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

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*

$$TypeOK \triangleq \land jarro\_pequeno \in 0 \,..\, 3 \\ \land jarro\_grande \in 0 \,..\, 5$$

$$Init \triangleq \land jarro\_grande = 0 \\ \land jarro\_pequeno = 0$$

Especificação: Sistema Jarros de Água (2/4)

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

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

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

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

Especificação: Sistema Jarros de Água (3/4)

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

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

Especificação: Sistema Jarros de Água (4/4)

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

Com essas fórmulas, a especificação do sistema é definida por

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

Propriedades

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

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

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

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

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

$$EsvaziaPequeno \triangleq \land jarro\_pequeno' = 0 \\ \land jarro\_grande' = jarro\_grande$$

```
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 um complexidade menor para as traduções.
3. Expressividade: código entendível, permitindo alta manutenabilidade.
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