Resumo

Especificar software traz garantias de segurança para o sistema e facilita modificações e otimizações sobre o programa especificado, o que é especialmente interessante para sistemas concorrentes ou distribuídos, onde há muitos comportamentos a se considerar. Especificações para esses sistemas podem ser escritas com a linguagem de especificação TLA$^+$, que é composta por definições próximas à matemática e ferramentas que permitem a verificação de propriedades descritas em lógica temporal. Este trabalho propõe um tradutor automático de especificações em TLA$^+$ para a linguagem de programação funcional e concorrente Elixir, permitindo a execução do sistema especificado.

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

Abstract

Specifying software provides safety guarantees for the system and facilitates modifications and optimizations on the specified program, which is specially interesting for concurrent or distributed systems, where there are many behaviors to be considered. Specifications for this systems can be written in the specification language TLA$^+$, which is composed by definitions close to mathematics and tools that allow verification of temporal logic described properties. This work proposes an automatic translator of TLA$^+$ specifications to the functional and concurrent programming language Elixir, allowing the execution of the specified system.

Contents

List of Figures **5**

List of Tables **7**

1 Introdução **8**

1.1 Objetivos 10

1.1.1 Objetivos Específicos 11

2 TLA$^+$ **12**

2.1 Lógica Temporal das Ações 13

2.1.1 Passos balbuciantes 17

2.2 Propriedades 17

2.2.1 Propriedades de Segurança 18

2.2.2 Propriedades de Vivacidade 18

2.3 Exemplo 1 - Jarros de Água 19

2.4 Exemplo 2 - Transações em Bancos de Dados 23

2.4.1 O sistema 23

2.4.2 O protocolo 25

3 O gerador de código **31**

3.1 Elixir 31

3.2 A tradução 32

3.2.1 Mapeamentos 33

3.2.2 Condições e Ações 36

3.2.3 Determinando a próxima ação 38

3.2.4 Concorrência 39
3.3 Regras de Tradução 40
3.3.1 Tradução de especificações 42
3.3.2 Tradução das definições 43
3.3.3 Estado Inicial 49
3.3.4 Função de próximo estado 50
3.4 Aplicação da tradução 52
3.4.1 Jarros de Água 52
3.4.2 Protocolo de efetivação em duas fases 56

4 Considerações **63**
4.1 Cronograma 64

Bibliography **66**

List of Figures

2.1 Sintaxe da linguagem de TLA 14

2.2 Especificação do problema dos Jarros de Água 20

2.3 Especificação de um sistema de transações em bancos de dados 24

2.4 Especificação do protocolo de efetivação em duas fases - Parte 1 29

2.5 Especificação do protocolo de efetivação em duas fases - Parte 2 30

3.1 Fórmula transicional *EsvaziaPequeno* como uma função em Elixir 33

3.2 Fórmula transicional *PequenoParaGrande* como uma função em Elixir . . 34

3.3 Disparo de processos para o sistema de Jarros de Água 34

3.4 Exploração de invariantes no código gerado 35

3.5 Ação para o recebimendo da mensagem "Efetive" por um gerenciador de recurso 37

3.6 Código gerado para a ação $GRRecebeMsgEfetive(g)$ 37

3.7 Sintaxe aceita para TLA$^+$ 41

3.8 Elementos gerados em Elixir 41

3.9 Tradução da especificação 42

3.10 Tradução de declarações de constantes e definições 43

3.11 Tradução intermediária de definições - Geração 44

3.12 Tradução intermediária de definições - Agregação 46

3.13 Organização das informações sobre uma ação 47

3.14 Tradução de predicados e ações 48

3.15 Tradução de valores 49

3.16 Tradução do Estado Incial de da função de próximo estado 51

3.17 Tradução do módulo Jarros de Água 52
3.18 Tradução das definições simples dos Jarros de Água 53
3.19 Tradução das definições com condicionais dos Jarros de Água 54
3.20 Tradução da função de próximo estado para os Jarros de Água 55
3.21 Tradução do módulo Efetivação em Duas Fases 57
3.22 Tradução da definição $GTEfetiva$ do protocolo 58
3.23 Tradução da definição $GRRecebeMsgEfetive$ do protocolo 59
3.24 Tradução da função de próximo estado para o protocolo 60

List of Tables

4.1 Cronograma Proposto 65

4.2 Cronograma Atualizado 65

1 Introdução

Desde a década de 60, com os trabalhos de Floyd e Hoare, são feitas as primeiras propostas de 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.

Para especificar sistemas concorrentes, contudo, são necessários formalismos diferentes daqueles direcionados a programas sequenciais. Nessa área, surgem as primeiras ideias com a proposta das redes de Petri, por Carl Adam Petri em (PETRI, 1962). Mais tarde, Milner, Parrow e Walker propõem o cálculo pi em (MILNER; PARROW; WALKER, 1992). Lamport apresenta suas primeiras ideias para um modelo de especificação em (LAMPORT, 1983), e apresenta a versão completa de TLA$^+$ em seu livro (LAMPORT, 2002).

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. Tal abstração permite manipular ações e não o sistema temporal puro. Essa lógica é chamada de TLA (*Temporal Logic of Actions* - Lógica Temporal das Ações).

Sistemas concorrentes são aqueles onde mais de uma computação acontece no mesmo intervalo de tempo e concorrendo por recursos - o que é comum em sistemas distribuídos. 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 são incluídos, além dos operadores de TLA, 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 intuito 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 formais escritas, contudo, não possuem nenhum vínculo com a implementação em uma linguagem de programação. O elo que correlaciona as duas partes é limitado ao entendimento do programador que as escreveu. Outras linguagens de especificação formal com objetivos semelhantes ao TLA$^+$, como Z, B-Method e ASM (*Abstract State Machine* - Máquina de Estado Abstrata), 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.

Observações sobre os benefícios da geração de código a partir de modelos de especificação formal já foram verificadas em trabalhos como o estudo de caso em (LEONARD; HEITMEYER, 2008). Práticas da engenharia de software vem tentado buscar maneiras de minimizar a geração de *bugs* por erro humano. Técnicas de revisão de código, programação em pares e examinadores automáticos de código são formas de detectar erros e quebra de certas propriedades. Práticas como essa são precedidas de uma fase de desenho de solução, onde podem ser feitos documentos e protótipos antes de uma implementação em linguagem de programação. Atualmente, a fase de desenho poderia ser feita formalmente, descrevendo a solução em TLA$^+$. Já a minimização de erros não teria benefícios, uma vez que a tradução do desenho para o código ainda seria feita por um humano e estaria sujeita a erros.

A motivação para automatizar esse processo é mitigar erros humanos na tradução de um desenho formalmente verificado para um código em linguagem de programação. Com o programa especificado, validado e traduzido, é possível aplicá-lo diretamente em casos reais com uma garantia de correspondência maior, assegurando as propriedades verificadas. A partir deste código gerado, ainda são permitidas modificações, como para melhorar a implementação em busca de uma versão mais otimizada. Essa nova versão estará partindo de uma base verificada e recebendo possivelmente novos comportamentos que não foram verificados - nesse caso, a garantia é reduzida, já que as mudanças não estavam representadas no modelo original.

A linguagem de programação escolhida para o código gerado é Elixir. Essa escolha se justifica pela proximidade do paradigma funcional, que a linguagem adota, com as representações matemáticas de uma especificação em TLA$^+$; assim como seu suporte à concorrência, que condiz com a motivação de criação de TLA$^+$, inclinada a seu uso em sistemas concorrentes.

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 da 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 - des-

crevendo as operações existentes daquele sistema. Contudo, 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 das Ações

TLA$^+$ combina a lógica TLA, 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 constantes são declaradas com a palavra-chave CONSTANTS e tem o mesmo valor para todos os estados de um comportamento - podendo diferir entre comportamentos. Já variáveis são declaradas com a palavra-chave VARIABLES e podem ter valores diferentes em cada estado de um comportamento.

Os operadores são classificados em constantes e não constantes. Os constantes são aqueles que podem ser escritos em lógica clássica de primeira ordem. Os não constantes dependem de mais fatores, tal como o operador *primed* ('), que depende do valor de uma variável em um estado diferente do atual. As definições em TLA$^+$ podem ser categorizadas em tipos de expressão. São denominadas fórmulas todas as expressões com valoração booleana.

- **Expressões constantes** são expressões com apenas constantes declaradas e operadores constantes. Pela definição de operador constante, o valor de uma expressão constante depende apenas do valor das constantes contidas nela.
- **Expressões de estado** contém expressões constantes e variáveis declaradas. O valor de uma expressão de estado depende do estado, já que os valores das variáveis

são definidos em um estado. Quando não são fórmulas, ou seja, sua valoração não é booleana, são chamadas também de funções de estado.

- **Expressões de ação** contém expressões de estado e operadores não constantes. O seu valor depende de um passo - um par de estados. Esse tipo de definição sobre ações dá o nome *actions* a TLA, e pode ser chamado simplesmente de ação.

- **Expressões temporais** são permitidas apenas com valoração booleana em TLA$^+$, sendo assim, chamadas sempre de fórmulas temporais. Elas contém expressões de ação e operadores $\Box$ e $\Diamond$ da lógica temporal (definidos posteriormente neste capítulo). O valor de uma fórmula temporal depende de uma sequência de passos - um comportamento.

Com essa estrutura, define-se a sintaxe na Figura 2.1. A hierarquia permite que toda a complexidade das definições em uma especificação esteja nas fórmulas de ações, e os operadores temporais sejam usados somente no momento de verificar propriedades de segurança, vivacidade e razoabilidade (*fairness*).

Constantes	c	Variáveis		v
Estados	s, t	Funções de estado		f, g
Conjuntos	S			
Ação		$\mathcal{A}$	::=	$c \mid v \mid v' \mid \neg\mathcal{A} \mid \mathcal{A} \wedge \mathcal{A}$
Predicado		P, Q	::=	$c \mid v \mid \neg P \mid P \wedge P \mid \text{ENABLED } \mathcal{A}$
Fórmula Simples		F	::=	$P \mid \Box[\mathcal{A}]_f \mid \Box\langle\mathcal{A}\rangle_f \mid \neg F \mid F \wedge F \mid \Box F$
Fórmula Geral		G	::=	$F \mid \exists\, x : G \mid \neg G \mid G \wedge G \mid G \implies G$

Figura 2.1: Sintaxe da linguagem de TLA

Uma fórmula temporal 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. Assim, a especificação de um sistema é dada por uma fórmula geral G e representa um sistema cujo conjunto de comportamentos permitidos é igual ao conjunto de comportamentos que satisfazem G.

Implementação é representada através de implicação. Uma especificação dada pela fórmula G_1 implementa outra especificação dada por G_0 se e somente se qualquer sistema cujo conjunto de comportamentos satisfaz G_1 também satisfaz G_0, ou seja, a fórmula $G_0 \implies G_1$ é válida.

Em (LAMPORT, 1994), são definidas também notações auxiliares. A lista abaixo apresenta a definição e atribui um possível significado a possíveis operadores para fórmulas temporais:

- $\Box F$ (F é sempre verdadeiro) para uma fórmula temporal F é satisfeito por um comportamento se e somente se F é verdadeiro para todos os sufixos (primeiro estado em um passo) do comportamento.
- $\Diamond F$ (Eventualmente F) é definido como $\neg\Box\neg F$.
- $F_0 \rightsquigarrow F_1$ (Em qualquer momento em que F_0 for verdadeiro, F_1 eventualmente será) é definido como $\Box(F_0 \implies \Diamond F_1)$
- $\exists x : F$ para uma variável x e uma fórmula temporal F é satisfeito por comportamento se e somente se existem alguns valores a serem atribuídos 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.

Para ações, são definidos ainda outros operadores:

- f' (f *primed*) para uma função de estado f é o valor de f no final de um passo. Em outras palavras, para um passo composto pelo par ordenado de estados s e t, f' é o valor de f para t. De forma semelhante, P' para um predicado P é o valor de P para o estado final de um passo. Assim, na avaliação da valoração de uma ação para um passo, predicados e variáveis sem o operador *primed* se referem aos seus respectivos valores no primeiro estado do passo, e sempre que forem marcados com o operador, fazem referência aos valores no segundo estado do passo.
- A fórmula $\Box[\mathcal{A}]_f$ para uma ação $\mathcal{A}$ e uma função de estado f é satisfeita por um comportamento se e somente se cada passo do comportamento satisfaz $\mathcal{A}$ ou mantém o valor de f - ou seja, $\mathcal{A} \vee (f = f')$.
- A fórmula $\Box\langle\mathcal{A}\rangle_f$ para uma ação $\mathcal{A}$ e uma função de estado f é satisfeita por um comportamento se e somente se cada passo do comportamento satisfaz $\mathcal{A}$ e altera o valor de f - ou seja, $\mathcal{A} \wedge (f \neq f')$.

- ENABLED $\mathcal{A}$ ($\mathcal{A}$ é ativável) para uma ação $\mathcal{A}$ é um predicado cujo valor é verdadeiro para um estado s se e somente se é possível fazer um passo $\mathcal{A}$ partindo de s. Isto é, existe um estado t tal que o passo formado pelo par s e t satisfaz $\mathcal{A}$.

- UNCHANGED f (f não é modificado) para uma fórmula de estado f em um passo (par de estados) é definido como $f' = f$ (o valor de f no estado atual é igual ao valor de f no próximo estado).

TLA conta com muitos operadores constantes, trazidos da matemática, da lógica, da teoria de conjuntos e de linguagens de programação. Abaixo, são apresentados os considerados menos triviais, sendo que a lista completa é definida em (LAMPORT, 2002).

- CHOOSE $x \in S : P$ (escolha algum x pertencente ao conjunto S que satisfaça P) para uma variável x e um predicado P resulta em algum valor de x que satisfaz P se $\exists\, x : P$ for verdadeiro. Sobre CHOOSE , é possível afirmar que se $\exists\, x : P$ então $P(\text{CHOOSE } x \in S : P)$ é verdadeiro, e que para todo predicado Q tal que $Q \equiv P$, é verdade que $(\text{CHOOSE } x \in S : P) = (\text{CHOOSE } x \in S : Q)$.

- f EXCEPT $![v] = e$ para uma função f, um elemento de seu domínio v e uma expressão e resulta em uma cópia de f exceto pelo valor $f[v]$ que é igual a e.

- $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ é uma estrutura do tipo registro (*record*) onde h_i são campos e e_i são expressões constantes para os valores de h_i. O valor de um campo h qualquer em um registro r pode ser obtido por $r.h$, e o operador r EXCEPT $!h = e$ funciona de forma semelhante ao definido no item anterior, retornando uma cópia de r exceto pelo valor de $r.h$, que é e.

Finalmente, os operadores lógicos $\wedge$ e $\vee$ podem ser prefixados às expressões, sendo

$$\begin{array}{l} \wedge\ e_1 \\ \vdots \\ \wedge\ e_n \end{array} \equiv e_1 \wedge \cdots \wedge e_n \qquad e \qquad \begin{array}{l} \vee\ e_1 \\ \vdots \\ \vee\ e_n \end{array} \equiv e_1 \vee \cdots \vee e_n$$

2.1.1 Passos balbuciantes

Os passos balbuciantes (*stuttering steps*) são parte importante das especificações em TLA. Eles permitem que o estado - formado pelos valores das variáveis da especificação - se mantenha igual durante um passo.

Supondo que as variáveis da especificação estejam declaradas como

$$vars = \langle var_1, var_2, \ldots, var_n \rangle$$

Então é possível usar o operador $\Box[\mathcal{A}]_f$ definido, com $f = vars$, no seguinte teorema sobre uma especificação $Spec$

$$\text{THEOREM } Spec \implies \Box[\mathcal{A}]_{vars}$$

o que, se verificado, garante que cada passo de um comportamento satisfeito por $Spec$ satisfaz a ação $\mathcal{A}$ ou é um passo balbuciante e mantém os valores das variáveis em $vars$. Isso é importante porque possibilita que nem todos os passos do sistema sejam especificados - seria muito complexo definir tudo o que pode ocorrer durante sua execução. Assim, definem-se apenas os passos relevantes para o sistema, e todos os outros passos - aqueles que não alteram as variáveis definidas - são permitidos. Um comportamento que passa a ter infinitos passos balbuciantes pode significar uma execução do sistema que finalizou. Com a definição de passos balbuciantes, é possível definir as propriedades apresentadas na Seção 2.2.

2.2 Propriedades

Sobre uma especificação definida, TLA$^+$ permite a verificação de algumas propriedades de segurança e vivacidade. Essas propriedades são descritas em forma de teoremas na especificação apenas com o intuito de documentar sua verificação, porém devem ser inseridas manualmente no modelo TLC para serem, de fato, checadas.

Propriedades são fórmulas temporais sobre ações definidas na especificação. Uma propriedade é satisfeita se a fórmula temporal que a define é válida.

2.2.1 Propriedades de Segurança

Propriedades de segurança definem o que o sistema pode fazer. Quando uma propriedade de segurança é violada, ela é violada em um instante específico de um comportamento. Esse tipo de propriedade é definido em TLA$^+$ como uma invariante.

Uma invariante é um predicado P que é verdadeiro em todos os passos de todos os comportamentos permitidos por uma especificação *Spec*, e pode ser verificada com a prova do teorema

$$\text{THEOREM } Spec \implies \Box P$$

2.2.2 Propriedades de Vivacidade

Propriedades de vivacidade definem o que o sistema deve fazer. Quando uma propriedade de vivacidade é violada, ela é violada em um comportamento. Em (LAMPORT, 2002), é apresentada uma especificação para um relógio. O ponteiro de um relógio deve, eventualmente, mexer. Esse é um tipo de propriedade que pode ser descrita com uma propriedade de vivacidade, tal qual a razoabilidade fraca (*weak fairness*).

A razoabilidade fraca para uma fórmula de estado f e uma ação $\mathcal{A}$ é escrita como $WF_f(\mathcal{A})$. Ela é satisfeita por um comportamento se e somente se $\mathcal{A} \wedge (f' \neq f)$ é infinitamente não ativável (ENABLED) ou infinitos passos $\mathcal{A} \wedge (f' \neq f)$ ocorrem. Sendo assim, essa propriedade garante que $\mathcal{A}$ não possa permanecer continuamente ativável para sempre sem que um passo $\mathcal{A}$ ocorra. Essa condição pode ser escrita de forma equivalente como

$$\Box(\text{ENABLED } \mathcal{A} \implies \Diamond\langle\mathcal{A}\rangle_f)$$

A conjunção com $(f' \neq f)$, expressada com a notação $\langle\mathcal{A}\rangle_f$, se deve ao fato de não ser desejável exigir que passos balbuciantes eventualmente ocorram. $\mathcal{A} \wedge (f' \neq f)$ pode ser lido como "todos os passos não balbuciantes que satisfazem $\mathcal{A}$ ".

A razoabilidade fraca recebe a denominação "fraca" porque exige que uma ação permaneça continuamente ativável para garantir a ocorrência de um passo satisfazendo-a. Se um comportamento repetidamente tornar a ação ativável e em seguida não ativável,

a razoabilidade fraca não garante nada sobre a ocorrência da ação neste comportamento. Para tal, é necessário garantir a propriedade de razoabilidade forte (*strong fairness*).

A razoabilidade forte para uma fórmula de estado f e uma ação $\mathcal{A}$ é escrita como $SF_f(\mathcal{A})$. Ela 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. Essa propriedade garante que $\mathcal{A}$ não possa ser repetidamente ativável para sempre sem que um passo $\mathcal{A}$ ocorra. Uma forma equivalente de representar essa condição é

$$\Box\Diamond \text{ENABLED } \mathcal{A} \implies \Box\Diamond\langle\mathcal{A}\rangle_f$$

que pode ser lida como "se sempre $\mathcal{A}$ for eventualmente ativável, então sempre um passo $\langle\mathcal{A}\rangle_f$ deve eventualmente ocorrer".

2.3 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 2.2.

Entendendo essa especificação no modelo de máquina de estado, é possível observar que as variáveis (VARIABLES) são um conjunto de valores 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

MODULE *JarrosDeAgua*

EXTENDS *Integers*

VARIABLES *jarro_pequeno*, *jarro_grande*

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

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

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

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

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

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

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

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

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

Figura 2.2: Especificação do problema dos Jarros de Água

que determina estados iniciais válidos - o que, nesse caso, é apenas o estado onde todas as variáveis do sistema tem valor 0.

As seis definições seguintes representam as transições através de ações. Em cada uma delas, as variáveis com o operador *primed* (') 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 propaga 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 ações definidas representam transições para vários estados do sistema. Cada transição da especificação do problema dos jarros pode ser aplicada 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 ações $PequenoParaGrande$ 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 fórmula temporal

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

Onde $vars$ é uma tupla contendo todas as variáveis declaradas. 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 ações ou de passos balbuciantes sobre $vars$. 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 deve ser verdadeiro para todos os estados da execução. Assim, ao definir 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 a especificação de um sistema real que pode ser implementado por especificações de programas para verificar que as propriedades definidas são mantidas.

2.4 Exemplo 2 - Transações em Bancos de Dados

Já tratando de 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 efetivar (fazer a ação *commit*) se todos os outros gerenciadores estiverem preparados para efetivar; e se algum gerenciador quiser abortar, então todos devem abortar. Ou seja, em nenhum momento pode haver um gerenciador abortado e outro cometido.

2.4.1 O sistema

Na Figura 2.3, 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 quando se trata de transações. Uma especificação de uma solução para o problema deve implementar essa especificação.

Um gerenciador de recursos pode estar em quatro estados diferentes, como definido em $TBDTypeOK$. Do estado "trabalhando", ele pode ir para o estado "preparado", no sentido de que ele está pronto para efetivar; ou então abortar, indo para o estado "abortado". Se todos os gerenciadores estão no estado "preparado", então qualquer um deles pode efetivar, indo para o estado "cometido"; ou então abortar, indo para o estado "abortado". Contudo, se exite algum gerenciador no estado "cometido", então nenhum outro gerenciador pode abortar.

A possibilidade de um gerenciador g ir do estado "trabalhando" ao "preparado" é representada pela ação $Prepara(g)$ onde, se o estado de g é "trabalhando", então o valor da variável $estadoGR$ no novo estado é igual ao seu valor no estado atual, exceto pelo valor de $estadoGR[g]$, que passa a ser "preparado".

A decisão de um gerenciador de abortar ou efetivar é representada pela ação $Decide(g)$. Nessa definição, as fórmulas $podeEfetivar$ e $naoEfetivado$ são definidas separadamente para minimizar a complexidade cognitiva da especificação - definí-las dentro de $Decide(g)$ seria semanticamente equivalente. $podeEfetivar$ verifica se todos os estados estão preparados ou cometidos, ou seja, qualquer um pode efetivar. Se $podeEfetivar$ for verdadeiro, e g ainda não cometeu, então g comete - o estado dos gerenciadores passa

MODULE *TransacoesBD*

CONSTANT GR

VARIABLE $estadoGR$

$$
\begin{array}{l}
TBDTypeOK \triangleq \\
\quad estadoGR \in [GR \to \{\text{“trabalhando”}, \text{“preparado”}, \text{“efetivado”}, \text{“abortado”}\}] \\
\\
TBDInit \triangleq estadoGR = [g \in GR \mapsto \text{“trabalhando”}] \\
\\
podeEfetivar \triangleq \forall\, g \in GR : estadoGR[g] \in \{\text{“preparado”}, \text{“efetivado”}\} \\
\\
naoEfetivado \triangleq \forall\, g \in GR : estadoGR[g] \neq \text{“efetivado”} \\
\\
Prepara(g) \triangleq \land\, estadoGR[g] = \text{“trabalhando”} \\
\qquad\qquad\quad \land\, estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“preparado”}] \\
\\
Decide(g) \triangleq \lor \land\, estadoGR[g] = \text{“preparado”} \\
\qquad\qquad\qquad\quad \land\, podeEfetivar \\
\qquad\qquad\qquad\quad \land\, estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“efetivado”}] \\
\qquad\qquad\qquad \lor \land\, estadoGR[g] \in \{\text{“trabalhando”}, \text{“preparado”}\} \\
\qquad\qquad\qquad\quad \land\, naoEfetivado \\
\qquad\qquad\qquad\quad \land\, estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“abortado”}] \\
\\
TBDNext \triangleq \exists\, g \in GR : Prepara(g) \lor Decide(g)
\end{array}
$$

$$
\begin{array}{l}
TBDConsistente \triangleq \\
\quad \forall\, r1,\, r2 \in GR : \neg \land\, estadoGR[r1] = \text{“abortado”} \\
\qquad\qquad\qquad\qquad\quad \land\, estadoGR[r2] = \text{“efetivado”}
\end{array}
$$

$$
\begin{array}{l}
TBDSpec \triangleq TBDInit \land \Box[TBDNext]_{estadoGR} \\
\\
\text{THEOREM } TBDSpec \Rightarrow \Box(TBDTypeOK \land TBDConsistente)
\end{array}
$$

Figura 2.3: Especificação de um sistema de transações em bancos de dados

a ser uma cópia do estado atual exceto por $estadoGR[g]$, que é "cometido". Outra decisão possível, separada da primeira por um operador de disjunção, é a de abortar. Para isso, verifica-se, com a fórmula $naoEfetivado$, se não há nenhum gerenciador cometido. Se $naoEfetivado$ for verdadeira, e g ainda não tiver abortado, então o novo estado dos gerenciadores passa a ter g como "abortado".

Com essas fórmulas, é possível definir a *next state function* $TBDNext$, onde um passo do sistema é dado por um gerenciador de recursos no conjunto GR que faz uma ação de preparar ou decidir. A fórmula temporal $TBDSpec$ consiste a especificação do sistema de transações bancárias e tem um formato semelhante à fórmula $Spec$ do exemplo

anterior, na Seção 2.3.

Para verificar que o sistema especificado por $TBDSpec$ está de acordo com a restrição do problema - em nenhum momento pode haver um gerenciador abortado e outro cometido - define-se $TBDConsistente$ onde, para cada possível par de gerenciadores de recursos, não é o caso de o primeiro estar abortado e o segundo, cometido. Essa fórmula é uma afirmação sobre um valor da variável $estadoGR$, porém precisa ser verdadeira para todos os valores dessa variável em qualquer comportamento que satisfaça $TBDSpec$. Para isso, ela é definida como uma invariante através do teorema

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

que permite verificar que, se um comportamento satisfaz $TBDSpec$ - isto é, seu estado inicial satisfaz $TBDInit$ e seus passos satisfazem $TBDNext$ - então os predicados $TBDTypeOK$ e $TBDConsistente$ são verdadeiros para todas as estados - todos os valores atribuídos para as variáveis - neste comportamento. Sendo satisfeito esse teorema, $TBDTypeOK$ e $TBDConsistente$ são ambos invariantes da especificação.

2.4.2 O protocolo

A partir da definição de ações válidas em um sistema de transações, é possível especificar um protocolo que restringe as ações de forma a atender propriedades desejadas. A estratégia de definir um protocolo com esse objetivo está presente em diversos contextos, como ao abrir uma nova conta bancária. Ao especificar um protocolo em TLA$^+$, é possível verificar as propriedades que ele busca atender e garantir que está cumprindo seu propósito.

Para o sistema de transações em bancos de dados, Lamport define em (LAMPORT, 2017) o protocolo de efetivação em duas fases. Esse protocolo estabelece que os gerenciadores de recurso se comuniquem com um gerenciador de transações (GT), avisando quando estão preparados para efetivar uma transação. Quando todos os gerenciadores de recurso estão preparados ($GRsPreparados = GR$), o gerenciador de transações envia uma mensagem a todos os gerenciadores de recurso para que efetivem.

A Figura 2.4 apresenta a primeira parte da especificação desse protocolo, onde são definidas as ações para o gerenciador de transações. A ação $GTRecebePrepara$ acontece quando uma mensagem "EstouPreparado" é recebida e colocao gerenciador de recurso g que enviou a mensagem no conjunto de gerenciadores preparados $GRsPreparados$. Quando todos os gerenciadores estão preparados, a ação $GTEfetiva$ se torna ativável, podendo enviar o comando "Efetive" para os gerenciadores.

Adicionalmente, o gerenciador de transações pode abortar espontaneamente. Essa característica é definida por $GTAborta$, que pode ocorrer a qualquer momento em que o gerenciador esteja ativo, isto é, $estadoGT = "inicio"$.

A parte 2 da especificação, na Figura 2.5, contém as definições de ações para os gerenciadores de recurso. Elas são semelhantes àquelas da especificação do sistema de transações, na Figura 2.2, exceto que nessa especificação o $estadoGR$ só alterado se uma mesagem de abortar ou efetivar for recebida, e não diretamente após o gerenciador escolher abortar ou efetivar.

$DFNext$ que qualquer ação para qualquer gerenciador de recurso pode acontecer a qualquer momento, e $DFSpec$ obedece o mesmo formato das especificações apresentadas até então. O teorema THEOREM $DFSpec \implies \Box DFTypeOK$ define o predicado sobre tipos comoo uma invariante, e com as declarações

INSTANCE $TransacoesBD$

THEOREM $DFSpec \implies TBDSpec$

as definições da especificação do sistema são importadas e, ao verificar que $DFSpec \implies TBDSpec$, garante-se que as ações do protocolo respeitam as propriedades exigidas por $TBDSpec$ - a especificação so sistema. Em outras palavras, $DFSpec$ implementa $TBDSpec$.

Troca de mensagens

O conjunto $msgs$ dessa especificação contém todas as mensagens que em algum momento foram enviadas, uma vez que em nenhum momento uma mensagem é retirada do conjunto. Isso significa que uma mesagem pode ser recebida qualquer número de vezes e, no contexto dessa especificação, isso não é um problema, já que a ação executada ao receber mensagens

pode ser feita uma ou infinitas vezes sem alterar o resultado.

Apesar de modelar a troca de mensagens da maneira mais simples possível fazer todo o sentido para uma especificação, fazer uma implementação com esse tipo de estrutura não seria viável, já que as ações são executadas por processos e os recursos são finitos. O protocolo de efetivação em duas fases não é um algoritmo a ser implementado, e isso permite que extrapolações como essas sejam presentes em sua especificação. O protocolo não determina como fazer a troca de mensagens - isso não faz parte da sua responsabilidade - e uma implementação desse protocolo deve definir uma forma de fazê-lo que seja viável e que satisfaça o protocolo.

Escolhas

$DFNext$ define que qualquer ação para qualquer gerenciador de recurso é válida como um próximo passo, ou seja, dado um passo de um estado s para outro estado t, qualquer ação satisfeita pela dupla $< s, t >$ pode ocorrer como próximo passo.

A influência dos valores de s e t na satisfação de ações, contudo, pode variar. Dado um momento onde o estado s possui o valor de $msgs$ igual a $\{[tipo \mapsto \text{“}Efetive\text{”}]\}$, as ações $GRRecebeMsgAborte(g)$ não poderiam ser satisfeitas para nenhum valor de t, já que há nelas o predicado $[tipo \mapsto \text{“}Aborte\text{”}] \in msgs$. Se o estado também possuir o valor de $estadoGR$ igual a “preparado” para todo gerenciador $g \in GR$, as ações $GRPrepara(g)$ e $GREscolheAbortar(g)$ também não poderiam ser satisfeitas, já que exigem um gerenciador no estado “trabalhando”.

Em alguns casos, apenas olhando para o valor do estado atual, é possível filtrar ações disjuntivas a ponto de apenas uma ação ser possivelmente satisfeita. Se essa ação define o valor de todas as variáveis no próximo estado - através de operadores *primed* e UNCHANGED - então esses valores são os únicos possíveis para as variáveis do próximo estado.

Em outros casos, encontrar apenas uma possível ação a partir do estado atual nunca será possível. As ações $GRPrepara(g)$ e $GREscolheAbortar(g)$ para um mesmo gerenciador g, por exemplo, tem um único predicado e ele é idêntico para ambas: $estadoGR[g] = \text{“}trabalhando\text{”}$. Uma vez que essas ações exigem próximos estados diferentes pelo valor de $estadoGR'$, sabe-se que a partir de um estado s que satisfaça-as, sempre haverá mais de um próximo estado possível que satisfaça a especificação.

Se a disjunção do conjunto de ações cujos predicados são satisfeitos pelo estado atual s é satisfeita por mais de um valor de próximo estado t, o estado atual exige uma escolha. Para a lógica temporal e o TLC, essas situações representam uma ramificação na árvore de estados.

Uma escolha no protocolo efetivação em duas fases, por exemplo, é a de um gerenciador de recurso preparar ($GRPrepara(g)$) ou abortar ($GREscolheAbortar(g)$). Escolhas não estão definidas pela especificação, e nesse caso poderia ser dada pelo resultado de uma etapa da transação de um gerenciador de recursos, onde ele pode escolher preparar (se o resultado foi sucesso) ou abortar (caso o resultado tenha sido fracasso); ou então poderia ser dada por um fator aleatório, isso não é relevante para o modelo especificado - qualquer escolha é válida.

MODULE *EfetivacaoEmDuasFases*

CONSTANT GR

VARIABLES $estadoGR$, $estadoGT$, $GRsPreparados$, $msgs$

$Mensagens \triangleq [tipo : \{\text{“EstouPreparado”}\}, gr : GR] \cup [tipo : \{\text{“Efetive”}, \text{“Aborte”}\}]$

$DFTypeOK \triangleq$
$\land\ estadoGR \in [GR \to \{\text{“trabalhando”}, \text{“preparado”}, \text{“efetivado”}, \text{“abortado”}\}]$
$\land\ estadoGT \in \{\text{“inicio”}, \text{“termino”}\}$
$\land\ GRsPreparados \subseteq GR$
$\land\ msgs \subseteq Mensagens$

$DFInit \triangleq$
$\land\ estadoGR = [g \in GR \mapsto \text{“trabalhando”}]$
$\land\ estadoGT = \text{“inicio”}$
$\land\ GRsPreparados = \{\}$
$\land\ msgs = \{\}$

$GTRecebePrepara(g) \triangleq$
$\land\ estadoGT = \text{“inicio”}$
$\land\ [tipo \mapsto \text{“EstouPreparado”}, gr \mapsto g] \in msgs$
$\land\ GRsPreparados' = GRsPreparados \cup \{g\}$
$\land$ UNCHANGED $\langle estadoGR, estadoGT, msgs\rangle$

$GTEfetiva \triangleq$
$\land\ estadoGT = \text{“inicio”}$
$\land\ GRsPreparados = GR$
$\land\ estadoGT' = \text{“termino”}$
$\land\ msgs' = msgs \cup \{[tipo \mapsto \text{“Efetive”}]\}$
$\land$ UNCHANGED $\langle estadoGR, GRsPreparados\rangle$

$GTAborta \triangleq$
$\land\ estadoGT = \text{“inicio”}$
$\land\ estadoGT' = \text{“termino”}$
$\land\ msgs' = msgs \cup \{[tipo \mapsto \text{“Aborte”}]\}$
$\land$ UNCHANGED $\langle estadoGR, GRsPreparados\rangle$

Figura 2.4: Especificação do protocolo de efetivação em duas fases - Parte 1

$$
\begin{array}{l}
GRPrepara(g) \triangleq \\
\quad \land\ estadoGR[g] = \text{“trabalhando”} \\
\quad \land\ estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“preparado”}] \\
\quad \land\ msgs' = msgs \cup \{[tipo \mapsto \text{“EstouPreparado”},\ gr \mapsto g]\} \\
\quad \land \text{ UNCHANGED } \langle estadoGT,\ GRsPreparados \rangle \\
\\
GREcolheAbortar(g) \triangleq \\
\quad \land\ estadoGR[g] = \text{“trabalhando”} \\
\quad \land\ estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“abortado”}] \\
\quad \land \text{ UNCHANGED } \langle estadoGT,\ GRsPreparados,\ msgs \rangle \\
\\
GRRecebeMsgEfetive(g) \triangleq \\
\quad \land\ [tipo \mapsto \text{“Efetive”}] \in msgs \\
\quad \land\ estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“efetivado”}] \\
\quad \land \text{ UNCHANGED } \langle estadoGT,\ GRsPreparados,\ msgs \rangle \\
\\
GRRecebeMsgAborte(g) \triangleq \\
\quad \land\ [tipo \mapsto \text{“Aborte”}] \in msgs \\
\quad \land\ estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“abortado”}] \\
\quad \land \text{ UNCHANGED } \langle estadoGT,\ GRsPreparados,\ msgs \rangle \\
\\
DFNext \triangleq \\
\quad \lor\ GTEfetiva \lor GTAborta \\
\quad \lor\ \exists\, g \in GR : \\
\qquad GTRecebePrepara(g) \lor GRPrepara(g) \lor GREcolheAbortar(g) \\
\qquad\quad \lor\ GRRecebeMsgEfetive(g) \lor GRRecebeMsgAborte(g) \\
\\
DFSpec \triangleq\ DFInit \land \Box[DFNext]_{\langle estadoGR,\ estadoGT,\ GRsPreparados,\ msgs \rangle} \\
\\
\text{THEOREM } DFSpec \Rightarrow \Box DFTypeOK \\
\\
\text{INSTANCE } TransacoesBD \\
\\
\text{THEOREM } DFSpec \Rightarrow TBDSpec
\end{array}
$$

Figura 2.5: Especificação do protocolo de efetivação em duas fases - Parte 2

3 O gerador de código

Dada uma especificação na linguagem TLA$^+$, contendo elementos da lógica TLA e da teoria de conjuntos, além de elementos sintáticos próprios, deseja-se obter uma definição equivalente em linguagem de programação. Equivalência para esse propósito é definida pela igualdade do conjunto de comportamentos permitidos. Isto é, todo comportamento especificado deve ser permitido na execução do código, e todo comportamento permitido pela execução do código deve ter sido especificado.

3.1 Elixir

Para esse propósito, a linguagem de programação escolhida para o código traduzido foi Elixir. As motivações são expostas abaixo por ordem de relevância na decisão:

1. A concorrência é facilitada por ter seu código traduzido para *bytecode* da máquina virtual do Erlang (BEAM). Suporte a concorrência é de extrema importância, já que TLA$^+$ foi criado para facilitar a especificação de sistemas concorrentes. É necessário que o código gerado seja capaz de refletir o sistema também nesse quesito.

2. Uma linguagem funcional tende a se(setq mouse-wheel-scroll-amount '(1 ((shift) . 1) ((control) . nil))) aproximar mais de definições matemáticas do que linguagens de outros paradigmas. Uma vez que a estrutura de TLA$^+$ foi construída principalmente no âmbito da matemática, a complexidade das traduções tende a ser menor para uma linguagem funcional.

3. O alto nível de abstração da sintaxe de Elixir, que se inspira em Ruby e sua busca por código facilmente entendível, faz com o programador que trabalhar com o código gerado possa entendê-lo de forma mais simples e rápida do que seria com uma linguagem de baixo nível. Com isso, otimizações podem ser feitas com mais segurança, e a manutenabilidade do código é favorecida.

4. A transparência de plataforma provida pela máquina virtual BEAM maximiza o número de ambientes aonde o código pode ser executado. Não seria de muito uso

gerar um código para um ambiente específico, e uma máquina virtual permite que o código gerado seja *cross plataform*.

5. O seu código é aberto sobre a licença Apache 2.0, permitindo que o funcionamento de suas estruturas possa ser verificado a qualquer momento. Não seria possível garantir nenhuma correspondência do código gerado com a especificação se não fosse conhecida a execução gerada pelos operadores usados no código.

Essa escolha vem de encontro com a finalidade de proporcionar um código modificável, de forma que o programador seja capaz de entender a correspondência entre as duas partes e minimizando a diferença do nível de abstração no qual ele está programando.

3.2 A tradução

A geração de código para uma especificação se dá pela tradução das estruturas de TLA$^+$ para Elixir. Esta tradução será feita de forma automática por uma ferramenta escrita em Haskell, implementada durante o corrente trabalho. A ferramenta será responsável pelo *parsing* do aquivo da especificação, no formato `.tla`, para estruturas internas e, então, transformação dessas estruturas internas em código Elixir.

A escolha da linguagem Haskell para implementação do gerador de código é motivada pela possibilidade da definição de tipos algébricos, que facilitam na representação das estruturas, e na tipagem forte, que ajuda a garantir consistência das relações entre estruturas definidas durante o processo, minimizando a possibilidade de erros no desenvolvimento. Haskell também conta com a biblioteca de *parsing* Parsec, que abstrai a complexidade de analisar sintaticamente um arquivo.

O escopo da tradução se limita à especificação definida, sendo suficiente para gerar código executável para o sistema definido. Traduzir teoremas e suposições não é necessário, uma vez que essas estruturas servem para fazer verificações sobre a especificação e não são necessárias para seu funcionamento. Ao código gerado não é atribuída a responsabilidade de refazer verificações, e sim de manter as propriedades já verificadas.

3.2.1 Mapeamentos

A tradução funciona como um grande mapeamento do conjunto de todas as especificações para um conjunto de programas em Elixir. Para viabilizar esse mapeamento, são definidos sub-mapeamentos que traduzem frações de uma especificação. Encontrar sub-mapeamentos suficientes para atender todo o domínio de especificações é suficiente para definir o processo de tradução.

Os primeiros mapeamentos definidos envolvem fórmulas transicionais e variáveis. Para cada fórmula transicional da especificação, é definida uma função, declarada com a sintaxe `def nome(parametros) do ... end`, que recebe as variáveis como parâmetro. O conjunto de variáveis do sistema é representado em uma Hash - estrutura de dados chave-valor de Elixir, equivalente a um dicionário - representada no padrão `variaveis = %{ variavel1: valor1, variavel2: valor2 }` e podendo ser acessada com `variaveis[:variavel1]` para obter `valor1`.

Cada função mapeada de uma ação recebe uma hash representando o estado atual e retorna outra hash representando o novo estado. O retorno, em Elixir, não exige uma palavra chave - a função retorna aquilo que a última linha retornou, sendo, para as funções geradas, a hash resultante da chamada do seu construtor.

A Figura 3.1 contém a função mapeada da fórmula *EsvaziaPequeno* definida na Figura 2.2.

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

Figura 3.1: Fórmula transicional *EsvaziaPequeno* como uma função em Elixir

Alguns operadores de TLA$^+$ permitem mapeamentos ainda mais diretos, como IF e CASE , devido a sua inspiração em linguagens de programação. A Figura 3.2 traz a função correspondente à fórmula *PequenoParaGrande* definida na Figura 2.2. A sintaxe para operadores IF em Elixir é na forma `if condição do ... else ... end`.

Com o conjunto inicial de mapeamentos apresentado, é possível definir todas as fórmulas transicionais do sistema definido na Seção 2.3. Ao traduzir as definições *Init*

```
def pequeno_para_grande(variaveis) do
  if variaveis[:grande] + variaveis[:pequeno] <= 5 do
    %{
      pequeno:  0,
      grande:  variaveis[:grande] + variaveis[:pequeno]
    }
  else
    %{
      pequeno:  variaveis[:pequeno] - (5 - variaveis[:grande]),
      grande:  5
    }
  end
end
```

Figura 3.2: Fórmula transicional *PequenoParaGrande* como uma função em Elixir

e *Next*, é possível executar concorrentemente todos os comportamentos permitidos pela especificação. A definição *Next* é traduzida para a função `main`, que recebe as variáveis para o estado atual e dispara um processo para cada passo permitido por *Next*. Como *Next* é uma disjunção de todas as ações, é disparado um novo processo com o resultado de cada função traduzida.

Para disparar processos, é chamada a função da biblioteca padrão de Elixir responsável por executar processos ligados: `spawn_link`. Essa função é chamada com três parâmetros: o módulo que receberá a chamada, a função a ser executada e uma lista contendo seus parâmetros. Para a tradução de *Next*, o módulo é sempre o módulo do arquivo gerado (`JarrosDeAgua`), a função é sempre `main` e os parâmetros são o resultado da aplicação de um dos passos permitidos. O último disparo corresponde à aplicação de um passo balbuciante. A definição dessa função encontra-se na Figura 3.3.

```
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 })
```

Figura 3.3: Disparo de processos para o sistema de Jarros de Água

A chamada `JarrosDeAgua.main(%{grande: 0, pequeno: 0})` é a tradução de *Init*. Como esse sistema permite um único estado inicial, apenas uma chamada a `main` é necessária. Com ela, todos os passos dados para iniciar novos processos terão iniciado com o valor para variáveis que satisfaz a condição inicial. Através da definição de `main`, é também garantido que todos os passos satisfazem $\Box[Next]_{vars}$. Assim, todos os comportamentos iniciados com essa chamada são permitidos por *Spec*, conforme definida na Seção 2.3.

O código gerado para esse sistema não permite, por si só, a solução do problema - uma vez que a especificação não tratava de uma solução. Entretanto, verificou-se que a invariante $jarro\_grande \setminus = 4$ não é satisfeita, e portanto um comportamento que leva à solução é permitido por esse sistema. É possível, apenas para fins exploratórios, encontrar os processos disparados pelo código que correspondem a esses comportamentos. Para isso, uma chamada que encerra o programa é invocada se o predicado da invariante for insatisfeito. Essa verificação é feita em todos os passos do comportamento, e portanto é definida como uma condição na função `main` como na Figura 3.4, que imprime os valores das variáveis com `IO.puts` e encerra o programa com um código de sucesso através de `:ok`.

```
def main(variaveis) do
  if variaveis[:grande] == 4 do
    IO.puts "#{variaveis[:grande]} #{variaveis[:pequeno]}"
    :ok
  end
  ...
end
```

Figura 3.4: Exploração de invariantes no código gerado

Com essa tradução inicial, é evidenciada a semelhança entre as definições matemáticas de TLA$^+$ e as estruturas do paradigma funcional presentes em Elixir. Entretanto, o disparo de múltiplos processos contemplando todos os caminhos possíveis do sistema não é viável dada a explosão de estados nem possui aplicação prática de muito valor. Assim, se apresenta a maior dificuldade encontrada durante o desenvolvimento deste trabalho: como traduzir especificações intermediárias que não descrevem uma implementação e sim um sistema ou protocolo.

A tradução da especificação completa de uma implementação não resultaria no

disparo de múltiplos processos pela definição $Next$ porque, ao especificar uma implementação, são definidas quais condições levam a execução de cada ação. Isso é evidenciado pela comparação das especificações do sistema (Figura 2.3), do protocolo e da implementação no Capítulo 2. Como o intuito do gerador de código é fornecer um protótipo funcional e viável, e não um disparador de processos interminável, são necessárias condições bem definidas para a decisão de que ação deve ser tomada. A Seção 3.2.2 descreve como a exigência dessas condições é feita.

3.2.2 Condições e Ações

Na definição de uma ação em TLA$^+$, existem expressões de estado e expressões de ação (descritas na seção 2.1). As expressões de estado em uma ação $\mathcal{A}$ são predicados que terão valoração verdadeira se e somente se o estado em que $\mathcal{A}$ será executada - ou seja, o primeiro estado do passo - seja um estado onde $\mathcal{A}$ é permitido. Essas expressões, quando dentro da definição de uma ação, podem ser interpretadas como condições de ativação dessa ação. A ação só pode ser ativada se suas condições de ativação forem satisfeitas.

Já uma expressão de ação envolve ambos os estados de um passo - o estado atual e o próximo estado. Se tratada puramente como uma especificação, esse próximo estado pode ser entendido como qualquer estado possível especificado, de forma que, de todos os estados possíveis, a expressão de ação só será verdadeira para aqueles pares de estado que respeitam a condição por ela determinada. Com essa perspectiva, as expressões de ação também são condições.

Contudo, no contexto de um programa executável viável, os estados são obtidos de maneira incremental. No código gerado, o estado inicial é obtido pela definicão do estado inicial da especificação, e os próximos estados são obtidos a partir da sucessiva aplicação de funções - que representam ações - nesse estado.

Gerar todos os estados possíveis e avaliar uma "condição de ação" para cada um dos passos para então filtrá-los seria uma forma possível de executar o programa, porém não é viável quando há muitos valores possíveis para as variáveis de estado. Ao especificar $x' = x + 1$, define-se que todos os pares de estado onde o segundo estado tem o valor de x igual ao valor $x - 1$ do primeiro estado. Contudo, em uma implementação, gerar pares de estado que para todos os valores inteiros e filtrar os que atendem essa condição é inviável. Nesse contexto, é possível simplesmente gerar os pares de estado válidos e, considerando

que o estado atual é conhecido durante a execução, isso significa apenas somar 1 ao valor de x para obter o segundo estado do par que satisfaz a ação.

As expressões de ação, no código gerado proposto, são operações efetuadas sobre um estado que resultam sempre em um novo estado válido. Para isso, elas só são ativadas quando as condições de ativação (definidas pelas expressões de estado) são satisfeitas. A divisão entre expressões de estado, que serão transformadas em condições, e expressões de ação, que serão transformadas em funções que alteram o estado, é feita pelo próprio parser já que essas expressões se distinguem na gramática da linguagem de TLA$^+$.

$$
\begin{array}{l}
GRRecebeMsgEfetive(g) \triangleq \\
\quad \land [tipo \mapsto \text{“Efetive”}] \in msgs \\
\quad \land estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“efetivado”}] \\
\quad \land \text{UNCHANGED } \langle estadoGT,\ GRsPreparados,\ msgs \rangle
\end{array}
$$

Figura 3.5: Ação para o recebimendo da mensagem “Efetive” por um gerenciador de recurso

```
def gr_recebe_msg_efetive_condition(variables, g) do
  Enum.member?(variables[:msgs], %{tipo:  "Efetive"})
end

def gr_recebe_msg_efetive(variables, g) do
  %{
    estado_gr:  Map.put(variables[:estado_gr], g, "efetivado"),
    estado_gt:  variables[:estado_gt],
    grs_preparados:  variables[:grs_preparados],
    msgs:  variables[:msgs]
  }
end
```

Figura 3.6: Código gerado para a ação $GRRecebeMsgEfetive(g)$

Na especificação do protocolo de efetivação em duas fases, na Figura 2.5, o tradutor transformara cada uma das ações em duas funções: uma de retorno booleano determinando se a condição de ativação é satisfeita, e outra retornando o estado resultante da aplicação da ação. A Figura 3.5 é um recorte dessa especificação, e será traduzida nas duas funções expostas na Figura 3.6.

3.2.3 Determinando a próxima ação

Como discutido na Seção 2.4.2, a ação de próximo estado $Next$ pode ser atendida por mais de um próximo estado t para o mesmo estado atual s. Isso não é um problema para o TLC, que avalia todos os pares estados possíveis distintos que atendem as condições da ação, verificando todas as ramificações da árvore de estados. No programa gerado, contudo, executar uma ação significa fazer possivelmente uma alteração de estado, e múltiplas alterações de estado não são interessantes para o propósito do código gerado, conforme discutido na Seção 3.2.1.

Assim, é necessário um processo bem definido para decidir qual ação executar a partir de um estado. Para garantir que seja possível tomar tais decisões sem uma situação que exija uma escolha, é necessário que a especificação defina, para cada estado do comportamento, condições de ativação mutualmente exclusivas para ações disjuntas com próximos estados distintos - isto é, sempre que há uma disjunção de ações, para todos os estados de um comportamento, todas as condições de ativação dessas ações devem ser falsas para aquele estado exceto pelas daquelas que levem ao próximo estado do comportamento.

Uma especificação que atenda essa restrição descreve uma função pura, onde não há efeitos colaterais e uma mesma entrada, dada pelo estado inicial, sempre gera uma mesma saída, dada pelo comportamento - sequência de estados - obtido. TLA$^+$ permite, contudo, especificar diversos comportamentos válidos a partir do mesmo estado inicial, o que possibilita que a sequência de estados possa ser eventualmente modificada por efeitos colaterais e permanecer válida.

Para garantir que o comportamento satisfaz a especificação, as influências de efeitos colaterais devem obedecer a função de próximo estado. Isso quer dizer que as influências são, para cada estado da execução, a escolha de uma ação que satisfaça a função de próximo estado. Como o estado atual s já é conhecido em tempo de execução, a influência precisa levar a um estado t tal que $< s, t >$ satisfaça tal função.

Quando é definido o conceito de escolha na Seção 2.4.2, define-se que é possível que exista apenas um valor possível para t que atenda essa restrição. Isso acontece quando a restrição de condições de ativação mutualmente exclusivas é respeitada pelo estado atual. Nesse caso, a influência externa não é necessária, já que é possível decidir o valor de t. Assim, a influência externa só é necessária quando uma escolha é exigida. Nesses casos, o

código gerado estabelece uma troca de mensagens entre o processo gerado pelo modelo e um novo processo denominado oráculo, que será responsável por toda influência externa no modelo.

O Algoritmo 1 é responsável por examinar as ações possíveis e decidir qual será aplicada ao estado atual - seja por uma decisão pura ou por uma escolha de um influência externa. Ela recebe uma identificação da ação origem, isto é, a definição cuja disjunção de ações precisa de uma decisão; e uma lista de ações que contém, para cada ação, um rótulo, o valor da sua condição de ativação para o estado atual e o próximo estado que sua execução traria.

Algorithm 1 Decisão da próxima ação

```
1: procedure DecideAção(origem, ações)
2:     acoes_possiveis ← ações com condição satisfeita
3:     estados_distintos ← estados únicos resultantes de acoes_possiveis
4:     if tamanho de estados_distintos = 1 then                ▷ Decisão pura
5:         return primeiro estado em estados_distintos
6:     else                                                    ▷ Influência Externa
7:         envie identificações das acoes_possiveis para o oráculo
8:         acao_escolhida ← resposta do oráculo
9:         return estado resultante de acao_escolhida
```

Se há apenas um próximo estado possível resultante das aplicações das ações ativadas, então o algoritmo decide por aquele estado. Caso contrário, ele abre uma conexão com o oráculo, enviando uma identificação das ações possíveis. O algoritmo permanece bloqueado até que o oráculo responda a ação escolhida, e então retorna o próximo estado correspondente àquela ação.

3.2.4 Concorrência

Mesmo TLA$^+$ sendo uma linguagem de especificação para sistemas concorrentes, ela não permite descrições para um *fork* ou disparo de novos processos. O valor de TLA$^+$ para sistemas concorrentes é possibilitar um conjunto de ordens de execução válido, e não limitar a apenas uma única ordem. Sistemas concorrentes usualmente apresentam muitos comportamentos distintos, e TLA$^+$ permite especificar um modelo que aceite vários desses comportamentos.

Assim, o modelo em si e sua tradução são sequenciais, e se comportam como uma troca sucessiva de estados. A ordem das transições que alteram esses estados é pos-

sivelmente determinada por uma influência externa - e esta, sim, pode possuir elementos concorrentes.

Considerando o sistema dos Jarros de Água do Capítulo 2, cada passo da execução exige uma escolha, já que nenhuma ação tem condição de ativação. Uma forma de fazer essa escolha é através do disparo de vários processos onde cada um seleciona uma das possíveis ações - esses processos podem ser representados como pessoas com intenções diferentes de resolver o problema dos jarros. Não é possível permitir que mais de uma pessoa (ou processo) altere o estado dos jarros ao mesmo tempo. É possível, contudo, a partir de uma fila ou de uma escolha aleatória de pessoas, estabelecer uma ordem de ações sobre os jarros.

Algumas operações computacionais, como a escrita em um endereço de memória, serializam ações - já que não podem ser feitos concorrentemente. Isso não significa que essas operações não precisam ser levadas em conta quando usadas em um sistema concorrente. Elas são justamente o ponto onde podem ocorrer problemas nesse tipo de sistema, nos momentos onde processos paralelos precisam de um recurso compartilhado e a ordem em que eles o obtém influencia o resultado final. Por isso, um procedimento de serialização como a ordem de obtenção de um recurso compartilhado pode acontecer de várias maneiras diferentes, e em razão disso sua especificação é importante.

Um exemplo mais tangível desse fenômeno pode ser dado pelo protocolo de efetivação em duas fases, discutido na Seção 2.4.2, onde existem vários gerenciadores de recursos enviando suas intenções de efetivar ou abortar para um gerenciador de transações compartilhado entre eles. O gerenciador de transações é um processo sequencial, mas pode ter vários comportamentos diferentes devido as possíveis ordens que as mensagens dos gerenciadores de recurso chegam.

3.3 Regras de Tradução

A partir de especificação escritas em TLA$^+$conforme a sintaxe da figura 3.7, deseja-se obter um código executável em Elixir com elementos da Figura 3.8. As regras de tradução são divididas em camadas conforme a recursão das definições, de forma que a especificação completa é traduzida no código pelas regras $\vdash$.

Em TLA$^+$, as definições $\mathcal{A}$ englobam predicados e ações e, como discutido na

Identificadores $I, C, k \subset v$		Valores v	Parâmetros p
Especificação	$Spec$	::=	$ModuleM$ CONSTANTS $C_o, \ldots, C_n$ VARIABLES $V_o, \ldots, V_n$ $D_0, \ldots, D_n$
Declaração	D	::=	$NomeAcao(p_0, \ldots, p_n) \triangleq \mathcal{A}$
Definição	$\mathcal{A}$	::=	$A \mid P \mid \mathcal{A} \wedge \mathcal{A} \mid \mathcal{A} \vee \mathcal{A}$
Predicado	P, Q	::=	$\neg P \mid P \wedge P \mid P \vee P \mid v_1 \in v_2 \mid v_1 = v_2$
		|	$v_1 \neq v_2 \mid$ ENABLED $\mathcal{A}$
Ação	A	::=	$I' = v \mid$ UNCHANGED $\langle I_0, \ldots, I_n \rangle$
Conjunto	S	::=	$v \mid S_a \cup S_b$
Registro	R	::=	$[k \mapsto v] \mid [I$ EXCEPT $![k] = v]$

Figura 3.7: Sintaxe aceita para TLA$^+$

Átomos `i, k`		Valores `x, y`	Parâmetros `p`
Estado	`a`	::=	`nome_acao(variables, p0, ..., pn) | variables`
		|	`%{ io: xo, ..., in: xn } | Map.merge(a, a)`
Condição	`c`	::=	`nome_acao_condition(variables, p0, ..., pn)`
		|	`not c | c and c | c or c`
Conjunto	`s`	::=	`MapSet.new([x]) | MapSet.union(sa, sb)`
Registro	`r`	::=	`%{ k: x } | Map.put(i, k, x)`
Definição	`def`	::=	`def nome_acao(variables, p0, . .., pn)`
		|	`def_decide_action([info])` (vide Algoritmo 1)
Informação	`info`	::=	`%{ action: "Nome", condition: c, state: a }`
		|	`Enum.map(x, f (i) -> [info] end)`

Figura 3.8: Elementos gerados em Elixir

Seção 3.2.2, deseja-se separá-los em condições `c` e ações sobre os estados `a`. Devido a essas definições com misturas de predicados e ações serem recursivas, é necessário que haja uma separação recursiva também. Para isso, ao traduzir uma definição com as regras $\vdash_d$, utiliza-se uma estrutura intermediária na forma de dupla, contendo uma lista das condições recursivamente encontradas e traduzidas como primeiro elemento, e das ações como segundo.

Além das traduções especificadas nas regras desta seção, o gerador de código também carrega os comentários da especificação, transformando comentários que antecedem as definições em documentação através das anotações `@doc` do Elixir. Também são omitidas das regras traduções básicas de expressões aritméticas e alguns detalhes imple-

mentados para melhorar a legibilidade do código gerado, como identação e conversão de *casing*.

A Seção 3.4 apresenta exemplos de aplicação das regras apresentadas nesta seção, e por isso é possivelmente um recurso para melhor compreendimento do funcionamento do tradutor. Por questão organizacional, as regras e exemplos são mantidos em seções distintas, mas recomenda-se a referência entre eles no momento da leitura deste capítulo.

3.3.1 Tradução de especificações

$$\boxed{\vdash Spec \rightarrowtail code}$$

$$\frac{\begin{array}{r} \Gamma \vdash_{const} C_0, \ldots, C_n \rightarrowtail \texttt{const}_0\texttt{, ..., const}_n \\ \{C_0 : const, \ldots, C_n : const\} \vdash_{dec} Def_0 \rightarrowtail \texttt{def}_0 \\ \vdots \\ \{C_0 : const, \ldots, C_n : const\} \vdash_{dec} Def_n \rightarrowtail \texttt{def}_n \\ \{C_0 : const, \ldots, C_n : const\} \vdash_{next} Def_{next} \rightarrowtail \texttt{def\_next} \\ \{M : module, C_0 : const, \ldots, C_n : const\} \vdash_{init} Def_{init} \rightarrowtail \texttt{state} \end{array}}{\begin{array}{l} \vdash \begin{array}{l} \text{MODULE } M \\ \text{CONSTANTS } C_0, \ldots, C_n \\ \text{VARIABLES } V_0, \ldots, V_n \\ Def_0 \\ \vdots \\ Def_n \\ Def_{init} \\ Def_{next} \end{array} \rightarrowtail \begin{array}{l} \texttt{defmodule M do} \\ \quad \texttt{@oracle spawn(Oracle, :listen, [])} \\ \quad \texttt{const}_0\texttt{, ..., const}_n \\ \quad \texttt{def}_0\texttt{, ..., def}_n \\ \quad \texttt{def\_next} \\ \quad \texttt{def\_decide\_action} \\ \texttt{end} \\ \\ \texttt{M.main(state)} \end{array} \end{array}} \text{(MOD)}$$

Figura 3.9: Tradução da especificação

A tradução de uma especificação para código Elixir é dada pela regra (MOD) na Figura3.15. O julgamento $\vdash$ é responsável pelo início da tradução, recebendo a especificação completa e traduzindo cada elemento com o julgamento devido. A partir dela, um módulo Elixir é obtido, assim como a invocação de sua função `main` com o estado inicial traduzido. Dentro do módulo, o oráculo é disparado e referenciado por uma constante, as constantes são declaradas e as definições traduzidas formam o restante das funções, juntamente com a função de próximo estado e a função `decide_action` definida pelo Algoritmo 1.

$$\boxed{\Gamma \vdash_{const} \overline{C} \rightarrowtail code}$$

$$\frac{}{\Gamma \vdash_{const} C_0, \ldots, C_n \rightarrowtail \begin{array}{l} @C_0 \text{ "<Value for } C_0\text{>"} \\ \texttt{def } C_0\texttt{, do: } @C_0 \\ \vdots \\ @C_n \text{ "<Value for } C_n\text{>"} \\ \texttt{def } C_n\texttt{, do: } @C_n \end{array}} \text{(CONST)}$$

$$\boxed{\Gamma \vdash_{dec} D \rightarrowtail code}$$

$$\frac{\Gamma \cup \{p_0 : param, \ldots, p_n : param\} \vdash_d \mathcal{A} \rightarrowtail (\{c_o, \ldots, c_n\}, \{a_0, \ldots, a_n\})}{\Gamma \vdash_{dec} NomeAcao(p_0, \ldots, p_n) \triangleq \mathcal{A} \rightarrowtail} \text{(DEF)}$$

```
def nome_acao_condition(variables, p0, ..., pn) do
  c0 and  ... and cn
end
def nome_acao(variables, p0, ..., pn) do
  Map.merge(a0, Map.merge(..., an))
end
```

Figura 3.10: Tradução de declarações de constantes e definições

As definições declaradas são traduzidas pela regra (DEF) (Figura 3.10) que, a partir das duplas obtidas recursivamente de uma definição $\mathcal{A}$, cria as funções `nome_acao_condition` e `nome_acao`, reduzindo as condições com o conector lógico `and` e as ações com a função `merge`, responsável por unificar duas estruturas do tipo `Map` - usada para representar estados - em uma só.

A declaração de constantes é traduzida pela regra (CONST), que define, para cada constante declarada na especificação, um atributo do módulo gerado em Elixir, a partir do operador `@`, assim como uma função que permite acessá-la de fora do módulo. O valor das constantes em TLA$^+$ só precisa ser definido para executar o TLC, e não está presente na especificação. Assim, o código gerado atribui um valor exemplo da forma "<Value for C>" a ser substituído pelo programador.

3.3.2 Tradução das definições

As definições de ações e predicados é recursivamente agregada em uma dupla de listas. Os predicados encontrados são traduzidos para condições com pelas regras $\vdash_P$ em (COND) e agregados na primeira lista da dupla, enquanto as ações são traduzidas em funções sobre o estado pelas regras $vdash_a$ em (ACT) e agregadas na segunda lista.

Quando uma definição é invocada dentro de outra definição, como acontece com *podeEfeitvar* e *naoEfetivado* na especificação de transações em bancos de dados na Figura 2.3 e em nas definições de próximo estado apresentadas, aplica-se a regra (CALL), que adiciona a condição e a ação daquela definição às respectivas listas, incorporando completamente a definição invocada.

$$\boxed{\Gamma \vdash_d \mathcal{A} \rightarrowtail (\overline{c}, \overline{a})}$$

$$\frac{\begin{array}{c}\Gamma \vdash_v v_0 \rightarrowtail x_0 \\ \vdots \\ \Gamma \vdash_v v_n \rightarrowtail x_n\end{array}}{\begin{array}{c}\Gamma \vdash_d \mathit{NomeAcao}(v_0, \ldots, v_n) \rightarrowtail \\ (\{\texttt{nome\_acao\_condition(variables, } \texttt{x}_0\texttt{, ..., } \texttt{x}_n\texttt{)}\}, \\ \{\texttt{nome\_acao(variables, } \texttt{x}_0\texttt{, ..., } \texttt{x}_n\texttt{)}\})\end{array}} \text{(CALL)}$$

$$\frac{\Gamma \vdash_p P \rightarrowtail c}{\Gamma \vdash_d P \rightarrowtail (\{c\},\ \{\})}\ \text{(COND)} \qquad \frac{\Gamma \vdash_a A \rightarrowtail a}{\Gamma \vdash_d A \rightarrowtail (\{\},\ \{a\})}\ \text{(ACT)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_p P \rightarrowtail c \\ \Gamma \vdash_d \mathcal{A}_t \rightarrowtail (\{\texttt{ct}_0, \ldots, \texttt{ct}_n\},\ \{\texttt{at}_0, \ldots, \texttt{at}_n\}) \\ \vdots \\ \Gamma \vdash_d \mathcal{A}_t \rightarrowtail (\{\texttt{ce}_0, \ldots, \texttt{ce}_n\},\ \{\texttt{ae}_0, \ldots, \texttt{ae}_n\})\end{array}}{\begin{array}{l}\Gamma \vdash_d \begin{array}{l}\text{IF } P \\ \text{THEN } \mathcal{A}_t \\ \text{ELSE } \mathcal{A}_e\end{array} \rightarrowtail (\{\mathit{condition}\},\ \{\mathit{action}\})\end{array}}\ \text{(IF)}$$

onde

condition =
```
if c do
  ct0 and ... and ctn
else
  ce0 and ... and cen
end
```

action =
```
if c do
  Map.merge(at0, Map.merge(..., atn))
else
  Map.merge(ae0, Map.merge(..., aen))
end
```

Figura 3.11: Tradução intermediária de definições - Geração

Condicionais

Estruturas IF definem ações com base em uma condição, e poderiam ser traduzidas para a disjunção de duas definições com suas próprias condições e ações, de forma que uma das definições tenha a condição idêntica a condicional de IF e a outra, sua negação. A as ações seriam enviadas para o algoritmo de decisão, já que se encontram em uma disjunção, mas nunca precisariam de uma escolha uma vez que a condicional torna suas condições mutualmente excludentes.

Para não adicionar indireções possivelmente confusas e preservar a semelhança com a especificação, contudo, optou-se por utilizar a própria estrutura `if` da linguagem destino. As definições dos blocos IF e ELSE podem conter tanto condições quanto ações, porém essas não podem ser mantidas em uma mesma expressão `if`, já que é necessário avaliar as condições antes de aplicar as ações. A expressão obtida da ação deve sempre resultar em uma alteração de estado, e no caso do `if`, a alteração dependente de uma condicional. Para que sempre haja o retorno de um novo estado, é necessário avaliar as condições antes e não durante a ação, e para uma essa ação IF ser ativável, é necessário atender as condições de ativação da definição em THEN se a condicional for verdadeira, e as condições de ativação de ELSE caso a condicional seja falsa - ou seja, a condição de ativação também depende da condicional. Assim, a regra (`IF`) gera duas estruturas `if`, uma para as condições e outra para as ações

Agregação

Quando as definições são conectadas pelo E lógico ($\wedge$), aplica-se a regra (`AND`) e as listas de condições e ações para cada definição são concatenadas - essas listas serão devidamente reduzidas a um elemento válido em outras regras.

Já quando o conector é o OU lógico ($\vee$), é necessário iniciar o processo de decisão discuto nas Seções 2.4.2 e 3.2.3. A regra (`OR`) não gera nenhuma nova condição, e encapsula o processo de decisão em uma única ação que se dá pela invocação da função `decide_action`, que implementa o Algoritmo 1 e retorna a ação que deve ser executada. Essa função recebe uma lista com três informações para cada definição conectada pelo OU, que são geradas pelas regras do julgamento $\vdash_i$.

Dada uma definição, a regra (`INFO-DEF`) gera uma estrutura `Map` com: Uma

$$\boxed{\Gamma \vdash_d \mathcal{A} \rightarrowtail (\overline{c}, \overline{a})}$$

$$\frac{\begin{array}{c}\Gamma \vdash_d \mathcal{A}_0 \rightarrowtail (\overline{c_0},\ \overline{a_0}) \\ \vdots \\ \Gamma \vdash_d \mathcal{A}_n \rightarrowtail (\overline{c_n},\ \overline{a_n})\end{array}}{\Gamma \vdash_d \begin{array}{l}\wedge\ \mathcal{A}_0 \\ \vdots \\ \wedge\ \mathcal{A}_n\end{array} \rightarrowtail (\overline{c_o} \cup \cdots \cup \overline{c_n},\ \overline{a_o} \cup \cdots \cup \overline{a_n})}\ \texttt{(AND)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_i \mathcal{A}_0 \rightarrowtail \texttt{info}_0 \\ \vdots \\ \Gamma \vdash_i \mathcal{A}_n \rightarrowtail \texttt{info}_n\end{array}}{\Gamma \vdash_d \begin{array}{l}\vee\ \mathcal{A}_0 \\ \vdots \\ \vee\ \mathcal{A}_n\end{array} \rightarrowtail \left(\begin{array}{l}\texttt{decide\_action(List.flatten[} \\ \{\},\ \{\ \ \texttt{info}_0\texttt{,} \\ \qquad \texttt{...,} \\ \qquad \texttt{info}_n \\ \texttt{])}\end{array}\ \begin{array}{l}\\ \\ \\ \\ \}\end{array}\right)}\ \texttt{(OR)}$$

Figura 3.12: Tradução intermediária de definições - Agregação

identificação da definição (gerada por uma função `show` de conversão da estrutura para `string`); A redução das condições encontradas na definição; e a redução das ações. Entre as definições a serem enviadas para o algoritmo de decisão, é possível que haja um quantificador existencial, determinando definições com parâmetros dependentes de algum valor. Nesse caso, a regra (`INFO-EX`) define uma chamada para a função `map`, que será capaz de gerar as informações conforme os valores em tempo de execução.

Tradução de Predicados

Os predicados em TLA$^+$ são fórmulas sobre o estado atual, e são traduzidos pelas regras $\vdash_p$ para expressões lógicas sobre o estado denominadas condições, e estarão sempre associados a execução de uma ação. As traduções são bem diretas e mapeiam os operadores de igualdade, desigualdade e pertence para suas representações em Elixir. Os operadores de desigualdade , , e são omitidos das regras por serem muito semelhantes às regras (`PRED-EQ`) e (`PRED-INEQ`).

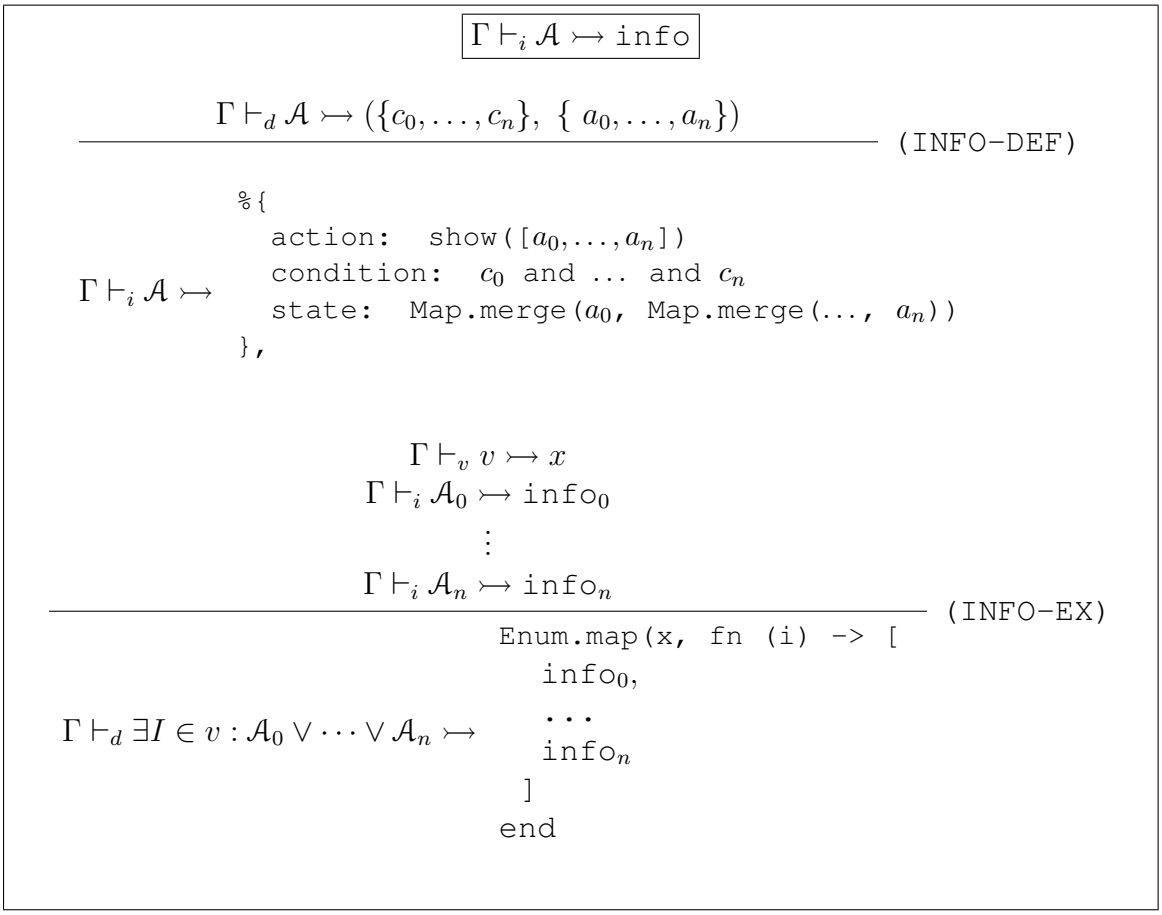

Figura 3.13: Organização das informações sobre uma ação

Tradução de Ações

As ações definidas em TLA$^+$ são fórmulas envolvendo o próximo estado. Em elixir, os estados são representados com a estrutura chave-valor `Map` em um parâmetro chamado `variables`, e uma ação traduz para uma nova estrutura desse tipo. Assim, ao aplicar a regra (ACT−UNCH) para uma ação do tipo UNCHANGED - isto é, o valor da variável no próximo estado permanece o mesmo - a nova estrutura indica que o valor para aquela chave, que representa a variável, é o valor que a estrutura anterior tinha para tal chave. A estrutura anterior é obtida pelo parâmetro `variables`, e o valor referente à chave `i`é obtido por `variables[:i]`.

De forma semelhante, quando há uma modificação do valor da variável na transição de estado, indicado pelo operador *primed*, a tradução é feita pela regra (ACT−PRIM) e a estrutura para o próximo estado apresenta o novo valor para a chave referente à variável sendo modificada.

Nas regras $\vdash_a$, cada ação de TLA$^+$ é transformada em uma estrutura do

$$\boxed{\Gamma \vdash_p P \rightarrowtail c}$$

$$\frac{\Gamma \vdash_v v_x \rightarrowtail x \quad \Gamma \vdash_v v_y \rightarrowtail y}{\Gamma \vdash_p v_x = v_y \rightarrowtail \texttt{x == y}}\ \texttt{(PRED-EQ)} \qquad \frac{\Gamma \vdash_v v_x \rightarrowtail x \quad \Gamma \vdash_v v_y \rightarrowtail y}{\Gamma \vdash_p v_x v_y \rightarrowtail \texttt{x != y}}\ \texttt{(PRED-NEQ)}$$

$$\frac{\Gamma \vdash_v v_e \rightarrowtail e \quad \Gamma \vdash_v v_l \rightarrowtail l}{\Gamma \vdash_p v_e \in v_l \rightarrowtail \texttt{Enum.member?(}l\texttt{, }e\texttt{)}}\ \texttt{(PRED-IN)}$$

$$\boxed{\Gamma \vdash_a A \rightarrowtail a}$$

$$\frac{}{\Gamma \vdash_a \text{UNCHANGED } \langle I_0, \ldots, I_n \rangle \rightarrowtail \texttt{\%\{ } I_0\texttt{: variables[:}I_0\texttt{], ..., } I_n\texttt{: variables[:}I_n\texttt{] \}}}\ \texttt{(ACT-UNCH)}$$

$$\frac{\Gamma \vdash_v v \rightarrowtail x}{\Gamma \vdash_a I' = v \rightarrowtail \texttt{\%\{ } I\texttt{: } x\texttt{ \}}}\ \texttt{(ACT-PRIM)}$$

Figura 3.14: Tradução de predicados e ações

tipo `Map`. Essas estruturas são posteriormente agregadas com a função `merge`, que teriam um formato como `Map.merge(variables, Map.merge(%{ i: 1 }, %{ j: 2 }))`. Para melhorar a legibilidade, após a tradução, o gerador agrega os valores literais em uma única estrutura, de forma que esse mesmo estado fica representado como `Map.merge(variables, %{ i: 1, j: 2 })`.

Tradução de valores

O mapeamento de valores é a camada mais baixa da tradução, e está bastante ligada a sintaxe. Conjuntos são traduzidos pra estruturas do tipo `MapSet`, que armazenam valores únicos em árvores binárias de busca; Registros são traduzidos para estruturas chave-valor `Map`, variáveis são acessadas pelo estado disponível no parâmetro `variables` e constantes pelo atributo do módulo.

Para decidir como traduzir um identificador, que pode representar uma variável, uma constante ou um parâmetro da de definição, utiliza-se o contexto Γ. Se o contexto indicar que um identificador é um parâmetro, ele é traduzido sem modificação.

Se ele for uma constante, é necessário verificar se a tradução está sendo solicitada de uma definição de dentro do módulo ou de fora - como é no caso da tradução do estado inicial. Se o contexto não tiver informação do módulo, significa que essa tradução

$$\boxed{\Gamma \vdash_v v \rightarrowtail x}$$

$$\frac{\{I : param\} \in \Gamma}{\Gamma \vdash_v I \rightarrowtail I} \text{ (VAL-PARAM)} \qquad \frac{\{I : param\} \notin \Gamma \quad \{I : const\} \notin \Gamma}{\Gamma \vdash_v I \rightarrowtail \texttt{variables[:}I\texttt{]}} \text{ (VAL-VAR)}$$

$$\frac{\begin{array}{c}\{I : const\} \in \Gamma \\ \{M : module\} \notin \Gamma\end{array}}{\Gamma \vdash_v I \rightarrowtail \texttt{@I}} \text{ (VAL-CONST)} \qquad \frac{\begin{array}{c}\{I : const\} \in \Gamma \\ \{M : module\} \in \Gamma\end{array}}{\Gamma \vdash_v I \rightarrowtail \texttt{M.I}} \text{ (VAL-ATTR)}$$

$$\frac{\Gamma \vdash_v v_0 \rightarrowtail x_o \ \ldots \ \Gamma \vdash_v v_n \rightarrowtail x_n}{\Gamma \vdash_v \{v_0, \ldots, v_n\} \rightarrowtail \texttt{MapSet.new([}x_0, \ldots, x_n\texttt{])}} \text{ (SET-LIT)}$$

$$\frac{\Gamma \vdash_v S_a \rightarrowtail s_a \quad \Gamma \vdash_v S_b \rightarrowtail s_b}{\Gamma \vdash_v S_a \cup S_b \rightarrowtail \texttt{MapSet.union(}s_a\texttt{, } s_b\texttt{)}} \text{ (SET-UNION)}$$

$$\frac{\Gamma \vdash_v v_0 \rightarrowtail x_o \ \ldots \ \Gamma \vdash_v v_n \rightarrowtail x_n}{\Gamma \vdash_v [k_0 \mapsto v_o, \ldots, k_n \mapsto v_n] \rightarrowtail \texttt{\%\{ } k_0\texttt{: } x_0, \ldots, k_n\texttt{: } x_n \texttt{ \}}} \text{ (REC-LIT)}$$

$$\frac{\Gamma \vdash_v v \rightarrowtail x \quad \Gamma \vdash_v I \rightarrowtail i}{\Gamma \vdash_v [I \text{ EXCEPT } ![k] = v] \rightarrowtail \texttt{Map.put(}i, k, x\texttt{)}} \text{ (REC-EXCEPT)}$$

$$\frac{\Gamma \vdash_v I \rightarrowtail i}{\Gamma \vdash_v I[k] \rightarrowtail i\texttt{[}k\texttt{]}} \text{ (REC-INDEX)}$$

Figura 3.15: Tradução de valores

vem de dentro do próprio módulo e então a constante pode ser acessada como uma constante do módulo, através do prefixo `@`. Caso haja uma indicação de módulo, é necessário traduzi-la para a chamada da função de acesso à constante daquele módulo, através de `NomeDoModulo.constante`.

Se o identificador não for nem um parâmetro e nem uma constante, então ele representa uma variável do estado e pode ser acessado por `variables[:i]`.

3.3.3 Estado Inicial

Para verificar propriedades no TLC, o TLA$^+$ exige uma definição dos estados iniciais possíveis. Essa definição tem a forma de um predicado sobre um estado - e não um passo - ou seja, não é uma ação.

Ao definir o predicado sobre estados iniciais para enviá-lo ao TLC, é possivel-

mente desejável que mais de um estado atenda-o, de forma que o TLC inicie a exploração do espaço de estados a partir de mais de um estado. Isso traria segurança de que, independentemente de quais dos estados iniciais definidos o programa se encontre no momento que inicia, seu comportamento será válido.

No contexto da execução desse programa, considera-se que não há benefício em permitir mais de um possível estado inicial. Uma possibilidade de lidar com uma definição de múltiplos estados iniciais seria delegar a escolha do estado inicial ao oráculo, porém isso significaria uma interação com o oráculo antes do início da execução do modelo, sendo que, nesse momento, o programador poderia fazer a escolha de forma mais simples. Assim, opta-se por encarregar o programador de definir o predicado para estados iniciais de forma a garantir que haja apenas um possível estado que satisfaça-o.

Após a ferramenta de tradução ser aplicada em contextos diferentes, será possível observar de que forma o oráculo é utilizado pelos programadores. Considera-se que se for comum o oráculo permanecer continuamente rodando e persistir o estado em que se encontra, é provável que seja interessante delegar a escolha do estado inicial ao oráculo em caso de reinicialização da execução do modelo, uma vez que ele pode ser capaz de restaurar o estado ou partes dele com as informações retidas. Essa é uma análise e possível trabalho a serem feitos futuramente.

De forma semelhante à tradução de ações, não é viável gerar todos os estados possíveis e então filtrá-los para encontrar um que atenda o predicado para o estado inicial. Assim, e exclusivamente nesse momento, deseja-se que o predicado gere um estado. Para isso, são necessárias regras de tradução que transforem um predicado na representação de um estado, onde igualdade é convertida para atribuição. Essas regras são expressadas por $\vdash_{init}$ na Figura 3.16. O estado inicial gerado é então enviado para a primeira chamada da função de próximo estado `main` conforme as regras $\vdash$ em 3.15.

3.3.4 Função de próximo estado

A definição *Next* é traduzida de maneira similar à definição das outras ações, exceto que o código gerado para a ação resultante é enviado como parâmetro para uma nova chamada da mesma função. O nome da definição é alterado para `main` e a função `main` gerada é chamada recursivamente, recebendo sempre um estado e retornando a invocação de `main` para o próximo estado gerado. A Figura 3.16 apresenta as regras de tradução

$\vdash_{next}$, responsáveis por gerar tal função.

Para que o programador seja capaz de acompanhar a execução do modelo, é também adicionada uma função de saída que exibe o estado atual sempre que a função `main` é invocada.

$$\boxed{\Gamma \vdash_{init} P \rightarrowtail a}$$

$$\frac{\Gamma \vdash_{v} v \rightarrowtail \mathtt{x}}{\Gamma \vdash_{init} I = v \rightarrowtail \mathtt{\%\{\ i:\ x\ \}}}\ \texttt{(INIT-EQ)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{init} P_0 \rightarrowtail \mathtt{a}_0 \\ \vdots \\ \Gamma \vdash_{init} P_n \rightarrowtail \mathtt{a}_n\end{array}}{\begin{array}{l}\Gamma \vdash_{init} \begin{array}{l}\wedge\ P_0 \\ \vdots \\ \wedge\ P_n\end{array} \rightarrowtail \mathtt{Map.merge(a_0,\ Map.merge(...,\ a_n))}\end{array}}\ \texttt{(INIT-AND)}$$

$$\boxed{\Gamma \vdash_{next} NomeAcao \triangleq \mathcal{A} \rightarrowtail code}$$

$$\frac{\Gamma \vdash_{d} \mathcal{A} \rightarrowtail (\{\},\ \mathtt{a}_0, \ldots, \mathtt{a}_n)}{\begin{array}{l}\Gamma \vdash_{next} NomeAcao \triangleq \mathcal{A} \rightarrowtail \\ \quad \begin{array}{l}\mathtt{def\ main(variables)\ do} \\ \quad \mathtt{IO.puts\ (inspect\ variables)} \\ \\ \quad \mathtt{main(} \\ \quad\quad \mathtt{Map.merge(a_0,\ Map.merge(...,\ a_n))} \\ \quad \mathtt{)} \\ \mathtt{end}\end{array}\end{array}}\ \texttt{(NEXT)}$$

Figura 3.16: Tradução do Estado Incial de da função de próximo estado

Sintaticamente as definições do estado inicial e da função de próximo estado são semelhantes à de qualquer ação. Assim, de forma semelhante a uma execução do TLC, o gerador de código recebe como parâmetro o nome dessas duas definições para que possa identificá-las e aplicar as regras de traduções devidas.

3.4 Aplicação da tradução

Com as regras de tradução definidas na Seção 3.3, é possível traduzir as especificações do Capítulo 2 em código Elixir executável. Esta seção tem como objetivo demonstrar os resultados, exemplificando a tradução para os exemplos de especificações dos Jarros de Água (Figura 2.2) e do protocolo de efetivação em duas fases (Figuras 2.4 e 2.5)

3.4.1 Jarros de Água

MODULE *JarrosDeAgua*

EXTENDS *Integers*

VARIABLES *jarro_pequeno*, *jarro_grande*

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

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

(MOD)

```
defmodule JarrosDeAgua do
  require Oracle
  @oracle spawn(Oracle, :listen, [])
   ...
end

JarrosDeAgua.main(
  %{
    jarro_grande:  0,
    jarro_pequeno:  0
  }
)
```

Figura 3.17: Tradução do módulo Jarros de Água

A especificação do sistema de Jarros de Água contém poucos elementos e é ideal para exemplificar os mapeamentos mais básicos. A tradução do módulo inicia pela aplicação da regra (MOD) demonstrada na Figura 3.17. A especificação não apresenta constantes, descartando a necessidade de aplicar a regra (CONST).

A definição *TypeOK* é uma invariante da especificação e, apesar de apresentar um predicado válido, não representa valor algum para a execução e, portanto, foi manualmente descartado. O estado inicial definido por *Init* - conforme identificado como

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

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

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

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

$\Downarrow$ (DEF)

```
def enche_pequeno_condition(variables) do
  True
end

def enche_pequeno(variables) do
  %{
    jarro_pequeno:  3,
    jarro_grande:  variables[:jarro_grande]
  }
end
...
```

Figura 3.18: Tradução das definições simples dos Jarros de Água

parâmetro da tradução - e gera a chamada para `JarrosDeAgua.main` apresentada. Os parâmetros com os quais a função é chamada são resultado da aplicação das regras (INIT-AND) e (INIT-EQ), e pela mesma agregação de literais feita para ações, que omite chamadas de `Map.merge` e foi exemplificada na Seção 3.3.

No interior do módulo, são listadas as definições. As definições dessa especificação se agrupam em duas formas similares: a primeira contendo apenas modificações de variáveis, e a segunda apresentando uma expressão `IF`. A Figura 3.18 demonstra o resultado da tradução para definições do primeiro grupo, apresentado também na figura.

O segundo grupo de definições traduz para uma condicional de Elixir conforme a Figura 3.19. Destaca-se a diferença entre uma ação com uma expressão `IF` e um predicado, que gera uma condição. A expressão $\mathit{jarro\_grande} + \mathit{jarro\_pequeno}5$ não é uma condição para a execução da ação, a ação pode ser ativada a qualquer momento. Essa expressão apenas altera os valores a serem atribuídos às variáveis do próximo estado.

Para ambos os grupos de definições, a regra (AND) é usada para agregar as

$$
\begin{array}{lll}
PequenoParaGrande \triangleq & \text{IF } jarro\_grande + jarro\_pequeno \leq 5 & \\
& \text{THEN} & \wedge jarro\_grande' = jarro\_grande + jarro\_pequeno \\
& & \wedge jarro\_pequeno' = 0 \\
& \text{ELSE} & \wedge jarro\_grande' = 5 \\
& & \wedge jarro\_pequeno' = jarro\_pequeno - (5 - jarro\_grande) \\
GrandeParaPequeno \triangleq & \text{IF } jarro\_grande + jarro\_pequeno \leq 3 & \\
& \text{THEN} & \wedge jarro\_grande' = 0 \\
& & \wedge jarro\_pequeno' = jarro\_grande + jarro\_pequeno \\
& \text{ELSE} & \wedge jarro\_grande' = jarro\_pequeno - (3 - jarro\_grande) \\
& & \wedge jarro\_pequeno' = 3
\end{array}
$$

$$\Big\downarrow \text{(DEF)}$$

```
def pequeno_para_grande_condition(variables) do
  True
end

def pequeno_para_grande(variables) do
  if variables[:jarro_grande] + variables[:jarro_pequeno] <= 5 do
    %{
      jarro_grande: variables[:jarro_grande] +
        variables[:jarro_pequeno],
      jarro_pequeno: 0
    }
  else
    %{
      jarro_grande: 5,
      jarro_pequeno: variables[:jarro_pequeno] -
        (5 - variables[:jarro_grande])
    }
  end
end
...
```

Figura 3.19: Tradução das definições com condicionais dos Jarros de Água

condições e ações encontradas para cada linha da definição. Em nenhuma definição há condições, resultando portanto em uma lista vazia que será compreendida como `True`. Todas as ações utilizam o operador *primed*, e portanto são traduzidas através da regra (ACT-PRIM).

Por fim, a definição de próximo estado por $Next$, assim como $Init$ identificado através de um parâmetro do tradutor, é traduzida pela regra ((Next)) conforme a Figura 3.20. A função `main` é recursiva, sempre chamando ela mesma com o próximo estado, sempre imprimindo o estado com que foi chamada para que seja possível acompanhar suas alterações. A definição $Next$ apresenta ações separadas pelo operador $\vee$, e portanto

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

$\Big\downarrow$ (NEXT)

```
def main(variables) do
  IO.puts (inspect variables)

  main(
    decide_action(
      List.flatten([
        %{
          action:  "EnchePequeno()",
          condition:  enche_pequeno_condition(variables),
          state:  enche_pequeno(variables)
        },
        %{
          action:  "EncheGrande()",
          condition:  enche_grande_condition(variables),
          state:  enche_grande(variables)
        },
        ...
      ])
    )
  )
end
```

Figura 3.20: Tradução da função de próximo estado para os Jarros de Água

requer uma decisão. Como discutido na Seção 3.2.3, a decisão é delegada para a função `decide_action`, que recebe uma lista com informações sobre as possíveis ações.

Essa lista de ações é exemplificada com as informações para as duas primeiras ações disponíveis, onde a chave `action` tem como valor uma identificação daquela ação, `condition` chama a função de condição que irá retornar se ela está satisfeita, e `state` informa qual o novo estado caso a decisão seja aquela ação.

Execução

Devido à ausência de condições, cada mudança de estado - isto é, cada chamada da função `main` - exige uma escolha, de forma que a maior parte do funcionamento do

código gerado dependerá da implementação do oráculo. Supondo um oráculo que escolha sempre uma ação aleatória entre as enviadas a ele, a execução dos jarros de água se dará pela alteração aleatória de quantidade de água nos jarros, porém sempre respeitando as mudanças especificadas.

Outra implementação possível para o oráculo é a leitura de alguma entrada que determine a ação a ser tomada. Com essa implementação, o resultado da execução pode ser entendido como um jogo ou simulação, onde algum usuário determina uma ação e observa um resultado, de forma que as restrições impostas pela especificação funcionem como regras que limitam as operações a serem feitas pelo usuário.

3.4.2 Protocolo de efetivação em duas fases

A especificação do protocolo tem elementos mais semelhantes a um modelo real, tal qual um conjunto arbitrário de gerenciadores de recurso podendo realizar ações em uma alta variedade de ordens. O código gerado a partir dela permite a execução de um gerenciador de transações, que executa ações escolhidas por gerenciadores de recursos e mantém o banco de dados consistente.

A tradução do módulo é, como sempre, iniciada pela regra (`MOD`), e traduz para um módulo Elixir conforme a Figura 3.21. A constante GR declarada na especificação é traduzida para `@gr` e declarada no início do módulo com sua função de acessibilidade `gr`. As definições $Mensagens$ e $DFTypeOK$ não tem valor para a execução e são manualmente removidas antecipadamente. O estado inicial é convertido em parâmetros para a chamada de `EfetivacaoEmDuasFases.main`, mapeando cada valor conforme as traduções definidas por $\vdash_v$.

Dentre as traduções de definições, exemplifica-se a tradução para $GTEfetiva$ na Figura 3.22. As duas primeiras definições da conjunção são condições, já que precisam apenas de informações constantes e do estado atual para serem avaliadas. Essas duas definições serão traduzidas para condições e declaradas em `gt_efetiva_condition`. As outras definições são traduzidas na ação que retorna o novo estado. Para as ações com o operador *primed*, o novo valor é atribuído; e para a ação UNCHANGED , o valor de cada variável na tupla no estado atual é atribuído no novo estado.

Exemplificando a tradução de uma definição parametrizada, a Figura 3.23 ap-

MODULE *EfetivacaoEmDuasFases*

CONSTANT GR

VARIABLES $estadoGR$, $estadoGT$, $GRsPreparados$, $msgs$

$Mensagens \triangleq [tipo : \{\text{“EstouPreparado”}\}, gr : GR] \cup [tipo : \{\text{“Efetive”}, \text{“Aborte”}\}]$

$DFTypeOK \triangleq$
$\wedge\ estadoGR \in [GR \to \{\text{“trabalhando”}, \text{“preparado”}, \text{“efetivado”}, \text{“abortado”}\}]$
$\wedge\ estadoGT \in \{\text{“inicio”}, \text{“termino”}\}$
$\wedge\ GRsPreparados \subseteq GR$
$\wedge\ msgs \subseteq Mensagens$

$DFInit \triangleq$
$\wedge\ estadoGR = [g \in GR \mapsto \text{“trabalhando”}]$
$\wedge\ estadoGT = \text{“inicio”}$
$\wedge\ GRsPreparados = \{\}$
$\wedge\ msgs = \{\}$

(MOD)

```
defmodule EfetivacaoEmDuasFases do
  require Oracle
  @oracle spawn(Oracle, :listen, [])

  @gr "<value for GR>"
  def gr, do:  @gr
  ...
end

EfetivacaoEmDuasFases.main(%{
  estado_gr:  EfetivacaoEmDuasFases.gr
    |> Enum.map(fn (g) -> {g, "trabalhando"} end)
    |> Enum.into(%{ }),
  estado_gt:  "inicio",
  g_rs_preparados:  MapSet.new([]),
  msgs:  MapSet.new([])
})
```

Figura 3.21: Tradução do módulo Efetivação em Duas Fases

resenta o código gerado para $GRRecebeMsgEfetive(g)$. A condição não depende do parâmetro já que apenas verifica a presença de uma mensagem, o que significa que sempre que essa ação for ativável, ela será ativável para qualquer valor de g. O estado resultante, contudo, depende do parâmetro, alterando o valor de $estadoGR$ apenas no valor referente ao parâmetro. O tradutor, contudo, gera as declarações das funções com os mesmos parâmetros - a omissão de parâmetros não utilizados, como é o caso de `g` em `gr_recebe_msg_efetive` não se mostrou relevante o suficiente para ser implementada, e esses parâmetros podem ser removidos posteriormente pelo programador com

$$
\begin{aligned}
GTEfetiva \triangleq\ & \\
& \wedge\ estadoGT = \text{“inicio”} \\
& \wedge\ GRsPreparados = GR \\
& \wedge\ estadoGT' = \text{“termino”} \\
& \wedge\ msgs' = msgs \cup \{[tipo \mapsto \text{“Efetive”}]\} \\
& \wedge\ \text{UNCHANGED}\ \langle estadoGR,\ GRsPreparados \rangle
\end{aligned}
$$

$$\Big\downarrow \text{(DEF)}$$

```
def gt_efetiva_condition(variables) do
  variables[:estado_gt] == "inicio" and
    variables[:g_rs_preparados] == @gr
end

def gt_efetiva(variables) do
  %{
    estado_gt:  "termino",
    msgs:  MapSet.put(variables[:msgs], %{ tipo:  "Efetive" }),
    estado_gr:  variables[:estado_gr],
    g_rs_preparados:  variables[:g_rs_preparados]
  }
end
```

Figura 3.22: Tradução da definição $GTEfetiva$ do protocolo

base em alertas do analisador de código do Elixir.

Por fim, a tradução da função de próximo estado $DFNext$ é disposta na Figura 3.24. As informações para as ações $GTEfetiva$ e $GTAborta$ são traduzidas de forma semelhante à tradução de $Next$ no problema dos Jarros de Água. Já as outras ações são quantificadas sobre os gerenciadores em GR, e as informações serão geradas com a regra (INFO-EX), produzindo a chamada de `map` que, para cada valor em `@gr`, aplicará uma função lambda que aplica a regra (INFO-ACT). A identificação de uma ação com parâmetro utiliza interpolação de *string* com o operador e a função `inspect`, que converte as estruturas de Elixir para *string*. Isso é necessário para que o oráculo identifique completamente aquela ação, e não receba várias cópias de um mesmo identificador sem o valor do parâmetro.

Execução

Para permitir a execução do modelo, é necessário definir valores para as constantes. O protocolo depende da constante GR que é o conjunto de gerenciadores de recurso que

$$GRRecebeMsgEfetive(g) \triangleq$$
$$\land [tipo \mapsto \text{“Efetive”}] \in msgs$$
$$\land estadoGR' = [estadoGR \text{ EXCEPT } ![g] = \text{“efetivado”}]$$
$$\land \text{UNCHANGED } \langle estadoGT,\ GRsPreparados,\ msgs \rangle$$

$$\Big\downarrow \text{(DEF)}$$

```
def gr_recebe_msg_efetive_condition(variables, g) do
  Enum.member?(variables[:msgs], %{ tipo:  "Efetive" })
end

def gr_recebe_msg_efetive(variables, g) do
  %{
    estado_gr:  Map.put(variables[:estado_gr], g, "efetivado"),
    estado_gt:  variables[:estado_gt],
    g_rs_preparados:  variables[:g_rs_preparados],
    msgs:  variables[:msgs]
  }
end
```

Figura 3.23: Tradução da definição $GRRecebeMsgEfetive$ do protocolo

poderão efetivar ou abortar durante a execução. Define-se então um conjunto simples de dois identificadores representando dois gerenciadores com `@gr MapSet.new(["g1", "g2"]`.

Cada passo da execução do protocolo exigirá uma escolha, delegada ao oráculo. Com o intuito de validar o modelo gerado para esse exemplo, uma implementação mais realística do oráculo foi explorada, e o resultado é fornecido como exemplo junto à implementação do tradutor. Este oráculo funciona como um gerenciador de transações (GT) do protocolo, se comunicando com os gerenciadores de recurso (GRs). Os GRs também são implementados com o propósito de validação, e fazem parte da influência no processo de escolha.

Todos os gerenciadores, GRs e GT, são processos que processam input do usuário no terminal, ao mesmo tempo que trocam mensagens sobre as ações a serem escolhidas. O GT, que recebe as mensagens do modelo (por ser essencialmente o oráculo), divide as possíveis ações em três grupos: as relacionadas a ele mesmo (como $GTAborta$), as relacionadas ao gerenciador g1 (como $GRPrepara(\text{“}g1\text{”})$) e as relacionadas ao gerenciador g2. Cada grupo de ações é enviado para o processo correspondente, de forma a serem exibidos para o usuário que decidirá, dentre as ações do grupo, qual será escolhida.

$$
\begin{array}{l}
DFNext \triangleq \\
\quad \lor GTEfetiva \lor GTAborta \\
\quad \lor \exists\, g \in GR : \\
\qquad GTRecebePrepara(g) \lor GRPrepara(g) \lor GREcolheAbortar(g) \\
\qquad \lor GRRecebeMsgEfetive(g) \lor GRRecebeMsgAborte(g)
\end{array}
$$

$$\Big\downarrow \text{(NEXT)}$$

```
def main(variables) do
  IO.puts (inspect variables)

  main(
    decide_action(
      List.flatten([
        %{ action: "GTEfetiva()",
           condition: gt_efetiva_condition(variables),
           state: gt_efetiva(variables) },
        %{ action: "GTAborta()",
           condition: gt_aborta_condition(variables),
           state: gt_aborta(variables) },
        Enum.map(@gr, fn (g) -> [
          %{ action: "GTRecebePrepara(#{inspect g})",
             condition: gt_recebe_prepara_condition(variables, g),
             state: gt_recebe_prepara(variables, g) },
          %{ action: "GRPrepara(#{inspect g})",
             condition: gr_prepara_condition(variables, g),
             state: gr_prepara(variables, g) }
           ...
        ] end)
      ])
    )
  )
end
```

Figura 3.24: Tradução da função de próximo estado para o protocolo

Quando uma entrada é lida em algum dos processos, a ação escolhida é enviada de volta ao oráculo, que por sua vez a envia para o modelo e espera por novas ações. Assim, é possível simular execuções, como por exemplo através dos passos:

1. Início

   Estado do Modelo:

   ```
   estado_gr: %{"g1" => "trabalhando", "g2" => "trabalhando"},
   estado_gt: "inicio",
   g_rs_preparados: #MapSet<[]>,
   msgs: #MapSet<[]>
   ```

Ações possíveis:

[GT]: `GTAborta`

[GR1]: `GRPrepara("g1"), GREscolheAbortar("g1")`

[GR2]: `GRPrepara("g2"), GREscolheAbortar("g2")`

2. GR1 escolhe `GRPrepara("g1")`

   Estado do Modelo:

   ```
   estado_gr: %{"g1" => "preparado", "g2" => "trabalhando"},
   estado_gt: "inicio",
   g_rs_preparados: #MapSet<[]>,
   msgs: #MapSet<[%{gr: "g1", tipo: "EstouPreparado" }]>
   ```

   Ações possíveis:

   [GT]: `GTAborta, GTRecebePrepara("g1")`

   [GR1]:

   [GR2]: `GRPrepara("g2"), GREscolheAbortar("g2")`

3. GR2 escolhe `GREscolheAbortar("g2")`

   Estado do Modelo:

   ```
   estado_gr: %{"g1" => "preparado", "g2" => "abortado"},
   estado_gt: "inicio",
   g_rs_preparados: #MapSet<[]>,
   msgs: #MapSet<[%{gr: "g1", tipo: "EstouPreparado" }]>
   ```

   Ações possíveis:

   [GT]: `GTAborta, GTRecebePrepara("g1")`

   [GR1]:

   [GR2]:

4. GT escolhe `GTAborta`

   Estado do Modelo:

   ```
   estado_gr: %{“g1” => “preparado”, “g2” => “abortado”},
   estado_gt: “termino”,
   g_rs_preparados: #MapSet<[]>,
   msgs: #MapSet<[%{tipo: “Aborte” }%{gr: “g1”, tipo:
   “EstouPreparado” }]>
   ```

   Ações possíveis:

   [GT]:

   [GR1]: `GRRecebeMsgAborte(“g1”)`

   [GR2]: `GRRecebeMsgAborte(“g2”)`

4 Considerações

A etapa inicial do desenvolvimento deste trabalho envolveu, na maior parte, revisão do material bibliográfico e didático disponível sobre TLA e TLA$^+$. O conhecimento obtido a partir desse estudo foi exposto no Capítulo 2. É esperado que novos conceitos sejam encontrados na continuação do trabalho, o que se transformará em atualizações e adições em tal capítulo. Deseja-se também, com a evolução do conhecimento sobre o assunto, tornar a descrição dos conceitos mais didática e organizada.

Além da apresentação dos conceitos necessários para o entendimento da lógica TLA e da linguagem de especificação TLA$^+$, foram expostos dois exemplos com a intenção de esclarecer o funcionamento das especificações na linguagem. Esses exemplos são resgatados no Capítulo 3 na demonstração de uma instância do objetivo deste trabalho, isto é, numa tradução de especificação para código. Essa demonstração assegura a viabilidade do trabalho, apesar de tratar um escopo restrito do problema. Com o desenvolvimento de mais mapeamentos para compor o tradutor, deseja-se apresentar mais exemplos como o incluso na Seção 3.2.1.

Para o exemplo de especificação apresentado na Seção 2.4, deseja-se definir uma outra especificação para exemplificar o conceito de implementação, apresentando um protocolo implementado sobre a especificação apresentada - assim como Lamport demonstra implementação em (LAMPORT, 2017). Com esse último exemplo devidamente explicado, é considerado que o Capítulo 2 é suficiente para um entendimento dos fundamentos e da aplicação de TLA$^+$.

Foram justificadas, no Capítulo 3, as escolhas das linguagens Elixir, para o código gerado, e Haskell, para a implementação do tradutor, respectivamente na Seção 3.1 e na Seção 3.2. Esse mesmo capítulo explica como será composto o tradutor, mostrando a possibilidade de descrevê-lo, neste trabalho e na implementação, através de mapeamentos. O processo de busca por mapeamentos se encontra em um estágio inicial, e deseja-se encontrar mapeamentos que permitam traduzir o maior subconjunto possível de especificações para linguagens, sendo o cenário ideal aquele em que são encontrados mapeamentos suficientes para traduzir corretamente qualquer especificação escrita em TLA$^+$.

A etapa de implementação do tradutor ainda não foi iniciada. Durante a revisão bibliográfica, contudo, foi encontrada a definição completa da gramática de TLA$^+$, em (LAMPORT, 2002). Esse artefato facilitará a implementação do *parser*, disponibilizando mais tempo do que o previsto para a implementação dos mapeamentos em si.

Com essas considerações, é evidenciado o progresso deste trabalho durante sua primeira fase, assim como sua viabilidade de conclusão no prazo previsto e a relevância de suas contribuições.

4.1 Cronograma

As etapas para este trabalho foram estabelecidas conforme a lista:

1. Etapa 1 - Revisão bibliográfica sobre TLA+: Inclui leitura de textos e tutoriais, busca de exemplos e outras fontes de conhecimento sobre a linguagem, como o curso em vídeo ensinado pelo criador da linguagem.

2. Etapa 2 - Traduções manuais: Se dá por tentativas de codificação de exemplos de modelos obtidos na Etapa 1. Essa codificação será em uma linguagem de programação funcional com concorrência, maximizando a proximidade com o modelo - possivelmente Elixir.

3. Etapa 3 - Estabelecimento de mapeamentos: Uma observação das traduções manuais ao lado do código, com objetivo de enumerar mapeamentos feitos no processo manual. Dessa etapa, se espera uma lista de possíveis correspondências a serem avaliadas na Etapa 4 e 5.

4. Etapa 4 - Escolha de mapeamentos: Dentre os possíveis mapeamentos, serão escolhidos aqueles que apresentam maior correspondência entre as definições, conforme estudos sobre os significados dos construtores utilizados nas duas linguagens.

5. Etapa 5 - Encontrar garantias para os mapeamentos: O processo de conjecturar, evidenciar ou provar a correspondência entre uma especificação formal qualquer e o código gerado a partir dela com os mapeamentos escolhidos na Etapa 4.

6. Etapa 6 - Implementação do gerador de código: Consiste em implementar um programa capaz de fazer o *parsing* da linguagem TLA+ e escrever um arquivo com o

código na linguagem de programação.

7. Etapa 7 - Análise de melhorias do ambiente: É o estudo da capacidade de testes para o código gerado manterem a correspondência perante modificações no mesmo. Será feito atráves da implementação de testes e exploração de quais mudanças teriam potencial de tornar o código incoerente com a especificação e passariam nos testes.

Etapas	**2019**											
	J	**F**	**M**	**A**	**M**	**J**	**J**	**A**	**S**	**O**	**N**	**D**
1												
2												
3												
4												
5												
6												
7												

Table 4.1: Cronograma Proposto

O cronograma proposto encontra-se na Tabela 4.1. Na primeira fase deste trabalho, contudo, foram finalizadas as etapas 1 e 2, e a etapa 3 está em andamento, juntamente com a 4 e a 5. Assim, o novo cronograma é proposto na Tabela 4.2.

Etapas	**2019**											
	J	**F**	**M**	**A**	**M**	**J**	**J**	**A**	**S**	**O**	**N**	**D**
1												
2												
3												
4												
5												
6												
7												

Table 4.2: Cronograma Atualizado

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. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, v. 5, n. 2, p. 190–222, 1983.

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.

LAMPORT, L. *Two-Phase Commit*. 2017. Disponível em: <http://lamport.azurewebsites.net/video/video6.html>. Acesso em: 15 jun. 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.

MILNER, R.; PARROW, J.; WALKER, D. A calculus of mobile processes, i. *Information and Computation*, v. 100, n. 1, p. 1 – 40, 1992. ISSN 0890-5401. Disponível em: <http://www.sciencedirect.com/science/article/pii/0890540192900084>.

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>.

PETRI, C. A. Fundamentals of a theory of asynchronous information flow. In: *IFIP Congress*. [S.l.: s.n.], 1962. p. 386–390.