UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Lucas Francisco Pereira de Gois Correia

Verificação Eficiente de Robôs Educacionais

Recife

2021

Lucas Francisco Pereira de Gois Correia

Verificação Eficiente de Robôs Educacionais

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

Universidade Federal Rural de Pernambuco – UFRPE

Departamento de Computação

Curso de Bacharelado em Ciências da Computação

Orientador: Sidney de Carvalho Nogueira

Recife

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal Rural de Pernambuco
Sistema Integrado de Bibliotecas
Gerada automaticamente, mediante os dados fornecidos pelo(a) autor(a)

C824v Correia, Lucas Francisco Pereira de Gois
Verificação Eficiente de Robos Educacionais / Lucas Francisco Pereira de Gois Correia. - 2021.
83 f.

Orientador: Sidney de Carvalho Nogueira.
Inclui referências e apêndice(s).

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal Rural de Pernambuco, Bacharelado em Ciência da Computação, Recife, 2021.

1. Engenharia de Software. 2. Métodos Formais. 3. Tradução Automática. 4. Verificador de Modelos. 5. Verificação de Software. I. Nogueira, Sidney de Carvalho, orient. II. Título

CDD 004

MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO
UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO (UFRPE)
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

http://www.bcc.ufrpe.br

FICHA DE APROVAÇÃO DO TRABALHO DE CONCLUSÃO DE CURSO

Trabalho defendido por Lucas Francisco Pereira de Gois Correia às 08 horas do dia 03 de março de 2021, no link meet.google.com/zqz-jpyv-qef, como requisito para conclusão do curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, intitulado Verificação Eficiente de Robôs Educacionais, orientado por Sidney de Carvalho Nogueira e aprovado pela seguinte banca examinadora:

Sidney de Carvalho Nogueira
DC/UFRPE

Lucas Albertins de Lima
DC/UFRPE

Dedico este trabalho aos meus pais, Jorge e Elda, à minha noiva, Jennifer e ao meu orientador, Sidney, por todo o apoio até aqui

Agradecimentos

Agradeço aos meus pais, Jorge e Elda, por tentarem compreenderem os momentos difíceis aos quais passei e me ajudarem a buscar apoio, quando eu não tinha capacidade ou coragem para buscar.

Agradeço à minha noiva, Jennifer, por estar sempre ao meu lado, seja nos meus melhores momentos ou seja nos piores, por sempre estar me apoiando e me fazer não desistir de tudo.

Agradeço ao meu orientador, Sidney, pela paciência e motivação para realizar este projeto, e por ser o primeiro a confiar a mim, um projeto que precisou de muita dedicação e tempo para ser completado.

Agradeço aos meus amigos de longa data, Adrielle, Amorim, Luiz e Marcos, que me acompanharam durante esta longa jornada até hoje.

Agradeço aos professores que encontrei durante essa jornada, que me mostraram como é incrível esta areá que é a ciência da computação, em especial, André Aziz, Jeane Melo, Rafael Dueire, Rafael Ferreira e Sidney Nogueira, com a qual tive o prazer de conhecer, por me mostrarem que escolhi um curso que não vou ter arrependimentos de ter concluído.

Agradeço aos meus colegas de jornada, que estão completando o curso comigo, com a qual tive o prazer de compartilhar experiencias, em especial, Emília, Fábio, Igor, Ismael, Larissa, Leonardo, Pedro e Raíssa, com a qual tive o prazer de compartilhar e receber experiencias vividas durante esses anos de curso.

Por fim, agradeço a Deus, por ter me iluminado neste caminho e ter me dado o prazer de conhecer tantas pessoas importantes para a minha vida e ter me dado forças para seguir essa longa jornada.

"A persistência é o caminho do êxito."
(Charles Chaplin)

Resumo

Robótica educacional é uma área de interesse crescente dentro das instituições de ensino. Devido ao seu baixo custo e facilidade de aquisição, ambientes virtuais de programação para robôs têm sido desenvolvidos para suportar o ensino de conceitos de computação, programação e robótica. A principal ferramenta de depuração disponível nestes ambientes é a simulação do robô dentro de um ambiente virtual. Nestes ambientes, a depuração acontece de forma visual: não é possível analisar de forma automática se um programa vai convergir para um objetivo específico. Soluções para analisar de forma automática programas de robôs virtuais são ferramentas de ensino importantes para a avaliação eficiente e precisa dos programas. O objetivo deste projeto é aperfeiçoar uma abordagem de verificação automática de programas de robô. Esta abordagem traduz programas na linguagem ROBO para a notação formal CSP e utiliza o verificador de modelos FDR para analisar o comportamento do programa. O resultado retornado pelo verificador é utilizado para informar se o programa analisado possui o comportamento esperado. O aperfeiçoamento corresponde a implementação de um tradutor de ROBO para CSP que gera um modelo CSP mais eficiente de ser analisado do que o modelo produzido pelo tradutor atual. Através de avaliação empírica, pode-se observar uma redução significativa no tempo para análise dos modelos CSP obtidos a partir do tradutor desenvolvido neste trabalho. O tempo de análise observado foi praticamente constante para os mapas analisados, enquanto o tempo de análise dos modelos gerados pelo tradutor anterior, produzido antes deste trabalho, apresentou um crescimento exponencial com relação ao tamanho do mapa onde o programa é analisado. Uma contribuição adicional deste trabalho é que o tradutor transforma em CSP programas com qualquer sintaxe da linguagem ROBO, enquanto o tradutor anterior lidava apenas com um subconjunto da sintaxe.

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

Abstract

Educational robotics is an area of growing interest within educational institutions. Due to it's low cost, program environments for virtual robots have been developed to support the teaching of computing, programming and robotics concepts. The main debug tool available in such environments is the simulation of the robot within a virtual map. Debugging is performed by observing the robot moving across the map: it is not possible to analyze automatically if a program will manage to conclude a specific objective. Approaches to automatically analyze virtual robots are an important teaching tool for an efficient and accurate evaluation of robots. The objective of this project is to improve an automatic verification approach of robot programs. This approach translates programs in the ROBO language to the formal notation CSP and uses the FDR model checker to automatically analyze the program's behavior. The result returned by the model checker is used to inform if the analyzed program has the expected behavior. The main improvement proposed by this project is the implementation of a ROBO to CSP translator that generates a more efficient CSP model to be analyzed, if compared to the model produced by the previous translator. We could observe, through empirical evaluation, a significant reduction in the time to analyze the CSP models obtained from the new translator. The proposed translator presents an almost constant analysis time for the maps considered in the empirical evaluation, while analysis time of the models produced by the previous translator shows an exponential growth in relation to the map's size where the program is analyzed. Another contribution of this work is that the new translator accepts ROBO programs with any command of the language, while the previous translator could only deal with a subset.

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

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

CSP	Communicating Sequential Processes
FDR	Failures-Divergences Refinement
AST	Abstract Syntax Tree

Sumário

1 Introdução

Robótica Educacional é uma área de grande interesse dentro das instituições de ensino, com o objetivo de promover o estudo de conceitos multidisciplinares como matemática, computação, análise de problemas, pensamento algorítmico, etc (ALIMISIS, 2013) (BERS; GONZÁLEZ-GONZÁLEZ; ARMAS-TORRES, 2019) (MILLER; NOURBAKHSH, 2016). Cada vez mais, a popularidade do uso da robótica nas escolas vem crescendo (MILLER; NOURBAKHSH, 2016) e uma maneira de ensinar robótica por um menor custo de aquisição, é através da adoção de ambientes com robôs virtuais. Nestes ambientes, é possível definir um terreno virtual onde o robô se movimenta e é possível acompanhar a sua simulação no terreno.

Um ambiente de desenvolvimento de robôs virtuais bastante popular é o RoboMind(KITCHEN; AMSTERDAM, 2014), que, além de realizar simulações com um robô virtual em um mapa, permite transferir o programa do robô virtual para um robô verdadeiro. Este ambiente utiliza como linguagem de programação ROBO, projetada para que o usuário consiga programar de forma imediata usando um conjunto conciso de comandos destinados ao controle de um robô. O RoboMind permite realizar a simulação de um robô virtual na tela e verificar o comportamento do robô visualmente.

Uma das grandes vantagens do ambiente RoboMind é sua simplicidade e facilidade de programação com a linguagem ROBO, garantindo uma curva de aprendizado pequena, mas, em contrapartida, sua maior desvantagem é a análise do programa. O ambiente limita o usuário a analisar o programa visualmente, e verificar se o programa consegue resolver um problema determinado. Neste ambiente, é preciso simular o programa de forma individual em cada um dos mapas e observar o seu funcionamento, o que torna difícil analisar se o programa desenvolvido é capaz de resolver os problemas propostos em um conjunto de mapas.

Uma evolução do RoboMind é a plataforma RoboMind Academy (ACADEMY, 2015), que permite desenvolver e simular online programas ROBO. Como limitações, esta plataforma é paga, e só realiza a verificação da corretude dos programas que estão cadastrados. Como exemplo de verificação realizada pela plataforma, temos que a mesma verifica se o robô segue um determinado caminho dentro de um mapa já cadastrado. Portanto, esta plataforma só funciona em um conjunto de mapas pré-determinados e não permite definir novos objetivos a serem verificados dentro do mapa.

1.1 Problema de pesquisa

Atualmente, existe uma considerável quantidade de plataformas gratuitas para verificação e análise de códigos de programação, como o TheHuxley(THEHUXLEY, 2020) e o Spoj(BRASIL, 2020), que analisam se programas submetidos pelos usuários passam nos testes cadastrados. Estas plataformas permitem que os usuários cadastrem novos desafios. Além disso, plataformas como o TheHuxley, permitem ao usuário criar uma turma e propor questões presentes nela como desafios para a turma e/ou criar novos desafios para serem usados e respondidos por outras pessoas. Entretanto, mesmo com plataformas que oferecem todas essas facilidades para verificação automática de desafios de programação, não há uma plataforma **gratuita** para análise e verificação de programas para robôs. A única opção conhecida para a linguagem ROBO é RoboMind Academy, que é paga.

Mesmo em um ambiente pago como RoboMind Academy, a verificação dos programas ROBO submetidos como solução é feita de forma visual. Se for levado em conta o nível de complexidade do desafio, e o tempo que o programa do usuário pode levar, o usuário pode nunca saber se seu programa consegue resolver o problema. O ambiente limita a uma análise visual do próprio usuário, e não fornece um feedback dizendo se resolve o problema, deixando o usuário se perguntando se o programa demora para resolver o problema, ou o programa não resolve o problema.

Para conseguir fazer essas verificações, é preciso buscar uma abordagem automática eficiente para analisar os programas do usuário na linguagem ROBO. A partir desta abordagem, será possível criar um sistema web que utilizará as verificações para analisar automaticamente os programas submetidos, que precisa fornecer um retorno rápido, onde vários alunos podem submeter seus programas e receber o resultado da análise deste.

1.2 Justificativa

Uma forma de verificar automaticamente programas ROBO é traduzindo a linguagem ROBO para uma versão equivalente em CSP (ROSCOE, 2011), e utilizar a ferramenta FDR (GIBSON-ROBINSON et al., 2014) para verificar as diferentes propriedades do programa. CSP é uma notação formal que pode ser usada para descrever sistemas, de forma a permitir a verificação de propriedades, como, por exemplo, verificar a existência de deadlock, que é uma propriedade interessante de ser analisada, uma vez que um desafio requer uma sequência finita de ações. Uma forma de automatizar a tradução de ROBO para CSP é utilizar o Spoofax (KATS; VISSER, 2010), um framework para desenvolvimento de linguagens específicas de domínio. Para isso, o Spoofax utiliza Stratego (KALLEBERG, 2006), uma linguagem que permite descrever regras de transformação, usando como entrada a árvore sintática de programas na linguagem ROBO, e definindo os respectivos elementos na linguagem CSP.

Mesmo usando verificação automática com FDR, ainda podem existir problemas de desempenho. O tempo para avaliação de um programa ROBO, usando a estratégia de tradução de ROBO para CSP apresentado em (PEREIRA, 2018), cresce exponencialmente com o tamanho do mapa onde o programa será analisado. Isto acontece porque a especificação CSP que representa o programa a ser analisado é a composição paralela dos processos que representam o mapa e o código ROBO. Composição paralela aumenta muito a quantidade de estados a serem analisados. Quanto mais estados para analisar, maior o tempo para avaliação. Por se tratar de um sistema web, o esforço computacional para verificar os problemas do lado do servidor tem que ser otimizado, uma vez que o tempo de resposta para verificar uma submissão crescerá rapidamente, devido ao crescente número de usuários e ao uso concorrente do servidor, o que pode tornar inviável a utilização do sistema web. Logo, o custo de verificar o modelo formal do programa deve ser otimizado de forma a não atrapalhar a experiência dos usuários do sistema.

Além disto, a solução em (PEREIRA, 2018) não abrange toda a linguagem ROBO, o que impede análises em programas que possuem os elementos não cobertos na tradução como a pintura do chão, que pode ser utilizado para criar uma trilha ou marcar o caminho do robô; bem como comandos para manipulação do beacon, um objeto ao qual o robô pode interagir.

Este trabalho visa criar um tradutor de programas na linguagem ROBO para o modelo formal CSP, que permite que sejam feitas análises automáticas do programa do usuário e que seja computacionalmente mais eficiente de ser analisado. A principal diferença do modelo proposto neste trabalho para o modelo proposto em (PEREIRA, 2018) é que o primeiro modelo não utiliza composição paralela, o que reduz a quantidade de estados a serem analisados por FDR (e o tempo de análise, como consequência).

Além da análise automática mais eficiente de programas ROBO, este trabalho também visa permitir a análise de programas ROBO que possuem todos os elementos da sintaxe de ROBO, não apenas parte da linguagem, como acontece no trabalho (PEREIRA, 2018).

1.3 Objetivos

O objetivo geral deste trabalho é a implementação de um tradutor que considere todos os elementos da linguagem de ROBO e que gere uma representação formal mais eficiente de ser analisada. Isto é necessário considerando que o compilador será utilizado em um sistema web que permite vários usuários submeterem seus problemas através de um navegador.

Objetivos Específicos:

1. Definir a semântica formal para os elementos de ROBO que seja eficiente de ser analisada;
2. Automatizar regras de tradução de comandos da linguagem ROBO para CSP;
3. Validar a corretude da tradução proposta; e,
4. Analisar a eficiência da verificação do modelo otimizado a partir de testes e experimentos;

1.4 Estrutura do trabalho

Este trabalho está estruturado nos seguintes capítulos:

Capítulo 2 apresenta uma visão geral dos conceitos fundamentais para a compreensão deste projeto;

Capítulo 3 apresenta os principais elementos da semântica CSP utilizada para traduzir programas ROBO para CSP;

Capitulo 4 apresenta as principais regras utilizadas para realizar a tradução de um programa ROBO para a respectiva representação em CSP;

Capitulo 5 apresenta como foi verificada a corretude e a eficiência do tradutor desenvolvido neste projeto;

2 Referencial Teórico

Este capítulo dá uma visão geral dos conceitos fundamentais para a compreensão deste trabalho.

2.1 ROBO

ROBO (KITCHEN; AMSTERDAM, 2014) é uma linguagem de programação educacional simples que foi projetada para que o usuário consiga programar de forma imediata, usando um conjunto conciso de comandos destinados a controlar um robô. A Figura 1 ilustra um programa na linguagem ROBO com comandos que fazem o robô andar para frente uma célula do mapa (forward(1)), andar para a frente duas células (forward(2)), e por fim andar para trás 3 células (backward(3)).

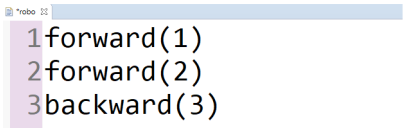

Figura 1 – Exemplo de código em ROBO

A Figura 2 apresenta a parte da interface do ambiente RoboMind que mostra a simulação do código do robô em um mapa predefinido. Do lado esquerdo da imagem fica o programa do usuário utilizado para manipular o robô, que se encontra na parte direita da imagem. Na parte inferior da figura, estão os comandos de simulação, como iniciar, parar, pausar, ajuste da velocidade do robô, entre outros.

Apesar de realizar a simulação de um robô, o ambiente RoboMind não consegue realizar validações automáticas no programa do usuário, sendo necessário a observação constante do usuário para saber se a simulação do programa vai resolver o problema proposto. O ambiente não produz um veredito se o programa resolve ou não resolve o problema.

2.2 CSP

CSP (ROSCOE, 2011) é uma linguagem formal para a descrição de padrões de interação em sistemas concorrentes. Também pode ser utilizado para descrever sistemas não concorrentes, assim como verificar diferentes propriedades, como, por exemplo, a existência de *deadlock*. No contexto deste projeto, que analisa propriedade de programas ROBO usando CSP, um programa que possui deadlock é um programa que entra em um estado

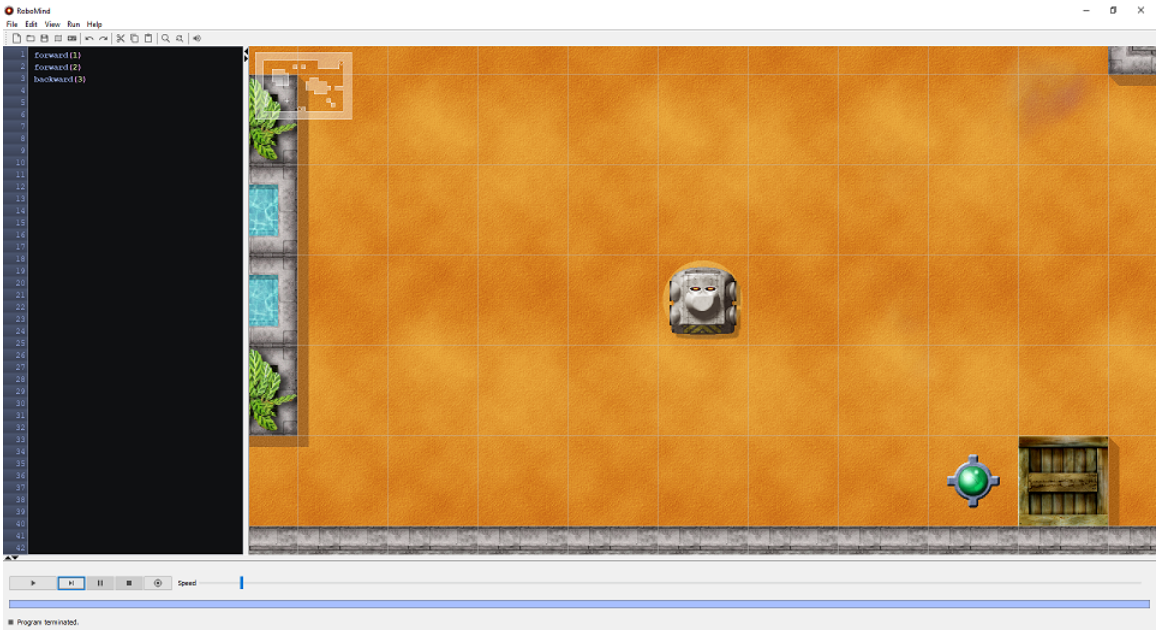

Figura 2 – Ambiente Robomind

terminal (não permanece em um laço infinito). A presença de deadlock é uma propriedade interessante de verificar em programas ROBO, uma vez que os desafios colocados para serem solucionados devem ser realizados com uma sequência finita de ações. No entanto, apenas observando a simulação não é possível ter certeza se o programa vai parar.

Em CSP, existem operadores e construtores que ajudam na criação de especificações. Entre eles, podemos citar o processo pré-definido **STOP**, que representa um processo em *deadlock* (um estado terminal), que não comunica eventos e não progride. O operador de prefixo `->` é usado para modelar um comportamento sequencial em um processo. Por exemplo, a especificação **PROGRAM = forward.1 -> STOP** define um processo chamado **PROGRAM** que comunica o evento **forward.1**, e em seguida se comporta como o processo **STOP**. Na sintaxe de CSP, canais são usados para comunicar valores. Por exemplo, a sintaxe **channel coin : boolean** define o canal **coin**, que comunica um valor booleano **c**, que é escolhido pelo ambiente. Como exemplo de uso de canal, usamos o processo CSP a seguir.

```
coin?c ->
if(c) then
   forward.1 -> STOP
else
   backward.1 -> STOP
```

No processo acima, a sintaxe **coin?c** representa que inicialmente o processo permite que o ambiente escolha se comunica o evento **coin.true** ou **coin.false**. Em sequência, existe um comando condicional escrito na notação de CSP que define o comportamento como

forward.1 -> STOP, caso o valor de **c** seja verdadeiro. Caso **c** seja falso, o comportamento do processo é **backward.1 -> STOP**. Usamos o canal **coin** para especificar as expressões da linguagem ROBO que chamam a função **flipCoin()** de ROBO, que gera um valor booleano de forma aleatória.

A Figura 3 apresenta uma especificação escrita em CSP, que corresponde ao código em ROBO da Figura 1. **PROGRAM** é o processo CSP que representa o comportamento do programa. Este processo se comporta como o processo **FORWARD1** que recebe como parâmetro **INIT**, que é um mapeamento que guarda o nome das variáveis do programa e os respectivos valores. **FORWARD1** recebe **m** como argumento, o qual representa o mapeamento das variáveis que será passado para o processo **FORWARD**. O processo **FORWARD** possui um comportamento predefinido e modela o avanço do robô em uma casa para a frente. Este processo faz parte de um arquivo de definições CSP que especifica os comandos básicos da linguagem ROBO. O processo **FORWARD** recebe o número de posições a avançar no mapa, o valor atual das variáveis e o processo que especifica o comportamento seguinte do robô. Na linha 2 da especificação o terceiro argumento do processo **FORWARD** corresponde ao processo **FORWARD2** que está definido na linha 3. O processo **BACKWARD3** corresponde ao comportamento do último comando do programa na Figura 1. O processo **TERMINATE** é um processo que representa que o programa termina e não possui mais comandos para executar.

```
1PROGRAM = FORWARD1(INIT)
2FORWARD1(m) = FORWARD(1, m, FORWARD2)
3FORWARD2(m) = FORWARD(2, m, BACKWARD3)
4BACKWARD3(m) = BACKWARD(3, m, TERMINATE)
```

Figura 3 – Exemplo de especificação em CSP

FDR (GIBSON-ROBINSON et al., 2014) é um verificador de refinamento para CSP que analisa propriedades (asserções) de processos CSP. Um exemplo de asserção é verificar se um processo está livre de deadlock. Outras propriedades de uma especificação podem ser verificadas com FDR, por exemplo, saber se uma sequência de eventos (traces) está presente na especificação. Isto é muito útil na análise do comportamento de programas ROBO que são traduzidos para CSP. FDR também permite analisar refinamento e equivalência entre processos CSP. Este tipo de análise permite avaliar se programas sintaticamente diferentes possuem o mesmo comportamento. Usando FDR podemos comprovar que o processo **PROGRAM** na Figura 3 termina.

2.3 Spoofax

Spoofax (KATS; VISSER, 2010) é um framework para desenvolvimento de linguagens específicas de domínio. Tem sido usado para desenvolver linguagens de script, de

fluxo de trabalho, de configuração, de descrição de trabalho, de modelagem de dados e de programação para web. Possui um conjunto de linguagens com funcionalidades específicas. Neste projeto, utilizamos as linguagens SDF3 e Stratego.

A linguagem SDF3 permite definir o parser (análise léxica e sintática) para uma linguagem. O parser é gerado automaticamente pelo framework. A linguagem Stratego (KALLEBERG, 2006) é usada para realizar transformações na árvore sintática dos programas, e também traduzir a árvore para uma linguagem diferente da linguagem analisada pelo parser. Possui regras de reescrita para transformações básicas, estratégias de reescrita programáveis para controlar a aplicação de regras, entre outros recursos. Este projeto usa Stratego para definir a semântica CSP para cada elemento sintático de um programa ROBO. As regras são aplicadas automaticamente pelo framework nos elementos da árvore sintática de um programa ROBO. Como resultado da aplicação das regras na árvore sintática do programa ROBO, é gerada uma string que corresponde a especificação CSP para o programa ROBO de entrada.

A Figura 4 ilustra regras de tradução de elementos da árvore sintática de um programa ROBO para texto em CSP. De forma geral, uma regra tem o formato $R : A \rightarrow B$, onde R corresponde ao nome da regra, A é um elemento da árvore sintática com o qual a regra casa padrão, e B é o resultado da aplicação da regra. As linhas 267 e 277 apresentam duas regras que são aplicadas em elementos da árvore sintática. Na linha 277, caso apareça um *LEFT()* na árvore, irá retornar a string *LEFT*, que é equivalente ao comando *left()* na linguagem ROBO, seguido por um *;* . Na linha 267, da mesma forma, se aparecer na árvore *BACKWARD(n)*, então ele irá gerar uma string que vai conter algumas variáveis representada por *[vars']*. Esta string é seguida por *BACKWARD* que recebe como parâmetro *[n']*. A string *[vars']* é retornada pela regra *put-get-var-exp-analyze*, que recebe o retorno da regra *get-vars-exp*, que por sua vez, recebe o valor de n; *[n']* é uma string que será o retorno da regra *to-csp-e*, que recebe como parâmetro n.

As regras implementadas atualmente em Stratego geram um modelo CSP, que corresponde a um programa ROBO, desde que o programa não possua comandos de pintura e de manipulação do beacon. Um dos objetivos do projeto atual é remover essa limitação.

```
robo2csp.str
267 to-csp:
268     BACKWARD(n) ->
269     $[[vars']
270     BACKWARD([n'])
271       ;
272     ]
273     with
274         vars' := <put-get-var-exp-analyze> <get-vars-exp> n;
275         n' := <to-csp-e> n
276
277 to-csp:
278     LEFT() ->
279     $[LEFT
280       ;
281     ]
282
```

Figura 4 – Exemplo de regras em Stratego

3 Semântica Formal CSP para um programa ROBO

Este capítulo apresenta os principais elementos da semântica CSP otimizada para ROBO. A formalização completa em CSP dos elementos do ambiente RoboMind pode ser visto no Apêndice A.

3.1 Representação do mapa

No ambiente RoboMind, os elementos de um mapa são definidos usando texto plano. Após carregar o texto do mapa, a ferramenta desenha o mapa que é usado na simulação (2.1). A seguir, um exemplo do texto de definição de mapa.

```
map:
    CHHHHHHD
    G@A   *I
    G      I
    G      I
    BFFFFFFE
```

No texto acima, os caracteres, [**A-Z**] representam um obstáculo que o robô não pode passar, @ representa a posição inicial que o robô começa no mapa, * representa o *Beacon* (um objeto que o robô pode interagir) e um *espaço* representa um espaço vazio.

Já na semântica CSP, o mapa é representado por uma sequência de cujos elementos representam objetos que existem no mapa original. O texto acima apresenta o mapa do ambiente RoboMind apresentado na Seção 2.1.

Em CSP, o mapa é representado pela constante `RAW_MAP` que contém uma sequência de sequências, onde cada sequência vai conter um elemento do mapa. Em CSP, os elementos do mapa são codificados como: **Empty** para um espaço vazio, **Obs** para um obstáculo, **Start** para a posição inicial do robô, e **Beacon** para um objeto que o robô pode interagir.

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

```
    <Obs,Obs,Obs,Obs,Obs,Obs,Obs,Obs>
>
```

3.2 Memória do programa

Considerando os valores obtidos na representação do mapa, é criado uma representação para os valores iniciais da memória. O texto a seguir apresenta os tipos que podem ser armazenados na representação da memória considerando a semântica CSP:

O tipo **DATA** representa a unificação de todos os tipos de dados que podem ser armazenados na memória. Este dado pode ser um booleano: representado por **B.Bool**, onde **B** é o label que representa o tipo booleano; **I** é o label que representa um valor inteiro; e, **BCN** é o label que representa o *Beacon* e sua respectiva posição **x** e **y** no mapa. O construtor **COLOR** representa a existência de uma tinta no mapa, com sua respectiva coordenada **x** e **y** e a cor armazenada naquela posição. O construtor **Paints** representa a cor de uma pintura. **UNDEFINED** representa o valor de uma variável que não existe na representação da memória.

```
datatype DATA = B.Bool
              | I.Int
              | BCN.Int.Int
              | COLOR.Int.Int.Colors
              | PAINTS.Colors
              | UNDEFINED
```

O valor inicial da memória é representado pela constante **INIT**, cujos valores são inicializados dependendo dos valores existentes no mapa. O valor dessa constante é uma lista de tuplas, onde cada tupla contém o label para uma variável e uma sequência de valores do tipo **DATA**.

```
INIT = {
        ("X", <I.startX>),
        ("Y", <I.startY>),
        ("ORIENTATION", <I.NORTH_>),
        ("BEACONS", INITIAL_BEACONS),
        ("CARRYING_BEACON", <B.false>),
        ("IS_PAITING", <B.false>),
        ("PAINT_COLOR", <PAINTS.White>),
        ("PAINTS", INITIAL_PAINTS),
        ("RETURN", <I.0>)
```

```
}
```

As variáveis dentro de INIT são:

- **X** e **Y** representam a posição inicial do robô no mapa, representado pelo **Start**;
- **ORIENTATION** representa a orientação inicial do robô no mapa (o robô sempre começa olhando para o norte);
- **BEACONS** representa a posição inicial dos *beacons* (representado no mapa por **Beacon**);
- **CARRYING_BEACON** representa se o robô está carregando um *beacon* (o robô não começa carregando um *beacon*);
- **IS_PAINTING** representa se o robô está pintando o mapa (o robô não começa pintando o mapa);
- **PAINT_COLOR** representa a cor que o robô está pintando no mapa (enquanto o robô não escolher uma tinta para pintar, o valor inicial é arbitrário).
- **PAINTS** representa as pinturas que podem existir no mapa.
- **RETURN** representa o valor que é retornado por um procedimento. Enquanto o programa do robô não utilizar retorno, o valor inicial é arbitrário.

Além desses, podem existir variáveis na memória, criadas pelo usuário em seu programa ROBO, que são adicionados à representação da memória sob demanda.

3.3 Funções

Para avaliar o comportamento de um programa ROBO, são necessárias algumas funções em CSP que ajudam na formalização do programa ROBO. A seguir, estão algumas dessas funções.

Todos os comandos e funções de CSP, usados para formalizar um programa ROBO, recebem como parâmetro **m**, que representa o valor atual da memória. O valor inicial de **m** é definido como o valor de **INIT**. A função get, apresentada a seguir, recupera da memória, representado pelo parâmetro **m**, o valor contido na variável **var**. Para isso, é checado se essa variável está presente na memória e retornamos o seu valor, se ela existir, caso contrário, é retornado o valor **UNDEFINED** e o processo que receber esse valor encerra a avaliação do programa.

```
single({x}) = head(x)

get(m,var) =
    let
        check = {val | (v,val) <- m, v == var}
    within
        if (check != {}) then
            single(check)
        else
            UNDEFINED
```

A função **single** é usada para retornar o conteúdo que é buscado na memória. Como as variáveis contidas em INIT são uma sequência, a função retorna o primeiro elemento da sequência.

A função auxiliar **thingsInFront** verifica no mapa se há algum objeto à frente do robô. Para isso, é verificada qual a posição atual do robô (representada por **x** e **y**), a sua orientação (parametro **o**), e os *beacons* (parâmetro **bcns**). Após verificar a orientação do robô, é passado para a função **thingsAt** a posição a qual se quer verificar e os *beacons*.

```
thingsAt(col,lin, bcns) =
    union({ Obs | (c,l,t) <- MAP, c==col, l==lin , t==Obs},
          { Beacon | (BCN.c.l) <- bcns, c==col, l==lin})

thingsInFront(x,y,o,bcns) =
  if(o == NORTH_) then
    thingsAt(x,y-1, bcns)
  else if (o == EAST_) then
    thingsAt(x+1,y, bcns)
  else if (o == SOUTH_) then
    thingsAt(x,y+1, bcns)
  else
    thingsAt(x-1,y, bcns)
```

A função auxiliar **toBool** recebe ou um booleano ou um inteiro, e retorna um booleano para ser usado e analisado em um processo. Caso a função receba um booleano, ela retorna o respectivo valor. Caso a função receba um inteiro, o valor retornado é true se o inteiro i for diferente de 0 como verdadeiro, e retorna falso caso contrário

```
toBool(B.b) = b
toBool(I.i) = (i != 0)
```

A função **toInt** recebe ou um booleano ou um inteiro, e retorna um inteiro. A linguagem ROBO trata o label *true* como o valor inteiro 1 e o label *false* como o valor inteiro 0, então essa função retorna esses valores respectivamente. Caso a função receba um inteiro, ela retorna o seu valor numérico.

```
toInt(B.true) = 1
toInt(B.false) = 0
toInt(I.i) = i
```

A função **frontIsObstacle** verifica se a posição a frente do robô é um obstáculo. Para isso, ele verifica se, no retorno da função **thingsInFront**, há um obstáculo ou um *beacon*. Um robô pode capturar e levar um beacon, porém, um beacon representa um obstáculo quando o robô executa um comando de movimentação em direção a uma célula que contém um beacon.

```
    frontIsObstacle(x,y,o,bcns) =
        member(Obs,thingsInFront(x,y,o,bcns))
        or
        member(Beacon,thingsInFront(x,y,o,bcns))
```

3.4 Comandos Básicos

Os comandos básicos da linguagem ROBO são formalizados em CSP como processos que leem e/ou alteram valores na memória do programa representada por **m**.

O processo **FORWARD** formaliza a movimentação do robô de andar para frente. Ele recebe **n**, que representa quantas casas o robô vai se movimentar, a memória **m** e **next**, O comportamento de `FORWARD` é equivalente ao processo **MOVE_STEPS**, que realiza a movimentação do robô no mapa. Este processo move o robô célula a célula do mapa, enquanto não esbarrar em um obstáculo. Caso o robô esbarre em um obstáculo, o processo é encerrado e é chamado o próximo processo (**next**).

```
FORWARD(n, m, next) = MOVE_STEPS(n,m,next)

MOVE_STEPS(0,m,next) = next(m)
MOVE_STEPS(n,m,next) =
  let
    ...
  within
    if(frontIsClear(x,y,o,bcns)) then(
```

```
      if(o == NORTH_) then (
        forward!1
        -> updatedY!(y-1)
        -> MOVE_STEPS(n-1, setVar(m_,"Y",I.(y-1)), next)
      ) else if(o == EAST_) then (
        forward!1
        -> updatedX!(x+1)
        -> MOVE_STEPS(n-1, setVar(m_,"X",I.(x+1)), next)
      ) else if(o == SOUTH_) then (
        forward!1
        -> updatedY!(y+1)
        -> MOVE_STEPS(n-1, setVar(m_,"Y",I.(y+1)), next)
      ) else if(o == WEST_) then (
        forward!1
        -> updatedX!(x-1)
        -> MOVE_STEPS(n-1, setVar(m_,"X",I.(x-1)), next)
      ) else
        error("Unexpected state")
    ) else
        forward!1 -> MOVE_STEPS(0, m_, next)
```

O processo **MOVE_STEPS** se encontra completo no Apêndice A.

O processo **PICKUP** formaliza a ação do robô capturar um *beacon*. Este processo verifica se a posição a frente do robô possui um beacon, se possuir e o robô não está carregando um beacon, se comporta como o processo **PICKUP_BEACON** formaliza a ação de coletar o beacon existente a frente do robô e para carregá-la. A especificação do processo **PICKUP_BEACON** pode ser visto no Apêndice A.

```
PICKUP(m, next) =
                  let
                    x = toInt(get(m,"X"))
                    y = toInt(get(m,"Y"))
                    o = toInt(get(m,"ORIENTATION"))
                    bcns = set(getAllBeacons(m))
                    carryingbeacon = toBool(get(m, "CARRYING_BEACON"))
                  within
                    if(frontIsBeacon(x, y, o, bcns)
                    and not(carryingbeacon)) then (
                      if(o == NORTH_) then (
```

```
        PICKUP_BEACON(x, y-1, m, next)
      ) else if(o == EAST_) then (
        PICKUP_BEACON(x+1, y, m, next)
      ) else if(o == SOUTH_) then (
        PICKUP_BEACON(x, y+1, m, next)
      ) else (
        PICKUP_BEACON(x-1, y, m, next)
      )
    )else(
      next(m)
    )
```

3.5 Variáveis, expressões e atribuição

Um programa ROBO permite a declaração e uso de variáveis. Na formalização CSP do programa, cada declaração de variável é adicionada na representação da memória, que é passada como parâmetro para os processos que representam a continuação do programa.

O processo **ASSIGN** formaliza a atribuição de um valor a uma variável. O parâmetro **var** corresponde ao nome da variável que será colocada na memória, o parâmetro **val** corresponde ao valor da variável que será salvo na memória **m**.

```
ASSIGN(var, val, m, next) =
    let
        check = {(var_, value) | (var_, value) <- m, var == var_}
        m_ = union(m,{(var, <val>)})
        others' = {(var_, head(value)) | (var_, value) <- m, var != var_}
    within
        if(check == {}) then
            next(m_)
        else
            UPDATE(var, val, m, next)
```

Na lógica do processo ASSIGN, quando a variável já está presente na memória e é atualizada através do comando de atribuição, um tratamento diferente é realizado. Em vez de adicionar na memória, o conteúdo da variável é atualizado com o novo valor. A especificação a seguir apresenta o processo **UPDATE**, que recebe o valor **val**, a ser adicionado em uma variável **var** já existente na memória **m**.

```
UPDATE(var, val, m, next) =
```

```
let
    check = {value | (var_, value) <- m, var == var_}
    others = {(var_, value) | (var_, value) <- m, var != var_ }
    m_ = union(others, {(var, <val>^tail(content(check)))})
within
    if(check != {}) then
        next(m_)
    else
        ERROR2(m)
```

3.6 Controle de Fluxo

A linguagem ROBO possui estruturas de controle de fluxo, como condicionais e repetição.

A especificação a seguir apresenta um exemplo de uso da estrutura condicional na linguagem ROBO que verifica se à frente do robô não existe um obstáculo. Caso a expressão frontIsClear retorne verdadeiro, o robô anda para a frente.

```
if(frontIsClear){
    forward(1)
}
```

O processo **IF1** (a seguir) é gerado como resultado da tradução da estrutura condicional ROBO apresentado acima. Este processo recebe a memória como parâmetro e verifica se é possível andar para frente avaliando a função CSP **frontIsClear**, que formaliza a função de mesmo nome em ROBO. A especificação de `frontIsClear` pode ser encontrada no Apêndice A. Se o caminho estiver livre, ele vai chamar **FORWARD2**, que se comporta como o processo **FORWARD**, que irá mover o robô uma casa para frente e se comporta como o processo **TERMINATE**. Caso o caminho não esteja livre, **IF1** se comporta como o processo **TERMINATE**.

No código ROBO mostrado anteriormente, não existe uma variável na expressão booleana do condicional. Portanto, o tradutor cria no processo IF1 um **if(true)**. Caso existam variáveis na expressão, a tradução substitui **true** por uma expressão que avalia se as variáveis utilizadas na expressão estão definidas. A expressão **coin?c** que aparece no processo **IF1** é sempre gerada pelo tradutor, para o caso de existir uma chamada para a função **flipCoin()** na expressão condicional. Neste exemplo, esta função não é chamada. Em ROBO, é possível passar variável como parâmetro para algumas funções ou condicionais. Para recuperar esse valor, é utilizado **let** e **within** para colocar no contexto do processo CSP as variáveis e seus valores. Dentro do **let** é sempre colocada a constante

dont_care, com o proposito de evitar que o **let** fique vazio e FDR acuse um erro sintático. A linguagem CSP não aceita **let** sem declarações.

```
IF1(m) =
    let
        x = toInt(get(m,"X"))
        y = toInt(get(m,"Y"))
        o = toInt(get(m,"ORIENTATION"))
        bcns = set(getAllBeacons(m))
        FORWARD2(m) =
            let
                dont_care = 0
            within
                if(true) then
                    FORWARD(toInt(I.1), m, TERMINATE)
                else
                    ERROR2(m)
    within
        coin?c ->
        if(true) then (
            if(toBool(B.frontIsClear(x, y, o, bcns))) then (
                FORWARD2(m)
            ) else (
                TERMINATE(m)
            )
        )else(
            ERROR2(m)
        )
```

A especificação a seguir apresenta um exemplo de estrutura de repetição na linguagem ROBO, que verifica se a posição a frente do robô está livre. Caso a expressão **frontIsClear** retorne verdadeiro, o robô irá andar para frente e verificar novamente se a posição a frente do robô está livre.

```
repeatWhile(frontIsClear){
    forward(1)
}
```

O processo **WHILE1** a seguir, corresponde ao processo que é resultado da tradução do código ROBO acima. Este processo recebe a memória como parâmetro e verifica se é

possível andar para frente avaliando a função CSP **frontIsClear**. Se o caminho estiver livre, ele vai se comportar como o processo **FORWARD2**, que por sua ver se comporta como o processo **FORWARD**, que irá mover o robô uma casa para frente e se comportar como o processo **WHILE1**. Caso o caminho não esteja livre, **WHILE1** se comporta como o processo **TERMINATE**

```
WHILE1(m) =
    let
        x = toInt(get(m, "X"))
        y = toInt(get(m, "Y"))
        o = toInt(get(m, "ORIENTATION"))
        bcns = set(getAllBeacons(m))
        FORWARD2(m) =
            let
                dont_care = 0
            within
                if(true) then
                    FORWARD(toInt(I.1), m, WHILE1)
                else
                    ERROR2(m)
    within
        if(true) then (
            if(toBool(B.frontIsClear(x, y, o, bcns))) then (
                FORWARD2(m)
            ) else (
                TERMINATE(m)
            )
        ) else (
            ERROR2(m)
        )
```

3.7 Procedimentos

A linguagem ROBO também permite a criação de procedimentos para o programa. Procedimentos na linguagem ROBO funcionam como sub-rotinas, sendo chamadas, executando uma sequência de passos e retornando ao fluxo original do programa. O texto a seguir apresenta uma função simples na linguagem ROBO que procura pelo *beacon* no mapa.

```
procedure findBeacon{
    repeatWhile(not(frontIsBeacon)){
        if(frontIsClear){
            forward(1)
        } else {
            backward(1)
            if(flipFlop()){
                right
            } else{
                left
            }
        }
    }
}
```

No texto a seguir, o processo **findBeacon** representa a função escrita acima no programa ROBO em CSP. Ele recebe a representação da memória **m** e o próximo processo a ser chamado. Este processo começa por criar um processo que representa uma estrutura de repetição, que verifica se a frente do robô não possui um beacon. Se não existir, o robô irá verificar se a frente dele possui um espaço livre, se possuir, ele anda para frente, e se não estiver livre, o robo irá recuar, e decidir se vai virar para a esquerda ou para a direita.

```
findBeacon(m, next) =
    let
        dont_care = 0
        WHILE1(m) =
            let
                x = toInt(get(m, "X"))
                y = toInt(get(m, "Y"))
                o = toInt(get(m, "ORIENTATION"))
                bcns = set(getAllBeacons(m))
                IF2(m) =
                    let
                        x = toInt(get(m,"X"))
                        y = toInt(get(m,"Y"))
                        o = toInt(get(m,"ORIENTATION"))
                        bcns = set(getAllBeacons(m))
                        FORWARD3(m) =
                            let
                                dont_care = 0
```

```
        within
            if(true) then
                FORWARD(toInt(I.1), m, WHILE1)
            else
                ERROR2(m)
    BACKWARD4(m) =
        let
            dont_care = 0
        within
            if(true) then
                BACKWARD(toInt(I.1), m, IF5)
            else
                ERROR2(m)
    IF5(m) =
        let
            x = toInt(get(m,"X"))
            y = toInt(get(m,"Y"))
            o = toInt(get(m,"ORIENTATION"))
            bcns = set(getAllBeacons(m))
            RIGHT6(m) =
                let
                    dont_care = 0
                within
                    if(true) then
                        RIGHT(m, WHILE1)
                    else
                        ERROR2(m)
            LEFT7(m) =
                let
                    dont_care = 0
                within
                    if(true) then
                        LEFT(m, WHILE1)
                    else
                        ERROR2(m)
        within
            coin?c ->
            if(true) then (
                if(toBool(B.c)) then (
```

```
                                    RIGHT6(m)
                                ) else (
                                    LEFT7(m)
                                )
                            )else (
                                ERROR2(m)
                            )
                    within
                        coin?c ->
                            if(true) then (
                                if(toBool(
                                    B.frontIsClear(x, y, o, bcns))) then (
                                    FORWARD3(m)
                                ) else (
                                    BACKWARD4(m)
                                )
                            )else (
                                ERROR2(m)
                            )
            within
                if(true) then (
                    if(toBool(
                        B.not(
                            toBool(
                                B.frontIsBeacon(x, y, o, bcns)
                            )
                        )
                    )) then (
                        IF2(m)
                    ) else (
                        next(m)
                    )
                ) else (
                    ERROR2(m)
                )
    within
        WHILE1(m)
```

A seguir apresenta a função fatorial, escrita na linguagem ROBO. A função recebe

um parâmetro **x**, e se **x** for menor que 2, então ele retorna 1, caso contrário, **x*fat(x-1)**.

```
procedure fat(x) {
    if(x < 2) {
        return(1)
    }else{
        return (x*fat(x-1))
    }
}
```

O texto a seguir apresenta a função fatorial traduzida da linguagem ROBO para sua representação em CSP. O processo **fat** recebe 2 argumentos, a representação da memoria **m** e o próximo processo **next**. O processo começa salvando em **x**, representado por **x_fat**, o valor passado por parâmetro. Em seguida, o processo **IF2** é chamado, verificando se o valor presente em **x_fat** é menor que 2. Caso o valor presente em **x_fat** seja menor que 2, então o processo **RETURN3** é chamado, indo para o próximo processo **next**. Caso **x_fat** seja maior ou igual a 2, então é iniciado a recursão, salvando na sequência da memória o novo valor que vai ser passado para a próxima chamada do processo, repetindo até satisfazer o processo **IF2**.

```
fat(m, next) =
    let
        dont_care = 0
        ASSIGN1(m) =
            let
                dont_care = 0
                ARG1 = get(m, "ARG1")
                m' = {(var, head(val)) | (var, val) <- m}
            within
                if(ARG1 != UNDEFINED and true) then
                    PUSH("x_fat", get(m, "ARG1"), m, IF2)
                else
                    ERROR2(m)
        IF2(m) =
            let
                x_fat = get(m, "x_fat")
                RETURN3(m) = ASSIGN("RETURN", I.1, m, next)
                ASSIGN4(m) =
                    let
                        dont_care = 0
```

```
                    x_fat = get(m, "x_fat")
                within
                    if(x_fat != UNDEFINED and true) then
                        ASSIGN("ARG1",
                        I.(toInt(get(m, "x_fat"))-toInt(I.1)),
                        m,
                        ASSIGN5)
                    else
                        ERROR2(m)
            ASSIGN5(m) = RETURN("AUX1",fat, m, POP6)
            POP6(m) = POP("x_fat", m, RETURN7)
            RETURN7(m) = ASSIGN("RETURN",
                I.(toInt(get(m, "x_fat"))*toInt(get(m, "AUX1"))),
                m,
                next)
        within
            coin?c ->
            if(x_fat != UNDEFINED and true) then (
                if(toBool(B.(toInt(get(m, "x_fat")) < toInt(I.2)))) then (
                    RETURN3(m)
                ) else (
                    ASSIGN4(m)
                )
            )else (
                ERROR2(m)
            )
within
    ASSIGN1(m)
```

4 Traduzindo programas ROBO para CSP

Na implementação da tradução, foram criados 3 arquivos SDF3 (para definição da sintaxe da linguagem ROBO) e 23 arquivos com código Stratego, que foram usados para definir as regras de tradução dos elementos sintáticos de ROBO para a respectiva representação em CSP. O módulo principal da definição da sintaxe é o **Robo2CSP**, já o módulo principal com o código da tradução é o **robo2csp**, que contém 15 outros módulos com regras específicas para expressões, procedimentos, parâmetros de processos, entre outros elementos da sintaxe.

A Seção 4.1 dá uma visão geral da gramática de ROBO usada para montar a AST, que é a entrada para as regras que são apresentadas nas seções seguintes. O programa ROBO a seguir será utilizado como exemplo para ilustrar o resultado da tradução.

```
right
repeat(4){
    if(frontIsObstacle){
        right
        forward(1)
        left
        forward(2)
        left
        forward(1)
        right
    }else{
        forward(1)
    }
}
```

4.1 Gramática ROBO

O módulo a seguir define a gramática de ROBO que é a entrada das regras de tradução de ROBO para CSP.

O módulo Robo2CSP é o principal na definição da gramática. O símbolo inicial da gramática é o símbolo **Start**, que é definido como um programa (**Program**), que por sua vez é um conjunto de 0 ou mais comandos (**Statement**). A produção **Statement** pode ser uma instrução (**Instr**) ou uma declaração(**Declaration**). A seguir, apresentamos fragmentos deste módulo.

```
module Robo2CSP

context-free syntax
    Start.Program = <<{Statement "\n"}*>>
    Statement.Instr = <<Instr>>
    Statement.Declaration = <<Declaration>>
```

Uma declaração pode ser a declaração de uma variável (**Variable**) ou a declaração de um procedimento, que pode ser parametrizado ou não ter parâmetros (**ProcedureParam** e **Procedure** respectivamente).

```
    Declaration.Variable = <<Identifier> = <Expr>>
    Declaration.Procedure =
        <procedure <Identifier>{ <{Statement "\n"}*> }>
    Declaration.ProcedureParam =
        <procedure <Identifier> <Params> { <{Statement "\n"}*> }>
```

Uma instrução é um comando da linguagem ROBO que quando executado faz o robô andar no mapa, pintar o chão ou interagir com o *beacon*. A seguir, algumas produções da gramática que representam os comandos da linguagem para: colocar o robô para frente, para trás, pintar o chão de branco (com e sem parênteses), pegar o *beacon* e comer o *beacon*.

```
    Instr.FORWARD = < forward(<Expr>) >
    Instr.BACKWARD = < backward(<Expr>) >

    Instr.PAINTWTE = < paintWhite >
    Instr.PAINTWTEP = < paintWhite() >

    Instr.PICKUP = < pickUp >
    Instr.EATUP = < eatUp >
```

Dentre as produções de uma instrução, estão os comandos de controle de fluxo, como estruturas condicionais e de repetição, além das produções para os comandos break, END e return.

```
    Instr.IF = < if (<Expr>) {<{Statement "\n"}*>} <Else?>>
    Else.ELSEIF = < else if (<Expr>) {<{Statement "\n"}*>} <Else?>>
    Else.ELSE = < else {<{Statement "\n"}*>} >
```

```
Instr.RPTWLE = < repeatWhile (<Expr>) {<{Statement "\n"}*>}>

Instr.BREAK = < break >

Instr.END = < end >

Instr.ReturnValue = < return(<Expr>) >
```

O texto a seguir apresenta a AST gerada pelo Spoofax para o programa ROBO apresentado na seção anterior. Esta AST é um exemplo do tipo de dado que é a entrada para as regras apresentadas a seguir, que como saída produzem uma especificação CSP.

```
Program(
  [ Instr(RIGHT())
  , Instr(
      RPTINT(
        Num("4")
      , [ Instr(
            IF(
              FROISOBS()
            , [ Instr(RIGHT())
              , Instr(FORWARD(Num("1")))
              , Instr(LEFT())
              , Instr(FORWARD(Num("2")))
              , Instr(LEFT())
              , Instr(FORWARD(Num("1")))
              , Instr(RIGHT())
              ]
            , Some(ELSE([Instr(FORWARD(Num("1")))]))
            )
          )
        ]
      )
    )
  ]
)
```

4.2 Traduzindo instruções básicas

A seguir, são apresentados exemplos de regras de tradução implementadas com Stratego que geram um modelo mais eficiente de ser analisado com FDR.

O texto a seguir mostra um exemplo de regra em Stratego. Neste exemplo, a regra chama-se *to-csp0*. Esta regra casa padrão com o elemento sintático *Program([])*. Este elemento corresponde a um programa sem comandos (vazio). O resultado da aplicação desta regra é uma string cujo conteúdo aparece entre os caracteres $[e]. O processo CSP chamado *PROGRAM* corresponde ao comportamento do programa. Para um programa vazio, o comportamento de *PROGRAM* corresponde ao processo *TERMINATE*. Este último é um processo que termina com sucesso sem produzir resultado algum.

```
to-csp0:
    Program([]) ->
    $[PROGRAM = TERMINATE(INIT)]
```

A regra apresentada a seguir, casa padrão com um programa não vazio, isto é, um programa que possui uma lista não vazia de instruções. O padrão é uma lista de instruções, *statements*, que contem o conteúdo do programa. Como parte da string resultante da regra, temos a substring *procedures'*, cujo conteúdo corresponde ao retorno da aplicação da regra *declared-procedures* com o argumento *procs*, que são os procedimentos sem parâmetros presentes na lista de instruções *statements*, e o numeral 1 um contador a ser utilizado na criação dos processos, assim, podemos diferenciar a chamada de dois processos que, por exemplo, chamam o processo **FORWARD**. O retorno são os processos que correspondem aos procedimentos sem parâmetros escritos no programa, caso existam. O conteúdo da substring *continuation'* corresponde ao retorno da aplicação da regra *to-csp0* no conteúdo do pre-processamento da lista de instruções *statements* sem os procedimentos coletados em *procs*, as variáveis declaradas no programa e a contagem de processos criados nos procedimentos. O retorno são os processos CSP que correspondem aos comandos do programa.

```
to-csp0:
    Program(statements) ->
    $[[procedures']
    [continuation']]
    with
        [statements',count'] := <pre-proccess0> [[],statements, "1"];
        procs := <get-declared-procedures> statements';
        procedures' := <declared-procedures> [procs, "1"];
        procedure_length := <procedure-length> procs;
```

```
        n’:= <addS> (procedure_length, "1");
        declared_vars’ := <to-get-declared-vars> statements’;
        continuation’ :=
            <to-csp0> [<diff> (statements’,procs), declared_vars’, n’]
```

A regra a seguir representa um programa que não contem comandos a serem executados. Ela casa padrão com um conjunto vazio de comandos (*[]*), uma lista de variaveis declaradas (*vars*) e um contador de comandos (n). O resultado da aplicação desta regra é a criação do processo *PROGRAM*, que corresponde ao processo *TERMINATE*, que termina sem produzir um resultado.

```
to-csp0:
    [[], vars, n] ->
    $[PROGRAM = TERMINATE(INIT)]
```

A regra a seguir representa um programa que contem pelo menos um comando a ser executado. Ela casa padrão com um conjunto de comandos *[statement | tail]*, onde *statement* é o primeiro comando na lista de instruções e *tail* é uma lista com o resto das instruções a serem traduzidas; uma lista de variáveis declaradas (*vars*); e o contador de processos (n). A string resultante desta regra contem a substring *procedures’*, que contem a declaração dos procedimentos com parâmetros; *vars’*, um conjunto com as variáveis presentes no programa ROBO; a declaração do processo *PROGRAM* que se comporta como o primeiro comando do programa (*process’*) e recebe como parâmetro *INIT*; e a string *continuation’*, que corresponde a tradução do restante do programa presente em *tail*.

```
to-csp0:
    [[statement | tail], vars, n] ->
    $[[procedures’]
    [vars’]
    PROGRAM = [process’](INIT)
    [continuation’]
    ]
    with
        procs’ :=
            <get-declared-procedures-with-params> [statement | tail];
        [statement’ | tail’] := <diff> ([statement | tail],procs’);
        procedures’ :=
            <declared-procedures> [procs’, n, [statement’ | tail’]];
        proc_vars’ := <proc-vars-to-vars> [[], procs’];
        proc_args’ := <proc-vars-to-args> [procs’, 0];
```

```
vars' :=
    <get-declared-vars0> <conc> (vars, proc_vars', proc_args');
n' := <addS> (<procedure-length> procs', n);
process':=
    <get-next-process>
        [[statement'], n', ["TERMINATE", "ERROR", "TERMINATE"]];
continuation' :=
    <to-csp>
        [[statement' | tail'], n',
        ["TERMINATE", "ERROR", "TERMINATE"]]
```

A regra *to-csp*, utilizada na regra a cima para definir o valor de continuation', é composta por três elementos:

- uma lista de elementos sintáticos a serem traduzidos para CSP;
- um valor n que corresponde ao contador de comandos; e
- uma lista que corresponde a continuação do programa. Esta lista é utilizada apenas quando a lista de comandos a serem traduzidos está dentro de uma estrutura condicional (if/else), de uma estrutura de repetição (repeat/repeatWhile), ou dentro de um procedimento (procedure).

No terceiro item acima, o primeiro elemento da lista é a instrução subsequente após a conclusão de uma estrutura condicional, o segundo elemento é a instrução subsequente após a conclusão da estrutura de repetição (esta instrução é executada se houver a chamada de um *break* dentro da estrutura de repetição); o terceiro elemento é a próxima instrução a ser chamada quando é realizado um retorno no procedimento. Esta separação é necessária para controlar o desvio de fluxo ocasionado pelo uso de uma estrutura condicional, por uma estrutura de repetição e de uma chamada de procedimento. Em CSP não existe estrutura de repetição, mas é possível modelar um laço utilizando recursão (conforme mostrado na Seção 3.6). Sendo assim, uma estrutura de repetição é uma estrutura condicional em que sua próxima instrução é ela mesma. Portanto o que existe do lado de fora da estrutura só será chamado se a condicional da estrutura avaliar como falso. Se o comando *break* for usado dentro do laço, será preciso saber qual a instrução que aguarda após a estrutura. Isto justifica porque a segunda posição do terceiro item da regra *to-csp* só é preenchida quando há uma estrutura de repetição no código ROBO, e também porque a primeira posição só é acessada pelas estruturas condicionais ou de repetição com finalidade de identificar qual a instrução que será chamada quando a execução das estruturas chegarem ao fim. O valor padrão para os elementos da lista é *TERMINATE*, pois para uma instrução fora de

estruturas, a próxima instrução, se não houver, é o encerramento, e *ERROR*, indicando que um *break* foi mal posicionado.

A seguir, temos a definição da regra *get-next-process*, que é usada para traduzir a próxima instrução a ser chamada no modelo CSP. Quando não há mais uma instrução no corpo de instruções, então a próxima instrução é o valor do parâmetro *nextIf*, que é a instrução que se encontra fora da condicional. O parâmetro *nextBreak* só é usado na chamada do *break* para pular para a instrução fora da estrutura de repetição, e o parâmetro *nextReturn* só é usado na chamada de um *return*. O elemento **c** representa o contador atual do próximo processo

```
get-next-process:
    [[], c, [nextIf, nextBreak, nextReturn]] -> $[[nextIf]]
```

Quando há uma instrução para ser traduzida, a regra *get-next-process* retorna a string *instruction'* acompanhado pelo contador *c*.

```
get-next-process:
    [[instruction | tail], c, next] ->
    $[[instruction'][c]]
    with
        instruction' := <get-process-name> instruction
```

A seguir, mostramos a regra *get-process-name* que gera o conteúdo para *process'* e *continuation'* na regra *to-csp0* apresentada anteriormente. Esta regra casa padrão com cada comando possível da linguagem. Para facilitar o entendimento da regra, indicamos que elemento da AST representa que comando da linguagem. Os comandos *forward(n)* e *backward(n)* são representados pelos elementos *FORWARD(Num(n))* e *BACKWARD(Num(n))*; *right* e *left* são representados por *RIGHT()* e *LEFT()*; *end* é representado por *END()*; *break* é representado por *BREAK()*; *if(expressão)* é representado pelo elemento *IF(expression, ifBody, elseBody)*, onde *expression* é a expressão da condicional; *ifBody* é o corpo de expressões para caso a condicional seja verdade; *elseBody* é o corpo de expressões para caso a condicional seja false, ou um conjunto vazio. Em ROBO, temos 3 formas de expressar repetição: *repeatWhile(expressão)*, que é equivalente a uma estrutura de repetição padrão; *repeat* equivalente a uma estrutura de repetição que é sempre verdade; *repeat(n)* que é equivalente a uma estrutura de repetição que repete n vezes. O comando *repeatWhile(expressão)* é representado pelo elemento *RPTWLE(expression, whileBody)*, onde *expression* é a condicional da estrutura de repetição e *whileBody* é o conjunto de instruções do laço. O comando *repeat* é representado por *RPTINF(whileBody)*. Diferente do *repeatWhile*, o *repeat* é uma estrutura de repetição cuja condicional é sempre verdade, então não é preciso carregar uma expressão.

```
get-process-name:
    FORWARD(n) -> $[FORWARD]

get-process-name:
    BACKWARD(n) -> $[BACKWARD]

get-process-name:
    IF(expression, ifBody, elseBody) -> $[IF]

get-process-name:
    RPTINF(whileBody) -> $[WHILE]

get-process-name:
    END() -> $[END]

get-process-name:
    BREAK() -> $[BREAK]

get-process-name:
    RPTWLE(expression, whileBody) -> $[WHILE]

get-process-name:
    LEFT() -> $[LEFT]

get-process-name:
    RIGHT() -> $[RIGHT]
```

Com finalidade de ilustração, se aplicarmos a regra *to-csp0* ao programa ROBO, que contém como único comando *forward(1)*, temos como resultado string *PROGRAM = FORWARD1(INIT).*

A regra *to-csp0* usa a regra *to-csp* para definir o valor de *continuation'*. A seguir, apresentamos o caso base da regra *to-csp*, que casa padrão quando não houver mais instruções no conjunto de instruções, indicando que o programa chegou à sua ultima instrução. Neste caso, como não há uma nova instrução a ser escrita, a string resultante é vazia.

```
to-csp:
    [[], c, next] ->
    $[]
```

Na sequência, apresentamos a regra *instruction-lenght*, que é utilizada na regra *to-csp* para calcular o contador de processos usado em *continuation'*. Esta regra retorna o tamanho dos corpos das estruturas condicionais e de repetição. Este conjunto de regras, conta quantos comandos existem em um bloco de comandos. O valor retornado por esta função é concatenado no fim do nome do processo, de forma que o nome do processo CSP gerado seja único. Lendo de baixo pra cima, se houver apenas uma instrução simples, então a contagem continua para a cauda da lista com o incremento de 1 no contador. Se houver uma condicional, então vai ser a soma dos tamanhos do corpo do *if* e do corpo do *else* da estrutura, incrementado de 1 no contador. Se for uma estrutura de repetição, então vai ser a soma do corpo da estrutura de repetição com a sua cauda incrementado em 1 no contador. Se não houver mais instruções, então a regra vai retornar o valor do contador. Como a definição é recursiva, ela está preparada para qualquer combinação de comandos. Por exemplo, estrutura condicional dentro de uma estrutura condicional ou de repetição, ou uma estrutura de repetição dentro de uma estrutura condicional ou de repetição.

```
instruction-length:
    [[], n] -> $[[n]]

instruction-length:
    [[Instr(RPTINF(whileBody)) | tail], n] ->
    $[[n']]
    with
        n1:= <instruction-length> [whileBody, "0"];
        n2:= <instruction-length> [tail, "0"];
        n':= <addS> (n1, <addS> (n2, "1"))

instruction-length:
    [[Instr(RPTWLE(expression, whileBody)) | tail], n] ->
    $[[n']]
    with
        n1 := <instruction-length> [whileBody,"0"];
        n2 := <instruction-length> [tail, "0"];
        n' := <addS> (n1, <addS> (n2, "1"))

instruction-length:
    [[Instr(IF(expression, ifBody, Some(ELSE(elseBody)))) | tail], n] ->
    $[[n']]
    with
        n1 := <instruction-length> [ifBody, "0"];
```

```
        n2 := <instruction-length> [elseBody, "0"];
        n3 := <instruction-length> [tail, "0"];
        n' := <addS> (n1, <addS> (n2, <addS> (n3, <addS> (n, "1"))))

instruction-length:
    [[instr | tail], n] ->
    $[[n']]
    with
        n':= <instruction-length> [tail, <addS> (n, "1")]
```

No texto a seguir, temos a definição da regra auxiliar *get-process-parameter*, que é usada para coletar o parâmetro de comandos, como por exemplo, os comandos *forward* e *backward*. Como *right*, *left* e *break* não recebem parâmetro, e também não retornam nenhum parâmetro, o resultado de aplicar a regra é uma lista vazia. O mesmo não acontece com *forward* e *backward*, que possuem um parâmetro numérico. Para estes comandos, a aplicação da regra retorna esse parâmetro numérico seguido por uma virgula e um espaço. O Stratego remove espaços da string colocada entre *$[* e *]*, se as mesmas estão no começo ou no fim. Por este motivo, na string que resulta da aplicação da regra em *forward* e *backward*, foi adicionado um *a*, apenas por questão estética para existir um espaço após a virgula.

```
get-process-parameter:
    RIGHT() -> $[]

get-process-parameter:
    LEFT() -> $[]

get-process-parameter:
    BREAK() -> $[]

get-process-parameter:
    Return() -> $[]

get-process-parameter:
    ReturnValue(value) -> $[toInt([value']),]
    with
        value' := <to-exp> value

get-process-parameter:
    FORWARD(n) ->
```

```
    $[toInt([n']),[a]]
    with
        n' := <to-exp> n;
        a := " "

get-process-parameter:
    BACKWARD(n) ->
    $[toInt([n']),[a]]
    with
        n' := <to-exp> n;
        a := " "
```

No que segue, exibimos o caso base da regra *to-csp* que é usada para gerar a string *continuation'* da regra *to-csp0*. Este caso da regra casa padrão quando a lista de comandos a ser traduzida consiste em apenas uma instrução simples, como *forward(n)*, *backward(n)*, *right()*, *left()*, *pickUp()*, *eatUp*, *putDown*, *paintWhite*, *paintBlack* e *stopPainting*. A regra recebe uma lista de instruções, sendo a cabeça da lista *Instr(instruction)* concatenada com a cauda *tail*; um contador de instruções *c*; e, a lista de próximas instruções *next*. Esta regra retorna uma string, onde *process'* é o nome da instrução atual, *c* é o contador atual da instrução, *n'* o valor recebido como parametro, *nextprocess'* é o nome do próximo processo a ser chamado, e *continuation'* a tradução das instruções contidas em *tail*.

```
to-csp:
    [[Instr(instruction)| tail], c, next] ->
    $[[process'][c](m) =
        let
            dont_care = 0
            [retrieve']
        within
            if([defined']) then
                [process']([n']m, [nextprocess'])
            else
                ERROR2(m)
    [continuation']]
    with
        retrieve' := <get-vars> <to-get-vars> instruction;
        defined' := <verify-vars> <to-get-vars> instruction;
        n' := <get-process-parameter> instruction;
        process' := <get-process-name> instruction;
        nextprocess' := <get-next-process> [tail, <addS> (c, "1"), next];
```

```
continuation' := <to-csp> [tail, <addS> (c, "1"), next]
```

4.3 Traduzindo instruções de controle de fluxo

Na tradução de uma estrutura condicional, temos duas situações, *if* com o *else* e *if* sem o *else*. O parser, quando tem uma condicional, pode retornar dois elementos na AST : um IF com um corpo de *else* (*Some(Else(elseBody))*), ou *None()*, quando o *if* não possui um *else*.

Apresentamos a seguir, o padrão da regra *to-csp* que traduz uma estrutura condicional com *else* da linguagem ROBO para a especificação CSP correspondente. A diferença destas regras a seguir para o padrão da regra *to-csp* da seção anterior, é que deixamos explicito aqui que essa regra é para a estrutura. Como saída desta regra, temos a definição do processo CSP que representa a estrutura condicional, cujo nome inicia com *IF* e cuja última parte do nome é o valor do contador *c*. Conforme apresentado no capítulo anterior, no modelo CSP, recuperamos da memória 3 itens: a posição do robô no mapa (x, y), a orientação na qual o robô está se movimento (norte, sul, leste, oeste) e a posição do *beacon* no mapa(bcns). CSP não permite a definição de processos dentro do *within*, apenas dentro do *let*, colocamos os processos CSP que representam *ifBody* e *elseBody* dentro do *let*, deixando dentro do *within* o processo que representa o comportamento inicial. Dentro do *with* da regra temos: as variáveis n1 e n2, que são os respectivos tamanhos de *ifBody* e *elseBody*, retornados pela regra *instruction-length*; *c_tail*, que é o contador que representa a cauda após medir o tamanho do *ifBody* e *elseBody* mais a estrutura; *expression'* que é a conversão da expressão da condicional para CSP; *tail'* que é o nome da instrução que vai ser chamado da cauda da lista de processos; *ifBody'* e *elseBody'* são strings que representam o nome do processo cujo comportamento especifica as primeiras instruções do *ifBody* e *elseBody*; e, por último, *tailDef* representa a definição das instruções contidas na cauda.

```
to-csp:
    [[Instr(IF(expression,ifBody,Some(ELSE(elseBody)))) | tail], c,
    [nextIf, nextBreak, nextReturn]] ->
    $[IF[c](m) =
        let
            x = toInt(get(m,"X"))
            y = toInt(get(m,"Y"))
            o = toInt(get(m,"ORIENTATION"))
            bcns = set(getAllBeacons(m))
            [retrieve']
            [ifBodyDef]
```

```
            [elseBodyDef]
        within
            coin?c ->
            if([defined']) then (
                if(toBool([expression'])) then (
                    [ifBody'](m)
                ) else (
                    [elseBody'](m)
                )
            )else (
                ERROR2(m)
            )
    [tailDef]]
    with
        retrieve' := <get-vars> <to-get-vars> expression;
        defined' := <verify-vars> <to-get-vars> expression;
        n1 := <instruction-length> [ifBody,"0"];
        n2 := <instruction-length> [elseBody,"0"];
        c_tail := <addS> (n2, <addS> (n1, <addS> (c, "1")));
        expression' := <to-exp> expression;
        tail' :=
            <get-next-process>
                [tail, c_tail, [nextIf, nextBreak, nextReturn]];
        ifBody' :=
            <get-next-process>
                [ifBody, <addS> (c, "1"), [tail',nextBreak, nextReturn]];
        elseBody' :=
            <get-next-process>
                [elseBody, <addS> (n1, <addS> (c, "1")),
                [tail',nextBreak, nextReturn]];
        ifBodyDef :=
            <to-csp>
                [ifBody, <addS> (c, "1"), [tail',nextBreak, nextReturn]];
        elseBodyDef :=
            <to-csp>
                [elseBody, <addS> (n1, <addS> (c, "1")),
                [tail',nextBreak, nextReturn]];
        tailDef := <to-csp>
            [tail, c_tail, [nextIf, nextBreak, nextReturn]]
```

4.4 Traduzindo instruções de repetição

Na linguagem ROBO, existem várias estruturas de repetição, entre elas: *repeatWhile*, que é a mais importante, uma vez que permite expressar qualquer padrão de repetição. Por este motivo, mostramos apenas a regra de tradução para esta estrutura.

No texto a seguir, temos a definição da regra para a tradução da estrutura de repetição *repeatWhile*. Na AST, o elemento que representa a estrutura *repeatWhile* tem uma expressão (*expression*) e um conjunto de instruções (*whileBody*). A tradução do *repeatWhile* para CSP, é muito semelhante à regra do if sem else, com a diferença de que, na composição do *whileBodyDef*, seu último processo chama a si mesmo, recursivamente. Desta forma, repetindo a estrutura enquanto a condição for verdade. A definição do *whileBody* chama o primeiro processo do *whileBodyDef* se *expression'* for verdade, ou a primeira instrução de *continuation* se for falso.

```
to-csp:
    [[Instr(RPTWLE(expression, whileBody)) | tail], c,
    [nextIf, nextBreak, nextReturn]] ->
    $[WHILE[c](m) =
        let
            x = toInt(get(m, "X"))
            y = toInt(get(m, "Y"))
            o = toInt(get(m, "ORIENTATION"))
            bcns = set(getAllBeacons(m))
            [retrieve']
            [whileBodyDef]
        within
            if([defined']) then (
                if(toBool([expression'])) then (
                    [whileBody'](m)
                ) else (
                    [endWhile](m)
                )
            ) else (
                ERROR2(m)
            )
    [continuation]]
    with
        retrieve' := <get-vars> <to-get-vars> expression;
        defined' := <verify-vars> <to-get-vars> expression;
```

```
n1 := <instruction-length> [whileBody, "0"];
endWhile :=
    <get-next-process>
        [tail, <addS> (<addS> (c, "1"), n1),
        [nextIf,nextBreak, nextReturn]];
next' :=
    <get-next-process>
        [[Instr(RPTWLE(expression, whileBody)) | tail],
        <to-get-vars> expression,
        c,
        [nextIf,nextBreak, nextReturn]];
whileBodyDef :=
    <to-csp>
        [whileBody, <addS> (c, "1"),
        [next',endWhile, nextReturn]];
expression' := <to-exp> expression;
whileBody' :=
    <get-next-process>
        [whileBody, <addS> (c, "1"),
        [next', endWhile, nextReturn]];
continuation :=
    <to-csp>
        [tail, <addS> (<addS> (c, "1"), n1),
        [nextIf, nextBreak, nextReturn]]
```

4.5 Traduzindo declarações de variáveis

A formalização para variáveis no modelo CSP consiste em definir um mapeamento $M : A \rightarrow B$, onde A é o nome da variável do usuário, e B é o valor associado à variável. Em ROBO, uma variável pode receber um número inteiro, ou um booleano, porém, em CSP, precisamos definir um único tipo para o imagem do mapeamento. Portanto, o domínio de M é do tipo string, que correspondem ao nome das variáveis do usuário, e o tipo da imagem é uma unificação de valores booleanos e inteiros (o **datatype** *DATA* apresentado no capítulo anterior). Desta forma, é possível salvar o mapeamento das variáveis em um único mapeamento.

Como padrão de qualquer programa ROBO, o mapeamento armazena as variáveis implícitas do programa como, por exemplo, **X** e **Y** que representam a posição do robô no mapa e **ORIENTATION** que representa o ponto cardeal ao qual o robô está olhando. Usamos este mapeamento para acrescentar as variáveis definidas pelos usuários. Para evitar

conflito do nome da variável do usuário com as variáveis implícitas, todas as variáveis do usuário são salvas no mapeamento seguidas do caractere _ (*underscore*), com a finalidade de diferenciar quais variáveis são implícitas e quais variáveis são definidas pelo usuário.

A regra apresentada a seguir, **get-declared-vars0**, coleta as variáveis declaradas pelo usuário e produz a constante *USER_VARS*, que representa o conjunto de variáveis declaradas pelo usuário. Esta regra recebe a lista de variáveis declaradas do usuário. Se não houver variáveis criadas pelo usuário, então será retornado pela regra uma lista vazia. Se a lista passada para a regra não for vazia, então a regra tem uma lista de **ID**s, que são os nomes das variáveis declaradas.

```
get-declared-vars0:
    [] -> $[USER_VARS={}]

get-declared-vars0:
    [Variable(ID(id_),_) | tail] ->
    $[USER_VARS = {"[id_]"[continuation']}
    ]
    with
        continuation' := <get-declared-vars> tail
```

O texto a seguir apresenta a regra *get-declared-vars*, que é uma regra auxiliar para *get-declared-vars0*. Nesta regra, enquanto tiver em sua lista, um identificador de variável(**ID**), esta irá adicionar o identificador no conjunto de variáveis do usuário (**USER_VARS**). Se não houver mais variáveis geradas pelo usuário, então a regra irá parar, mas se houver uma variável, então irá adicionar a variável precedida de uma virgula e continuar.

```
get-declared-vars:
    [] -> $[]

get-declared-vars:
    [Variable(ID(id_),_) | tail] ->
    $[,"[id_]"[continuation']]
    with
        continuation' := <get-declared-vars> tail
```

Exibimos a seguir a regra para a tradução de uma atribuição de variável em ROBO para CSP. Esta regra recebe três parâmetros. O primeiro parâmetro é uma lista de comandos (*statements*), o segundo o contador de comandos (c), e o terceiro as próximas instruções a serem traduzidas (*next*). Esta regra casa padrão quando o comando corresponde a

declaração de uma variável, que é um par, cujo primeiro elemento é o nome da variável (*id_*) e cujo segundo elemento é o valor atribuído (*value*). A aplicação desta regra produz um processo CSP chamado *ASSIGN*, cujo nome é seguido do valor de *c*, e que recebe o parâmetro *m*, que é o estado atual da memória do programa. Dentro do *let*, a tradução recupera o valor de todas as variáveis que aparecem na expressão em *value* na memoria *m* e salva em *retrieve_vars'*. Se alguma variável não está definida, a especificação se comporta como o processo *ERROR2*, que indica que uma variável não existe na memoria. Se a variável estiver definida na memória, então a especificação atualiza o valor da variável no mapeamento. A atualização na memória é realizada pelo processo pré-definido, chamado *ASSIGN*, que foi introduzido no capítulo anterior. Se assumirmos que o programa em análise está correto sintaticamente/semanticamente este tipo de verificação de erro é opcional e poder ser eliminada da tradução. Em CSP, o *let* não pode ser vazio, então se *retrieve_vars* for vazio, a especificação estará errada, por este motivo adicionamos uma atribuição *dont_care = 0* dentro do *let*, que evita que o compilador de CSP acuse erro quando *retrieve_vars'* for vazio.

```
to-csp:
    [[Declaration(Variable(ID(id_), value)) | tail], c, next] ->
    $[ASSIGN[c](m) =
        let
            dont_care = 0
            [retrieve_vars']
        within
            if([defined']) then
                ASSIGN("[id_]", [value'], m, [nextprocess'])
            else
                ERROR2(m)
    [continuation']]
    with
        retrieve_vars' := <get-vars> <to-get-vars> value;
        defined' := <verify-vars> <to-get-vars> value;
        value' := <to-exp> value;
        nextprocess' := <get-next-process> [tail, <addS> (c, "1"), next];
        continuation' := <to-csp> [tail, <addS> (c, "1"), next]
```

A string *defined'*, que aparece na regra acima, corresponde a uma expressão booleana que avalia para verdadeiro se todas as variáveis em que aparecem em *value* estão declaradas. Se alguma não estiver declarada, então o processo CSP resultante para a atribuição se comporta como *ERROR2*. Já se todas as variáveis estiverem declaradas, o processo se

comporta como **ASSIGN**, que salva na memoria m a variável *id_* (o nome original da variável seguido de um caractere _) com o valor *value'*.

4.6 Traduzindo procedimentos

A metodologia para representar formalmente procedimentos é criar uma função em CSP que corresponde a uma chamada de procedimento em ROBO. A primeira coisa a ser feita ao traduzir um programa é extrair da árvore sintática do programa todas as declarações de procedimentos.

Esta extração pode ser vista na regra *to-csp0* apresentada em seção anterior. Naquela regra, é utilizada a regra auxiliar *get-declared-procedures*, que recupera todas as definições de procedimento e guarda na string *procs*. Esta string é a entrada para a regra apresentada a seguir, que transforma cada declaração na respectiva representação CSP.

O texto a seguir apresenta a regra **declared-procedures** que é usada para a geração de um processo CSP que representa um procedimento definido pelo usuário no programa ROBO. O processo CSP gerado tem o nome do procedimento definido no programa ROBO seguido de um caractere _ (*id_*). Este processo também recebe como parâmetro o mapeamento m, que é a representação do estado atual do programa; e, o parâmetro *next*, que é o processo que será chamado quando o procedimento encerrar. Este procedimento, contem no *let* a string *body_def'*, que corresponde a tradução das instruções contidas dentro do corpo do procedimento (*body*). Este corpo é passado para a regra *to-csp*, que produz a especificação CSP para o corpo do procedimento. É colocado como *next* nesta regra, uma lista que contém a função *next* no lugar de *TERMINATE*, assim, quando a string estiver pronta, em vez de chamar *TERMINATE* para encerrar o processo, o processo chamará o processo que representa o próximo comando do programa a ser executado.

```
declared-procedures:
    [[], c] -> $[]

declared-procedures:
    [[Declaration(Procedure(ID(id_), body)) | tail], c] ->
    $[[id_]](m, next) =
        let
            dont_care = 0
            [body_def']
        within
            [body']](m)
    [continuation']]
```

```
    with
        next’:= ["next", "ERROR", "next"];
        body_def’ := <to-csp> [body, c, next’];
        body’ := <get-next-process> [body, c, next’];
        continuation’ :=
            <declared-procedures>
                [tail, <addS> (c, <procedure-length> tail)]
```

A regra para a tradução de chamadas de procedimentos sem parâmetros é apresentada a seguir. Esta regra traduz uma instrução que representa a chamada do procedimento identificado por *id_* com a lista de parâmetros vazia. A regra gera, como resultado, o processo cujo nome inicia com *PROC_CALL* seguido pelo valor do contador. O comportamento deste processo é chamar o processo que representa o comportamento do procedimento, passando como parâmetro para este processo o mapeamento m e a próxima instrução *next’*.

```
to-csp:
    [[Instr(ProcCall(ID(id_), ExprParams(params))) | tail], c,
    [nextIf, nextBreak, nextReturn]] ->
    $[PROC_CALL[c’](m) = [id_](m, [next’])
    [continuation’]]
    with
        params’ := <init-proccall-vars> [[],params,"1"];
        c’ := <addS> (c, <instruction-length> [params, "0"]);
        c’’ := <addS> (c’, "1");
        next := <conc-strings> ("PROC_CALL", c’);
        next’ :=
            <get-next-process>
                [tail, c’’, [nextIf, nextBreak, nextReturn]];
        assign-params’ :=
            <to-csp>
                [params’, c, [next, nextBreak, nextReturn]];
        continuation’ :=
            <to-csp>
                [tail, c’’, [nextIf, nextBreak, nextReturn]]
```

4.6.1 Procedimentos com valor de retorno

Uma chamada de procedimento pode retornar um valor, portanto, na linguagem ROBO, uma expressão pode conter constantes, nomes de variáveis e chamadas a procedi-

mentos. Como uma forma de facilitar a implementação da transformação de expressões com chamadas a procedimentos, é realizada na AST uma normalização. Esta normalização traduz expressões que contém chamadas de procedimento de forma que é criado uma variável auxiliar para guardar o valor de retorno da chamada de cada procedimento na expressão. Então a expressão é escrita substituindo as chamadas de procedimento pelas variáveis auxiliares.

Na regra *to-csp0* que casa padrão com *Program(...)*, é realizada esta normalização na AST. A regra *pre-process0* é usada para este propósito. O pré-processamento cria uma variável auxiliar para guardar o valor de retorno de cada chamada de procedimento. Na expressão, as chamadas de procedimento são então substituídas pelas variáveis auxiliares.

A Figura 5 apresenta um exemplo de código a ser pré-processado, onde é criado uma variável x com a soma do retorno dos procedimentos *Proc1* e *Proc2*. Após aplicar o pré-processamento, o valor de cada chamada de procedimento estará atribuído a uma variável auxiliar: *AUX1* guarda o resultado da chamada do procedimento *Proc1*, e *AUX2* guarda o resultado da chamada do procedimento *Proc2*.

```
1 x = Proc1() + Proc2()
2
3
4
5
6
7
8
9
10
```

```
1 AUX1 = Proc1()
2 AUX2 = Proc2()
3 x = AUX1 + AUX2
4
5
6
7
8
9
10
```

Figura 5 – Exemplo de código antes e depois do pré-processamento (normalização)

O processo a seguir representa a atribuição do retorno de um procedimento em uma variável. O processo *RETURN* recebe 4 argumentos, o nome da variável que vai receber o retorno, a função que vai ser executada para se obter um retorno, o estado atual do programa e o próximo processo. Este processo começa executando a função, que vai salvar em uma variável implícita o retorno do procedimento. Ao fim deste, o processo então salva na variável desejada o valor retornado pelo procedimento e voltar ao estado inicial da variável implícita.

```
RETURN(var, func, m, next) =
    let
        ASSIGN_VALUE(m) = ASSIGN(var, get(m, "RETURN"), m, UPDATE_VALUE)
        UPDATE_VALUE(m) = UPDATE("RETURN", I.0, m, next)
    within
```

```
        func(m, ASSIGN_VALUE)
```

O texto a seguir apresenta a regra que traduz para CSP a atribuição do retorno de um procedimento à uma variável auxiliar, considerando que a AST está normalizada. A regra recebe uma lista, em que o primeiro elemento da lista é a declaração da variável *id_*, que recebe como valor a chamada de um procedimento *proc_name* com seus parâmetros *params*. A regra produz um processo chamado *ASSIGN* seguido por *c*, onde *c* é o valor de um contador. A semântica CSP para este processo é definida como o processo *RETURN*, que salva em *id_* o retorno presente em *proc_name*, passando *m* e o próximo processo *nextprocess'*.

```
to-csp:
    [[Declaration(
        Variable(ID(id_),
            Instr(ProcCall(ID(proc_name),
                    ExprParams(params)
                )
            )
        )
    ) | tail],
    c, [nextIf, nextBreak, nextReturn]] ->
    $[ASSIGN[c](m) = RETURN("[id_]",[proc_name], m, [nextprocess'])
    [continuation']]
    with
        next' :=
            <get-next-process>
                [tail, c, [nextIf, nextBreak, nextReturn]];
        nextprocess' :=
            <get-next-process>
                [tail, <addS> (c, "1"), [next', nextBreak, nextReturn]];
        continuation' :=
            <to-csp>
                [tail, <addS> (c, "1"), [nextIf, nextBreak, nextReturn]]
```

O texto a seguir apresenta um exemplo de código ROBO que declara um procedimento *walk* que chama a função *forward* com o parâmetro *1*, e declara o procedimento *dois* que retorna o valor 2. o programa salva em uma variável x o valor 10, chama a função *walk* e salva na variável y o retorno do procedimento *dois*.

```
procedure walk{
```

```
    forward(1)
}

procedure dois{
    return(2)
}

x = 10
walk()
y = dois()
```

O texto a seguir apresenta a tradução para CSP do código ROBO apresentado acima. Nesta especificação, destacamos o conjunto com as variáveis (do usuário e auxiliares) representado pela constante *USER_VARS*, e a tradução do programa ROBO, que corresponde ao processo *PROGRAM*. O comportamento do processo *PROGRAM* é inicialmente atribuir o valor *I.10* na variável x (processo *ASSIGN3*), que é identificada como $x\_$ na especificação. Na especificação CSP valores recebem uma tag para identificar o tipo do valor; *I.10* representa o valor 10 do tipo inteiro. Após a atribuição, a especificação se comporta como o processo *PROC_CALL4*, que corresponde a chamada da função *walk*. Em seguida, a especificação se comporta como o processo *ASSIGN5* que guarda o retorno da chamada do procedimento *dois* na variável *AUX2*. Por último, o valor de *AUX2* é atribuído a variável *y*. Conforme explicado anteriormente, o pré-processamento cria a variável auxiliar *AUX2* para salvar o valor de retorno da chamada da função *dois*.

```
walk(m, next) =
    let
        dont_care = 0
        FORWARD1(m) =
            let
                dont_care = 0
            within
                if(true) then
                    FORWARD(toInt(I.1), m, next)
                else
                    ERROR2(m)
    within
        FORWARD1(m)

dois(m, next) =
    let
```

```
        dont_care = 0
        RETURN2(m) = ASSIGN("RETURN", I.2, m, next)
    within
        RETURN2(m)

USER_VARS = {"x_", "AUX2", "y_"}

PROGRAM = ASSIGN3(INIT)

ASSIGN3(m) =
    let
        dont_care = 0
    within
        if(true) then
            ASSIGN("x_", I.10, m, PROC_CALL4)
        else
            ERROR2(m)

PROC_CALL4(m) = walk(m, ASSIGN5)
ASSIGN5(m) = RETURN("AUX2", dois, m, ASSIGN6)
ASSIGN6(m) =
    let
        dont_care = 0
        AUX2 = get(m, "AUX2)
    within
        if(AUX2 != UNDEFINED and true) then
            ASSIGN("y_", get(m, "AUX2"), m, TERMINATE)
        else
            ERROR2(m)
```

4.6.2 Procedimentos com parâmetros

A forma de representar em CSP a passagem de parâmetros para os procedimentos é criar variáveis auxiliares que guardam os valores a serem passados por parâmetro, e ler o valor das variáveis auxiliares dentro do procedimento, como primeira parte da especificação. Esta transformação é feita através de regras que realizam o pre-processamento na AST, de forma que as chamadas de procedimento com passagem de argumentos sejam refatoradas em duas partes: guarda de valores em variáveis auxiliares e chamada do procedimento sem parâmetros.

O texto a seguir apresenta a regra que faz o pré-processamento da chamada de procedimento: primeiro, recupera os argumentos da chamada; em seguida, os argumentos são armazenados nas variáveis auxiliares; por último, são removidos os parâmetros da chamada. Os parâmetros (*params*) são pre-processados de forma que recebem o caractere _ seguido do nome do procedimento (para evitar conflito de nomes com variáveis globais com o mesmo nome) e, em seguida, o valor dos parâmetros são guardados em variáveis auxiliares que serão usados no inicio do procedimento.

```
pre-proccess0:
    [proccessed, [Instr(ProcCall(id_, ExprParams(params)))| tail], count]
    -> continuation'
    with
        params' := <pre-proccess-params> params;
        params'' := <init-proccall-vars> [[],params', "1"];
        proccessed' := <conc>(proccessed, params'');
        continuation' :=
            <pre-proccess0>
                [<conc> (proccessed',
                    [Instr(ProcCall(id_, ExprParams([])))]
                ), tail, count]
```

O texto a seguir apresenta a tradução de procedimentos com parâmetros. A diferença da tradução de procedimentos com argumentos e sem argumentos, está na coleta das variáveis auxiliares responsáveis por carregar os argumentos do procedimento, antes de traduzir o corpo do procedimento. Desta forma, é coletado do procedimento, os parâmetros *params*, renomeados para conter o caractere _ seguido do nome do procedimento. Após isso, as variáveis dentro do procedimento, que fazem referencia as variáveis contidas nos parâmetros são renomeadas com o acréscimo do caractere _ (*underscore*) e o nome do procedimento, evitando acesso a uma variável global com mesmo nome. Por fim, é adicionado ao inicio do corpo do procedimento, declaração de variáveis que irão receber os valores contidos nas variáveis auxiliares de argumentos, que serão utilizados no procedimento.

```
declared-procedures:
    [[Declaration(ProcedureParam(ID(id_), Params(params), body)) | tail],
    c, statements] ->
    $[[id_](m, next) =
        let
            dont_care = 0
            [body_def']
        within
```

```
        [body'](m)
[continuation']]
with
    next' := ["next", "ERROR", "next"];
    params' := <rename-proc-vars> [[], params, id_];
    new_body' := <rename-proc-vars-in-body> [[], body, params, params'];
    init' := <init-proc-vars> [[], params',"1"];
    init_length' := <instruction-length> [params', "0"];
    body_def' :=
        <to-csp-procedure0>
            [<conc> (init', new_body'), "1", init_length',
            c, id_, params', next'];
    body' := <get-next-process> [<conc> (init', new_body'), c, next'];
    continuation' :=
        <declared-procedures>
            [tail,
            <addS> (c, <int-to-string> <length> <conc>(init', new_body')),
            statements]
```

Os textos abaixo apresentam, de forma simples, a mudança existente no pre-processamento de procedimentos com variáveis. As figuras representam, o código antes do pre-processamento e o código depois do pre-processamento, respectivamente. O parâmetro x é removido do procedimento, entrando em seu lugar, *x_fib* no corpo do procedimento. Toda referencia a x dentro do corpo do procedimento é substituída por *x_fib*, evitando que façam referencia ao $x\_$ definido no escopo de fora do procedimento. O parâmetro *x_fib* recebe o valor de *ARG1*, que representa o argumento 1 passado como parâmetro antes da chamada.

Antes do pré-processamento:

```
procedure fib(x){
    if(x < 2){
        return(x)
    }else{
        return(fib(x-1)+fib(x-2))
    }
}

x = fib(5)
```

```
Depois do pré-processamento:

    procedure fib(){
        x_fib = ARG1
        if(x_fib < 2){
            return(x_fib)
        }else{
            ARG1 = x_fib-1
            AUX1 = fib()
            ARG1 = x_fib-2
            AUX2 = fib()
            return(AUX1+AUX2)
        }
    }

    ARG1 = 5
    x_ = fib()
```

A Figura 6 apresenta o resultado obtido na tradução do procedimento que corresponde a função de fibonacci em ROBO, passando como parâmetro *10* e esperando como resultado *55*, que deve ser salvo na variável $x\_$. A variável $x\_$ recebe seu valor da variável auxiliar $AUX3$, que corresponde a soma das variáveis $AUX1$ e $AUX2$, que carregam, respectivamente, os valores 34 e 21, que são os valores obtidos pela função de fibonacci passando como argumento 9 e 8.

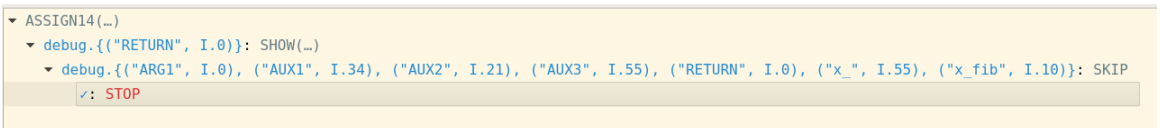

Figura 6 – Resultado obtido na tradução da recursão de fibonacci

5 Validação

As seções seguintes mostram como foi testada a corretude da tradução, e como foi analisada a eficiência da verificação do modelo CSP gerado pelo compilador.

5.1 LGEN

Lua Language Generator (HENTZ; MOREIRA, 2009), ou LGEN, é um gerador de sentenças (dados de teste) baseado na descrição da sintaxe, ao qual usa critérios de cobertura para restringir as sentenças geradas. Ele recebe como entrada a descrição da gramática em uma notação baseada na EBNF e retorna um conjunto de sentenças da linguagem gerada pela gramática. O LGEN foi utilizado neste trabalho para gerar casos de testes da linguagem (programas ROBO), com intuito de validar o tradutor que desenvolvemos.

A Figura 7 mostra um exemplo de uma gramatica em notação baseada na EBNF, onde o símbolo inicial é **PROGRAM**, que possui um ou mais comandos (**STATEMENTS**), um **STATEMENTS** é um comando (**INSTR**), um **INSTR** pode ser um **FORWARD**, um **BACKWARD**, um **RIGHT**, ou um **LEFT**, que levam as produções **"forward(1)"**, **"backward(1)"**, **"right()"** e **"left()"** respectivamente.

```
1   PROGRAM = STATEMENTS+;
2   STATEMENTS = INSTR;
3   INSTR = FORWARD | BACKWARD | RIGHT | LEFT;
4   FORWARD="forward(1)";
5   BACKWARD="backward(1)";
6   RIGHT="right()";
7   LEFT="left()";
8
```

Figura 7 – Exemplo de EBNF

A ferramenta LGEN recebe como entrada esta EBNF e gera um arquivo *.lua* que é usado para criar as sentenças (dados de teste) da gramática. Ao passar o arquivo *.lua* gerado para o LGEN, passamos, obrigatoriamente, como parâmetro a quantidade máxima de ciclos que controla a quantidade de recursões de uma regra não-terminal da EBNF passada, e, a quantidade máxima de derivações, que controla a profundidade que o LGEN irá gerar as sentenças na árvore de derivação. Opcionalmente, podemos dizer qual tipo de cobertura queremos usar, cujas opções são *term* para cobertura terminal, *prod* para cobertura de produção, ou nenhuma cobertura.

A Figura 8 mostra o resultado gerado pela EBNF da Figura 7.

```
1    forward(1)
2    backward(1)
3    right()
4    left()
5
```

Figura 8 – Exemplo de resultado gerado pelo LGEN na EBNF

5.2 Validando a semântica com testes automáticos

Utilizando o LGEN, foram gerados programas de teste como entrada para o tradutor. O texto a seguir mostra alguns exemplos gerados pelo LGEN, onde cada linha representa um programa ROBO usado como caso de teste, criados a partir de uma EBNF que é subconjunto da linguagem ROBO.

```
forward(1)
backward(1)
right()
left()
if(true){ forward(1) }else{ forward(1) }
if(true){ forward(1) }else{ backward(1) }
if(true){ forward(1) }else{ right() }
if(true){ forward(1) }else{ left() }
if(true){ forward(1) }else{ if(true){ forward(1) }else{ forward(1) } }
if(true){ forward(1) }else{ if(true){ forward(1) }else{ backward(1) } }
if(true){ forward(1) }else{ if(true){ forward(1) }else{ right() } }
...
```

Como cada linha representa um programa ROBO, cada linha foi separada em um arquivo, a ser usado na tradução. Para cada programa ROBO, foi verificado o determinismo do programa utilizando a ferramenta FDR. Foram realizadas validações em mais de 5000 programas gerados pelo LGEN, que foram verificados usando FDR. Caso houvesse algum erro sintático ou de tipo, FDR acusaria durante a verificação.

Por limitações de tempo, os testes gerados pelo LGEN não contemplaram procedimentos e variáveis, uma vez que quanto maior a quantidade de testes, maior é o tempo para gerar as sentenças, e para gerar os 5000 testes utilizados, foi necessário aguardar 30 minutos para gerar todos os testes. Como trabalho futuro, planejamos ampliar a EBNF de LGEN para gerar testes que cubram mais elementos da gramática de ROBO.

5.3 Validando a eficiência do modelo

Um dos objetivos deste trabalho é obter um modelo CSP que representa um programa ROBO e seja mais eficiente de ser analisado do que o modelo introduzido em (PEREIRA, 2018). Para comparar a eficiência do modelo antigo com o novo, proposto neste trabalho, foram coletados os tempos de análise utilizados pela ferramenta FDR (tempo de CPU) durante a análise de diferentes especificações CSP. Para um mesmo programa ROBO, foram gerados dois modelos CSP: o modelo CSP introduzido em (PEREIRA, 2018) e o modelo otimizado introduzido neste trabalho. Isto foi feito para mapas de diferentes tamanhos (pequeno, médio e grande).] A analise feita em ambos os modelos foi de verificar a existência de *deadlock* no programa ROBO.

Os tempos para analisar cada um dos modelos em diferentes mapas foram coletados usando o comando *time* no terminal do Ubuntu 20.04.2 LTS, é possível coletar o tempo que o FDR utiliza para avaliar os programas traduzidos. O comando *time* retorna 3 informações, *real time*, *user time* e *sys time*, onde *real time* é o tempo total da execução do comando, *user time* é o tempo do processo no processador a nível de usuário (este tempo é acumulativo entre threads, se rodado em paralelo, vai pegar o tempo em cada thread e somar, passando do tempo real do processo) e o *sys time* que é o tempo que o kernel passou no processo.

Na Tabela 1, temos o resultado da coleta do *real time* para análise dos dois modelos apresentados. É apresentado o tempo do modelo antigo em mapas de tamanho pequeno (10x11), tamanho médio (23x17) e de tamanho grande (74x25). Os testes foram executados 30 vezes e seus respectivos resultados coletados em uma tabela. Os desvios padrões apresentados, na realidade, não são 0, mas são tão pequenos que chegam a ser desprezíveis. Speedup é uma formula utilizada para comparação de desempenho em arquiteturas semelhantes. O resultado do speedup é obtido dividindo o tempo antigo pelo tempo novo. Um resultado menor que 1, significa que o novo tempo é maior que o tempo antigo e que a maquina antiga apresenta uma performance melhor. Um resultado igual a 1 significa que os dois modelos apresentam a mesma performance. Enquanto que um resultado maior que 1, significa que o novo modelo apresenta um tempo menor que o modelo antigo.

Tabela 1 – Resultado em segundos do primeiro algoritmo em 3 mapas diferentes

	Mapa 1 (Pequeno)	Mapa 2 (Médio)	Mapa 3 (Grande)
Modelo Antigo	0,28 $\pm 0,00$	0,99 $\pm 0,06$	7,09 $\pm 0,81$
Modelo Otimizado	0,12 $\pm 0,00$	0,14 $\pm 0,00$	0,17 $\pm 0,00$
Speedup	2,28	6,95	41,36

$$Speedup = \frac{TempoAntigo}{TempoNovo}$$

Para ter uma melhor certeza, a Tabela 2 apresenta os resultados obtidos por um

algoritmo com maior quantidade de linhas de código. Este novo algoritmo, apresentou aproximadamente 400 linhas em CSP nos dois modelos utilizados.

Tabela 2 – Resultado em segundos do segundo algoritmo em 3 mapas diferentes

	Mapa 1 (Pequeno)	Mapa 2 (Médio)	Mapa 3 (Grande)
Modelo Antigo	0,54±0, 03	3,48±0, 32	27,79±1, 02
Modelo Otimizado	0,17±0, 00	0,22±0, 00	0,21±0, 00
Speedup	3,06	15,82	131,3

A partir destas análise, podemos observar uma redução significativa no tempo para análise dos modelos CSP obtidos a partir do tradutor desenvolvido nesta pesquisa. Modelos CSP obtidos a partir do novo tradutor apresentam um tempo de análise praticamente constante para os mapas analisados (nos dois algoritmos, considerando os diferentes mapas). Já o tempo de análise dos modelos antigos, antes deste trabalho, apresentam um crescimento exponencial com relação ao tamanho do mapa onde o programa é analisado.

6 Conclusão

A principal contribuição deste trabalho é uma extensão de uma abordagem atual de verificação automática eficiente de programas ROBO para considerar toda a gramatica da linguagem. A extensão consistiu em permitir que o tradutor aceite programas ROBO que usem comandos de pintura (como fazer pintura e detectar a cor pintada) e também comandos para capturar e movimentar o *beacon*. O tradutor anterior não lidava com estes comandos. Para isso, foi necessário reformular e estender as regras em Stratego utilizadas para realizar a tradução de programas ROBO de forma a considerar os novos elementos, mantendo a possibilidade de utilizar todos os outros elementos da linguagem. Além disso, foram atualizados na gramática elementos sintáticos para a declaração de variáveis e procedimentos e a chamada de procedimentos.

A corretude da tradução foi validada através de testes gerados por LGen. Já a eficiência dos modelos CSP gerados pelo tradutor foram analisados comparando o tempo de análise de *deadlock* com o modelo CSP gerado por este trabalho e o modelo CSP proposto em (NOGUEIRA et al., 2016). Analisando os tempos de verificação obtidos pelo modelo CSP gerado pelo tradutor deste trabalho, a verificação apresentou um tempo praticamente constante em relação ao crescimento do mapa, mostrando ser mais eficiente em relação ao tempo de analise de um programa ROBO do que o modelo proposto em (NOGUEIRA et al., 2016).

Uma das limitações do tradutor é que ele não realiza verificação semântica antes de traduzir o programa para CSP. Portanto os programas usados como entrada devem estar corretos sintaticamente e semanticamente, de forma a evitar erros na tradução. Outra limitação está ligada a tecnologia utilizada por FDR para analisar os modelos CSP; FDR analisa apenas modelos com um número finito de estados. Caso o programa em análise tenha uma quantidade infinita de estados, FDR vai expandir os estados do modelo de forma infinita, e não vai concluir a análise.

Entretanto, apesar das limitações, este é o primeiro tradutor do nosso conhecimento que traduz programas ROBO completos para a semântica de CSP, de forma a permitir análises automáticas dos programas usando ferramentas de verificação de modelos, como FDR.

6.1 Trabalhos Relacionados

Assim como é feito em RoboMind, no contexto educacional a simulação de robôs tem sido a principal forma de analisar o comportamento do robô de forma visual. Como exemplos

de ferramentas e plataformas que usam simulação, podemos citar Miranda (MIRANDA, 2020) e Robomind Academy (ACADEMY, 2015).

Miranda é um simulador de robôs virtuais, onde, a partir de programação em blocos (ou usando a linguagem Python), o usuário consegue criar um programa para um robô virtual e observar o seu comportamento através de uma simulação. O ambiente da simulação pode ser definido pelo usuário. A simulação pode ser feita considerando os diferentes tipos de robôs cadastrados na plataforma, que vão desde simples carros a drones.

Robomind Academy permite que o usuário crie um programa para um robô escrito na linguagem ROBO e observe o seu comportamento em mapas e desafios pré-cadastrados. A plataforma só disponibiliza um único robô virtual para o controle do usuário.

Nem Miranda nem Robomind Academy possuem a possibilidade de verificar de forma automática e não visual o comportamento do robô.

A abordagem de verificação proposta neste trabalho é muito similar a abordagem da ferramenta SVA (ROSCOE; HOPKINS, 2007), que é usada para verificar de forma automática programas desenvolvidos no contexto de ensino de programação concorrente. O SVA compila programas programas de variáveis compartilhadas escritas em uma linguagem imperativa para um modelo em CSP e usa o FDR para verificar as propriedades dos programas. A ferramenta LTSA (MAGEE; KRAMER, 1999) é outra ferramenta educacional que permite o ensino de programação concorrente e usa verificação de modelos. Todavia, enquanto nossa abordagem esconde a especificação formal do programador, no LTSA os programas são expressos diretamente em um cálculo de processos.

6.2 Trabalhos Futuros

Apesar de que o novo modelo é mais eficiente do que o proposto em (NOGUEIRA et al., 2016), não foi analisado se o CSP do novo modelo é equivalente ao CSP gerado anteriormente. Esta análise fica como um trabalho futuro.

A ideia de trabalho futuro é usar o tradutor que foi desenvolvido neste trabalho como parte de ferramentas educacionais. Este tradutor será usado como parte de um sistema web (OLIVEIRA et al., 2017) que recebe submissões de programas na linguagem ROBO, traduz para CSP e verifica se os programas convergem para um objetivo proposto usando FDR.

Até então, o tipo de análise utilizado neste trabalho é se o programa termina (entra em deadlock). Entretanto muitas outras propriedades podem ser verificadas, conforme descrito no paper (NOGUEIRA et al., 2016), onde é proposto o uso de propósitos de teste para especificar as propriedades dos programas. Propósitos de teste são formais, então para esconder a notação que descreve as propriedades esperadas para o programa, é preciso

usar uma linguagem que ajude a definir as propriedades esperadas para o programa. Como trabalho futuro, pretendemos usar a linguagem proposta em (OLIVEIRA et al., 2017) como entrada para definir as propriedades a serem analisadas.

A notação CSP é muito útil para a definição do mapeamento composicional de ROBO para CSP. Outra vantagem em potencial do uso de CSP é a possibilidade de usar os refinamentos de CSP para estabelecer equivalências entre programas ROBO e explorar refinamentos entre robôs. Esta possibilidade deixamos para explorar em trabalhos futuros.

Referências

ACADEMY, R. *Disponível em: https://www.robomindacademy.com/robomind/home*. 2015. Citado 2 vezes nas páginas 14 e 69.

ALIMISIS, D. Educational robotics: Open questions and new challenges. *Themes in Science and Technology Education*, v. 6, n. 1, p. 63–71, 2013. Citado na página 14.

BERS, M. U.; GONZÁLEZ-GONZÁLEZ, C.; ARMAS-TORRES, M. B. Coding as a playground: Promoting positive learning experiences in childhood classrooms. *Computers & Education*, Elsevier, v. 138, p. 130–145, 2019. Citado na página 14.

BRASIL, S. *Disponível em: https://br.spoj.com. Acesso em: 20 set. 2020*. 2020. Citado na página 15.

GIBSON-ROBINSON, T. et al. FDR3 17 A Modern Refinement Checker for CSP. In: *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.: s.n.], 2014. p. 187–201. Citado 2 vezes nas páginas 15 e 20.

HENTZ, C.; MOREIRA, A. M. *Geraçao de sentenças para testes a partir de descriç oes de linguagens*. [S.l.]: SAST, 2009. Citado na página 64.

KALLEBERG, K. T. Stratego. *Crossroads*, Association for Computing Machinery (ACM), v. 12, n. 3, p. 4–4, May 2006. ISSN 1528-4972. Disponível em: ⟨http://dx.doi.org/10.1145/1144366.1144370⟩. Citado 2 vezes nas páginas 15 e 21.

KATS, L. C.; VISSER, E. The spoofax language workbench. *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '10*, ACM Press, 2010. Disponível em: ⟨http://dx.doi.org/10.1145/1869542.1869592⟩. Citado 2 vezes nas páginas 15 e 20.

KITCHEN, R.; AMSTERDAM, U. V. RoboMind — http://robomind.net/. 03 2014. Disponível em: ⟨http://robomind.net/⟩. Citado 2 vezes nas páginas 14 e 18.

MAGEE, J.; KRAMER, J. *Concurrency: State Models &Amp; Java Programs*. New York, NY, USA: John Wiley & Sons, Inc., 1999. ISBN 0-471-98710-7. Citado na página 69.

MILLER, D. P.; NOURBAKHSH, I. Robotics for education. In: *Springer handbook of robotics*. [S.l.]: Springer, 2016. p. 2115–2134. Citado na página 14.

MIRANDA. *Disponivel em: https://www.miranda.software/*. 2020. Citado na página 69.

NOGUEIRA, S. et al. An approach for verifying educational robots. In: SPRINGER. *Brazilian Symposium on Formal Methods*. [S.l.], 2016. p. 59–77. Citado 2 vezes nas páginas 68 e 69.

OLIVEIRA, E. et al. Ambiente para avaliação automática de robôs virtuais: uma forma de apoio à aprendizagem de robótica. 2017. Citado 2 vezes nas páginas 69 e 70.

PEREIRA, I. L. *Uma Abordagem para Tradução de uma Linguagem de Programação de Robôs para um Modelo Formal*. 2018. Trabalho de Conclusão do Curso — Bacharel em Ciências da Computaçao — UFRPE. Citado 2 vezes nas páginas 16 e 66.

ROSCOE, A. W. *Understanding Concurrent System*. [S.l.]: Springer, 2011. Citado 2 vezes nas páginas 15 e 18.

ROSCOE, A. W.; HOPKINS, D. SVA, a tool for analysing shared-variable programs. In: *Proceedings of AVoCS 2007*. [S.l.: s.n.], 2007. p. 177–183. Citado na página 69.

THEHUXLEY. *Disponível em: https://www.thehuxley.com. Acesso em: 20 set. 2020*. 2020. Citado na página 15.

A Especificações CSP

Listing A.1 – Exemplo CSP

```
1  -- root: aux.csp
2  {-
3
4  ROBO CSP specification 16/03/2018
5  Sidney de Carvalho Nogueira (sidney.ufrpe@gmail.com)
6
7  Notes:
8  - RAW_MAP, PAINT and COMMANDS are assumed to be in context
9  - RAW_MAP is assumed to be a retangule
10 - there is no command to move the beacon
11 - paint is static
12 - procedures are not supported
13 - there are no commands to move the beacon
14
15 coordinate system is:
16
17 0,0 1,0 ... "X",0
18 0,1 1,1 ... "X",1
19 0,2 1,2 ... "X",2
20 ...
21 0,"Y" 1,"Y" ... "X","Y"
22
23 where "Y" is YMAX and "X" is XMAX
24
25 ERROR(m) is when "break" is outside of a loop
26 ERROR2(m) is when a variable is not declared and called to
      get a value
27
28 -}
29
30 --
31 -- map elements
32 --
33
34 MIN_MAX = 10
35
36 datatype Colors = White | Black
37
38 datatype DATA = B.Bool
39                | I.{(-MIN_MAX)..MIN_MAX}
40                | BCN.{(-MIN_MAX)..MIN_MAX}.{(-MIN_MAX)..
      MIN_MAX}
41                | COLOR.{(-MIN_MAX)..MIN_MAX}.{(-MIN_MAX)..
      MIN_MAX}.Colors
42                | PAINTS.Colors
43                | UNDEFINED
44 --datatype DATA = B.Bool | I.Int | BCN.Int.Int | UNDEFINED
45
46
47 datatype StrokeTypes = Dot | Hor | Ver
48
49 -- Obs represents obstacles as Tiles and Palms
50
```

```
51 datatype Things = Empty | Obs | Beacon | Start | Paint.Colors
52
53 -- map building
54 --
55
56 -- inputs a sequence that represents the map (RAW_MAP)
57 -- outputs a sequence of tuples of the type (x,y,element)
58 -- x (column number) starts at 0
59 -- y (line number) starts at 0
60
61 map2Seq(raw) =
62   let
63     map2Seq_(<>,  y) =  <>
64     map2Seq_(<line>^tail,  y) = processLine(line,y) ^
      map2Seq_(tail,y+1)
65   within
66     map2Seq_(raw, 0)
67
68 processLine(line, y) =
69   let
70     processLine_(<>, _, _) = <>
71     processLine_(<Empty>^tail, x, y) = processLine_(tail, x
      +1, y)
72     processLine_(<element>^tail, x, y) = <(x, y, element)> ^
      processLine_(tail, x+1, y)
73   within
74     processLine_(line, 0, y)
75
76 map2Set(map) =
77   let
78     map2Set_(<>) = {}
79     map2Set_(<(x,y,element)>^tail) = union({(x,y,element)},
      map2Set_(tail))
80   within
81     map2Set_(map)
82
83 -- inputs a paint sequence
84 -- outputs a set of map coordinates that match the paint
85
86 formatPaint(<>) = {}
87 formatPaint(<(c, t, x, y)>^tail) =
88   let
89     format(c,t,x,y) =
90       if(t == Dot) then
91         {(x,y,Paint.c)}
92       else if (t == Hor) then
93         {(x,y,Paint.c), (x+1,y,Paint.c)}
94       else -- Ver
95         {(x,y,Paint.c), (x,y+1,Paint.c)}
96   within
97     union(format(c,t,x,y), formatPaint(tail))
98
99 -- validates whether the input map is a retangule
100 isRetangule(map) = let
101     numberOfCells(<>) = 0
102     numberOfCells(<line>^tail) = #line + numberOfCells(tail)
103   within
104     (numberOfCells(map) / #map ) == #head(map)
105
106 IS_RETANGULE = isRetangule(RAW_MAP)
```

```
107
108 -- initialising the map
109 MAP = union(formatPaint(PAINT) , map2Set(map2Seq(RAW_MAP)) )
110
111 --
112 -- variables and memory
113 --
114
115 -- map limits
116 COLUMNS = length(head(RAW_MAP))
117 LINES = length(RAW_MAP)
118 XMAX = COLUMNS-1
119 YMAX = LINES-1
120
121 -- orientation
122 NORTH_ = 0
123 EAST_ = 1
124 SOUTH_ = 2
125 WEST_ = 3
126
127 -- variable types
128 {-
129 P = debug!INIT -> STOP
130 -}
131
132 -- channel debug : Set({(var,val) | var <- VARS, val <- DATA
    })
133
134 -- memory initial bindings
135 single({x}) = head(x)
136
137 startX = single({ <x> | (x,y,Start) <- MAP })
138 startY = single({ <y> | (x,y,Start) <- MAP })
139
140 -- Para fins de depura  o ,    preciso pelo menos 1 beacon
    na memoria
141 INITIAL_BEACONS =
142   let
143     BEACONS = {BCN.x.y | (x,y,Beacon) <- MAP }
144   within
145     if(BEACONS == {}) then (
146       <BCN.-1.-1>
147     ) else (
148       seq(union({BCN.10.10},BEACONS))
149     )
150
151 INITIAL_PAINTS =
152   let
153     paint = {COLOR.x.y.color | (x, y, Paint.color) <-
     formatPaint(PAINT)}
154   within
155     if(paint == {}) then (
156       <>
157     ) else (
158       seq(paint)
159     )
160
161 PREDEFINED_VARS = {"X","Y","ORIENTATION","BEACONS", "RETURN",
    "CARRYING_BEACON", "IS_PAITING", "PAINTS", "PAINT_COLOR"}
162
```

```
163 VARS = union(PREDEFINED_VARS,USER_VARS)
164
165 INIT = {
166          ("X", <I.startX>),
167          ("Y", <I.startY>),
168          ("ORIENTATION", <I.NORTH_>),
169          ("BEACONS", INITIAL_BEACONS),
170          ("CARRYING_BEACON", <B.false>),
171          ("IS_PAITING", <B.false>),
172          ("PAINT_COLOR", <PAINTS.White>),
173          ("PAINTS", INITIAL_PAINTS),
174          ("RETURN", <I.0>)
175        }
176 -- reading and updating memory
177
178 get(m,var) =
179           let
180             check = {val | (v,val) <- m, v == var}
181           within
182             if (check != {}) then
183               single(check)
184             else
185               UNDEFINED
186
187 getAllBeacons(m) = content({val | (v,val) <- m, v == "BEACONS
      "})
188 getAllPaints(m) = content({val | (v,val) <- m, v == "PAINTS
      "})
189 PaintColor(PAINTS.color) = color
190 getPaintColor(m) = PaintColor(head(content({val | (v, val) <-
       m, v == "PAINT_COLOR"})))
191
192 setVar(m,var,val) = union({(v,val) | (v,val) <- m, v != var},
       {( v,<val>^tail ) | (v,<head>^tail) <- m, v == var})
193
194 --
195 -- auxiliary functions
196 --
197
198 -- things within a given position
199 thingsAt(col,lin, bcns) =  union({ Obs | (c,l,t) <- MAP, c==
      col, l==lin , t==Obs},
200                                  { Beacon | (BCN.c.l) <- bcns
      , c==col, l==lin})
201
202 -- calculates the maximum number of movies to NORTH
203 max2NORTH(c, l, bcns) = let
204     obs = { t | t <- thingsAt(c,l-1, bcns), t == Obs }
205     beaconFree = frontIsBeacon(c, l, NORTH_, bcns)
206   within
207     if( l-1 >= 0 and card(obs) == 0 and beaconFree ) then
208       1 + max2NORTH(c, l-1, bcns)
209     else
210       0
211
212 -- calculates the maximum number of movies to EAST
213 max2EAST(c, l, bcns) = let
214     obs = { t | t <- thingsAt(c+1,l,bcns), t == Obs }
215     beaconFree = frontIsBeacon(c, l, EAST_, bcns)
216   within
```

```
217     if( c+1 <= XMAX and card(obs) == 0 and beaconFree ) then
218       1 + max2EAST(c+1, l, bcns)
219     else
220       0
221
222 -- calculates the maximum number of movies to SOUTH
223 max2SOUTH(c, l, bcns) = let
224     obs = { t | t <- thingsAt(c,l+1,bcns), t == Obs }
225     beaconFree = frontIsBeacon(c, l, SOUTH_, bcns)
226   within
227     if( l+1 <= YMAX and card(obs) == 0 and beaconFree ) then
228       1 + max2SOUTH(c, l+1, bcns)
229     else
230       0
231
232 -- calculates the maximum number of movies to WEST
233 max2WEST(c, l, bcns) = let
234     obs = { t | t <- thingsAt(c-1,l,bcns), t == Obs }
235     beaconFree = frontIsBeacon(c, l, WEST_, bcns)
236   within
237     if( c-1 >= 0 and card(obs) == 0 and beaconFree ) then
238       1 + max2WEST(c-1, l, bcns)
239     else
240       0
241
242 --
243 -- moving commands
244 --
245
246 -- auxiliary for movement commands
247
248 channel updatedX : {0..XMAX}
249 channel updatedY : {0..YMAX}
250 channel updatedORIENTATION : {0..3}
251
252 channel backward, forward : {1}
253
254 MOVE_STEPS(0,m,next) = next(m)
255 MOVE_STEPS(n,m,next) =
256   let
257     x = toInt(get(m,"X"))
258     y = toInt(get(m,"Y"))
259     o = toInt(get(m,"ORIENTATION"))
260     bcns = set(getAllBeacons(m))
261     m' = {(var_, head(value)) | (var_, value) <- m}
262     x_ = if(o == EAST_) then
263           x+1
264         else if(o == WEST_) then
265           x-1
266         else
267           x
268     y_ = if(o == NORTH_) then
269           y-1
270         else if(o == SOUTH_) then
271           y+1
272         else
273           y
274     isPaiting = toBool(get(m, "IS_PAITING"))
275     paints_ = { COLOR.c.l.color | COLOR.c.l.color <- set(
    getAllPaints(m)), x_==c and l==y_}
```

```
276     paints = diff(set(getAllPaints(m)), paints_)
277     paint_color = getPaintColor(m)
278     paint = if(frontIsClear(x,y,o,bcns) and isPaiting) then
279               {COLOR.x_.y_.paint_color}
280             else
281               paints_
282     others = {(var_, value) | (var_, value) <- m, var_ != "
    PAINTS" }
283     m_ = union(others, {("PAINTS", seq(union(paints, paint)))
    })
284
285   within
286     --debug!m' ->
287     if(frontIsClear(x,y,o,bcns)) then(
288       if(o == NORTH_) then (
289         forward!1 -> updatedY!(y-1) -> MOVE_STEPS(n-1, setVar
    (m_,"Y",I.(y-1)), next)
290       ) else if(o == EAST_) then (
291         forward!1 -> updatedX!(x+1) -> MOVE_STEPS(n-1, setVar
    (m_,"X",I.(x+1)), next)
292       ) else if(o == SOUTH_) then (
293         forward!1 -> updatedY!(y+1) -> MOVE_STEPS(n-1, setVar
    (m_,"Y",I.(y+1)), next)
294       ) else if(o == WEST_) then (
295         forward!1 -> updatedX!(x-1) -> MOVE_STEPS(n-1, setVar
    (m_,"X",I.(x-1)), next)
296       ) else
297         error("Unexpected state")
298     ) else
299         forward!1 -> MOVE_STEPS(0, m_, next)
300
301 MOVE_STEPS_BACK(0,m,next) = next(m)
302 MOVE_STEPS_BACK(n,m,next) =
303   let
304     x = toInt(get(m,"X"))
305     y = toInt(get(m,"Y"))
306     o = toInt(get(m,"ORIENTATION"))
307     bcns = set(getAllBeacons(m))
308     m' = {(var_, head(value)) | (var_, value) <- m}
309     x_ = if(o == EAST_) then
310           x-1
311         else if(o == WEST_) then
312           x+1
313         else
314           x
315     y_ = if(o == NORTH_) then
316           y+1
317         else if(o == SOUTH_) then
318           y-1
319         else
320           y
321     isPaiting = toBool(get(m, "IS_PAITING"))
322     paints_ = { COLOR.c.l.color | COLOR.c.l.color <- set(
    getAllPaints(m)), x_==c and l==y_}
323     paints = diff(set(getAllPaints(m)), paints_)
324     paint_color = getPaintColor(m)
325     paint = if(frontIsClear(x,y,(o+2)%4,bcns) and isPaiting)
    then
326               {COLOR.x_.y_.paint_color}
327             else
```

```
328                 paints_
329       others = {(var_, value) | (var_, value) <- m, var_ != "
          PAINTS" }
330       m_ = union(others, {("PAINTS", seq(union(paints, paint)))
          })
331     within
332       --debug!m' ->
333       if(frontIsClear(x,y,(o+2)%4,bcns)) then(
334         if(o == NORTH_) then (
335             backward!1 -> updatedY!(y+1) -> MOVE_STEPS_BACK(n-1,
          setVar(m_,"Y",I.(y+1)), next)
336         ) else if(o == EAST_) then (
337             backward!1 -> updatedX!(x-1) -> MOVE_STEPS_BACK(n-1,
          setVar(m_,"X",I.(x-1)), next)
338         ) else if(o == SOUTH_) then (
339             backward!1 -> updatedY!(y-1) -> MOVE_STEPS_BACK(n-1,
          setVar(m_,"Y", I.(y-1)), next)
340         ) else if(o == WEST_) then (
341             backward!1 -> updatedX!(x+1) -> MOVE_STEPS_BACK(n-1,
          setVar(m_,"X",I.(x+1)), next)
342         ) else
343           error("Unexpected state")
344       ) else
345         backward!1 -> MOVE_STEPS_BACK(0, m_, next)
346
347 -- moving commands
348 channel right, left, north, south, east, west
349
350 FORWARD(n, m, next) = MOVE_STEPS(n,m,next)
351 BACKWARD(n, m, next) = MOVE_STEPS_BACK(n,m,next)
352
353 RIGHT(m,next) =
354   let
355     o = toInt(get(m,"ORIENTATION"))
356     m' = {(var_, head(value)) | (var_, value) <- m}
357   within
358     right ->
359     --debug.m' ->
360     updatedORIENTATION!((o+1)%4) -> next(setVar(m,"
          ORIENTATION", I.((o+1)%4)))
361
362 LEFT(m,next) =
363   let
364     o = toInt(get(m,"ORIENTATION"))
365   within
366     left -> updatedORIENTATION!((o-1)%4) -> next(setVar(m,"
          ORIENTATION", I.((o-1)%4)))
367
368 NORTH(n,m,next) =
369   let
370     m_ = setVar(m,"ORIENTATION",I.NORTH_)
371   within
372     north -> updatedORIENTATION!NORTH_ -> FORWARD(n,m_,next)
373
374 SOUTH(n,m,next) =
375   let
376     m_ = setVar(m,"ORIENTATION",I.SOUTH_)
377   within
378     south -> updatedORIENTATION!SOUTH_ -> FORWARD(n,m_,next)
379
```

```
380 EAST(n,m,next) =
381   let
382     m_ = setVar(m,"ORIENTATION",I.EAST_)
383   within
384     east -> updatedORIENTATION!EAST_ -> FORWARD(n,m_,next)
385
386 WEST(n,m,next) =
387   let
388     m_ = setVar(m,"ORIENTATION",I.WEST_)
389   within
390     west -> updatedORIENTATION!WEST_ -> FORWARD(n,m_,next)
391
392 --
393 -- see functions
394 --
395
396 -- looks for obstacles
397 thingsInFront(x,y,o,bcns) =
398   if(o == NORTH_) then
399     thingsAt(x,y-1, bcns)
400   else if (o == EAST_) then
401     thingsAt(x+1,y, bcns)
402   else if (o == SOUTH_) then
403     thingsAt(x,y+1, bcns)
404   else
405     thingsAt(x-1,y, bcns)
406
407 -- looks for beacon
408 beaconInFront(x,y,o,bcns) =
409   let
410     bcn_north = { true | BCN.c.l <- bcns, c==x, l==(y-1), o==
      NORTH_}
411     bcn_east = { true | BCN.c.l <- bcns, c==(x+1), l==y, o==
      EAST_}
412     bcn_south = { true | BCN.c.l <- bcns, c==x, l==(y+1), o==
      SOUTH_}
413     bcn_west = { true | BCN.c.l <- bcns, c==(x-1), l==y, o==
      WEST_}
414     within
415   if(bcn_north != {}) then
416     true
417   else if (bcn_east != {}) then
418     true
419   else if (bcn_south != {}) then
420     true
421   else if(bcn_west !={}) then
422     true
423   else
424     false
425
426
427 frontIsObstacle(x,y,o,bcns) = member(Obs,thingsInFront(x,y,o,
      bcns)) or
428                               member(Beacon,thingsInFront(x,y,o,
      bcns))
429
430 frontIsBeacon(x,y,o,bcns) = beaconInFront(x,y,o,bcns)
431
432 frontIsClear(x,y,o,bcns) = not(frontIsObstacle(x,y,o,bcns))
      and
```

```
433                                 not(frontIsBeacon(x,y,o,bcns))
434
435 leftIsObstacle(x,y,o,bcns) = member(Obs,thingsInFront(x,y,(o
    -1)%4,bcns)) or
436                             member(Beacon,thingsInFront(x,y,(o-1)
    %4,bcns))
437
438 leftIsBeacon(x,y,o,bcns) = beaconInFront(x,y,(o-1)%4,bcns)
439
440 leftIsClear(x,y,o,bcns) = not(frontIsObstacle(x,y,(o-1)%4,
    bcns)) and
441                               not(frontIsBeacon(x,y,(o-1)%4,bcns
    ))
442
443 rightIsObstacle(x,y,o, bcns) = member(Obs,thingsInFront(x,y,(
    o+1)%4, bcns)) or
444                                member(Beacon,thingsInFront(x,y,(o
    +1)%4, bcns))
445
446 rightIsBeacon(x,y,o, bcns) = beaconInFront(x,y,(o+1)%4, bcns)
447
448 rightIsClear(x,y,o, bcns) = not(frontIsObstacle(x,y,(o+1)%4,
    bcns)) and
449                               not(frontIsBeacon(x,y,(o+1)%4,
    bcns))
450
451 ASSIGN(var, val, m, next) =
452                               let
453                                 check = {(var_, value) | (var_,
     value) <- m, var == var_}
454                                 m_ = union(m,{(var, <val>)})
455                                 others' = {(var_, head(value))
    | (var_, value) <- m, var != var_  }
456                                 m' = union(others', {(var, val)
    })
457                               within
458                                 if(check == {}) then
459                                   -- debug.m' ->
460                                   next(m_)
461                                 else
462                                   UPDATE(var, val, m, next)
463
464 UPDATE(var, val, m, next) =
465                               let
466                                 check = {value | (var_, value)
    <- m, var == var_}
467                                 others = {(var_, value) | (var_
    , value) <- m, var != var_ }
468                                 m_ = union(others, {(var, <val
    >^tail(content(check)))})
469                                 others' = {(var_, head(value))
    | (var_, value) <- m, var != var_  }
470                                 m' = union(others', {(var, val)
    })
471                               within
472                                 if(check != {}) then
473                                   -- debug.m' ->
474                                   next(m_)
475                                 else
476                                   ERROR2(m)
```

```
477
478 RETURN(var, func, m, next) =
479                         let
480                           ASSIGN_VALUE(m) = ASSIGN(var, get(m,
    "RETURN"), m, UPDATE_VALUE)
481                           UPDATE_VALUE(m) = UPDATE("RETURN", I
    .0, m, next)
482                         within
483                           func(m, ASSIGN_VALUE)
484
485 content({}) = <>
486 content({x}) = x
487
488 PUSH(var, value, m, next) =
489                         let
490                           check = {value | (var_, value) <- m,
    var == var_}
491                           others = {(var_, value) | (var_,
    value) <- m, var != var_ }
492                           m_ = union(others, {(var, <value>^
    content(check))})
493                         within
494                           next(m_)
495
496 POP(var, m, next) =
497                       let
498                         others = {(var_, value) | (var_, value)
     <- m, var != var_ }
499                         check = single({<value> | (var_, value)
     <- m, var_ == var})
500                         m_ = union(others, {(var, tail(check))
    })
501                         m' = {(var, head(val)) | (var, val) <-
    m_}
502                       within
503                         if(#check > 1) then
504                           -- debug.m' ->
505                           next(m_)
506                         else
507                           next(others)
508
509 PICKUP_BEACON(x, y, m,next) =
510                                 let
511                                   bcns = set(getAllBeacons(m))
512                                   bcn = { BCN.c.l | BCN.c.l <-
    bcns, c == x and l == y}
513                                   bcns_ = seq(diff(bcns, bcn))
514                                   others = {(var, value) | (var
    , value) <- m, var != "BEACONS", var != "CARRYING_BEACON"}
515                                   m_ = union(others, {("BEACONS
    ", bcns_),("CARRYING_BEACON", <B.true>)})
516                                 within
517                                   next(m_)
518 PICKUP(m, next) =
519                     let
520                       x = toInt(get(m,"X"))
521                       y = toInt(get(m,"Y"))
522                       o = toInt(get(m,"ORIENTATION"))
523                       bcns = set(getAllBeacons(m))
```

```
524                     carryingbeacon = toBool(get(m, "
    CARRYING_BEACON"))
525                   within
526                     if(frontIsBeacon(x, y, o, bcns) and not(
    carryingbeacon)) then (
527                       if(o == NORTH_) then (
528                         PICKUP_BEACON(x, y-1, m, next)
529                       ) else if(o == EAST_) then (
530                         PICKUP_BEACON(x+1, y, m, next)
531                       ) else if(o == SOUTH_) then (
532                         PICKUP_BEACON(x, y+1, m, next)
533                       ) else (
534                         PICKUP_BEACON(x-1, y, m, next)
535                       )
536                     )else(
537                       next(m)
538                     )
539
540 EATUP_BEACON(x, y, m, next) =
541                                 let
542                                   bcns = set(getAllBeacons(m))
543                                   bcn = { BCN.c.l | BCN.c.l <-
    bcns, c == x and l == y}
544                                   bcns_ = seq(diff(bcns, bcn))
545                                   others = {(var, value) | (var
    , value) <- m, var != "BEACONS"}
546                                   m_ = union(others, {("BEACONS
    ", bcns_)})
547                                 within
548                                   next(m_)
549 EATUP(m, next) =
550                   let
551                     x = toInt(get(m,"X"))
552                     y = toInt(get(m,"Y"))
553                     o = toInt(get(m,"ORIENTATION"))
554                     bcns = set(getAllBeacons(m))
555                   within
556                   if(frontIsBeacon(x, y, o, bcns)) then (
557                     if(o == NORTH_) then (
558                       EATUP_BEACON(x, y-1, m, next)
559                     ) else if(o == EAST_) then (
560                       EATUP_BEACON(x+1, y, m, next)
561                     ) else if(o == SOUTH_) then (
562                       EATUP_BEACON(x, y+1, m, next)
563                     ) else (
564                       EATUP_BEACON(x-1, y, m, next)
565                     )
566                   )else(
567                     next(m)
568                   )
569
570 PUTDOWN(m, next) =
571                     let
572                       x = toInt(get(m,"X"))
573                       y = toInt(get(m,"Y"))
574                       o = toInt(get(m,"ORIENTATION"))
575                       bcns = set(getAllBeacons(m))
576                       carryingbeacon = toBool(get(m, "
    CARRYING_BEACON"))
```

```
577                       others = {(var, value) | (var, value) <-
    m, var != "BEACONS", var != "CARRYING_BEACON"}
578                       m_ = union(others, {("CARRYING_BEACON", <
    B.false>)})
579                   within
580                     if(carryingbeacon and not(frontIsObstacle
    (x, y, o, bcns))) then (
581                       if(o == NORTH_) then (
582                         next(union(m_, {("BEACONS", seq(union
    (bcns, {BCN.x.(y-1)})))}))
583                       ) else if(o == EAST_) then (
584                         next(union(m_, {("BEACONS", seq(union
    (bcns, {BCN.(x+1).y})))}))
585                       ) else if(o == SOUTH_) then (
586                         next(union(m_, {("BEACONS", seq(union
    (bcns, {BCN.x.(y+1)})))}))
587                       ) else (
588                         next(union(m_, {("BEACONS", seq(union
    (bcns, {BCN.(x-1).y})))}))
589                       )
590                     ) else (
591                       next(m)
592                     )
593
594 PAINTWHITE(m, next) =
595   let
596     others = {(var, val) | (var, val) <- m, var != "
    IS_PAITING", var != "PAINTS", var != "PAINT_COLOR"}
597     x = toInt(get(m, "X"))
598     y = toInt(get(m, "Y"))
599     paints_ = { COLOR.c.l.color | COLOR.c.l.color <- set(
    getAllPaints(m)), x==c and l==y}
600     paints = seq(diff(set(getAllPaints(m)), paints_))
601     m_ = union(others, {("IS_PAITING", <B.true>),("PAINTS",<
    COLOR.x.y.White>^paints),("PAINT_COLOR", <PAINTS.White>)})
602   within
603     next(m_)
604
605 PAINTBLACK(m, next) =
606   let
607     x = toInt(get(m,"X"))
608     y = toInt(get(m,"Y"))
609     others = {(var, val) | (var, val) <- m, var != "
    IS_PAITING", var != "PAINTS", var != "PAINT_COLOR"}
610     paints_ = { COLOR.c.l.color | COLOR.c.l.color <- set(
    getAllPaints(m)), x==c and l==y}
611     paints = seq(diff(set(getAllPaints(m)), paints_))
612     m_ = union(others, {("IS_PAITING", <B.true>),("PAINTS",<
    COLOR.x.y.Black>^paints),("PAINT_COLOR", <PAINTS.Black>)})
613   within
614     next(m_)
615
616 STOPPAINTING(m, next) =
617   let
618     others = {(var, val) | (var, val) <- m, var != "
    IS_PAITING"}
619     m_ = union(others, {("IS_PAITING", <B.false>)})
620   within
621     next(m_)
622
```

```
623
624
625 ERROR2(m) = END(m) {-Variavel fora de escopo-}
626
627 toBool(B.b) = b
628 toBool(I.i) = (i != 0)
629
630 toInt(B.true) = 1
631 toInt(B.false) = 0
632 toInt(I.i) = i
633
634 --
635 -- program snipets
636 --
637
638
639 channel coin : Bool
640
641 END(m) = STOP
642
643 TERMINATE(m) =
644                 let
645                   m' = {(var_, head(value)) | (var_, value) <-
    m}
646                 within
647                 --debug.m' ->
648                 SKIP
```

Fonte – O autor