommentChar
  LAYOUT          = "//" ~[\n\r]* NewLineEOF
  NewLineEOF      = [\n\r]
  NewLineEOF      = EOF
  EOF             =

lexical restrictions

  // Ensure greedy matching for lexicals

  CommentChar   -/- [\/]
  INT           -/- [0-9]
  ID            -/- [a-zA-Z0-9\_]

  // EOF may not be followed by any char

  EOF           -/- ~[]

  // Backslash chars in strings may not be followed by "

  BackSlashChar -/- [\"]

context-free restrictions

  // Ensure greedy matching for comments

  LAYOUT? -/- [\ \t\n\r]
  LAYOUT? -/- [\/].[\/]
  LAYOUT? -/- [\/].[\*]
```

Fonte – O autor

Código A.2 – Módulo ExpressionsBoolean

```
module ExpressionsBoolean

```

```
3  imports
4
5    Common
6    ExpressionsMath
7
8  sorts ExprBoolean
9
10 context-free syntax
11
12     ExprBoolean = <(<ExprBoolean>)> {bracket}
13
14     ExprBoolean.True      = <true>
15     ExprBoolean.False       = <false>
16
17     ExprBoolean.Not       = <not <ExprBoolean>> //Negates the
       value of the argument
18     ExprBoolean.NotSimb     = <~ <ExprBoolean>>
19     ExprBoolean.NotVar      = <not <ExprMath>>
20     ExprBoolean.NotVarSimb    = <~ <ExprMath>>
21
22     ExprBoolean.And       = <<ExprBoolean> and <ExprBoolean>> {
       assoc}//Only true when both arguments are true
23     ExprBoolean.AndSimb     = <<ExprBoolean> & <ExprBoolean>> {
       assoc}
24
25     ExprBoolean.Or        = <<ExprBoolean> or <ExprBoolean>> {
       assoc} //True when at least one of the arguments is true
26     ExprBoolean.OrSimb      = <<ExprBoolean> | <ExprBoolean>> {
       assoc}
27
28     ExprBoolean.EqMath      = <<ExprMath> == <ExprMath>> {non-
       assoc} //Check for equality
29     ExprBoolean.NeqMath     = <<ExprMath> ~= <ExprMath>> {non-
       assoc} //Check for inequality
30     ExprBoolean.EqBoolean   = <<ExprBoolean> == <ExprBoolean>>
       {non-assoc} //Check for equality
31     ExprBoolean.NeqBoolean    = <<ExprBoolean> ~= <ExprBoolean
       >> {non-assoc} //Check for inequality
32     ExprBoolean.LessT     = [[ExprMath] < [ExprMath]] {left}
       //"less than"
33     ExprBoolean.LessTorE    = [[ExprMath] <= [ExprMath]] {left}
        //"less than or equals"
34     ExprBoolean.GreaterT    = [[ExprMath] > [ExprMath]] {left}
       //"greater than"
35     ExprBoolean.GreaterTorE   = [[ExprMath] >= [ExprMath]] {
       left} //"greater than or equals"
36
37   //ProgrammingStructures
38
39     ExprBoolean.FLIPCOIN  = < flipCoin() > //Flip a coin to
       make a random choice. flipCoin() will either be true or
       false with a chance of 50%-50%.
40
41     ExprBoolean.LEFISOBS  = < leftIsObstacle >
42     ExprBoolean.LEFISCLR  = < leftIsClear >
43     ExprBoolean.LEFISBEA  = < leftIsBeacon >
44     ExprBoolean.LEFISWTE  = < leftIsWhite >
45     ExprBoolean.LEFISBLK  = < leftIsBlack >
```

```
46
47      ExprBoolean.FROISOBS  = < frontIsObstacle >
48      ExprBoolean.FROISCLR  = < frontIsClear >
49      ExprBoolean.FROISBEA  = < frontIsBeacon >
50      ExprBoolean.FROISWTE  = < frontIsWhite >
51      ExprBoolean.FROISBLK  = < frontIsBlack >
52
53      ExprBoolean.RIGISOBS  = < rightIsObstacle >
54      ExprBoolean.RIGISCLR  = < rightIsClear >
55      ExprBoolean.RIGISBEA  = < rightIsBeacon >
56      ExprBoolean.RIGISWTE  = < rightIsWhite >
57      ExprBoolean.RIGISBLK  = < rightIsBlack >
58
59      ExprBoolean.LEFISOBSP = < leftIsObstacle() >
60      ExprBoolean.LEFISCLRP = < leftIsClear() >
61      ExprBoolean.LEFISBEAP = < leftIsBeacon() >
62      ExprBoolean.LEFISWTEP = < leftIsWhite() >
63      ExprBoolean.LEFISBLKP = < leftIsBlack() >
64
65      ExprBoolean.FROISOBSP = < frontIsObstacle() >
66      ExprBoolean.FROISCLRP = < frontIsClear() >
67      ExprBoolean.FROISBEAP = < frontIsBeacon() >
68      ExprBoolean.FROISWTEP = < frontIsWhite() >
69      ExprBoolean.FROISBLKP = < frontIsBlack() >
70
71      ExprBoolean.RIGISOBSP = < rightIsObstacle() >
72      ExprBoolean.RIGISCLRP = < rightIsClear() >
73      ExprBoolean.RIGISBEAP = < rightIsBeacon() >
74      ExprBoolean.RIGISWTEP = < rightIsWhite() >
75      ExprBoolean.RIGISBLKP = < rightIsBlack() >
76
77  context-free priorities
78
79      {left: ExprBoolean.EqMath
80      ExprBoolean.NeqMath
81      ExprBoolean.EqBoolean
82      ExprBoolean.NeqBoolean
83      ExprBoolean.LessT
84      ExprBoolean.LessTorE
85      ExprBoolean.GreaterT
86      ExprBoolean.GreaterTorE}
87      >{left: ExprBoolean.Not ExprBoolean.NotSimb}
88      >{left: ExprBoolean.And ExprBoolean.AndSimb}
89      >{left: ExprBoolean.Or ExprBoolean.OrSimb}
90
91  lexical syntax // reserved words
92      Keyword = "true"
93      Keyword = "false"
```

Fonte – O autor

Código A.3 – Módulo ExpressionsMath

```
1  module ExpressionsMath
2
3  imports
4
5    Common
```

```
6
7  sorts ExprMath
8
9  context-free syntax
10
11     //arithmetic
12     ExprMath = <(<ExprMath>)> {bracket}
13     ExprMath.Var    = ID
14     ExprMath.Num  = INT
15     ExprMath.Min  = <-<ExprMath>>
16     ExprMath.Add  = <<ExprMath> + <ExprMath>> {left}
17     ExprMath.Sub  = <<ExprMath> - <ExprMath>> {left}
18     ExprMath.Mul  = <<ExprMath> * <ExprMath>> {left}
19     ExprMath.Div  = <<ExprMath> / <ExprMath>> {left}
20
21
22 context-free priorities
23
24      ExprMath.Min
25      > {left: ExprMath.Mul ExprMath.Div}
26      > {left: ExprMath.Add ExprMath.Sub}
```

Fonte – O autor

Código A.4 – Módulo Robo2CSP

```
1  module Robo2CSP
2
3  imports
4
5    Common
6    ExpressionsBoolean
7    ExpressionsMath
8
9  sorts Else
10
11 context-free start-symbols
12
13   Start
14
15 context-free syntax
16
17     Start.Program = <<{Statement "\n"}*>>
18
19   Statement.Instr = <<Instr>>
20   Statement.Declaration = <<Declaration>>
21
22   Declaration.Variable = <<Identifier> = <Expr>>
23   Declaration.Procedure = <procedure <Identifier>{ <{Statement
       "\n"}*> }>
24   Declaration.ProcedureParam = <procedure <Identifier> <Params>
       { <{Statement "\n"}*> }>
25
26   Params.Params = <(<{ Identifier ", "}*>)>
27   Identifier.ID = <<ID>>
28
29   Instr.ProcCall = <<Identifier> <TypeParams>>
30   //TypeParams.ProcParams = <(<{ Call ", "}*>)>
```

```
31    TypeParams.ExprParams = <(<{ Expr ", "}*>)>
32
33    Expr.ExpBool = <<ExprBoolean>>
34    Expr.ExpMath = <<ExprMath>>
35
36  //Basic instructions
37
38    Instr.SHOW       = < show(<Expr>) >
39
40    //Move
41
42    Instr.FORWARD = < forward(<ExprMath>) > //Move n steps
       forward
43    Instr.BACKWARD  = < backward(<ExprMath>) > //Move n steps
       backward
44    Instr.LEFT    = < left > //Turn left over 90 degrees
45      Instr.LEFTP   = < left() > //Turn left over 90 degrees
46      Instr.RIGHT   = < right > //Turn right over 90 degrees
47    Instr.RIGHTP  = < right() > //Turn right over 90 degrees
48      Instr.NORTH   = < north(<ExprMath>) > //Turn to head north
       and move n steps forward
49    Instr.SOUTH   = < south(<ExprMath>) > //Turn to head south
       and move n steps forward
50    Instr.EAST    = < east(<ExprMath>) > //Turn to head east and
       move n steps forward
51    Instr.WEST    = < west(<ExprMath>) > //Turn to head west and
       move n steps forward
52
53
54    //Paint
55
56    Instr.PAINTWTE  = < paintWhite > //Put the brush with white
       paint to the ground.
57    Instr.PAINTWTEP = < paintWhite() > //Put the brush with white
        paint to the ground.
58    Instr.PAINTBLK  = < paintBlack > //Put the brush with black
       paint to the ground.
59    Instr.PAINTBLKP = < paintBlack() > //Put the brush with black
        paint to the ground.
60    Instr.STOPPAINT = < stopPainting > //Stop painting, hide the
       brush
61    Instr.STOPPAINTP  = < stopPainting() > //Stop painting, hide
       the brush
62
63    //Grab
64
65    Instr.PICKUPP = < pickUp() > //Get the beacon in front of the
        robot
66    Instr.PICKUP  = < pickUp > //Get the beacon in front of the
       robot
67    Instr.PUTDOWNP  = < putDown() > //Put a beacon in front of
       the robot
68    Instr.PUTDOWN = < putDown > //Put a beacon in front of the
       robot
69    Instr.EATUPP  = < eatUp() > //Pick up and destroy the beacon
       in front.
70    Instr.EATUP   = < eatUp > //Pick up and destroy the beacon in
        front.
71
```

```
72
73 //Programming structures
74
75   //Loops
76
77   Instr.RPTINT  = < repeat (<ExprMath?>) {<{Statement "\n"}*>}>
       //repeats the instructions between curly brackets exactly n
       times.
78   Instr.RPTINF  = < repeat {<{Statement "\n"}*>}> // just keeps
       repeating the instructions between curly brackets for ever.
79   Instr.RPTWLE  = < repeatWhile (<ExprBoolean>) {<{Statement "\
      n"}*>}> // repeats the instructions between curly brackets
      as long as the condition holds. This condition must be a
      perception/seeing instruction (for example frontIsClear)
80   Instr.BREAK   = < break > // allows you to jump out of the
      loop (e.g. a repeat section) so it stops performing the
      instructions between curly brackets. The robot will resume
      performing the instructions left after the closing curly
      bracket of the loop.
81
82   // If-structures
83
84   Instr.IF    = < if (<ExprBoolean>) {<{Statement "\n"}*>} <
      Else?>>  // will perform the the instructions between curly
      brackets, only if the condition holds. Else the robot
      immediately steps to the instructions written after the
      closing curly bracket. The condition must be a perception/
      seeing instruction (for example: frontIsClear)
85   Else.ELSEIF   = < else if (<ExprBoolean>) {<{Statement "\n
      "}*>} <Else?>>
86   Else.ELSE     = < else {<{Statement "\n"}*>} >
87
88   //end
89
90   Instr.END = < end > // will cause the entire program to stop
      when this instruction is performed
91
92 context-free priorities
93
94   Instr.IF > Else.ELSEIF > Else.ELSE
95
96 // reserved words
97 lexical syntax
98   ID      = Keyword {reject}
99   Keyword = "True"
100  Keyword = "repeat"
101  Keyword = "repeatWhile"
102  Keyword = "break"
103  Keyword = "if"
104  Keyword = "else"
105  Keyword = "north"
106  Keyword = "south"
107  Keyword = "east"
108  Keyword = "west"
109  Keyword = "right"
110  Keyword = "left"
111  Keyword = "forward"
112  Keyword = "backward"
113  Keyword = "paintWhite"
```

```
114   Keyword = "paintBlack"
115   Keyword = "stopPainting"
116   Keyword = "pickUp"
117   Keyword = "putDown"
118   Keyword = "eatUp"
119   Keyword = "show"
120   Keyword = "COMMANDS"
121   Keyword = "INIT"
122   Keyword = "VarType"
123   Keyword = "MAXVAR"
```

Fonte – O autor

B Árvore de Sintaxe Abstrata - AST

Código B.1 – AST do programa ROBO para o problema Contando Caixas (adaptado)

```
Program(
  [ Declaration(Variable(ID("counter"), ExpMath(Num("0"))))
  , Declaration(Variable(ID("lookLeft"), ExpMath(Num("1"))))
  , Declaration(
      ProcedureParam(
        ID("countBoxes")
      , Params([ID("side")])
      , [ Instr(
            IF(
              EqMath(Var("side"), Num("1"))
            , [ Instr(
                  IF(
                    LEFISOBS()
                  , [ Declaration(
                        Variable(
                          ID("counter")
                        , ExpMath(Add(Var("counter"), Num("1"))
  )
                        )
                    )
                  ]
                , None()
                )
              )
            ]
          , Some(
              ELSE(
                [ Instr(
                    IF(
                      RIGISOBS()
                    , [ Declaration(
                          Variable(
                            ID("counter")
                          , ExpMath(Add(Var("counter"), Num
  ("1")))
                          )
                      )
                    ]
                  , None()
                  )
                )
              ]
            )
          )
        )
      ]
    )
  , Instr(RIGHT())
  , Instr(
```

```
50        RPTWLE(
51          FROISCLR()
52        , [ Instr(
53              ProcCall(ID("countBoxes"), ExprParams([ExpMath(Var
      ("lookLeft"))]))
54            )
55          , Instr(FORWARD(Num("1")))
56          ]
57        )
58      )
59    , Instr(
60        ProcCall(ID("countBoxes"), ExprParams([ExpMath(Var("
      lookLeft"))]))
61      )
62    , Instr(SHOW(ExpMath(Var("counter"))))
63    ]
64 )
```

Fonte – O autor

C Regras em Stratego

Código C.1 – Padrões da regra <to-csp-e>

```
1  to-csp-e:
2    Var(name) -> $[[name]Var]
3
4  to-csp-e:
5    Num(val) -> $[[val]]
6
7  to-csp-e:
8    Add(expression1,expression2)-> $[([expression1'] + [
       expression2'])]
9    with
10       expression1':= <to-csp-e> expression1
11   with
12       expression2':= <to-csp-e> expression2
13
14 to-csp-e:
15   True() ->$[true]
16
17  to-csp-e:
18   False()->$[false]
19
20 to-csp-e:
21   FROISCLR() -> $[frontIsClear(x,y,o,bx,by)]
22
23 to-csp-e:
24     LEFISOBS() -> $[leftIsObstacle(x,y,o)]
25
26 to-csp-e:
27     RIGISOBS() -> $[rightIsObstacle(x,y,o)]
```

Fonte – O autor

Código C.2 – Padrões da regra <to-csp>

```
1  to-csp:
2    RPTWLE(expression,whileBody) ->
3    $[(let
4         WHILE =
5         get.X?x ->
6         get.Y?y ->
7         get.ORIENTATION?o ->
8         get.BX?bx ->
9         get.BY?by ->
10        coin?c ->
11        [vars']
12          if ([expression']) then (
13            [whileBody']
14            ;
15            WHILE
16          ) else (
17            SKIP
18          )
```

```
19        within
20        WHILE
21      )
22      ;
23    ]
24    with
25     vars' := <put-get-var-exp-analyze> <get-vars-exp> expression
26      with
27       expression' := <to-csp-e> expression
28      with
29       whileBody'  := <statement-definition> whileBody
30
31  to-csp:
32    FORWARD(n) ->
33    $[[vars']
34    FORWARD([n'])
35      ;
36    ]
37    with
38      vars' := <put-get-var-exp-analyze> <get-vars-exp> n;
39      n' := <to-csp-e> n
40
41   to-csp:
42    IF(expression,ifBody,elseBody) ->
43      $[(let
44           IFELSE =
45           get.X?x ->
46         get.Y?y ->
47         get.ORIENTATION?o ->
48         get.BX?bx ->
49         get.BY?by ->
50         coin?c ->
51         [vars']
52         if([expression']) then (
53           [ifBody']
54         ) else (
55           [elseBody']
56         )
57       within
58         IFELSE
59      )
60      ;
61       ]
62       with
63        vars' := <put-get-var-exp-analyze> <get-vars-exp>
     expression
64       with
65        expression' := <to-csp-e> expression
66       with
67        ifBody'     := <statement-definition> ifBody
68     with
69        elseBody'   := <to-csp> elseBody
```

Fonte – O autor

D Especificação em CSP

Código D.1 – Especificação CSP do problema Contando Caixas (adaptado)

```
counterConst = 0
lookLeftConst = 1

nametype INTVALUES = {0..11}
datatype VarType = side.INTVALUES |counter.INTVALUES | lookLeft.INTVALUES |  X.TX | Y.TY | ORIENTATION.TO | BX.TBX | BY.TBY
INIT = { (side,0),(counter,0),(lookLeft,1), (X, startX), (Y, startY), (ORIENTATION, NORTH_), (BX, startBX), (BY, startBY) }

countBoxesProc(sideParam) =
  set.side!(sideParam) ->
  (let
     IFELSE =
       get.X?x ->
       get.Y?y ->
       get.ORIENTATION?o ->
       get.BX?bx ->
       get.BY?by ->
       coin?c ->
       get.side?sideVar ->
       if((sideVar == 1)) then (
         (let
            IFONLY =
              get.X?x ->
             get.Y?y ->
             get.ORIENTATION?o ->
             get.BX?bx ->
             get.BY?by ->
             coin?c ->
             if(leftIsObstacle(x,y,o)) then (
               get.counter?counterVar ->
               get.counter?counterVar ->
               member((counterVar + 1), INTVALUES) & set.counter!((counterVar + 1)) ->
               SKIP
             ) else (
               SKIP
             )
           within
             IFONLY
         )
         ;
         SKIP
       ) else (
         (let
            IFONLY =
              get.X?x ->
             get.Y?y ->
             get.ORIENTATION?o ->
             get.BX?bx ->
```

```
48                get.BY?by ->
49                coin?c ->              if(rightIsObstacle(x,y,o))
    then (
50                  get.counter?counterVar ->
51                  get.counter?counterVar ->
52                  member((counterVar + 1), INTVALUES) & set.
    counter!((counterVar + 1)) ->
53                  SKIP
54                ) else (
55                  SKIP
56                )
57              within
58                IFONLY
59            )
60            ;
61            SKIP
62          )
63        within
64          IFELSE
65    )
66    ;
67    SKIP
68
69  COMMANDS =
70    RIGHT
71    ;
72    (let
73       WHILE =
74         get.X?x ->
75         get.Y?y ->
76         get.ORIENTATION?o ->
77         get.BX?bx ->
78         get.BY?by ->
79         coin?c ->
80         if (frontIsClear(x,y,o,bx,by)) then (
81           get.lookLeft?lookLeftVar ->
82           countBoxesProc(lookLeftVar)
83           ;
84           FORWARD(1)
85             ;
86           SKIP
87           ;
88           WHILE
89         ) else (
90           SKIP
91         )
92      within
93        WHILE
94    )
95    ;
96    get.lookLeft?lookLeftVar ->
97    countBoxesProc(lookLeftVar)
98    ;
99    get.counter?counterVar ->
100   showInt!(counterVar) ->
101   SKIP
```

Fonte – O autor

Código D.2 – Especificação CSP do problema Contando Caixas

```
countFirstLineConst = 0
countLastLineConst = 0
findFirstBoxLeftConst = 0
findFirstBoxRightConst = 0
firstBoxLeftConst = 10
firstBoxRightConst = 10
lastBoxLeftConst = 0
lastBoxRightConst = 0
countForwardConst = 0

nametype INTVALUES = {0..11}
datatype VarType = side.INTVALUES |countFirstLine.INTVALUES |
   countLastLine.INTVALUES | findFirstBoxLeft.INTVALUES |
   findFirstBoxRight.INTVALUES | firstBoxLeft.INTVALUES |
   firstBoxRight.INTVALUES | lastBoxLeft.INTVALUES |
   lastBoxRight.INTVALUES | countForward.INTVALUES |  X.TX | Y.
   TY | ORIENTATION.TO | BX.TBX | BY.TBY
INIT = { (side,0),(countFirstLine,0),(countLastLine,0),(
   findFirstBoxLeft,0),(findFirstBoxRight,0),(firstBoxLeft,10)
   ,(firstBoxRight,10),(lastBoxLeft,0),(lastBoxRight,0),(
   countForward,0), (X, startX), (Y, startY), (ORIENTATION,
   NORTH_), (BX, startBX), (BY, startBY) }

countBoxesProc(sideParam) =
  set.side!(sideParam) ->
  (let
     IFELSE =
       get.X?x ->
       get.Y?y ->
       get.ORIENTATION?o ->
       get.BX?bx ->
       get.BY?by ->
       coin?c ->
       get.side?sideVar ->
       if((sideVar == 1)) then (
         (let
            IFONLY =
              get.X?x ->
             get.Y?y ->
             get.ORIENTATION?o ->
             get.BX?bx ->
             get.BY?by ->
             coin?c ->
             if(leftIsObstacle(x,y,o)) then (
               get.countFirstLine?countFirstLineVar -> get.
   countFirstLine?countFirstLineVar ->
               member((countFirstLineVar + 1), INTVALUES) & set
   .countFirstLine!((countFirstLineVar + 1)) ->
               (let
                  IFONLY =
                    get.X?x ->
                   get.Y?y ->
                   get.ORIENTATION?o ->
                   get.BX?bx ->
                   get.BY?by ->
```

```
45                    coin?c ->
46                    get.findFirstBoxLeft?findFirstBoxLeftVar ->
47                    if((findFirstBoxLeftVar == 0)) then (
48                      get.firstBoxLeft?firstBoxLeftVar ->
49                      get.countForward?countForward ->
50                      member(countForwardVar, INTVALUES) & set.
    firstBoxLeft!(countForwardVar) ->
51                      get.findFirstBoxLeft?findFirstBoxLeftVar
    ->
52                      member(1, INTVALUES) & set.
    findFirstBoxLeft!(1) ->
53                      SKIP
54                    ) else (
55                      SKIP
56                    )
57                  within
58                    IFONLY
59                )
60                ;
61                get.lastBoxLeft?lastBoxLeftVar ->
62                get.countForward?countForward ->
63
64                member(countForwardVar, INTVALUES) & set.
    lastBoxLeft!(countForwardVar) ->
65                SKIP
66              ) else (
67                SKIP
68              )
69            within
70              IFONLY
71          )
72          ;
73          SKIP
74        ) else (
75          (let
76             IFONLY =
77               get.X?x ->
78              get.Y?y ->
79              get.ORIENTATION?o ->
80              get.BX?bx ->
81              get.BY?by ->
82              coin?c ->
83              if(rightIsObstacle(x,y,o)) then (
84                get.countLastLine?countLastLineVar ->
85                get.countLastLine?countLastLineVar ->
86                member((countLastLineVar + 1), INTVALUES) & set.
    countLastLine!((countLastLineVar + 1)) ->
87                (let
88                   IFONLY =
89                     get.X?x ->
90                    get.Y?y ->
91                    get.ORIENTATION?o ->
92                    get.BX?bx ->
93                    get.BY?by ->
94                    coin?c ->
95                    get.findFirstBoxRight?findFirstBoxRightVar
    ->
96                    if((findFirstBoxRightVar == 0)) then (
```

```
97                      get.firstBoxRight?firstBoxRightVar ->
98                      get.countForward?countForwardVar ->
99
100                     member(countForwardVar, INTVALUES) & set.
    firstBoxRight!(countForwardVar) ->
101                     get.findFirstBoxRight?findFirstBoxRightVar
     ->
102
103                     member(1, INTVALUES) & set.
    findFirstBoxRight!(1) ->
104                     SKIP
105                   ) else (
106                     SKIP
107                   )
108                 within
109                   IFONLY
110               )
111               ;
112
113               get.lastBoxRight?lastBoxRightVar ->
114               get.countForward?countForwardVar ->
115               member(countForwardVar, INTVALUES) & set.
    lastBoxRight!(countForwardVar) ->
116               SKIP
117             ) else (
118               SKIP
119             )
120           within
121             IFONLY
122         )
123         ;
124         SKIP
125       )
126     within
127       IFELSE
128   )
129   ;
130   SKIP
131
132 showsMoreBoxesProc() =
133   (let
134     IFELSE =
135       get.X?x ->
136       get.Y?y ->
137       get.ORIENTATION?o ->
138       get.BX?bx ->
139       get.BY?by ->
140       coin?c ->
141       get.countFirstLine?countFirstLineVar ->
142       get.countLastLine?countLastLineVar ->
143       if((countFirstLineVar > countLastLineVar)) then (
144         showInt!(1) ->
145         get.countFirstLine?countFirstLineVar ->
146         get.countLastLine?countLastLineVar ->
147         showInt!((countFirstLineVar - countLastLineVar)) ->
148         SKIP
149       ) else (
150         (let
```

```
151             ELSEIFELSE =
152               get.X?x ->
153               get.Y?y ->
154               get.ORIENTATION?o ->
155               get.BX?bx ->
156               get.BY?by ->
157              coin?c ->
158              get.countFirstLine?countFirstLineVar ->
159              get.countLastLine?countLastLineVar ->
160               if((countFirstLineVar < countLastLineVar)) then (
161                 showInt!(2) ->
162                 get.countLastLine?countLastLineVar ->
163                 get.countFirstLine?countFirstLineVar ->
164                 showInt!((countLastLineVar - countFirstLineVar)
    ) ->
165                 SKIP
166               ) else (
167                 showInt!(3) ->
168                 get.countLastLine?countLastLineVar ->
169                 get.countFirstLine?countFirstLineVar ->
170
171                 showInt!((countLastLineVar - countFirstLineVar)
    ) ->
172                 SKIP
173               )
174             within
175               ELSEIFELSE
176         )
177       )
178     within
179       IFELSE
180   )
181   ;
182   SKIP
183
184 getBoxFirstLineProc() =
185   (let
186      IFELSE =
187        get.X?x ->
188        get.Y?y ->
189        get.ORIENTATION?o ->
190        get.BX?bx ->
191        get.BY?by ->
192        coin?c ->
193        get.firstBoxLeft?firstBoxLeftVar ->
194        get.firstBoxRight?firstBoxRightVar ->
195        if((firstBoxLeftVar < firstBoxRightVar)) then (
196          showInt!(1) ->
197          SKIP
198        ) else (
199          (let
200             ELSEIFELSE =
201               get.X?x ->
202               get.Y?y ->
203               get.ORIENTATION?o ->
204               get.BX?bx ->
205               get.BY?by ->
206              coin?c ->
```

```
207                get.firstBoxLeft?firstBoxLeftVar ->
208                get.firstBoxRight?firstBoxRightVar ->
209                 if((firstBoxLeftVar > firstBoxRightVar)) then (
210                   showInt!(2) ->
211                   SKIP
212                 ) else (
213                   showInt!(3) ->
214                   SKIP
215                 )
216               within
217                 ELSEIFELSE
218           )
219         )
220       within
221         IFELSE
222   )
223   ;
224   SKIP
225
226 getBoxLastLineProc() =
227   (let
228       IFELSE =
229         get.X?x ->
230         get.Y?y ->
231         get.ORIENTATION?o ->
232         get.BX?bx ->
233         get.BY?by ->
234         coin?c ->
235         get.lastBoxLeft?lastBoxLeftVar ->
236         get.lastBoxRight?lastBoxRightVar ->
237         if((lastBoxLeftVar > lastBoxRightVar)) then (
238           showInt!(1) ->
239           SKIP
240         ) else (
241           (let
242              ELSEIFELSE =
243                get.X?x ->
244                get.Y?y ->
245                get.ORIENTATION?o ->
246                get.BX?bx ->
247                get.BY?by ->
248               coin?c ->
249               get.lastBoxLeft?lastBoxLeftVar ->
250               get.lastBoxRight?lastBoxRightVar ->
251                if((lastBoxLeftVar < lastBoxRightVar)) then (
252                  showInt!(2) ->
253                  SKIP
254                ) else (
255
256                  showInt!(3) ->
257                  SKIP
258                )
259              within
260                ELSEIFELSE
261           )
262         )
263       within
264         IFELSE
```

```
265   )
266   ;
267   SKIP
268
269 COMMANDS =
270   RIGHT
271   ;
272   (let
273      WHILE =
274        get.X?x ->
275        get.Y?y ->
276        get.ORIENTATION?o ->
277        get.BX?bx ->
278        get.BY?by ->
279        coin?c ->
280        if (frontIsClear(x,y,o,bx,by)) then (
281          countBoxesProc(1)
282          ;
283          countBoxesProc(2)
284          ;
285          FORWARD(1)
286          ;
287          get.countForward?countForwardVar ->
288          get.countForward?countForwardVar ->
289          member((countForwardVar + 1), INTVALUES) & set.
    countForward!((countForwardVar + 1)) ->
290          SKIP
291          ;
292          WHILE
293        ) else (
294          SKIP
295        )
296    within
297      WHILE
298   )
299   ;
300   countBoxesProc(1)
301   ;
302   countBoxesProc(2)
303   ;
304   get.countFirstLine?countFirstLineVar ->
305   showInt!(countFirstLineVar) ->
306   get.countLastLine?countLastLineVar ->
307   showInt!(countLastLineVar) ->
308   showsMoreBoxesProc()
309   ;
310   getBoxFirstLineProc()
311   ;
312   getBoxLastLineProc()
313   ;
314   SKIP
```

Fonte – O autor

Código D.3 – Especificação CSP do problema *findBeacon*

```
1 datatype VarType =  X.TX | Y.TY | ORIENTATION.TO | BX.TBX | BY.
    TBY
```

```
INIT = {  (X, startX), (Y, startY), (ORIENTATION, NORTH_), (BX,
    startBX), (BY, startBY) }

COMMANDS =
  (let
     WHILE =
       get.X?x ->
       get.Y?y ->
       get.ORIENTATION?o ->
       get.BX?bx ->
       get.BY?by ->
       coin?c ->
       if (not (frontIsBeacon(x,y,o,bx,by))) then (
         (let
            IFELSE =
              get.X?x ->
             get.Y?y ->
             get.ORIENTATION?o ->
             get.BX?bx ->
             get.BY?by ->
             coin?c ->
             if(frontIsClear(x,y,o,bx,by)) then (
               FORWARD(1)
                 ;
               SKIP
             ) else (
               BACKWARD(1)
                 ;
               (let
                  IFELSE =
                    get.X?x ->
                   get.Y?y ->
                   get.ORIENTATION?o ->
                   get.BX?bx ->
                   get.BY?by ->
                   coin?c ->
                   if(c) then (
                     RIGHT
                     ;
                     SKIP
                   ) else (
                     LEFT
                     ;
                     SKIP
                   )
                 within
                   IFELSE
               )
               ;
               SKIP
             )
           within
             IFELSE
       )
       ;
       SKIP
       ;
       WHILE
```

```
59          ) else (
60            SKIP
61          )
62      within
63        WHILE
64    )
65    ;
66    SKIP
```

Fonte – O autor