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

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 matemáticas, 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 - de-

screvendo 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 a seguir, serão explicadas especificações de sistemas e de implementações.

2.1 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, 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 abaixo.

MODULE *JarrosDeAgua*

EXTENDS *Integers*

VARIABLES *jarro_pequeno*, *jarro_grande*

$$
\begin{aligned}
\mathit{TypeOK} \triangleq\ & \land\ \mathit{jarro\_pequeno} \in 0\,..\,3 \\
& \land\ \mathit{jarro\_grande}\ \in 0\,..\,5
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Init} \triangleq\ & \land\ \mathit{jarro\_grande}\ = 0 \\
& \land\ \mathit{jarro\_pequeno} = 0
\end{aligned}
$$

$$
\begin{aligned}
\mathit{EnchePequeno} \triangleq\ & \land\ \mathit{jarro\_pequeno}' = 3 \\
& \land\ \mathit{jarro\_grande}'\ = \mathit{jarro\_grande}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{EncheGrande} \triangleq\ & \land\ \mathit{jarro\_grande}'\ = 5 \\
& \land\ \mathit{jarro\_pequeno}' = \mathit{jarro\_pequeno}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{EsvaziaPequeno} \triangleq\ & \land\ \mathit{jarro\_pequeno}' = 0 \\
& \land\ \mathit{jarro\_grande}'\ = \mathit{jarro\_grande}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{EsvaziaGrande} \triangleq\ & \land\ \mathit{jarro\_grande}'\ = 0 \\
& \land\ \mathit{jarro\_pequeno}' = \mathit{jarro\_pequeno}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{PequenoParaGrande} \triangleq\ & \text{IF } \mathit{jarro\_grande} + \mathit{jarro\_pequeno} \leq 5 \\
& \text{THEN } \land\ \mathit{jarro\_grande}'\ = \mathit{jarro\_grande} + \mathit{jarro\_pequeno} \\
& \qquad\ \land\ \mathit{jarro\_pequeno}' = 0 \\
& \text{ELSE } \land\ \mathit{jarro\_grande}'\ = 5 \\
& \qquad\ \land\ \mathit{jarro\_pequeno}' = \mathit{jarro\_pequeno} - (5 - \mathit{jarro\_grande})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{GrandeParaPequeno} \triangleq\ & \text{IF } \mathit{jarro\_grande} + \mathit{jarro\_pequeno} \leq 3 \\
& \text{THEN } \land\ \mathit{jarro\_grande}'\ = 0 \\
& \qquad\ \land\ \mathit{jarro\_pequeno}' = \mathit{jarro\_grande} + \mathit{jarro\_pequeno} \\
& \text{ELSE } \land\ \mathit{jarro\_grande}'\ = \mathit{jarro\_pequeno} - (3 - \mathit{jarro\_grande}) \\
& \qquad\ \land\ \mathit{jarro\_pequeno}' = 3
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Next} \triangleq\ & \lor\ \mathit{EnchePequeno} \\
& \lor\ \mathit{EncheGrande} \\
& \lor\ \mathit{EsvaziaPequeno} \\
& \lor\ \mathit{EsvaziaGrande} \\
& \lor\ \mathit{PequenoParaGrande} \\
& \lor\ \mathit{GrandeParaPequeno}
\end{aligned}
$$

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 especificado um estado inicial do qual o sistema executa.

As seis definições seguintes representam as transições. 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 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 transições para todas as combinações de variáveis. Dessa forma, as transições definidas são genéricas e podem ser aplicadas a qualquer estado do sistema. Cada transição da especificação do problema dos jarros pode ser aplicada nos estados `(jarro_pequeno = 0, jarro_grande = 0), (jarro_pequeno = 0, jarro_grande = 1), ...`.

No sentido de aproveitar informações do estado atual, é possível utilizar condicionais, como nas transições `PequenoParaGrande` e `GrandeParaPequeno`. Com isso, é fácil definir transições generalizadas 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 transições do sistema, incluindo qualquer composição de transições 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 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 da lógica temporal:

$$Spec \triangleq Init \wedge \Box[Next]$$

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 transições. 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 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 os valores das variáveis estiverem de acordo com essas restrições, e falso caso contrário. 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. Como o estado inicial 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:

$$\text{THEOREM } Spec \implies \Box(TypeOK)$$

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:

$$\text{THEOREM } Spec \implies \Box(TypeOK \wedge jarro\_grande \setminus = 4)$$

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.

2.2 Exemplo 2 - Transações de um Banco de Dados

MODULE *TransacoesBD*

CONSTANT GR

VARIABLE $estadoGR$

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

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

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

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

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

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

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

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

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

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. 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. *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>.

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>.