Gabriela Moreira Mafra

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

Joinville

2019

UNIVERSIDADE DO ESTADO DE SANTA CATARINA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Gabriela Moreira Mafra

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

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

Cristiano Damiani Vasconcellos
Orientador

Karina Girardi Rôggia
Co-Orientador

Joinville, Março de 2019

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

Gabriela Moreira Mafra

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

Banca Examinadora

Cristiano Damiani Vasconcellos - Doutor
(orientador)

Adelaine Franciele Gelain - Mestre

Paulo Henrique Torrens - Mestre

Resumo

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

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

Abstract

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

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

Contents

List of Figures

List of Tables

1 Introdução

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

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

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

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

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

Baseando-se na lógica definida como TLA, Lamport propõe a linguagem de especificação formal TLA$^+$ (*Temporal Logic of Actions*), com o objetivo de escrever provas

formais para sistemas concorrentes da maneira mais simples possível (LAMPORT, 2008). Nessa linguagem, além dos operadores de TLA, são incluídos elementos da teoria de conjuntos e alguns açúcares sintáticos para fórmulas temporais como cláusulas IF e CASE.

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

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

A partir das definições de propriedades desejadas e da possibilidade de verificá-las, se torna possível alterar uma especificação no 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 lógica 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, 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 tem 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.

1.1 Objetivos

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

1.1.1 Objetivos Específicos

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

2 TLA$^+$

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

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

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

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

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

screvendo as operações existentes daquele sistema.

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

2.1 Lógica Temporal das Ações

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

Lamport sumariza em (CHAUDHURI et al., 2008) o uso de TLA em TLA$^+$. TLA é uma lógica temporal linear. Em TLA$^+$, as variáveis rígidas do TLA são chamadas constantes, enquanto as flexíveis são chamadas variáveis. As 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 VARIABLESe 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.

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

Uma fórmula 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. Uma especificação F implementa outra especificação G se e somente se qualquer sistema que satisfaz F também satisfaz G, ou seja, a fórmula $G \implies F$ é válida.

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

- ENABLED $\mathcal{A}$ ($\mathcal{A}$ 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 $s \to t$ satisfaz $\mathcal{A}$.

- $\Box F$ (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 uma passo) do comportamento.

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

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

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

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

- 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).

- $\exists x : F$ 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.

- CHOOSE $x : P$ (escolha algum x 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 , e possível afirmar que se $\exists x : P$ então P(CHOOSE $x : P$) é verdadeiro, e que para todo predicado Q tal que $Q \equiv P$, é verdade que (CHOOSE $x : P$) = (CHOOSE $x : Q$).

- f EXCEPT $![v] = e$ para uma função f, um elemento de seu domínio v e uma expressão e é definida por $[x \in D \mapsto$ IF $x = d$ THEN e ELSE $f[x]]$ onde D é o domínio de f. Ou seja, o valor da expressão EXCEPT é uma cópia de f exceto pelo valor $f[v]$ que é igual a e.

- ASSUME c (assuma c) para uma fórmula constante c define c como verdadeiro. Esse operador não tem nenhum efeito nas definições de uma especificação, apenas pode facilitar a verificação de teoremas.

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

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 especifi-

cação - se mantenha igual durante um passo.

Supondo que as variáveis da especificação estejam declaradas como

$$vars =< var_1, var_2, \ldots, var_n >$$

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 $vars$.

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 define 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$^+$ através de invariantes.

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 através 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

$$\Box(\text{ENABLED } \mathcal{A} \implies \Diamond[\mathcal{A}]_f)$$

A conjunção com $(f' \neq f)$ 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

$$\Box\Diamond \text{ENABLED } \mathcal{A} \implies \Box\Diamond[\mathcal{A}]_f$$

que pode ser lida como "se sempre $\mathcal{A}$ for eventualmente ativável, então sempre um passo $[\mathcal{A}]_f$ 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 2.1.

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 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 símbolo de linha representam os valores no estado seguinte, e sempre precisam ser definidas. Na transição $EnchePequeno$, o valor de $jarro\_grande$ se mantém o mesmo entre os estados atual e seguinte, mas é necessário explicitar isso com $jarro\_grande' = jarro\_grande$. Essa necessidade vem da aproximação

Figure 2.1: Especificação do problema dos Jarros de Água

MODULE *JarrosDeAgua*

EXTENDS *Integers*

VARIABLES *jarro_pequeno*, *jarro_grande*

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

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

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

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

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

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

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

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

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

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 nos em qualquer um dos estados, isto é: $(jarro\_pequeno = 0,\ jarro\_grande = 0), (jarro\_pequeno = 0,\ jarro\_grande = 1), \ldots$.

No sentido de aproveitar informações do estado atual, é possível utilizar condicionais, como nas 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 seguinte 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 não deve ser modificado em nenhum estado da execução. Já que o estado inicial definido em $Init$ faz $TypeOK$ verdadeiro, ao colocar essa invariante, todos os estados devem fazer $TypeOK$ verdadeiro, ou o TLC retornará um erro. $TypeOk$ pode ser definido como uma invariante através do teorema:

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

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

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

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

Esse exemplo é apresentado com o intuito de demonstrar a estrutura da especificação de um sistema e o funcionamento das invariantes. A seguir, é proposto um exemplo com especificações de um sistema real e de um protocolo implementado sobre ele.

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 cometer (fazer a ação *commit*) se todos os outros gerenciadores estiverem preparados para cometer, 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 2.2, encontra-se uma especificação para um sistema de transações consistente. Ela não apresenta uma proposta de solução para o problema, e sim traz uma descrição formal do que significa ser consistente 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 cometer; ou então abortar, indo para o estado "abortado". Se todos os gerenciadores estão no estado "preparado", então qualquer um deles pode cometer, 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 cometer é representada pela ação $Decide(g)$. Nessa definição, as fórmulas *podeCometer* e *naoCometido* são definidas separadamente para minimizar a complexidade cognitiva da especificação - definí-las dentro de $Decide(g)$ seria semanticamente equivalente. *podeCometer* verifica se todos os estados estão preparados ou cometidos, ou seja, qualquer um pode cometer. Se *podeCometer* for verdadeiro, e g ainda não cometeu, então g comete - o estado dos gerenciadores passa

Figure 2.2: Especificação de um sistema de transações em bancos de dados

MODULE *TransacoesBD*

CONSTANT GR

VARIABLE $estadoGR$

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

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

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

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 $naoCometido$, se não há nenhum gerenciador cometido. Se $naoCometido$ 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

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

que permite verificar que, se um comportamento satisfaz $TBDSpec$ - isto é, seu estado inicial satisfaz $TBDInit$ e seus passos satisfazem $TBDNext$ - então as fórmulas de estado $TBDTypeOK$ e $TBDConsistente$ são verdadeiras 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 A implementação

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 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 generalizados, 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 o valor.

Cada função mapeada de uma fórmula transicional 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 2.1.

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

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

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 2.1. 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

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

```
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
```

as fórmulas transicionais do sistema definido na Seção 2.3. Ao traduzir as definições *Init* 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 fórmulas transicionais, 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.

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

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

```
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 })
```

- 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`.

Figure 3.4: Exploração de invariantes no código gerado

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

Com essa tradução inicial, é evidenciada a semelhança entre as definições matemáticas de TLA$^+$ e as estruturas do paradigma funcional presentes em Elixir. Espera-se obter mapeamentos claros tais quais os encontrados até então para o restante das estruturas das duas linguagens, de forma que o tradutor finalizado seja intuitivo, e que o código gerado seja trivialmente relacionado com a especificação para o programador que a escreveu.

3.3 Cronograma

Bibliography

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

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

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

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

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

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

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

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

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

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

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