Gabriela Moreira Mafra

Tradução automática de especificação formal modelada em TLA+ para linguagem de programação

Joinville

2019

UNIVERSIDADE DO ESTADO DE SANTA CATARINA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Gabriela Moreira Mafra

TRADUÇÃO AUTOMÁTICA DE ESPECIFICAÇÃO FORMAL MODELADA EM TLA+ PARA LINGUAGEM DE PROGRAMAÇÃO

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Cristiano Damiani Vasconcellos
Orientador

Karina Girardi Rôggia
Co-Orientador

Joinville, Março de 2019

TRADUÇÃO AUTOMÁTICA DE ESPECIFICAÇÃO FORMAL MODELADA EM TLA+ PARA LINGUAGEM DE PROGRAMAÇÃO

Gabriela Moreira Mafra

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Ciência da Computação Integral do CCT/UDESC.

Banca Examinadora

Cristiano Damiani Vasconcellos - Doutor
(orientador)

Adelaine Gelain - Mestre

Paulo Torrens - Mestre

Agradecimentos

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Resumo

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Palavras-chaves: Especificação de software, Lógica temporal, Geração de código, Métodos formais, Model checking

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Keywords: Software specification, Temporal Logic, Code Generation, Formal Methods, Model Checking

Contents

List of Figures

List of Tables

Lista de Abreviaturas

1 Introdução

Desde a década de 60, com os trabalhos de Floyd e Hoare, são feitos trabalhos com propostas para especificar software formalmente. Com especificações, o grau de confiança na correção do programa aumenta, e se torna possível provar formalmente algumas propriedades, com base na semântica da especificação.

Esses trabalhos, contudo, são direcionados a programas sequenciais. Especificar um software concorrente necessita uma modelagem diferente, e não era possível até os primeiros trabalhos de Leslie Lamport na década de 90.

Os métodos de especificação mais bem sucedidos são baseados em modelar transformações de estados com alguma lógica formal. Pensando em sistemas concorrentes, Lamport propõe uma lógica que estende os termos básicos da lógica temporal para permitir predicados sobre pares de estados, o que ele chama de ações. Essa abstração permite manipular ações e não o sistema temporal puro. Essa lógica é chamada de TLA - *Temporal Logic of Actions*.

Sistemas concorrentes são aqueles onde mais de uma computação acontece no mesmo intervalo de tempo - concorrentemente - podendo ou não interagir entre si. Na lógica temporal, os passos executados por todas essas computações concorrentes são descritos como um comportamento, e definidos por uma sequência infinita de estados. Assim, uma fórmula da lógica pode ser verdadeira ou falsa para um comportamento, assim como pode ser válida ou não para todos os comportamentos possíveis.

Com essa abordagem, é possível verificar propriedades sobre um sistema especificado. Especificar um sistema significa definir todos os seus comportamentos possíveis. Tratando-se de um sistema concorrente, é esperado que existam muitos comportamentos, e listá-los exaustivamente seria uma tarefa extremamente passível de erro. Para viabilizar a definição dos comportamentos, é empregada uma modelagem semelhante a de uma máquina de estados, onde é definida a fórmula para o estado inicial e as fórmulas para as transições.

Baseando-se na lógica definida como TLA, Lamport propõe a linguagem de especificação formal TLA^+ (*Temporal Logic of Actions*), com o objetivo de escrever provas formais para sistemas concorrentes da maneira mais simples possível (LAMPORT, 2008). Nessa linguagem, além dos operadores de TLA, são incluídos elementos da teoria de conjuntos e alguns açúcares sintáticos para fórmulas temporais como cláusulas IF e CASE.

No viés de permitir verificações de propriedades, surge o *model checker* TLC. Um *model checker* busca todos os estados atingíveis de um modelo, de forma que todos os comportamentos possíveis são verificados. O TLC recebe uma especificação e uma configuração, e verifica se as fórmulas temporais dadas são válidas para a especificação. Se nenhuma fórmula temporal for dada, o TLC checará a presença de erros na semântica de TLA$^+$ e de situações de *deadlock*. A checagem de *deadlock* pode ser desativada, já que pode significar terminação em alguns sistemas.

Mais recentemente, outra ferramenta para verificar propriedades de uma especificação está em desenvolvimento : o sistema de provas TLAPS (*TLA Proof System*) (CHAUDHURI et al., 2010). Esse sistema permite checar mecanicamente algumas provas, semelhantemente a Coq e Isabelle, mas ainda está incompleto.

A partir das definições de propriedades desejadas e da possibilidade de verificá-las, se torna possível alterar uma especificação no sentido de buscar por otimizações ou propostas diferentes para o sistema e, através das verificações, encontrar potenciais problemas como *bugs* e inconsistências com as propriedades exigidas. Esses benefícios foram reportados pela *Amazon Web Services* (NEWCOMBE et al., 2015), que afirma ter usado TLA+ em 10 sistemas complexos e, para cada um deles, ter encontrado *bugs* ou adquirido entendimento e confiança para implementar otimizações agressivas.

As especificações escritas, contudo, não possuem nenhum vínculo com a implementação se não pelo entendimento do programador que as escreveu. Outras linguagens de especificação formal com objetivos semelhantes ao TLA+, como Z, B-Method e ASM, fornecem formas de gerar código a partir do modelo. Contudo, até a data da escrita desse texto, não foram encontrados geradores de código a partir de modelos escritos em TLA+, impossibilitando a conversão das especificações em linguagens de programação com garantia de correspondência.

1.1 Objetivos

Esse trabalho é feito com a intenção de elaborar um método de tradução, através do mapeamento de estruturas e construtores, de especificações formais descritas em TLA+ para código em linguagem de programação com possibilidade de ser executado e modificado; assim como implementar um tradutor que aplique esse método.

1.1.1 Objetivos Específicos

- Encontrar mapeamentos entre as estruturas de especificação em TLA+ e estruturas de linguagens de programação
- Implementar um gerador de código Elixir, com capacidade de fazer *parsing* de especificações em TLA+ e aplicar os mapeamentos necessários.

2 TLA$^+$

TLA$^+$ é uma linguagem de especificação de software, criada por Leslie Lamport (LAMPORT, 2008) voltada à modelagem de sistemas concorrentes. Ela se propõe a oferecer uma maneira mais simples de escrever um algoritmo, ao utilizar um nível de abstração acima do que há ao escrever código em uma linguagem de programação. Assim, ao programar, não é necessário atentar-se a detalhes de implementação, permitindo o foco no comportamento do algoritmo - e não das suas dependências.

As especificações são descritas em fórmulas lógicas, com pequenas adaptações de sintaxe. Para facilitar a curva de aprendizado para engenheiros, foi criada a linguagem PlusCal (LAMPORT, 2009), com uma sintaxe semelhante a linguagens de programação imperativas, e que traduz seus programas para TLA$^+$. A linguagem PlusCal não permite especificar sistemas tão complexos quanto os que podem ser escritos diretamente em TLA$^+$, mas, devido à tradução para a linguagem original, aproveita completamente as capacidades dela de verificação de propriedades.

O método de especificação é baseado em máquinas de estados (LAMPORT, 2008) e, sendo assim, a descrição de um modelo é composta por uma condição inicial, que determina os possíveis estados inciais, e por uma relação de transições, que determina os possíveis estados que podem suceder cada estado em uma execução. Dessa forma, o conjunto de comportamentos especificado é composto por todos os comportamentos cujo estado inicial satisfaz a condição inicial e todas as transições fazem parte relação.

Lamport destaca (LAMPORT, 2015) que as especificações deveriam ser sobre modelos de uma abstração do sistema, e não algo retirado do próprio sistema. Semelhante à planta de um edifício, a especificação pode ser consultada para obter informações sobre o edifício (ou programa) de forma mais conveniente, além de ser capaz de facilitar uma série de verificações e perceber problemas enquanto a mudança ainda não é inviavelmente custosa.

Sendo assim, uma especificação em TLA$^+$ pode ser sobre comportamentos do ambiente no qual o programa funciona - como ao especificar um sistema e verificar possíveis comportamentos indesejáveis, entendendo aonde o programa deve atuar - descrevendo as operações existentes daquele sistema.

Não limitada a definição de um sistema, uma especificação pode incluir comportamentos do programa em si, compostas por operações existentes do sistema e novas operações definidas pelo programa. Em seu livro (LAMPORT, 2002), Lamport define um sistema de memória linear e, então, propõe uma implementação de um programa de escrita através de *cache* que atua sobre um sistema de memória linear. Assim, ele verifica que a especificação da implementação dele satisfaz a especificação do sistema e prova a implementação. Nos exemplos deste capítulo, serão explicadas especificações de sistemas e de implementações.

2.1 Lógica Temporal

TLA$^+$ combina a Lógica Temporal das Ações, TLA (*Temporal Logic of Actions*), proposta por Lamport em (LAMPORT, 1994), com teoria dos conjuntos - mais especificamente, a teoria de conjuntos de Zermelo-Fraenkel (ZFC), como detalhado em (MERZ, 2003).

Lamport sumariza em (CHAUDHURI et al., 2008) o uso de TLA em TLA$^+$. TLA é uma lógica temporal linear. Em TLA$^+$, as variáveis rígidas do TLA são chamadas constantes, enquanto as flexíveis são chamadas variáveis. As expressões podem ser construídas em lógica clássica de primeira ordem, e são denominadas operadores constantes. Uma função de estado é uma expressão construída de operadores constantes, constantes e variáveis. Por fim, as fórmulas não temporais construídas a partir de operadores constantes, constantes, variáveis e expressões sobre uma função de estado são denominadas ações - correspondendo a *actions* em TLA.

Com essa estrutura, toda a complexidade das definições estão nas fórmulas de ações. Os operadores temporais são usados somente no momento de verificar propriedades de segurança, vivacidade e razoabilidade (*fairness*).

Uma fórmula em TLA é verdadeira ou falsa em um comportamento, que é definido por uma sequência infinita de estados. Uma fórmula é dita válida se e somente se ela é verdadeira para todos os comportamentos. Uma especificação F implementa outra especificação G se e somente se qualquer sistema que satisfaz F também satisfaz G, ou seja, a fórmula $G \implies F$ é válida.

Aos operadores de TLA, são atribuídos os seguintes significados (LAMPORT, 1994):

- ENABLED $\mathcal{A}$ para uma ação $\mathcal{A}$ é um predicado que é verdadeiro para um estado se e somente se é possível fazer um passo $\mathcal{A}$ partindo daquele estado.

- Um predicado P deve ser verdadeiro para os valores das variáveis definidas no estado inicial de um comportamento para ser satisfeito por ele.

- A fórmula $\Box[\mathcal{A}]_f$ para uma ação $\mathcal{A}$ é satisfeita por um comportamento se e somente se cada passo do comportamento satisfaz $\mathcal{A}$ ou mantém os valores de f.

- $\Box F$ (F é sempre verdadeiro) para uma fórmula TLA F é satisfeito por um comportamento se e somente se F é verdadeiro para todos os sufixos do comportamento.

- $\boldsymbol{\exists}\, x : F$ para uma variável x e uma fórmula TLA F é satisfeito por comportamento se e somente se existem alguns valores a serem atribuído a x que produzem um comportamento que satisfaz F. Esse operador é uma especialização do quantificador existencial comum $\exists$ porque ele asserte a existência de uma sequência infinita de valores para x, e não um único valor.

- $WF_f(\mathcal{A})$ – Razoabilidade fraca (*Weak fairness*) de $\mathcal{A}$ – para uma função de estado f e uma ação $\mathcal{A}$ é satisfeita por um comportamento se e somente se $\mathcal{A} \wedge (f' \neq f)$ é infinitamente não ativo ou infinitos passos $\mathcal{A} \wedge (f' \neq f)$ ocorrem.

- $SF_f(\mathcal{A})$ – Razoabilidade forte (*Strong fairness*) de $\mathcal{A}$ – para uma função de estado f e uma ação $\mathcal{A}$ é satisfeita por um comportamento se e somente se $\mathcal{A} \wedge (f' \neq f)$ ocorre finitas vezes ou infinitos passos $\mathcal{A} \wedge (f' \neq f)$ ocorrem.

- $F \stackrel{+}{\rightarrow} G$ para fórmulas temporais F e G é verdadeiro para um comportamento se e somente se G é verdadeiro, pelo menos, enquanto F é.

- $\Diamond F$ (Eventualmente F) é definido como $\neg\Box\neg F$.

- $F \leadsto G$ (Em qualquer momento em que F for verdadeiro, G eventualmente será) é definido como $\Box(F \implies \Diamond G)$

Além deles, existem os operadores lógicos $\wedge$, $\vee$ e $\neg$ com seus significados padrões.

As fórmulas não temporais são chamadas também de fórmulas transicionais. Na notação de TLA$^+$, elas devem ser escritas entre colchetes e sucedidas por uma definição do conjunto de variáveis para o qual são permitidos passos balbuciantes (*stuttering steps*) - passos nos quais nenhuma variável dentro do conjunto tem seu valor alterado.

2.2 Exemplo 1 - Jarros de Água

Para exemplificar uma especificação de um sistema, é possível definir um problema combinatório simples como o dos jarros de água. Nesse problema, são fornecidos dois jarros inicialmente vazios, um com capacidade de 3 litros e outro com capacidade de 5 litros, assim como uma fonte inesgotável de água. Sendo assim, é possível despejar a água dos jarros no chão, transferir a água de um jarro ao outro ou encher um jarro com a fonte de água.

O objetivo do problema é ter exatamente 4 litros de água em um dos jarros. Isso é, dada uma máquina de estados, é necessário encontrar uma sequência de transições que leva a algum estado onde o jarro maior tem exatamente 4 litros de água. No entanto, para esse exemplo, deseja-se apenas especificar os comportamentos do sistema em si, e não de um possível programa que buscaria atingir esse objetivo.

Uma possível especificação em TLA$^+$ para esse sistema se encontra na Figura 2.1.

Entendendo essa especificação no modelo de máquina de estado, é possível observar que as variáveis (VARIABLES) são propriedades que variam nos estados, de forma que o conjunto com todas as combinações dos valores possíveis para cada uma das variáveis forma o conjunto de estados da máquina. Um estado desse sistema seria $jarro\_pequeno = 0, jarro\_grande = 1$. Na definição $Init$, é especificada uma fórmula que determina estados iniciais válidos - o que, nesse caso, é apenas o estado onde todas as variáveis tem valor 0.

As seis definições seguintes representam as transições através de fórmulas transicionais. Em cada uma delas, as variáveis com o símbolo de linha representam os valores no estado seguinte, e sempre precisam ser definidas. Na transição $EnchePequeno$, o valor de $jarro\_grande$ se mantém o mesmo entre os estados atual e seguinte, mas é necessário explicitar isso com $jarro\_grande' = jarro\_grande$. Essa necessidade vem da

Figure 2.1: Especificação do problema dos Jarros de Água

$$
\begin{array}{l}
\textsc{module}\ JarrosDeAgua \\
\textsc{extends}\ Integers \\[1ex]
\textsc{variables}\ jarro\_pequeno,\ jarro\_grande \\[1ex]
TypeOK \triangleq \begin{array}[t]{l} \land\ jarro\_pequeno \in 0\,..\,3 \\ \land\ jarro\_grande \ \ \in 0\,..\,5 \end{array} \\[1ex]
Init \triangleq \begin{array}[t]{l} \land\ jarro\_grande \ \ = 0 \\ \land\ jarro\_pequeno = 0 \end{array} \\[1ex]
EnchePequeno \triangleq \begin{array}[t]{l} \land\ jarro\_pequeno' = 3 \\ \land\ jarro\_grande' \ \ = jarro\_grande \end{array} \\[1ex]
EncheGrande \triangleq \begin{array}[t]{l} \land\ jarro\_grande' \ \ = 5 \\ \land\ jarro\_pequeno' = jarro\_pequeno \end{array} \\[1ex]
EsvaziaPequeno \triangleq \begin{array}[t]{l} \land\ jarro\_pequeno' = 0 \\ \land\ jarro\_grande' \ \ = jarro\_grande \end{array} \\[1ex]
EsvaziaGrande \triangleq \begin{array}[t]{l} \land\ jarro\_grande' \ \ = 0 \\ \land\ jarro\_pequeno' = jarro\_pequeno \end{array} \\[1ex]
PequenoParaGrande \triangleq \begin{array}[t]{l} \textsc{if}\ jarro\_grande + jarro\_pequeno \leq 5 \\ \quad \textsc{then}\ \begin{array}[t]{l} \land\ jarro\_grande' \ \ = jarro\_grande + jarro\_pequeno \\ \land\ jarro\_pequeno' = 0 \end{array} \\ \quad \textsc{else}\ \ \begin{array}[t]{l} \land\ jarro\_grande' \ \ = 5 \\ \land\ jarro\_pequeno' = jarro\_pequeno - (5 - jarro\_grande) \end{array} \end{array} \\[1ex]
GrandeParaPequeno \triangleq \begin{array}[t]{l} \textsc{if}\ jarro\_grande + jarro\_pequeno \leq 3 \\ \quad \textsc{then}\ \begin{array}[t]{l} \land\ jarro\_grande' \ \ = 0 \\ \land\ jarro\_pequeno' = jarro\_grande + jarro\_pequeno \end{array} \\ \quad \textsc{else}\ \ \begin{array}[t]{l} \land\ jarro\_grande' \ \ = jarro\_pequeno - (3 - jarro\_grande) \\ \land\ jarro\_pequeno' = 3 \end{array} \end{array} \\[1ex]
Next \triangleq \begin{array}[t]{l} \lor\ EnchePequeno \\ \lor\ EncheGrande \\ \lor\ EsvaziaPequeno \\ \lor\ EsvaziaGrande \\ \lor\ PequenoParaGrande \\ \lor\ GrandeParaPequeno \end{array}
\end{array}
$$

aproximação da sintaxe de TLA$^+$ com a matemática, onde não existe efeito colateral e, portanto, o valor da variável $jarro\_grande$ não propagaria de um estado para outro.

É possível, sintaticamente, utilizar a informação das variáveis do estado atual para definir o estado seguinte - não é necessário definir exaustivamente transições para todas as combinações de variáveis. Dessa forma, as fórmulas transicionais definidas representam transições para vários estados do sistema. Cada transição da especificação do problema dos jarros pode ser aplicada nos em qualquer um dos estados, isto é: $(jarro\_pequeno = 0, jarro\_grande = 0), (jarro\_pequeno = 0, jarro\_grande = 1), \ldots$.

No sentido de aproveitar informações do estado atual, é possível utilizar condicionais, como nas fórmulas *PequenoParaGrande* e *GrandeParaPequeno*. Com isso, é fácil definir transições diferentes para conjuntos de estados com propriedades diferentes. Na definição de *PequenoParaGrande*, os estados que atualmente possuem 5 litros ou menos de água nos jarros em total recebem uma transição para um estado onde o jarro pequeno está vazio. Já os estados que possuem mais de 5 litros de água recebem uma transição para um estado onde o jarro grande está cheio.

Ao fim dessa especificação, em $Next$, é definida a *next state function* (função de próximo estado), na qual são declaradas as fórmulas transicionais do sistema, incluindo qualquer composição dessas fórmulas que possa levar um estado a outro. No caso do problema dos jarros, apenas é definido que qualquer transição pode ser utilizada para obter um novo estado.

As definições $Init$ e $Next$ são buscadas pelo *model checker* TLC na construção da máquina de estados. É possível renomear essas definições, mas é preciso informar ao TLC os novos nomes para o estado inicial e a *next state function*. A especificação - chamada *Spec* - é descrita a partir dessas definições com a seguinte fórmula temporal:

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

Onde $vars$ é uma tupla contendo todas as variáveis. Com essa especificação, o sistema está definido. As operações permitidas e as variáveis relevantes foram descritas e, a partir do estado inicial, cada passo do sistema pode ser executado a partir de uma das seis diferentes fórmulas transicionais. Essas informações são suficientes para o TLC fazer verificações sobre o sistema, é apenas necessário definir tais verificações.

A definição $TypeOK$ na especificação apresentada pode ser utilizada para verificar os tipos desse sistema. Ela define que a variável $jarro\_pequeno$ é sempre um inteiro entre 0 e 3, e a variável $jarro\_grande$ é sempre um inteiro entre 0 e 5. Ou seja, $TypeOK$ será verdadeiro se e somente se os valores das variáveis estiverem de acordo com essas restrições. Isso não é uma verificação em si, e sim uma definição. Para que essa definição seja verificada em todos os estados alcançáveis pelo sistema, é necessário adicioná-la como uma invariante do modelo. Como uma invariante, o valor dela não deve ser modificado em nenhum estado da execução. Já que o estado inicial definido em $Init$ faz $TypeOK$ verdadeiro, ao colocar essa invariante, todos os estados devem fazer $TypeOK$ verdadeiro, ou o TLC retornará um erro. $TypeOk$ pode ser definido como uma invariante através do teorema:

$$\textsc{theorem}\ Spec \implies \Box(TypeOK)$$

Outra propriedade interessante de ser verificada para esse problema antes da implementação de um programa para resolvê-lo é a possibilidade de resolução, isto é, se é possível alcançar um estado onde onde o jarro maior contém 4 litros de água. Para isso, define-se uma invariante para o predicado $jarro\_grande \setminus = 4$, que não será satisfeita. Como esse predicado é verdadeiro para o estado inicial, o fato de ele não ser satisfeito significa que, em algum momento da execução, o predicado foi falso, ou seja, $jarro\_grande = 4$. Adicionando essa invariante, um possível teorema seria:

$$\textsc{theorem}\ Spec \implies \Box(TypeOK \land jarro\_grande \setminus = 4)$$

O TLC, ao encontrar uma execução que insatisfaz a invariante, traz a sequência de transições que levam ao estado onde o predicado é falso, o que, no caso do simples problema dos jarros, é a solução buscada.

Esse exemplo é apresentado com o intuito de demonstrar a estrutura da especificação de um sistema e o funcionamento das invariantes. A seguir, é proposto um exemplo com especificações de um sistema real e de um programa que atua nele.

2.3 Exemplo 2 - Transações em Bancos de Dados

Já em um contexto de um problema real de sistemas concorrentes, define-se uma especificação para o problema da consistência das transações em bancos de dados. Esse é um problema clássico onde, dado um conjunto de gerenciadores de recursos fazendo operações sobre um mesmo banco, um gerenciador só pode cometer (fazer a ação *commit*) se todos os outros estiverem preparados para cometer, e se algum gerenciador quiser abortar, então todos devem abortar.

Na Figura 2.2, encontra-se uma especificação para um sistema de transações consistente. Ela não apresenta uma proposta de solução para o problema, e sim traz uma descrição formal do que significa ser consistente. Uma especificação de uma solução para o problema deve implementar essa especificação.

Figure 2.2: Especificação do problema das Transações

$$\text{MODULE } \mathit{TransacoesBD}$$

$\text{CONSTANT } GR$

$\text{VARIABLE } estadoGR$

$$TBDTypeOK \triangleq estadoGR \in [GR \rightarrow \{\text{“trabalhando”}, \text{“preparado”}, \text{“cometido”}, \text{“abortado”}\}]$$

$$TBDInit \triangleq estadoGR = [g \in GR \mapsto \text{“trabalhando”}]$$

$$podeCometer \triangleq \forall g \in GR : estadoGR[g] \in \{\text{“preparado”}, \text{“cometido”}\}$$

$$naoCometido \triangleq \forall g \in GR : estadoGR[g] \neq \text{“cometido”}$$

$$\begin{aligned} Prepara(g) \triangleq\ & \land\ estadoGR[g] = \text{“trabalhando”} \\ & \land\ estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“preparado”}] \end{aligned}$$

$$\begin{aligned} Decide(g) \triangleq\ & \lor\ \land\ estadoGR[g] = \text{“preparado”} \\ & \quad\ \land\ podeCometer \\ & \quad\ \land\ estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“cometido”}] \\ & \lor\ \land\ estadoGR[g] \in \{\text{“trabalhando”}, \text{“preparado”}\} \\ & \quad\ \land\ naoCometido \\ & \quad\ \land\ estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“abortado”}] \end{aligned}$$

$$TBDNext \triangleq \exists g \in GR : Prepara(g) \lor Decide(g)$$

$$\begin{aligned} & TBDConsistente \triangleq \\ & \quad \forall r1, r2 \in GR : \neg \land\ estadoGR[r1] = \text{“abortado”} \\ & \quad\quad\quad\quad\quad\quad\quad\quad\ \land\ estadoGR[r2] = \text{“cometido”} \end{aligned}$$

$$TBDSpec \triangleq TBDInit \land \Box[TBDNext]_{estadoGR}$$

$$\text{THEOREM } TBDSpec \Rightarrow \Box(TBDTypeOK \land TBDConsistente)$$

3 O gerador de código

Em vista da relação das especificações em TLA+ e sistemas concorrentes, é interessante que a linguagem do código gerado seja capaz de suportar concorrência em um nível alto de abstração. Adicionalmente, devido à natureza matemática dessas especificações, espera-se minimizar a complexidade e a quantidade de mapeamentos ao traduzi-las para uma linguagem funcional. Ambos estes requistos se fazem necessários pela finalidade de proporcionar um código modificável, de forma que o programador seja capaz de entender a correspondência e minimizando a diferença do nível de abstração no qual ele está programando. Uma linguagem de programação que atende esses requisitos é Elixir.

Bibliography

CHAUDHURI, K. et al. A TLA+ proof system. In: *LPAR Workshops*. [S.l.]: CEUR-WS.org, 2008. (CEUR Workshop Proceedings, v. 418).

CHAUDHURI, K. et al. Verifying Safety Properties With the TLA+ Proof System. In: GIESL, J.; HAEHNLE, R. (Ed.). *Fifth International Joint Conference on Automated Reasoning - IJCAR 2010*. Edinburgh, United Kingdom: Springer, 2010. (Lecture Notes in Artificial Intelligence, v. 6173), p. 142–148. The original publication is available at www.springerlink.com. Disponível em: <https://hal.inria.fr/inria-00534821>.

LAMPORT, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, v. 16, n. 3, p. 872–923, 1994.

LAMPORT, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. Disponível em: <https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/>.

LAMPORT, L. The specification language tla+. In: HENSON, D. B. e M. C. (Ed.). *Logics of specification languages*. Berlin: Springer, 2008. p. 616–620. ISBN 3540741062. Disponível em: <http://lamport.azurewebsites.net/pubs/commentary-web.pdf>.

LAMPORT, L. The pluscal algorithm language. In: *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*. [s.n.], 2009. p. 36–60. Disponível em: <https://doi.org/10.1007/978-3-642-03466-4_2>.

LAMPORT, L. *The TLA Hyperbook*. 2015. Disponível em: <http://lamport.azurewebsites.net/tla/hyperbook.html>. Acesso em: 25 mai. 2019.

LEONARD, E. I.; HEITMEYER, C. L. Automatic program generation from formal specifications using apts. In: DANVY, O. et al. (Ed.). *Automatic Program Development: A Tribute to Robert Paige*. Dordrecht: Springer Netherlands, 2008. p. 93–113. ISBN 9781402065859. Disponível em: <https://doi.org/10.1007/978-1-4020-6585-9_10>.

MERZ, S. On the logic of tla+. *Computers and Artificial Intelligence*, v. 22, p. 351–379, 01 2003.

NAJAFI, M.; HAGHIGHI, H. A formal mapping from object-z specification to c++ code. *Scientia Iranica*, v. 20, p. 1953–1977, 12 2013.

NEWCOMBE, C. et al. How amazon web services uses formal methods. *Commun. ACM*, ACM, New York, NY, USA, v. 58, n. 4, p. 66–73, mar. 2015. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/2699417>.