

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

Gabriela Moreira Mafra

Universidade do Estado de Santa Catarina
gabrielamoreiramafra@gmail.com

24 de Junho de 2019



Fundamentos

TLA⁺

Gerador de código

Próximos passos

Especificação Formal

Especificar Software é como fazer a planta de um edifício

- Permite verificações
- Serve como base para consulta
- Mais fácil de modificar do que o produto final
- Vem antes da produção

Sistemas Concorrentes

Um sistema é concorrente quando há mais de uma computação concorrendo pelo mesmo recurso.

O resultado pode depender da ordem em que essas computações conseguem os recursos.

- Muitas ordens possíveis
- Muitos comportamentos possíveis

Especificar pode ser ainda mais relevante.

Geração de código

A implementação - ou a tradução da especificação em linguagem de programação - pode ser feita por um programador ou um tradutor automático.

Problemas da tradução manual:

- Sucetível a erro - causando a perda de propriedades verificadas.
- Custosa

Z, B-Method e ASM (*Abstract State Machine*) tem geradores de código.

Temporal Logic of Actions⁺

Linguagem de especificação baseada na lógica TLA.

- Sintaxe matemática
- Ideal para especificar sistemas concorrentes

Não é disponibilizado um gerador de código.

Temporal Logic of Actions⁺

Linguagem de especificação baseada na lógica TLA.

- Sintaxe matemática
- Ideal para especificar sistemas concorrentes

Não é disponibilizado um gerador de código.

Objetivo

Elaborar um método para gerar código a partir de especificações em TLA⁺

- Encontrar mapeamentos
- Implementar um tradutor, aplicando os mapeamentos e gerando código Elixir

TLA

TLA = Lógica. TLA⁺ = linguagem de especificação.

Ação

Fórmula sobre um passo. Passo = dupla de estados.

Permite definir quais transições são permitidas.

Comportamento

Sequência de passos. Representa uma execução no sistema.

Uma fórmula é verdadeira para um comportamento se ela é verdadeira para cada passo dele.

Exemplo: Problema dos Jarros de Água

- Jarro de 3 litros
- Jarro de 5 litros
- Fonte infinita de água
- Local para descarte

Como conseguir um jarro com exatos 4 litros de água?

Primeiro é definido o sistema e, só então, um programa que atua sobre ele.

Especificação: Sistema Jarros de Água (1/4)

EXTENDS *Integers*

VARIABLES *jarro_pequeno*, *jarro_grande*

$$\textit{TypeOK} \triangleq \wedge \textit{jarro_pequeno} \in 0 \dots 3 \\ \wedge \textit{jarro_grande} \in 0 \dots 5$$
$$\textit{Init} \triangleq \wedge \textit{jarro_grande} = 0 \\ \wedge \textit{jarro_pequeno} = 0$$

Especificação: Sistema Jarros de Água (2/4)

$$\begin{aligned} \textit{EnchePequeno} &\triangleq \wedge \textit{jarro_pequeno}' = 3 \\ &\quad \wedge \textit{jarro_grande}' = \textit{jarro_grande} \end{aligned}$$

$$\begin{aligned} \textit{EncheGrande} &\triangleq \wedge \textit{jarro_grande}' = 5 \\ &\quad \wedge \textit{jarro_pequeno}' = \textit{jarro_pequeno} \end{aligned}$$

$$\begin{aligned} \textit{EsvaziaPequeno} &\triangleq \wedge \textit{jarro_pequeno}' = 0 \\ &\quad \wedge \textit{jarro_grande}' = \textit{jarro_grande} \end{aligned}$$

$$\begin{aligned} \textit{EsvaziaGrande} &\triangleq \wedge \textit{jarro_grande}' = 0 \\ &\quad \wedge \textit{jarro_pequeno}' = \textit{jarro_pequeno} \end{aligned}$$

Especificação: Sistema Jarros de Água (3/4)

$PequenoParaGrande \triangleq$ IF $jarro_grande + jarro_pequeno \leq 5$
THEN $\wedge jarro_grande' = jarro_grande + jarro_pequeno$
 $\wedge jarro_pequeno' = 0$
ELSE $\wedge jarro_grande' = 5$
 $\wedge jarro_pequeno' = jarro_pequeno - (5 - jarro_grande)$

$GrandeParaPequeno \triangleq$ IF $jarro_grande + jarro_pequeno \leq 3$
THEN $\wedge jarro_grande' = 0$
 $\wedge jarro_pequeno' = jarro_grande + jarro_pequeno$
ELSE $\wedge jarro_grande' = jarro_pequeno - (3 - jarro_grande)$
 $\wedge jarro_pequeno' = 3$

Especificação: Sistema Jarros de Água (4/4)

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

Com essas fórmulas, a especificação do sistema é definida por

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

Propriedades

TypeOK é um predicado para a consistência dos tipos. Um predicado pode ser definido como uma invariante do sistema com o teorema:

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

É possível verificar que o sistema permite, a partir da condição inicial definida, que o estado desejado seja alcançado:

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

Tradução

O método de tradução é descrito através de **mapeamentos**.
Alguns mapeamentos já definidos são:

- Ações \mapsto Funções
- Estado \mapsto Hash mapeando as variáveis aos seus valores
- Ação \vee Ação \mapsto Disparo de novo processo
- ...

Exemplo: Tradução de uma ação

$$\begin{aligned} \text{EsvaziaPequeno} &\triangleq \wedge \text{jarro_pequeno}' = 0 \\ &\quad \wedge \text{jarro_grande}' = \text{jarro_grande} \end{aligned}$$

```
def esvazia_pequeno(variaveis) do
  %{
    jarro_pequeno: 0,
    jarro_grande:  variaveis[:jarro_grande]
  }
end
```


Tradução do sistema

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

```
def main(variaveis) do
  spawn_link JarrosDeAgua, :main, [grande_para_pequeno(variaveis)]
  spawn_link JarrosDeAgua, :main, [pequeno_para_grande(variaveis)]
  spawn_link JarrosDeAgua, :main, [esvazia_grande(variaveis)]
  spawn_link JarrosDeAgua, :main, [esvazia_pequeno(variaveis)]
  spawn_link JarrosDeAgua, :main, [enche_grande(variaveis)]
  spawn_link JarrosDeAgua, :main, [enche_pequeno(variaveis)]
  spawn_link JarrosDeAgua, :main, [variaveis]
end

JarrosDeAgua.main(%{ grande: 0, pequeno: 0 })
```

Por que Elixir?

- 1 Concorrência: gera *bytecode* da máquina virtual do Erlang (BEAM).
- 2 Paradigma funcional: se aproxima de definições matemáticas, proporcionando uma complexidade menor para as traduções.
- 3 Expressividade: código entendível, permitindo alta manutenibilidade.
- 4 Transparência de plataforma
- 5 Open Source

Considerações

Até aqui:

- Estudo dos construtores de TLA⁺ e entendimento da lógica.
- Validação da estrutura inicial do código traduzido.
- Início da listagem de mapeamentos.

Próximos passos

A continuidade do trabalho será feita em duas frentes paralelas:

- Busca por mais mapeamentos
- Implementação do tradutor em Haskell

Próximos passos

A continuidade do trabalho será feita em duas frentes paralelas:

- Busca por mais mapeamentos
- Implementação do tradutor em Haskell

Exploração

- Fornecimento de garantias para os mapeamentos
- Gerar complementos para melhorar o ambiente de desenvolvimento (e.g. testes unitários)

Obrigada!

Fim :D

Gabriela Moreira Mafra

`gabrielamoreiramafra@gmail.com`