UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JONES MARTINS VIEIRA DA CRUZ

Relato de experiência em modelagem e verificação de algoritmos concorrentes com TLA+

RIO DE JANEIRO
2024

JONES MARTINS VIEIRA DA CRUZ

Relato de experiência em modelagem e verificação de algoritmos concorrentes com TLA+

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Profª. Anamaria Martins Moreira
Co-orientadora: Profª. Silvana Rossetto

RIO DE JANEIRO
2024

CIP - Catalogação na Publicação

C957r

Cruz, Jones Martins Vieira da
Relato de experiência em modelagem e verificação de algoritmos concorrentes com TLA+ / Jones Martins Vieira da Cruz. -- Rio de Janeiro, 2024.
96 f.

Orientadora: Anamaria Martins Moreira.
Coorientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. Computação concorrente. 2. Métodos formais. 3. Verificação de modelos. 4. TLA+. I. Moreira, Anamaria Martins, orient. II. Rossetto, Silvana, coorient. III. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

JONES MARTINS VIEIRA DA CRUZ

Relato de experiência em modelagem e verificação de algoritmos concorrentes com TLA+

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 26 de junho de 2024

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
ANAMARIA MARTINS MOREIRA
Data: 12/08/2024 17:53:24-0300
Verifique em https://validar.iti.gov.br

Prof.ª Anamaria Martins Moreira
Doutora (IC-UFRJ)

gov.br Documento assinado digitalmente
SILVANA ROSSETTO
Data: 11/08/2024 15:06:44-0300
Verifique em https://validar.iti.gov.br

Prof.ª Silvana Rossetto
Doutora (IC-UFRJ)

gov.br Documento assinado digitalmente
DANIEL SADOC MENASCHE
Data: 12/08/2024 21:07:54-0300
Verifique em https://validar.iti.gov.br

Prof. Daniel Sadoc Menasché
Doutor (IC-UFRJ)

gov.br Documento assinado digitalmente
JOAO ANTONIO RECIO DA PAIXAO
Data: 13/08/2024 04:37:59-0300
Verifique em https://validar.iti.gov.br

Prof. João Antônio Récio da Paixão
Doutor (IC-UFRJ)

AGRADECIMENTOS

Agradeço às professoras Anamaria e Silvana, pelo rigor, paciência, e apoio inestimável durante toda a longa extensão de tempo que este trabalho tomou.

Agradeço novamente à professora Silvana por me conseguir um espaço no LC3 para eu escrever este trabalho. Dizer que foi uma contribuição tremenda ainda seria pouco.

Agradeço à galera do LC3 com quem compartilhei o espaço por mais de um ano, especialmente ao Bruno Carno pelo acolhimento no espaço, e pelos cafés que davam para "passar no pão".

Agradeço à comunidade de TLA$^+$, principalmente a Stephan Merz e Markus Kuppe pela constante disponibilidade na resolução de dúvidas, a Hillel Wayne e Calvin Loncaric por percepções que afetaram diretamente este trabalho.

Agradeço a meus pais pelo suporte constante durante esse percurso.

Agradeço aos amigos de longa data: do Martins, de São Paulo, de 15.2; e aos novos amigos da UFRJ: do LABIC, do LC3, de Libras, do Cores, do SIGA, do VoIP.

Muito obrigado a todos!

RESUMO

Sistemas concorrentes corretos são difíceis de se implementar. Abordagens em nível de programação, desde linguagens de programação modernas a ferramentas sofisticadas de testes, são incapazes de verificar propriedades importantes de sistemas concorrentes. Analisamos métodos formais como abordagem complementar para verificar tais propriedades, e, principalmente, para projetar sistemas concorrentes corretos. Métodos formais aqui são representados pela linguagem de especificação TLA+ e o verificador de modelos TLC. Estudamos TLA+ e sua lógica subjacente (TLA), desenvolvemos dois exemplos de modelagem e especificação em TLA+ – uma *lock* e um contador compartilhado – e, finalmente, discutimos os aprendizados e dificuldades dessa experiência. Concluímos que TLA$^+$ nos dá uma base sólida sobre a qual futuras implementações de sistemas concorrentes podem ser construídas, principalmente porque a linguagem e seu ecossistema nos permitem definir e verificar propriedades relevantes impossíveis de se verificar com testes e simulações de programas.

Palavras-chave: Computação concorrente; Métodos formais; Verificação de modelos; TLA+.

ABSTRACT

Correct concurrent systems are difficult to implement. Programming-level approaches, from modern programming languages to sophisticated testing techniques, are incapable of verifying important concurrent system properties. We analyze formal methods as a complementary approach to verifying important properties and especially designing correct concurrent systems. Formal methods here are represented by the TLA+ specification language and the TLC model checker. We study TLA+ and its underlying logic (TLA), we develop two examples in modelling and specification–a lock and a concurrent counter–, and, finally, we discuss lessons and challenges throughout this experience. We conclude that TLA$^+$ gives us solid ground onto which future implementations of concurrent systems may be built, especially because the language and its ecosystem allow us to define and verify relevant properties that are impossible to verify with program tests and simulations.

Keywords: Concurrent systems; Formal methods; Model checking; TLA+.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

ASCII	*American Standard Code for Information Interchange*
AWS	*Amazon Web Services*
CTL	*Computation Tree Logic*
LTL	*Linear Temporal Logic*
PDF	*Portable Document Format*
SAT	Problema de satisfatibilidade lógica
SMT	*Satisfiability Modulo Theory*
TLA	*Temporal Logic of Actions*
TLAPS	*TLA*$^+$ *Proof System*
TLC	*TLA*$^+$ *Model Checker*
UFRJ	Universidade Federal do Rio de Janeiro

LISTA DE SÍMBOLOS

$\vee$	Disjunção lógica
$\wedge$	Conjunção lógica
$\Rightarrow$	Implicação lógica
$\neg$	Negação lógica
$\equiv$	Equivalência entre fórmulas
$=$	Igualdade de valores
$\exists\, x$	Existe uma variável x
$\forall\, x$	Para toda variável x
$s \in S$	Valor s pertence a conjunto S
$R \subseteq S$	Conjunto R é subconjunto de S
$E \triangleq expr$	E é definido como igual a $expr$
$\Box P$	Predicado P é sempre verdadeiro
$\Diamond P$	Predicado P é futuramente verdadeiro
$P \rightsquigarrow Q$	Sempre que o predicado P é verdadeiro, futuramente Q é verdadeiro

SUMÁRIO

1 INTRODUÇÃO

Sistemas concorrentes corretos são difíceis de se implementar. Desde o compartilhamento de uma variável entre *threads*, até a garantia de consistência eventual de componentes de um sistema concorrente, a ampla variedade e complexidade dos desafios que o não determinismo impõe sobre desenvolvedores de sistemas concorrentes requer atenção especial. Como garantiremos a corretude de um sistema concorrente se cada execução possui um resultado diferente, ou, pior, se os resultados são frequentemente iguais, exceto em raríssimas ocasiões, devido a uma sequência infeliz de eventos? Ou, ainda, como nos certificaremos que um sistema distribuído é robusto frente a mensagens corrompidas ou atrasadas, conexões instáveis e quedas de energia?

A resposta – ou uma tentativa de resposta – para grande parte dessas dificuldades que a concorrência traz costuma aparecer na forma de linguagens de programação e ferramentas de testagem, ou seja, soluções em nível de implementação. No caso de linguagens de programação, pode-se implementar um sistema em Rust (MATSAKIS; KLOCK, 2014), uma linguagem imperativa que reduz a possibilidade de conflitos de acesso a memória através do conceito de empréstimo e posse de variáveis. Pode-se implementar um sistema em Go (GOOGLE, 2009), outra linguagem imperativa, aproveitando-se de canais e *gorotinas*, que facilitam a implementação de programas concorrentes. No paradigma funcional, Clojure (HICKEY, 2007) dificulta o uso de variáveis mutáveis, em geral (como outras linguagens deste paradigma), e implementa memória transacional em software (*software transactional memory*) (SHAVIT; TOUITOU, 1995). Ou ainda, o gerenciamento de *threads* e processos pode ser abstraído completamente em linguagens baseadas no modelo de atores como Erlang (ARMSTRONG, 2003), Elixir (VALIM, 2012) e Pony (CLEBSH, 2015), que só permitem o compartilhamento de dados por mensagens trocadas entre esses atores, removendo a necessidade de *locks* para sincronização. No caso de ferramentas de testagem, programas sofisticados como ThreadSanitizer (GOOGLE, 2015), Helgrind (VALGRIND™, 2007b), DRD (VALGRIND™, 2007a) buscam detectar uma série de erros de concorrência em programas, desde corrida de dados até ordenamentos inconsistentes de *locks* que podem causar impasses.

Cada uma dessas linguagens e ferramentas é notável frente ao desafio de escrever programas concorrentes corretos. Porém, são técnicas insuficientes. Apesar de cada uma dessas linguagens e ferramentas conseguirem reduzir a quantidade de erros de concorrência presentes, elas são incapazes de garantir programas livres desses erros: linguagens sofisticadas podem acrescentar erros de novas categorias, e ferramentas de testes são limitadas nos erros detectáveis.

O objetivo deste trabalho é explorar métodos formais, mais especificamente, a linguagem de especificação TLA$^+$ (LAMPORT, 2002) e o verificador de modelos (*model checker*)

de TLA$^+$, TLC (YU; MANOLIOS; LAMPORT, 1999), como uma abordagem complementar a essas ferramentas. Sair do domínio da programação em direção a um formalismo matemático facilitará o projeto de sistemas concorrentes ao nos permitir abstraí-los em diferentes níveis de complexidade e granularidade. Além disso, esse formalismo garantirá que tais projetos satisfaçam uma vasta gama de propriedades complexas que as linguagens e técnicas mencionadas raramente conseguem garantir.

Visamos atingir esse objetivo expondo a teoria de TLA$^+$, dando dois exemplos de modelagem e especificação em TLA$^+$ – uma *lock* e um contador compartilhado – , e discutindo sobre a experiência de aprender e usar TLA$^+$ no contexto do curso de Ciência da Computação. Apesar de não podermos preferir TLA$^+$ a outros formalismos (esta não é a função deste trabalho), concluímos que TLA$^+$ nos dá uma base sólida para futuras implementações de sistemas concorrentes e distribuídos – todo algoritmo distribuído é concorrente –, principalmente porque a linguagem e seu ecossistema nos permitem definir e verificar propriedades relevantes impossíveis de serem verificadas com testes e simulações de programas.

1.1 POR QUE TLA+?

Este trabalho foi escrito pela perspectiva de um aluno de graduação que descobriu métodos formais por acaso na forma de verificação de modelos. Os assuntos ensinados durante o curso de Ciência da Computação mais próximos ao tópico deste trabalho lidam com linguagens formais – máquinas de Turing, categorias de linguagens, teoria de autômatos – e lógica – lógica proposicional, lógica de primeira ordem, dedução e resolução lógica. Portanto, perceber que é possível gerar todas as execuções possíveis de determinado algoritmo e verificar ou até provar propriedades sobre este algoritmo foi uma surpresa bem-vinda.

A decisão de avaliar TLA$^+$ não foi resultado de um processo rigoroso de escolha entre alternativas. Trabalhos que fazem esse tipo de análise podem ser encontrados em (MAZZANTI; FERRARI, 2018), num contexto de supervisão automática de trens, e (DAY; BANDALI, 2022) em múltiplos contextos.

Dito isso, a decisão de avaliar TLA$^+$ também não foi por acaso. A linguagem possui um histórico de uso interessante principalmente para quem não conhece métodos formais e para quem imagina que métodos formais só seriam aplicados em contextos específicos, como *hardware* e *software* embarcado, e especialmente em sistemas de alto risco à vida humana, como veículos e aparelhos médicos. Apesar de TLA$^+$ poder contribuir positivamente no desenvolvimento desses exemplos, destacamos aqui seu uso no desenvolvimento de sistemas distribuídos complexos.

Em sua página *web* "*Industrial Use of TLA*$^+$" (LAMPORT, 2022), Lamport apresenta depoimentos de uso de TLA$^+$ por funcionários de empresas como Intel, Amazon, Dropbox

e Microsoft, que utilizaram (e ainda utilizam) a linguagem para especificar e verificar que porções críticas de seus algoritmos de escala global funcionam como esperado. Todas as citações em português são nossa tradução.

Relato de Chuck Thacker sobre o uso de TLA$^+$ no desenvolvimento do console de jogos Xbox 360:

> Durante o desenvolvimento do Xbox 360, trabalhei com o grupo de engenheiros no protocolo de coerência de memória. Trabalhando com um estagiário, desenvolvemos um modelo em TLA$^+$ para o protocolo. Durante esse desenvolvimento, descobrimos um *bug* bem sutil. Reportamos o *bug* à IBM, que respondeu que o erro era impossível de acontecer. Duas semanas depois, eles cederam e nos disseram que não apenas o *bug* era real, mas que seus testes de regressão não o teriam encontrado. Se não o tivessem consertado, teriam nos enviado *chips* que entrariam em *deadlock* após cerca de quatro horas de uso. Se isso tivesse ocorrido, o plano de lançamento na época do Natal quase certamente teria fracassado.

Uso de TLA$^+$ na Azure, serviço de nuvem da Microsoft, relatado por Albert Greenberg, vice-presidente empresarial da Azure Networking:

> Achamos TLA$^+$ especialmente adequado para escrever especificações de alto nível de sistemas concorrentes e distribuídos, definindo concretamente o problema, escrevendo uma especificação próxima da implementação desejada, e provando que refinamentos sucessivos implementam a especificação, ou, senão, encontrando falhas e corrigindo o *design*. Entre os projetos na Azure Networking em que aplicamos TLA$^+$ estão a Azure DNS (propagação de registros de DNS), RingMaster (replicação global distribuída e coordenação de checkpoints), distribuição de demanda distribuída, e Macsec (orquestração de substituição de chaves criptográficas). Por exemplo, em RingMaster, identificamos um *bug* que apareceu em falhas intermitentes de testes unitários, mas que era incrivelmente difícil de localizar no código. Escrevemos uma especificação em TLA$^+$ para o comportamento desejado, encontramos rapidamente o *bug* no projeto lógico, e consertamos o código.

Uso de TLA$^+$ em Cosmos DB, serviço de banco de dados distribuído globalmente pela Microsoft, relatado por Dharma Shukla, parceiro técnico da Microsoft:

> O time de engenheiros do Cosmos DB tem usado TLA$^+$ para especificar e validar a corretude de algoritmos centrais, assim como especificações de alto-nível para cinco modelos de consistência que o sistema oferece aos nossos clientes. Achamos TLA$^+$ extremamente útil de duas maneiras fundamentais:
>
> i. Para projetar algoritmos distribuídos. Por exemplo, enquanto projetamos o algoritmo para a capacidade de escrita multi-local, encontramos um *bug* crítico de corretude que violava uma propriedade importante de segurança (*safety*) do sistema. O *bug* foi encontrado enquanto escrevemos a especificação em TLA$^+$, e conseguimos corrigir o problema antes de escrever uma única linha de código.
>
> ii. Especificar precisamente as garantias providas por um algoritmo ou sistema. Por exemplo, o Cosmos DB usou TLA$^+$ para especificar as garantias providas pelos cinco modelos de consistência que o serviço oferece a seus usuários. Essas especificações ficam disponíveis aos usuários.

Lamport também menciona um artigo escrito por Newcombe et al. (NEWCOMBE et al., 2015), onde os autores descrevem sua experiência com TLA$^+$ na AWS, suas dificuldades, e sobre convencer outros engenheiros a aprenderem e utilizarem a linguagem. Os pontos-chave destacados no próprio artigo são:

- Métodos formais encontram *bugs* em projetos de sistemas que não são encontrados por meio de outras técnicas que conhecemos.
- Métodos formais são surpreendentemente viáveis para desenvolvimento de *software* convencional e dão bons retornos em investimento.
- Na Amazon, métodos formais são rotineiramente aplicados no projeto de *softwares* complexos reais, incluindo serviços públicos da nuvem.

E os autores concluem:

> Métodos formais foram um grande sucesso na AWS, ajudando-nos a prevenir *bugs* sutis, porém sérios, de alcançarem o ambiente de produção, *bugs* que não teríamos encontrado por meio de outras técnicas. Métodos formais nos ajudaram a conceber otimizações agressivas para algoritmos complexos sem sacrificar qualidade. Durante a escrita deste artigo, sete times da Amazon usaram TLA$^+$, todos encontrando valor em seu uso, e mais times da Amazon começaram a usá-lo. Usar TLA$^+$ melhorará o *time-to-market* e a qualidade de nossos sistemas. A gerência executiva encoraja times a escreverem especificações em TLA$^+$ para novos recursos e outras mudanças significativas de design. No planejamento anual, gerentes passaram a alocar tempo de engenharia para TLA$^+$.
>
> Embora nossos resultados sejam encorajadores, temos algumas ressalvas importantes. Métodos formais lidam com modelos de sistemas, e não com sistemas em si, então o provérbio "todos os modelos estão errados; alguns são úteis" se aplica. O projetista deve se certificar de que o modelo captura aspectos significativos do sistema real. Conseguir isso é uma habilidade especial cuja aquisição requer prática cuidadosa. Além disso, nos preocupamos somente em obter benefícios práticos em nosso domínio de problema específico, e não realizamos uma pesquisa abrangente. Portanto, os resultados podem variar com outras ferramentas e em outros domínios de problema.

1.2 ESTRUTURA DO TRABALHO

O restante deste texto está organizado da seguinte forma. No Capítulo 2 apresentamos conceitos de engenharia de requisitos, verificação formal de *software* e lógica temporal. No Capítulo 3 explicamos a lógica temporal de ações, a linguagem TLA$^+$ e o verificador de modelos TLC. No Capítulo 4 damos dois exemplos de modelagem e especificação formal em TLA$^+$, e os desafios encontrados. No Capítulo 5 discutimos TLA$^+$ e experiência de aprendê-la no contexto do curso de Ciência da Computação. No Capítulo 6, abordamos a conclusão e trabalhos futuros.

2 FUNDAMENTOS

Neste capítulo apresentamos os fundamentos por trás do nosso trabalho divididos em duas seções: engenharia de software e lógica matemática. Na seção de engenharia de software contextualizaremos o uso de especificações matemáticas em projetos de *software*, ao definirmos o que são especificações de requisitos, e como métodos formais auxiliam no desenvolvimento desses projetos. Na seção de lógica matemática explicaremos brevemente as duas principais lógicas temporais para verificação de *software* para, no próximo capítulo, detalhar a lógica TLA.

2.1 ESPECIFICAÇÃO E VERIFICAÇÃO FORMAL DE *SOFTWARE*

Segundo Sommerville (SOMMERVILLE, 2016, p. 8), o projeto de *software* tem quatro atividades fundamentais comuns a todos os processos de *software*, como desenvolvimento incremental, modelo em cascata e método ágil. São elas:

1. Especificação do *software*, etapa em que clientes e engenheiros definem o *software* que deve ser produzido e as restrições impostas à sua operação.
2. Desenvolvimento de *software*, etapa em que o *software* é projetado e programado.
3. Validação de *software*, etapa em que o programa é analisado para garantir que seja aquilo de que o cliente precisa.
4. Evolução do *software*, etapa de modificação para refletir a mudança de requisitos tanto do cliente quanto do mercado.

Cada atividade é organizada e descrita de diferentes maneiras e diferentes níveis de detalhe dependendo do tipo de *software* a ser desenvolvido. Por exemplo, enquanto é essencial que um *software* de tempo real de uma aeronave seja completamente especificado antes de começar a ser desenvolvido, um *software* de comércio eletrônico não: a especificação e o programa podem desenvolvidos em conjunto (SOMMERVILLE, 2016, p. 8).

2.1.1 Especificação de *software*

A especificação de *software* faz parte do processo de engenharia de requisitos. Requisitos são a descrição das funcionalidades de um sistema e suas restrições. Os requisitos podem ser divididos entre requisitos funcionais e não funcionais. Requisitos funcionais lidam com as funcionalidades e serviços do sistema, sobre como o sistema deve reagir a determinadas entradas e como deve se comportar em determinadas situações. Requisitos

não funcionais são restrições sobre os serviços e funções oferecidas pelo sistema – confiabilidade, tempo de resposta, uso de memória – e restrições à implementação – capacidade dos dispositivos de entrada e saída, representação dos dados utilizados nas interfaces com outros sistemas. Ainda assim, a distinção entre requisito funcional e não funcional não é sempre clara: um requisito funcional pode gerar (ou limitar) um requisito não funcional e vice-versa (SOMMERVILLE, 2016, cap. 4).

A engenharia de requisitos envolve três processos fundamentais: a elicitação e análise de requisitos, a especificação de requisitos, e a validação de requisitos. Destes processos, o mais relevante para nós será o de especificação de requisitos.

Um projeto possui diversas partes interessadas, sejam elas gerentes do projeto, desenvolvedores, e também usuários e clientes, então existem diversas notações possíveis para os requisitos dependendo do seu público-alvo.

Sommerville (SOMMERVILLE, 2016, p. 104) lista quatro tipos principais de notações para especificação de requisitos: frases em linguagem natural, formulários, notações e modelos gráficos, e especificações matemáticas. Focaremos em especificações matemáticas.

2.1.2 Especificações matemáticas

Segundo Sommerville, especificações matemáticas se baseiam em conceitos matemáticos como máquinas de estado finito ou conjuntos (SOMMERVILLE, 2016, p. 104). O rigor matemático não é necessário para todo projeto de *software*. Seu uso está ligado às categorias de falha de um sistema e suas consequências. Falhas importantes podem afetar inúmeras pessoas de alguma forma, causar perda ou vazamentos de dados, gerar grandes custos econômicos, causar danos ao meio ambiente, ou causar lesões, ou morte humana. Para evitar esses tipos de falha, espera-se que o sistema seja confiável. Os cinco principais atributos que um sistema confiável deve implementar são resumidos por Sommerville (SOMMERVILLE, 2016, p. 259, 260):

- *Disponibilidade.* O sistema funciona e presta serviços úteis a qualquer momento.
- *Confiabilidade.* O sistema presta serviços corretamente, conforme o esperado pelo usuário.
- *Segurança* (safety). O sistema não causa danos a pessoas ou ao seu ambiente.
- *Segurança da informação* (security). O sistema resiste a intrusões acidentais ou propositais.
- *Resiliência.* O sistema mantém a continuidade de seus serviços críticos na presença de eventos disruptivos.

O rigor matemático nos permite expressar os requisitos de confiabilidade do sistema de forma clara, precisa, livre de ambiguidades. Expressamos requisitos formalmente utilizando especificações formais. Segundo Lamsweerde (LAMSWEERDE, 2000, p. 149), uma especificação formal é a expressão, em alguma linguagem formal e em algum nível de abstração, de um conjunto de propriedades que um sistema deve satisfazer. Uma especificação é *formal* se for expressa em uma linguagem composta por regras que determinam a gramática de sentenças bem formadas (a sintaxe), regras para interpretar precisamente essas sentenças no domínio considerado (a semântica), e regras para inferir informações úteis sobre a especificação (a teoria de provas).

Lamsweerde (LAMSWEERDE, 2000, p. 151) distingue os paradigmas de linguagens de especificação formal entre especificações baseadas em históricos, especificações baseadas em estados, especificações baseadas em transições, especificações operacionais, especificações funcionais algébricas, e especificações funcionais de alta ordem. É possível que uma linguagem pertença a mais de um desses paradigmas.

Especificações formais por si só são úteis, claro, pelas vantagens mencionadas acima, mas é possível ir além. Especificações formais são apenas um elemento dos chamados métodos formais. **Métodos formais** é um grande conjunto de abordagens formais para nos certificarmos que a especificação de um sistema não possui falhas, seja analisando-o formalmente para procurar erros e inconsistências com verificação formal, provando que um programa é consistente com sua especificação, ou aplicando no modelo do sistema uma série de transformações que preservem a corretude para gerar um programa (SOMMERVILLE, 2016, p. 273).

2.1.3 Verificação formal

As principais técnicas para verificar sistemas formalmente são verificação de modelos e provas matemáticas.

Verificação de modelos é um processo automático que opera sobre o espaço de estados de um sistema de transição para determinar se ele respeita uma determinada especificação formal. Os principais métodos de verificação são verificação de modelos de estado explícito e verificação de modelos de estado simbólico.

Um **verificador de modelos de estados explícitos** explora todos os estados possíveis descritos por uma especificação. Se o domínio de uma variável for infinito, há infinitos estados para se verificar, o que costuma tornar a verificação de modelos um problema indecidível. Porém, isso não quer dizer que a verificação de modelos com domínios infinitos seja um problema indecidível (BAIER; KATOEN, 2008, p. 78).

Se todas as variáveis do sistema possuírem domínios finitos, a quantidade de estados a se visitar será proporcional ao produtório do tamanho de seus domínios:

$$\prod_{v \in Var} |\mathrm{dom}(v)|$$

onde Var é o conjunto de variáveis do sistema. Este crescimento exponencial da quantidade de estados possíveis é conhecido como o **problema da explosão do espaço de estados**. Se todos os estados forem armazenados na memória primária durante a verificação, o espaço em memória ocupado pelos estados visitados pode exceder a memória disponível. O tempo em que isso ocorre depende da velocidade de geração de novos estados, da quantidade de memória disponível, e de como cada estado é armazenado em memória.

Uma alternativa para a verificação de modelos de estados explícitos é a **verificação de modelos de estados simbólicos**. O problema de explosão de espaço de estados ainda existe, mas ele é significativamente superado, porque a verificação simbólica aproveita regularidades no espaço de estados verificando conjuntos de estados em vez de estados individuais.

O algoritmo de verificação simbólica tradicional utiliza diagramas de decisões binárias (*binary decision diagrams*, ou BDDs) para representar fórmulas booleanas, mas é um algoritmo geral no sentido de que qualquer modelo com domínio finito pode ser traduzido para um modelo com domínio booleano (BURCH et al., 1992). Algoritmos de verificação simbólica mais recentes também utilizam resolvedores de problemas SAT (BIERE et al., 1999) e resolvedores de problemas SMT (ARMANDO; MANTOVANI; PLATANIA, 2006) para acelerar a verificação de modelos.

Por outro lado, **provas matemáticas** evitam o problema de explosão de estados ao lidarem com argumentos matemáticos, e não com estados da especificação. Provas matemáticas são extremamente poderosas devido à sua confiabilidade e robustez. Por exemplo, pode-se provar propriedades sobre um domínio inteiro, inclusive domínios infinitos. Porém, esse processo de argumentação matemática é, no melhor dos casos, um processo semi-automático: mesmo com o uso de provadores de teoremas interativos, o usuário ainda precisa conhecer estratégias para provar teoremas (SOMMERVILLE, 2016, cap. 12).

2.2 CONCEITOS DE LÓGICA TEMPORAL

Lógica modal formaliza a aplicação de modalidades (qualificadores) sobre expressões lógicas de modo que a verdade de uma expressão depende de sua relação com outro aspecto da realidade. A lógica modal básica expande a lógica proposicional acrescentando dois operadores modais unários: $\Diamond$ (diamante) e $\Box$ (caixa). Por exemplo, numa lógica modal básica que estuda a necessidade e possibilidade de expressões, o símbolo $\Box$ é interpretado como "necessariamente" e o símbolo $\Diamond$ é interpretado como "possivelmente" (HUTH; RYAN, 2004).

Lógica temporal é uma lógica modal onde a verdade de uma fórmula está relacionada a um instante no tempo, e o tempo é definido como uma sequência ordenada de eventos:

não há noção explícita de tempo, como um relógio global. A lógica temporal possui dois principais modelos de tempo: o modelo de tempo linear e o modelo de tempo ramificado. Para ilustrar ambos os modelos, examinaremos brevemente as duas principais lógicas para verificação formal de algoritmos: a **lógica temporal linear** (**LTL**, ou *linear temporal logic* em inglês) e a **lógica de árvore computacional** (**CTL**, ou *computation tree logic* em inglês).

LTL (PNUELI, 1977) e CTL (CLARKE; EMERSON, 1982) podem ser usadas para descrever propriedades sobre os mais diversos sistemas, desde programas de computador a sistemas biológicos, contanto que sejam restritos ao tempo discreto. Em geral, estas lógicas são usadas para descrever propriedades de **sistemas cíclicos não terminais**, como sistemas operacionais ou servidores, onde alcançar um estado terminal é indesejado. Isso significa que noções de corretude de um sistema cíclico devem ser expressas por **implicações temporais** ($\rightsquigarrow$) (PNUELI, 1977). Por exemplo, a propriedade de um servidor sempre responder a um pedido após recebê-lo pode ser expressa informalmente por

$$\text{para todo pedido } p : \{\text{pedido } p \text{ é recebido}\} \rightsquigarrow \{\text{resposta a } p \text{ é enviada}\}$$

Para definirmos sistemas discretos e suas propriedades em lógica temporal, precisamos formalizá-los. Enquanto propriedades são expressas em alguma lógica temporal, os sistemas discretos costumam ser descritos através de **sistemas de transição**: um conjunto de estados $\mathcal{S}$, uma relação de transições $\mathcal{R}$ entre estes estados tal que $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, e um conjunto de estados iniciais $\mathcal{I}$ tal que $\mathcal{I} \subseteq \mathcal{S}$.

Veremos que TLA está próxima de LTL no sentido de ambas serem lógicas de tempo linear, mas não requer um sistema de transição *a priori*, pois TLA em si descreve um sistema de transição.

3 LÓGICA TEMPORAL DE AÇÕES E TLA+

A lógica temporal de ações (TLA) é o conceito central deste trabalho, e é a partir dela que a linguagem TLA$^+$ foi criada. Por questões de espaço e clareza, focamos o capítulo no que faz a lógica temporal de ações se destacar das demais lógicas, mas sem focar em sua semântica formal como já elaborado no artigo que a deu origem (LAMPORT, 1994) na Seção 3.1. Além disso, passaremos pela linguagem TLA$^+$ na Seção 3.2, que adiciona módulos e estruturas de dados à sintaxe de TLA, e, por fim, explicamos o verificador de modelos de TLA$^+$ (TLC) e seu funcionamento na Seção 3.3.

Neste capítulo evitaremos, principalmente na seção sobre TLA$^+$, expor operadores ou módulos de TLA$^+$ que não são possíveis de serem verificados com TLC. O ecossistema de TLA$^+$ inclui o assistente de provas TLAPS (TLA$^+$ *Proof System*) que auxilia o usuário a provar os teoremas mais diversos sem as restrições presentes no TLC (CHAUDHURI et al., 2010). Não focaremos neste tópico.

3.1 LÓGICA TEMPORAL DE AÇÕES

A lógica temporal de ações é uma lógica para especificação formal de algoritmos, principalmente algoritmos concorrentes ou distribuídos. Nesta lógica, execuções são chamadas de comportamentos. Um comportamento é uma sequência de estados, onde um estado é uma atribuição de valores a variáveis. Uma especificação em TLA é uma fórmula booleana que representa um conjunto de comportamentos. Comportamentos válidos são comportamentos que satisfazem a especificação, ou seja, que fazem com que esta fórmula seja avaliada como verdadeira.

Diferente de lógicas temporais como *Linear Temporal Logic* (LTL) ou *Computation Tree Logic* (CTL), TLA se destaca ao evitar o uso de operadores temporais na definição de fórmulas. Como explica Lamport:

> "TLA difere de outras lógicas temporais porque é baseada no princípio de que a lógica temporal é um mal necessário que deve ser evitado o máximo possível. Fórmulas temporais tendem a ser mais difíceis de entender do que fórmulas de lógica de primeira ordem comum, e o raciocínio de lógica temporal é mais complicado do que o raciocínio matemático (não-modal) comum."(LAMPORT, 1994, p. 46, tradução nossa)[1]

Além disso, TLA é uma lógica de ações. Enquanto outras lógicas temporais constroem fórmulas com predicados sobre estados, TLA constrói fórmulas com ações, predicados

[1] "TLA differs from other temporal logics because it is based on the principle that temporal logic is a necessary evil that should be avoided as much as possible. Temporal formulas tend to be harder to understand than formulas of ordinary first-order logic, and temporal logic reasoning is more complicated than ordinary mathematical (nonmodal) reasoning."

sobre *pares* de estados, mais especificamente, sobre determinado estado e o próximo em um comportamento, representando transições atômicas. Se uma ação $\mathcal{A}$ é verdadeira para os estados s e t, ou seja, o predicado é verdadeiro para esse par de estados, chamamos essa transição entre s e t de um passo $\mathcal{A}$.

Para diferenciar entre o valor de uma variável em um estado e no próximo utiliza-se o operador linha ($'$). Por exemplo, se x representa um número, x' representa este número no próximo estado. A ação onde o próximo valor de x é o valor de x no estado atual somado a 1 seria expressa pela fórmula $x' = x + 1$. Destacamos que igualdade aqui é uma igualdade matemática, e não uma atribuição como em linguagens de programação.[2] Além disso, não é permitido mais de um operador linha por variável, por exemplo, escrever x'' é proibido.

3.1.1 Um contador

Para exemplificar, descreveremos agora um contador: um sistema onde um número natural x é somado a 1 sem parar. Se nosso contador sempre começa do zero, então a fórmula $x = 0$ é verdadeira no primeiro estado de todo comportamento válido. Nomeando esta fórmula de *Ini*, escrevemos $Ini \triangleq x = 0$ (lê-se "Ini é definida como igual a $x = 0$").

O contador só possui uma transição possível, uma ação onde o próximo valor de x é igual ao anterior somado a 1: $x' = x + 1$. Nomeando esta ação de *Prox* ("próximo"), escrevemos $Prox \triangleq x' = x + 1$.

Um comportamento é composto por vários estados, enquanto uma ação só menciona um estado e o próximo. Para afirmarmos sobre todos os estados de um comportamento, precisamos introduzir o operador "sempre" (também conhecido como "daqui para frente"). Se F é um predicado, $\Box F$ afirma que F é verdadeiro em determinado estado e em todo estado posterior. A Figura 1 ilustra a intuição da semântica do operador $\Box$.

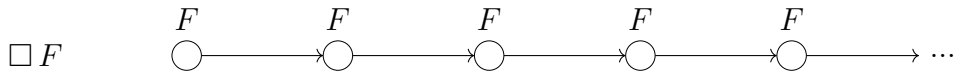

Figura 1 – Intuição da semântica do operador $\Box$

Como queremos dizer que a ação *Prox* é verdadeira em todas as transições de um comportamento (do estado inicial para frente) escrevemos $\Box Prox$.

As fórmulas separadas *Ini* e $\Box Prox$ não formam uma especificação: precisamos unir-las. Para afirmarmos que o predicado inicial *Ini* é verdadeiro no estado inicial e a ação *Prox* é verdadeira daqui para frente ($\Box Prox$), fazemos a conjunção de ambas as fórmulas. Nomeando de *Spec* esta fórmula que representa nossa especificação, teremos $Spec \triangleq Ini \wedge \Box Prox$, resultando na especificação da figura 2.

[2] Seria possível escrever $x = x' - 1$ ou $x' - 1 = x$, mas é recomendado manter a variável-linha do lado esquerdo da igualdade (LAMPORT, 1994, p. 22).

$$Ini \triangleq x = 0$$
$$Prox \triangleq x' = x + 1$$
$$Spec \triangleq Ini \wedge \Box Prox$$

Figura 2 – Especificação de um contador único

3.1.2 Contadores independentes

Imaginemos, agora, que o contador único de x, agora representado por $SpecX$, é acompanhado por um contador y idêntico a ele, representado por $SpecY$, como vemos na figura 3.

$$IniX \triangleq x = 0 \qquad IniY \triangleq y = 0$$
$$ProxX \triangleq x' = x + 1 \qquad ProxY \triangleq y' = y + 1$$
$$SpecX \triangleq IniX \wedge \Box ProxX \qquad SpecY \triangleq IniY \wedge \Box ProxY$$

Figura 3 – Especificações $SpecX$ e $SpecY$

Queremos especificar um sistema que inclui dois contadores funcionando independentemente. A união das especificações $SpecX$ e $SpecY$ em $SpecXY$ requer alguns cuidados:

- Esperamos que no estado inicial ambos x e y sejam iguais a zero, ou seja, uma conjunção das fórmulas $IniX$ e $IniY$ em $IniXY \triangleq IniX \wedge IniY$, que equivale a

$$IniXY \triangleq x = 0 \wedge y = 0$$

- A união de ações das especificações acontece de outra forma. A lógica temporal de ações descreve execuções concorrentes como passos atômicos intercalados (LAMPORT, 1994, p. 44). Se temos dois contadores independentes executando em passos intercalados, espera-se que, ou o contador de x mude a variável x, ou que o contador de y mude a variável y. Para simplificar a especificação, não permitiremos passos onde x e y mudam simultaneamente. Assim, representamos a opção entre as duas ações utilizando a disjunção $ProxX \vee ProxY$. Só que a disjunção não basta: precisamos expressar que, quando determinada ação acontece, o valor de outras variáveis do sistema permanece o mesmo. Caso contrário, seria como se estivéssemos dizendo "ou valor de x' é $x + 1$ e o valor de y' é um valor qualquer, ou o valor de y' é $y + 1$ e o valor de x' é um valor qualquer". Portanto, as ações $ProxX$ e $ProxY$ modificadas tornam-se, respectivamente, $MudaX \triangleq x' = x + 1 \wedge y' = y$ e $MudaY \triangleq y' = y + 1 \wedge x' = x$, resultando na fórmula

$$ProxXY \triangleq MudaX \vee MudaY$$

A figura 4 demonstra a especificação *SpecXY* completa até agora.

$$
\begin{aligned}
\mathit{IniXY} &\triangleq x = 0 \wedge y = 0 \\
\mathit{MudaX} &\triangleq x' = x + 1 \wedge y' = y \\
\mathit{MudaY} &\triangleq y' = y + 1 \wedge x' = x \\
\mathit{ProxXY} &\triangleq \mathit{MudaX} \vee \mathit{MudaY} \\
\mathit{SpecXY} &\triangleq \mathit{IniXY} \wedge \Box \mathit{ProxXY}
\end{aligned}
$$

Figura 4 – Especificação de dois contadores independentes

3.1.3 Passos balbuciantes

Façamos uma breve visualização do nosso sistema com os dois contadores funcionando lado a lado. A figura 5 mostra algumas possibilidades de alternância de passos permitidos por *SpecXY*.

$$
\begin{aligned}
&[\,x = 0\,, \,y = 0\,] \rightarrow [\,x = 1\,, y = 0] \rightarrow [\,x = 2\,, y = 0] \rightarrow [\,x = 3\,, y = 0] \rightarrow ... \\
&[\,x = 0\,, \,y = 0\,] \rightarrow [x = 0, \,y = 1\,] \rightarrow [x = 0, \,y = 2\,] \rightarrow [\,x = 1\,, y = 2] \rightarrow ... \\
&[\,x = 0\,, \,y = 0\,] \rightarrow [\,x = 1\,, y = 0] \rightarrow [x = 1, \,y = 1\,] \rightarrow [\,x = 2\,, y = 1] \rightarrow ...
\end{aligned}
$$

Figura 5 – Alguns comportamentos que satisfazem a especificação *SpecXY* (variáveis que mudaram em destaque).

Se imaginarmos ambos os contadores funcionando lado a lado, e ignorarmos um desses lados, é evidente para nós que o lado não ignorado deveria funcionar como um contador único. Por exemplo, se decidirmos ignorar o contador y e focarmos apenas no contador x, espera-se que o valor de x mude como na especificação *SpecX*. Para ilustrar, se escondermos a variável y dos comportamentos da figura 5, teremos os comportamentos da figura 6.

$$
\begin{aligned}
&[\,x = 0\,] \rightarrow [\,x = 1\,] \rightarrow [\,x = 2\,] \rightarrow [\,x = 3\,] \rightarrow ... \\
&[\,x = 0\,] \rightarrow [x = 0] \rightarrow [x = 0] \rightarrow [\,x = 1\,] \rightarrow ... \\
&[\,x = 0\,] \rightarrow [\,x = 1\,] \rightarrow [x = 1] \rightarrow [\,x = 2\,] \rightarrow ...
\end{aligned}
$$

Figura 6 – Comportamentos da figura 5 com variável y escondida (variáveis que mudaram em destaque).

Constatamos que no segundo e terceiro comportamentos da figura 6, a variável x não muda em alguns passos. Porém, a especificação *SpecX* espera que x sempre mude:

$\Box ProxX$ expressa que a ação $ProxX$ é verdade em toda transição. Ou seja, escondendo y, a especificação $SpecXY$ deixa de respeitar $SpecX$. Do mesmo modo, escondendo x, $SpecXY$ deixa de respeitar $SpecY$.

A correção deste problema será adicionar passos onde a variável x não muda à especificação $SpecX$. Chamaremos esta especificação $SpecX$ com passos balbuciantes de ${SpecX}_{balb}$. Fazendo o equivalente para $SpecY$, temos seu resultado na figura 7.

$$
\begin{aligned}
{SpecX}_{balb} &\triangleq IniX \wedge \Box(ProxX \vee x' = x) \\
{SpecY}_{balb} &\triangleq IniY \wedge \Box(ProxY \vee y' = y)
\end{aligned}
$$

Figura 7 – Especificações $SpecX$ e $SpecY$ modificadas para ${SpecX}_{balb}$ e ${SpecY}_{balb}$

E se uníssemos dois, ou quatro, ou oito contadores independentes? Cada contador independente individual também teria de aceitar comportamentos com passos em que não somente uma ou algumas variáveis não mudem, e sim **nenhuma** variável mude. Em geral, toda especificação em TLA deve permitir comportamentos com passos em que nenhuma variável muda, passos chamados de passos balbuciantes (*stuttering steps*).

Nomeando a fórmula $SpecXY$ com passos balbuciantes de ${SpecXY}_{balb}$, definimos:

$$
{SpecXY}_{balb} \triangleq IniXY \wedge \Box(ProxXY \vee (x' = x \wedge y' = y))
$$

Por questões de notação, a expressão do passo balbuciante pode se alongar, sendo que estamos apenas listando variáveis que se mantêm iguais. Podemos expressar sequências ordenadas de valores com tuplas: $(x' = x \wedge y' = y) \equiv (\langle x', y' \rangle = \langle x, y \rangle) \equiv (\langle x, y \rangle' = \langle x, y \rangle)$.

$$
{SpecXY}_{balb} \triangleq IniXY \wedge \Box(ProxXY \vee \langle x, y \rangle' = \langle x, y \rangle)
$$

Podemos ir além evitando a repetição de $\langle x, y \rangle$ na fórmula acima ao utilizar a seguinte notação, onde $\mathcal{A}$ é uma ação, e v é uma variável ou tupla de variáveis.

$$
[\mathcal{A}]_v \equiv \mathcal{A} \vee v' = v
$$

Deste modo, simplificamos as fórmulas ${SpecX}_{balb}$, ${SpecY}_{balb}$ e ${SpecXY}_{balb}$ como demonstrado na figura 8.

$$
\begin{aligned}
{SpecX}_{balb} &\triangleq \quad IniX \wedge \Box[ProxX]_x \\
{SpecY}_{balb} &\triangleq \quad IniY \wedge \Box[ProxY]_y \\
{SpecXY}_{balb} &\triangleq IniXY \wedge \Box[ProxXY]_{\langle x,y \rangle}
\end{aligned}
$$

Figura 8 – Especificações ${SpecX}_{balb}$, ${SpecY}_{balb}$ e ${SpecXY}_{balb}$ com novas notações

3.1.4 Comportamentos infinitos e o universo de comportamentos

O primeiro contador que vimos (Figura 2) modifica sua variável x para sempre e não dá passos balbuciantes. Sua especificação *SpecX* descreve um único comportamento com um número infinito de estados. Acrescentando passos balbuciantes a *SpecX*, a especificação $SpecX_{balb}$ passa a descrever *infinitos* comportamentos com um número infinito de estados, porque um passo onde nada acontece pode ser dado um número indeterminado de vezes.

Pensemos, então, em um contador finito que soma 1 a x um determinado número de vezes. A especificação deste contador descreve um número finito de estados. Após incluir passos balbuciantes a este contador finito, poderíamos supor que comportamentos finitos existiriam quando a ação de x mudar acontecesse até x alcançar seu valor máximo. Por exemplo, se x só alcançasse 2, um comportamento possível deste contador seria

$$[x = 0] \rightarrow [x = 1] \rightarrow [x = 2]$$

Porém, em TLA esta suposição estaria incorreta. Na realidade, todo comportamento é infinito, pois todo comportamento representa o avanço do universo, não só o avanço de determinado sistema. Um estado de um comportamento, representando um estado potencial de todo o universo, torna-se uma atribuição de valores a *todas* as variáveis do universo. Quando escrevemos determinada especificação com variáveis x e y, por exemplo, estamos as destacando como as variáveis que nos importam. Variáveis não mencionadas podem assumir qualquer valor.

Como fazemos, então, para especificar sistemas ou algoritmos que terminam? Um sistema que termina é um sistema onde todas as suas variáveis param de mudar em um ponto específico e assim continuam para sempre. Lamport explica:

> A observação de que um comportamento pode representar a execução de dois ou mais programas independentes explica o motivo de representarmos execuções que terminam ou não com comportamentos infinitos. O término de um programa significa que ele parou; não significa que o universo inteiro parou. Uma execução que termina é representada por um comportamento em que todas as variáveis do programa param de mudar (LAMPORT, 1994, p. 15, tradução nossa).[3]

Como estamos especificando um sistema que faz parte do universo, os passos balbuciantes seriam passos que permitem que as mudanças do sistema especificado possam intercalar com mudanças do universo de infinitas maneiras.

Uma consequência de acrescentar passos balbuciantes é que toda fórmula temporal que escrevermos terá de ser invariante sob balbuciação (*invariant under stuttering*). Uma

[3] "The observation that a single behavior can represent an execution of two or more noninteracting programs explains why we represent terminating as well as nonterminating executions by infinite behaviors. Termination of a program means that it has stopped; it does not mean that the entire universe has come to a halt. A terminating execution is represented by a behavior in which eventually all of the program's variables stop changing."

fórmula temporal F é invariante sob balbuciação se e somente se adicionar ou remover um passo balbuciante de um comportamento σ não afeta se σ satisfaz F. Caso contrário, por exemplo, determinadas quantidades de passos balbuciantes satisfariam F, e outras quantidades, não. Seria como se o fato de nada acontecer no universo tivesse que seguir alguma regra, ou como se a especificação mandasse em como o universo evolui.

Um predicado P é invariante sob balbuciação, pois afirma sobre um único estado, então P não é afetado ao acrescentar ou remover passos balbuciantes de comportamentos.

Em geral, a fórmula $\Box\mathcal{A}$, onde $\mathcal{A}$ é uma ação, não é invariante sob balbuciação. Por exemplo, a fórmula $\Box(x' = x+1)$ não aceita passos onde x não muda. Como toda fórmula temporal precisa ser invariante sob balbuciação, $\Box\mathcal{A}$ não é uma fórmula temporal válida em TLA.

Uma fórmula $[\mathcal{A}]_v$ também não é invariante sob balbuciação, portanto, não é fórmula temporal válida em TLA. Ela descreve uma transição *somente do primeiro passo* de um comportamento. Por exemplo, a fórmula $[x' = x + 1]_x$ é satisfeita pelo seguinte comportamento onde o primeiro passo é balbuciante:

$$[x = 0] \rightarrow [x = 0] \rightarrow [x = \sqrt{2}] \rightarrow ...$$

Porém, não é satisfeita por um comportamento sem este passo:

$$[x = 0] \rightarrow [x = \sqrt{2}] \rightarrow ...$$

A fórmula $\Box[\mathcal{A}]_v$ é invariante sob balbuciação porque lida com todas as transições de um comportamento. Por exemplo, $\Box[x' = x + 1]_x$ é satisfeita por um comportamento se e somente se todo passo que mudar x for um passo $x' = x + 1$ (LAMPORT, 2002, p. 90).

Assim, fórmulas temporais invariantes sob balbuciação geralmente três formas:

- P, onde P é um predicado de estado, ou seja, um predicado que descreve propriedades de um único estado;
- $\Box[\mathcal{A}]_v$, onde $\mathcal{A}$ é uma ação e v é uma variável ou uma tupla de variáveis;
- Combinação das duas formas acima utilizando o operador $\Box$ e conectivos booleanos ($\vee$, $\neg$, etc.)(LAMPORT, 2002, p. 90)

3.1.5 Vivacidade

Lembramos que uma especificação é uma fórmula temporal, ou seja, uma fórmula booleana que aceita ou rejeita comportamentos. O predicado de estados iniciais *Ini* aceita um comportamento se *Ini* for verdadeira para o primeiro estado de um comportamento. A relação de transições $\Box[Prox]_v$ aceita um comportamento se as mudanças de valores das variáveis do comportamento inteiro condizem com as transições esperadas em $\Box[Prox]_v$.

Com a adição de passos balbuciantes, toda especificação pode parar a qualquer momento ao sempre aceitar comportamentos onde nada acontece.

Assim, uma especificação com passos balbuciantes apenas diz o que é possível de acontecer, e não o que tem de acontecer. Simplificadamente, chamamos de **segurança** ("safety" em inglês) de um sistema tudo aquilo que ele pode fazer, e chamamos de **vivacidade** ("liveness" em inglês) de um sistema tudo aquilo que ele tem que fazer.

Se quisermos rejeitar comportamentos onde nada acontece, teremos que adicionar alguma nova fórmula temporal V a *Spec* na forma $Spec \triangleq Ini \wedge \Box[Prox]_v \wedge V$. O papel de V é garantir que algo seja verdade pelo menos uma vez em todo comportamento, ou seja, seu papel é garantir vivacidade sobre o sistema ou sobre determinadas ações.

Dada uma fórmula temporal arbitrária F, se queremos garantir que F seja verdadeira pelo menos uma vez em um comportamento, dizemos que F é *futuramente*[4] verdadeira. Formalmente, se a fórmula F é verdadeira ou no estado atual, ou em algum estado posterior, escrevemos $\Diamond F$, segundo a equivalência

$$\Diamond F \equiv \neg\Box\neg F$$

A Figura 9 ilustra a intuição da semântica do operador $\Diamond$ sobre uma fórmula temporal F.

Figura 9 – Intuição da semântica da fórmula $\Diamond F$

Estamos buscando um formato para a fórmula temporal V. Como vimos na subseção anterior, fórmulas temporais invariantes sob balbuciação vêm em dois formatos: predicado P ou fórmula $\Box[\mathcal{A}]_v$ (além de suas combinações por operadores lógicos). Se a fórmula V for definida como $\Diamond F$, e substituirmos F por um predicado P ou $\Box[\mathcal{A}]_v$, teríamos $\Diamond P$ ou $\Diamond\Box[\mathcal{A}]_v$. A primeira fórmula garante que o predicado P seja verdadeiro pelo menos uma vez, mas a segunda fórmula *não garante* que $\mathcal{A}$ seja verdadeira pelo menos uma vez. Como $\Box[\mathcal{A}]_v$ é verdadeira também para comportamentos onde nada acontece, $\Diamond\Box[\mathcal{A}]_v$ não garante que $\mathcal{A}$ seja verdadeira em algum momento, ou seja, não garante vivacidade.

Em vez de $\Diamond\Box[\mathcal{A}]_v$, tentemos $\Diamond\neg\Box[\neg\mathcal{A}]_v$, pois $\neg\Box[\neg\mathcal{A}]_v$ também é fórmula temporal invariante sob balbuciação.

$$\begin{aligned}
\Diamond\neg\Box[\neg\mathcal{A}]_v &\equiv \Diamond(\neg\Box(\neg\mathcal{A} \vee v' = v)) && \text{pela definição de } [A]_v \\
&\equiv \Diamond(\Diamond\neg(\neg\mathcal{A} \vee v' = v)) && \text{pois } \neg\Box F \equiv \Diamond\neg F \\
&\equiv \Diamond\Diamond(\neg\neg\mathcal{A} \wedge v' \neq v) && \text{pois } \neg(A \vee B) \equiv (\neg A \wedge \neg B) \\
&\equiv \Diamond(\mathcal{A} \wedge v' \neq v) && \text{pois } \Diamond\Diamond F \equiv \Diamond F \text{ e } \neg\neg\mathcal{A} \equiv \mathcal{A}
\end{aligned}$$

[4] A semântica dos operadores $\Box$ e $\Diamond$ é mais complexa do que as palavras "sempre" e "futuramente" expressam. Nomes mais precisos (e mais extensos) para os operadores $\Box$ e $\Diamond$ podem ser "sempre daqui para frente" e "em algum momento daqui para frente", respectivamente.

Isso resolve nosso problema de vivacidade sobre $\mathcal{A}$ porque, além de $\Diamond(\mathcal{A} \wedge v' \neq v)$ ser uma fórmula temporal invariante sob balbuciação, ela garante que a ação $\mathcal{A}$ acontece. Dizemos que uma ação $\mathcal{A}$ **acontece** quando uma ação $\mathcal{A}$ é verdadeira em um par de estados e v mudou: $(\mathcal{A} \wedge v' \neq v)$. Assim, a fórmula $\Diamond(\mathcal{A} \wedge v' \neq v)$ expressa que, agora ou futuramente, a ação $\mathcal{A}$ acontece.

Introduzindo uma notação similar a $[\mathcal{A}]_v$, temos:[5]

$$\langle \mathcal{A} \rangle_v \equiv (\mathcal{A} \wedge v' \neq v)$$

Assim, quando se trata da fórmula de vivacidade V, as fórmulas temporais que a compõem terão duas formas: um predicado P ou fórmula $\Diamond\langle \mathcal{A} \rangle_v$, além, claro, de suas combinações por operadores lógicos.

Expressar que a fórmula temporal F é verdadeira em algum momento pode ser insuficiente. Como já se pode ter percebido, é possível compor os operadores $\Box$ e $\Diamond$ para expressar outras frequências com as quais F é verdadeira.

- $\Box\Diamond F$ ("sempre futuramente" F): F é infinitamente frequentemente verdadeira. Dizemos, informalmente, que F é *repetidamente* verdadeira.

  O operador $\Box\Diamond$ se distribui sobre $\vee$: $\Box\Diamond(F \vee G) \equiv (\Box\Diamond F) \vee (\Box\Diamond G)$

- $\Diamond\Box F$ ("futuramente sempre" F): F passa a ser verdadeira em determinado estado, e continua a ser verdadeira sem parar. Dizemos, informalmente, que F é *continuamente* verdadeira.

  O operador $\Diamond\Box$ se distribui sobre $\wedge$: $\Diamond\Box(F \wedge G) \equiv (\Diamond\Box F) \wedge (\Diamond\Box G)$

Voltando à nossa especificação dos contadores independentes da figura 4, gostaríamos de expressar que o sistema não para de funcionar. Podemos expressar esta condição afirmando que a ação $ProxXY$ acontece não apenas futuramente, mas sim *infinitamente frequentemente* ($\Box\Diamond$): $\Box\Diamond\langle ProxXY \rangle_{\langle x,y \rangle}$.

A fórmula $\Box\Diamond\langle ProxXY \rangle_{\langle x,y \rangle}$ garante que o sistema nunca para, mas é uma fórmula injusta, pois $ProxXY$ equivale a $MudaX \vee MudaY$. Devido à disjunção entre as ações, tal fórmula permite comportamentos, por exemplo, onde somente $MudaX$ acontece infinitamente frequentemente.

Mudaremos nossa especificação para expressarmos que ambos $MudaX$ e $MudaY$ aconteçam infinitamente frequentemente: em vez de aplicarmos $\Box\Diamond$ sobre $ProxXY$, os aplicaremos sobre cada sub-ação de $ProxXY$, resultando em

$$\Box\Diamond\langle MudaX \rangle_{\langle x,y \rangle} \wedge \Box\Diamond\langle MudaY \rangle_{\langle x,y \rangle}$$

[5] Nota-se que o formato da notação $[\mathcal{A}]_v$ para ações que balbuciam reflete o formato de $\Box$, e que o formato da notação $\langle \mathcal{A} \rangle_v$ para ações que acontecem reflete o formato de $\Diamond$, ou seja, notações que refletem o formato dos operadores temporais que as acompanham.

Finalmente, a inclusão da propriedade de vivacidade na fórmula $SpecXY_{balb}$ resulta na fórmula $SpecVivaXY_{balb}$ tal que:

$$VivacidadeXY \triangleq \Box\Diamond\langle MudaX\rangle_{\langle x,y\rangle} \wedge \Box\Diamond\langle MudaY\rangle_{\langle x,y\rangle}$$
$$SpecVivaXY_{balb} \triangleq SpecXY_{balb} \wedge VivacidadeXY$$

Tendo alcançado uma especificação de contadores que não só podem avançar, mas têm que avançar, já adiantamos que toda execução de um algoritmo pode ser especificada pela conjunção de fórmulas de segurança e vivacidade (ALPERN; SCHNEIDER, 1985), ou seja, podemos descrever um sistema completamente usando esses dois tipos de fórmulas.

3.1.6 Justiça

Na seção anterior (3.1.5) mencionamos que a fórmula de vivacidade $\Box\Diamond\langle ProxXY\rangle_{\langle x,y\rangle}$ podia ser injusta ao permitir que somente uma das ações acontecessem infinitamente frequentemente. A questão aqui é que fórmulas arbitrárias de vivacidade podem ser arbitrariamente sutis. Uma solução que Lamport propõe em TLA é de, sempre que possível, expressar vivacidade em termos de justiça, como explica Lamport em (LAMPORT, 2002, p. 107, tradução nossa):

> Propriedades de vivacidade podem ser sutis. Expressá-las com fórmulas temporais *ad hoc* pode levar a erros. Especificaremos vivacidade como a conjunção de propriedades de justiça fraca e/ou forte sempre que possível, e é quase sempre possível. Ter um modo uniforme de expressar vivacidade facilita a compreensão de especificações.[6]

Assim, escrever vivacidade sem utilizar justiça é relegado a casos especiais (LAMPORT, 2002, p. 113).

Simplificadamente, uma propriedade de justiça ("*fairness*" em inglês) afirma que algo é frequentemente verdadeiro. Porém, expressar que algo é frequentemente verdadeiro pode ser uma propriedade forte demais, uma propriedade "irrealista". Assim, propriedades de justiça costumam ser expressas acrescentando condições para que algo aconteça frequentemente, propriedades chamadas de condições de justiça. Por exemplo, "todo agente acessará a seção crítica frequentemente" é mais forte e menos realista do que a propriedade "todo agente, *se puder acessar a seção crítica frequentemente*, acessará a seção crítica frequentemente" (HUTH; RYAN, 2004).

Como conhecemos operadores da lógica temporal, expressamos estes tipos de justiça, ainda informalmente, na lógica temporal de ações da seguinte maneira:

[6] "Liveness properties can be subtle. Expressing them with ad hoc temporal formulas can lead to errors. We will specify liveness as the conjunction of weak and/or strong fairness properties whenever possible and it almost always is possible. Having a uniform way of expressing liveness makes specifications easier to understand."

- Condição de justiça fraca: se uma ação é habilitada em algum momento e continua habilitada para sempre – habilitada sem interrupções –, então esta ação acontece infinitamente frequentemente.

$$\Diamond\Box\{\text{ação habilitada}\} \Rightarrow \Box\Diamond\{\text{ação acontece}\}$$

- Condição de justiça forte: se uma ação é habilitada infinitamente frequentemente – habilitada com possíveis interrupções –, então esta ação acontece infinitamente frequentemente.

$$\Box\Diamond\{\text{ação habilitada}\} \Rightarrow \Box\Diamond\{\text{ação acontece}\}$$

- Justiça incondicional: uma ação acontece infinitamente frequentemente.

$$\Box\Diamond\{\text{ação acontece}\}$$

Justiça incondicional implica em justiça forte, e justiça forte implica em justiça fraca.

Podemos observar que justiça depende de um conceito de habilitar uma ação. Uma ação ser habilitada ("*enabled*" em inglês) significa que existe um próximo estado possível. Múltiplas ações podem estar habilitadas em determinado momento, mas somente uma delas acontece naquele momento. Em outras palavras, uma ação ser habilitada num instante seria dizer que uma ação é possível de acontecer naquele instante.

Lamport recomenda que fórmulas de justiça sejam escritas sobre sub-ações da relação de transições da especificação (LAMPORT, 2002, p. 110). Uma ação $\mathcal{A}$ é sub-ação de uma ação $\mathcal{P}$ se, e somente se, $\mathcal{A} \Rightarrow \mathcal{P}$. Como a relação de transição *Prox* é frequentemente definida como a disjunção de diversas ações, por exemplo, $Prox \triangleq A_1 \vee A_2 \vee ... \vee A_N$, sempre definiremos justiça sobre A_1, sobre A_2, etc.

Definamos os três tipos de justiça formalmente em TLA: dada uma ação $\mathcal{A}$ e tupla de variáveis v, a ação $\mathcal{A}$ acontecer sendo expressa como $\langle\mathcal{A}\rangle_v$, e uma ação $\mathcal{A}$ ser habilitada sendo expressa como *Enabled* $\langle\mathcal{A}\rangle_v$:

- Condição de justiça fraca é expressa logicamente como

$$\Diamond\Box Enabled\ \langle\mathcal{A}\rangle_v \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_v$$

  TLA denota a condição de justiça fraca sobre $\mathcal{A}$ e v por $\mathrm{WF}_v(\mathcal{A})$

- Condição de justiça forte.

$$\Box\Diamond Enabled\ \langle\mathcal{A}\rangle_v \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_v$$

  TLA denota a condição de justiça forte sobre $\mathcal{A}$ e v por $\mathrm{SF}_v(\mathcal{A})$

- Justiça incondicional.

$$\Box\Diamond\langle\mathcal{A}\rangle_v$$

  TLA não possui notação especial para justiça incondicional.

Especifiquemos as condições de justiça dos contadores independentes $SpecXY_{balb}$ da figura 8. Assim como a fórmula temporal de vivacidade da seção 3.1.5, a fórmula temporal para justiça faz parte da conjunção de fórmulas temporais da especificação. Por exemplo, podemos estruturar a especificação $SpecJustaXY$ a partir de $SpecXY_{balb}$ tal que

$$SpecJustaXY_{balb} \triangleq SpecXY_{balb} \wedge JusticaXY_{balb}$$

Evidentemente, quando especificamos um sistema, esperamos que ele satisfaça determinadas propriedades. Neste caso, esperamos que ambos os contadores de x e de y cresçam indefinidamente. Poderíamos esperar que somente um dos contadores crescesse indefinidamente, ou que nenhum dos dois crescesse indefinidamente. Independentemente de como o sistema deve se comportar, especificar justiça requer que analisemos as frequências com as quais as ações do sistema são habilitadas.

As ações $MudaX$ e $MudaY$ são sempre habilitadas: nada impede que passos $MudaX$ e $MudaY$ sejam dados. Como $ProxXY$ é a disjunção de $MudaX$ e $MudaY$, então $ProxXY$ também é sempre habilitada.

Uma fórmula F qualquer ser sempre verdadeira implica que F é infinitamente frequentemente verdadeira, e que F é verdadeira de um momento para frente. Como todas as ações do sistema sempre são habilitadas, tanto a condição de justiça fraca quanto forte garantirá que estas ações aconteçam infinitamente frequentemente, como demonstrado na derivação da figura 10 para uma ação $\mathcal{A}$ que é sempre habilitada.

$$\begin{aligned}
\mathrm{WF}_v(\mathcal{A}) &\equiv \Diamond\Box \mathit{Enabled}\ \langle\mathcal{A}\rangle_v \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_v && \text{Pela definição de } \mathrm{WF}_v(\mathcal{A}) \\
&\equiv \Box \mathit{Enabled}\ \langle\mathcal{A}\rangle_v \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_v && \text{Pois } \Box \mathit{Enabled}\ \langle\mathcal{A}\rangle_v \Rightarrow \Diamond\Box \mathit{Enabled}\ \langle\mathcal{A}\rangle_v \\
&\equiv \mathrm{TRUE} \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_v && \text{Pois } \Box \mathit{Enabled}\ \langle\mathcal{A}\rangle_v \text{ é verdade} \\
&\equiv \Box\Diamond\langle\mathcal{A}\rangle_v && \text{Pois } (\mathrm{TRUE} \Rightarrow P) \equiv P
\end{aligned}$$

$$\begin{aligned}
\mathrm{SF}_v(\mathcal{A}) &\equiv \Box\Diamond \mathit{Enabled}\ \langle\mathcal{A}\rangle_v \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_v && \text{Pela definição de } \mathrm{SF}_v(\mathcal{A}) \\
&\equiv \Box \mathit{Enabled}\ \langle\mathcal{A}\rangle_v \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_v && \text{Pois } \Box \mathit{Enabled}\ \langle\mathcal{A}\rangle_v \Rightarrow \Diamond\Box \mathit{Enabled}\ \langle\mathcal{A}\rangle_v \\
&\equiv \mathrm{TRUE} \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_v && \text{Pois } \Box \mathit{Enabled}\ \langle\mathcal{A}\rangle_v \text{ é verdade} \\
&\equiv \Box\Diamond\langle\mathcal{A}\rangle_v && \text{Pois } (\mathrm{TRUE} \Rightarrow P) \equiv P
\end{aligned}$$

Figura 10 – Derivação de $\Box\Diamond\langle\mathcal{A}\rangle_v$ quando $\Box \mathit{Enabled}\ \langle\mathcal{A}\rangle_v$

Quando especificamos um sistema onde as condições de justiça fraca e forte de uma determinada ação obtêm o mesmo comportamento, Lamport recomenda que se aplique a condição mais fraca (LAMPORT, 2002, p. 107):

> Justiça forte pode ser mais difícil de se implementar do que justiça fraca, e é um requisito menos comum. Uma condição de justiça forte deve ser usada em uma especificação apenas se necessário. Quando justiça forte e fraca são equivalentes, a propriedade de justiça deve ser escrita como justiça fraca[7].

As condições de justiça sobre a ação $ProxXY_{balb}$ nos levam de volta à situação encontrada no final da seção 3.1.5, pois ambas equivalem a $\Box\Diamond\langle ProxXY_{balb}\rangle_{\langle x,y\rangle}$. Então temos de aplicar justiça sobre as sub-ações de $ProxXY_{balb}$[8]. Assim, das duas condições de justiça escolheremos a condição fraca, e a aplicaremos sobre as sub-ações de $ProxXY_{balb}$ para garantirmos que x e y cresçam indefinidamente, como mostra a figura 11.

$$JusticaXY \triangleq \mathrm{WF}_{\langle x,y\rangle}(MudaX_{balb}) \wedge \mathrm{WF}_{\langle x,y\rangle}(MudaX_{balb})$$
$$SpecJustaXY_{balb} \triangleq SpecXY_{balb} \wedge JusticaXY$$

Figura 11 – Especificação justa dos contadores independentes $SpecJustaXY_{balb}$

Para ilustramos um exemplo de especificação com justiça forte, especificaremos um sistema um pouco diferente. Aproveitaremos a especificação de contadores independentes $SpecXY_{balb}$ da figura 8 para definir um sistema concorrente onde o contador de y dependerá do valor do contador de x para avançar. Acrescentaremos uma condição à ação $MudaY_{balb}$ de modo que o contador de y só poderá avançar, mas não necessariamente avançará, quando o contador de x for ímpar. Chamaremos este sistema de **contadores dependentes** (apesar de somente um contador depender do outro). Definimos a especificação $Spec_{dep}$, como demonstrado na figura 12.

[7] "Strong fairness can be more difficult to implement than weak fairness, and it is a less common requirement. A strong fairness condition should be used in a specification only if it is needed. When strong and weak fairness are equivalent, the fairness property should be written as weak fairness."

[8] $ProxXY_{balb}$ é uma sub-ação de si mesma pois $ProxXY_{balb} \Rightarrow ProxXY_{balb}$. Portanto, nos referenciamos às ações $MudaX_{balb}$ e $MudaY_{balb}$.

$$
\begin{aligned}
Ini_{dep} &\triangleq x = 0 \wedge y = 0 \\
MudaX_{dep} &\triangleq x' = x + 1 \wedge y' = y \\
MudaY_{dep} &\triangleq x \bmod 2 = 1 \wedge y' = y + 1 \wedge x' = x \\
Prox_{dep} &\triangleq MudaX_{dep} \vee MudaY_{dep} \\
Spec_{dep} &\triangleq Ini_{dep} \wedge \Box[Prox_{dep}]_{\langle x,y \rangle}
\end{aligned}
$$

Figura 12 – Especificação de contadores dependentes $Spec_{dep}$

A especificação $Spec_{dep}$ é bastante similar à especificação $SpecXY_{balb}$. As ações $Prox_{dep}$ e $MudaX_{dep}$ continuam sendo sempre habilitadas, e, seguindo o raciocínio da especificação justa anterior, aplicaremos a condição de justiça fraca sobre $MudaX_{dep}$. Portanto, nos resta analisar a frequência com a qual a ação $MudaY_{dep}$ é habilitada.

A ação $MudaY_{dep}$ **não** é sempre habilitada, porque depende do valor de x ser ímpar: se o valor de x não for ímpar, então $MudaY_{dep}$ não é habilitada. Porém, x só é ímpar se $MudaX_{dep}$ acontecer, então só é possível garantir que y avança se x também avançar. A noção de dependência entre ações é importante. Por exemplo, a especificação de um contador dependente onde somente y deve crescer indefinidamente (e x *pode* crescer indefinidamente) é uma especificação incoerente.

Como $MudaY_{dep}$ não é sempre habilitada, e queremos garantir que $MudaY_{dep}$ aconteça infinitamente frequentemente para que y cresça indefinidamente, a frequência com a qual $MudaY_{dep}$ é habilitada é relevante.

$$\mathrm{WF}_{\langle x,y \rangle}(MudaY_{dep}) \triangleq \Diamond\Box Enabled\ \langle MudaY_{dep} \rangle_{\langle x,y \rangle} \Rightarrow \Box\Diamond\langle MudaY_{dep} \rangle_{\langle x,y \rangle}$$

$$\mathrm{SF}_{\langle x,y \rangle}(MudaY_{dep}) \triangleq \Box\Diamond Enabled\ \langle MudaY_{dep} \rangle_{\langle x,y \rangle} \Rightarrow \Box\Diamond\langle MudaY_{dep} \rangle_{\langle x,y \rangle}$$

Figura 13 – Condições de justiça fraca e forte sobre $MudaY_{dep}$

A condição de justiça fraca garante que $MudaY_{dep}$ acontecerá infinitamente frequentemente se $MudaY_{dep}$ for habilitada de um momento para sempre. Por exemplo, se o valor de x parasse em um valor ímpar e nunca mais mudasse, então poderíamos garantir a propriedade que queremos, mas a condição de justiça sobre $MudaX_{dep}$ não permite esse tipo de comportamento. Como a habilitação de $MudaY_{dep}$ pode ser frequentemente interrompida por $MudaX_{dep}$ acontecer, concluímos que a condição de justiça forte sobre $MudaY_{dep}$ é a condição que nos garantirá que ambos x e y cresçam indefinidamente.

Portanto, a partir das definições do contador dependente da figura 12, definimos um contador dependente justo através especificação $SpecJusta_{dep}$ na figura 14

$$Justica_{dep} \triangleq \mathrm{WF}_{\langle x,y \rangle}(MudaX_{dep}) \wedge \mathrm{SF}_{\langle x,y \rangle}(MudaY_{dep})$$
$$SpecJusta_{dep} \triangleq Spec_{dep} \wedge Justica_{dep}$$

Figura 14 – Especificação do contador dependente justo $SpecJusta_{dep}$

3.1.7 Propriedades temporais

Ao longo da exposição mencionamos que nossas especificações tinham de satisfazer determinadas propriedades. TLA permite que definamos tais propriedades formalmente, e que afirmemos quais dessas propriedades uma especificação satisfaz. Podemos denotar uma especificação satisfazer uma propriedade por implicação lógica de uma fórmula temporal a outra. Se uma especificação *Spec* implementa uma propriedade *Prop*, expressamos este requisito através da fórmula $Spec \Rightarrow Prop$: todo comportamento que satisfaz *Spec* satisfaz *Prop* também.

Em TLA, há três principais tipos de propriedades: propriedades invariantes, propriedades de futuro, e refinamento.

Invariantes

Uma invariante afirma que algo é sempre verdadeiro. Por exemplo:

- Exclusão mútua: dois agentes nunca acessam a mesma seção crítica simultaneamente (HERLIHY; SHAVIT, 2012, p. 8).

$$\forall\, a_1 \in Agentes : \forall\, a_2 \in Agentes :$$
$$\Box((a_1 \neq a_2) \Rightarrow \neg(EmSeçãoCrítica(a_1) \ \wedge \ EmSeçãoCrítica(a_2)))$$

  Nota-se que aqui fórmulas na forma $\forall\, v \in S : P$ equivalem a $\forall\, v : v \in S \wedge P$ em lógicas como a lógica proposicional.

- Ausência de impasse: o funcionamento do sistema nunca é interrompido (HERLIHY; SHAVIT, 2012, p. 8).

$$\Box Enabled\ \langle Prox \rangle_v$$

  onde *Prox* é a relação de transição de uma especificação e v são suas variáveis.

- Corretude parcial: um procedimento que termina nunca dá a resposta errada.

$$\Box(ProcedimentoTerminou \Rightarrow RespostaCorreta)$$

A invariante mais comum em especificações é a invariante de tipos. Diferente de linguagens de programação, TLA não possui tipos, então, para garantimos a corretude

do domínio das variáveis da especificação, utilizamos este invariante. Definindo uma invariante de tipos do contador dependente $Spec_{dep}$, esperamos que o valor de x e de y sejam números inteiros maiores ou iguais a zero. Ou, ainda melhor, que sejam números naturais. Expressamos tal invariante e o requisito de que $Spec_{dep}$ satisfaz esta propriedade respectivamente como:

$$CorretudeDeTipos \triangleq x \in \mathbf{N} \land y \in \mathbf{N}$$
$$Spec_{dep} \Rightarrow \Box CorretudeDeTipos$$

Invariantes são as propriedades mais importantes que podemos afirmar sobre um sistema, e temos grande liberdade em defini-las. Por exemplo, na especificação do contador dependente $Spec_{dep}$, o contador de y depende do contador de x para avançar. Isso significa que o valor de x sempre será maior ou igual ao valor de y em qualquer momento. Este invariante é afirmada da seguinte maneira:

$$XMaiorOuIgualAY == \Box(x \geq y)$$
$$Spec_{dep} \Rightarrow XMaiorOuIgualAY$$

Propriedades de futuro

Propriedades de futuro afirmam que algo acontece futuramente. Assim como invariantes, propriedades de futuro utilizam predicados sobre as variáveis do sistema. Exemplos de propriedades de futuro são:

- Término: um procedimento termina em algum momento.

$$\Diamond\Box EmEstadoTerminal$$

- Garantia de serviço: todo pedido é respondido em algum momento.

$$\forall\, pedido \in Pedidos : FoiEnviado(pedido) \rightsquigarrow RespostaRecebida(pedido)$$

  Nota-se que $A \rightsquigarrow B$ equivale a $\Box(A \rightarrow \Diamond B)$.

- Ausência de inanição: todo agente que desejar acessar um recurso o acessa futuramente.

$$\forall\, agente \in Agentes : DesejaAcesso(agente) \rightsquigarrow AcessaRecurso(agente)$$

Os contadores que especificamos são infinitos: o valor de suas variáveis x e y podem crescer indefinidamente. Expressamos que x e y crescem indefinidamente como o fato de seus valores alcançarem todos os números naturais em algum momento:

$$SempreCrescem \triangleq \forall\, n \in \mathbb{N} : \Diamond(x = n) \land \Diamond(y = n)$$
$$SpecJusta_{dep} \Rightarrow SempreCrescem$$

As especificações $SpecJustaXY_{balb}$ e $Spec_{dep}$ satisfazem a propriedade de futuro *SempreCrescem* porque são especificações com garantia de vivacidade: se podemos garantir que os valores de x e y mudam infinitamente frequentemente, então eles crescem indefinidamente. Por outro lado, não há garantia que a especificação $SpecXY_{balb}$ satisfaz *SempreCrescem* devido à ausência de vivacidade: é possível que os valores de x e y cresçam indefinidamente, mas não é garantido que *todos* os comportamentos satisfaçam *SempreCrescem*. Ou seja, a fórmula $SpecXY_{balb} \Rightarrow SempreCrescem$ é falsa.

Refinamento

A implicação entre duas fórmulas temporais permite que afirmemos que uma especificação satisfaz outra, já que uma especificação é uma fórmula temporal. Informalmente, neste caso, dizemos que uma especificação $\mathcal{C}$ **implementa** uma especificação $\mathcal{A}$ quando $\mathcal{C} \Rightarrow \mathcal{A}$.

A partir de uma especificação de alto nível podemos aumentar gradativamente a granularidade de suas operações. Podemos afirmar iterativamente que cada nova especificação implementa uma versão um pouco menos abstrata até alcançarmos um nível de abstração baixo o suficiente para a especificação ser implementável como um programa. Este processo de transformar uma especificação abstrata em uma menos abstrata é chamado de **refinamento**. Frequentemente será necessário traduzir os símbolos de uma especificação para outra, uma tradução chamada de **mapeamento de refinamento** (*refinement mapping*).

3.2 TLA+

TLA$^+$ (LAMPORT, 2002) é uma linguagem de especificação que se baseia na lógica temporal de ações, adicionando módulos e notações para funções, conjuntos, tuplas e estruturas condicionais, nos permitindo descrever sistemas concorrentes complexos em um alto nível de abstração.

TLA$^+$ é escrita com símbolos ASCII. É possível gerar arquivos PDF das especificações com tipografia similar a fórmulas em LaTeX, mas como nosso foco é explorar a experiência de uso da linguagem e do ecossistema, decidimos manter as especificações neste trabalho em ASCII. Para facilitar a leitura, colorimos certos elementos da sintaxe, obtendo um visual similar ao presente em editores de texto para programação.

Nesta seção explicaremos breve e informalmente a sintaxe de TLA$^+$ como exposta no capítulo 15 de (LAMPORT, 2002). A sintaxe exposta no livro não inclui a gramática de

comentários. TLA$^+$ possui dois tipos de comentários: comentários de linha, que começam com `\*` e comentários em bloco, que podem preencher múltiplas linhas, começando com `(*` e terminando com `*)`. Para mais informações sobre sua sintaxe e semântica, recomendamos os capítulos 15, 16, 17 e 18 de (LAMPORT, 2002).

3.2.1 Estrutura da linguagem

Módulos

TLA$^+$ facilita o reúso de especificações e operadores através de módulos. Todo arquivo TLA$^+$ é um módulo, e espera-se que o arquivo `.tla` em que habita tenha o mesmo nome do módulo. Por exemplo, espera-se que o módulo *Mod* esteja salvo em um arquivo chamado `Mod.tla`.

Todas as fórmulas no arquivo residem entre um cabeçalho e um rodapé. O cabeçalho deve conter quatro ou mais símbolos '-' (hífen) de cada lado da expressão `MODULE <nome do módulo>` (os lados não precisam estar simétricos). O rodapé contém quatro ou mais símbolos '=', e nenhum texto.

```
---- MODULE Mod ----

====================
```

Todo módulo em TLA$^+$ pode ter o seguinte, nesta ordem:

1. zero ou uma extensão de outros módulos;
2. zero ou mais do seguinte (em qualquer ordem): declaração de variáveis; declaração de constantes; definição de operador; definição de função; definição de premissa; definição de teorema; instanciação de módulo; definição de módulo.

Estendendo módulos

Quando estendemos um módulo `M`, copiamos suas definições para o módulo local. Isso significa que é possível que definições de mesmo nome nos módulos `M` e módulo local conflitem.

A extensão de módulos, se existir, deve ser a primeira declaração de um módulo. Um módulo somente estende outros módulos que estejam **em um mesmo diretório**. Suponhamos que um diretório contenha os módulos `Mod1`, `Mod2` e `Mod3`, e que o módulo `Mod3` requer fórmulas definidas nos módulos `Mod1` e `Mod2`. Como TLA$^+$ permite zero ou uma expressão `EXTENDS` em todo o módulo, declararíamos a extensão dos módulos separados por vírgula:

```
---- MODULE Mod3 ----
```

```
EXTENDS Mod1, Mod2
...
=====================
```

A exceção para a regra de extensão de módulos do mesmo diretório são os módulos-padrão, dos quais os módulos `Naturals`, `Integers`, `Sequences`, `FiniteSets` e `TLC` nos serão relevantes. Os módulos-padrão restantes são os módulos `Reals`, `Bags` e `RealTime`. Para mais informações, ver (LAMPORT, 2002).

Instanciando módulos

A instanciação de módulos costuma aparecer quando queremos compor ou refinar módulos. Quando instanciamos um módulo `Externo` num módulo `Local`, espera-se que exista um mapeamento das variáveis de `Local` para as de `Externo`. Caso existam variáveis de mesmo nome em ambos os módulos, o mapeamento entre ambas é feito implicitamente.

Aproveitando o exemplo de módulos estendidos, suponhamos que `Mod3` precisa instanciar os módulos `Mod1` e `Mod2`. `Mod1` contém as variáveis `a` e `b`, e `Mod2` contém as variáveis `x` e `y`. Se o módulo `Mod3` declarar as variáveis `x`, `y`, `a` e `b`, instanciaremos ambos os módulos individualmente dentro de `Mod3` assim:

```
VARIABLES a, b, x, y, ... \* Outras variáveis
M1 == INSTANCE Mod1
M2 == INSTANCE Mod2
```

Os identificadores `M1` e `M2` referenciam todo conteúdo de seus respectivos módulos. Acessamos suas fórmulas utilizando a sintaxe $Identificador\texttt{!}Identificador$. É possível se referir a fórmulas por múltiplas camadas de identificadores, por exemplo, `M!N!O!P!Spec`

Para casos onde é necessário um mapeamento entre variáveis, escreveríamos como no exemplo seguinte:

```
VARIABLES p, q
M1 == INSTANCE Mod1 WITH a <- p, b <- q
M2 == INSTANCE Mod2 WITH x <- p + q, y <- 2 * q
```

Neste exemplo, dizemos que as variáveis `a` e `b` do módulo `Mod1` são substituídas por `p` e `q`, respectivamente: uma simples renomeação. Já as variáveis do módulo `Mod2` são substituições mais complexas: todo valor de `x` em `Mod2` é substituído por `p + q`; todo valor de `y` é substituído por `2 * q`.

Definição de operadores

TLA$^+$ permite definir operadores da seguinte maneira:

```
Op1 == exp1
```

```
Op2(a1, a2) == exp2
```

No exemplo acima temos dois operadores. O operador `Op1` não possui argumentos. Já o operador `Op2(a1, a2)` tem dois argumentos representados pelos identificadores `a1` e `a2`. `exp1` e `exp2` são expressões. A avaliação da expressão `Op2(e1, e2)`, onde `e1` e `e2` também são expressões, é feita substituindo todo identificador `a1` e `a2` na expressão `exp2` por `e1` e `e2`, respectivamente.

Declaração de variáveis e constantes

Variáveis são declaradas com `VARIABLES` e uma lista de identificadores separados por vírgula. Convenciona-se que esses identificadores começam com letras minúsculas.

```
VARIABLES x, y, z
```

Constantes são declaradas com `CONSTANTS` e uma lista de identificadores separados por vírgula. Convenciona-se que esses identificadores começam com letras maiúsculas. Um detalhe é que a declaração de constantes tem uma sintaxe especial quando um identificador representa um operador com um ou mais argumentos.

```
CONSTANTS A, Func, Op(_, _)
```

Neste exemplo, `A` e `Func` podem ser valores quaisquer, enquanto `Op` é obrigatoriamente um operador com dois argumentos.

Definição de premissas

Uma especificação supõe certas propriedades sobre constantes, e é possível declarar formalmente estas propriedades na forma de **premissas** (*assumptions* em inglês). Definir premissas, porém, é opcional.

```
ASSUME expr1              \* Premissa sem nome
ASSUME Premissa == expr2  \* Premissa com nome "Premissa"
```

No exemplo acima, espera-se que `expr1` e `expr2` afirmem sobre as constantes declaradas na especificação.

Definição de funções

Uma função em TLA$^+$ é uma função matemática com domínio e imagem. Por exemplo, dada uma função `f` cujo domínio (`DOMAIN f`) é o produto cartesiano dos conjuntos `X` e `Y` ($X \times Y$), e sua imagem é o resultado do operador `Op` sobre os elementos de $X \times Y$, podemos definir `f` de duas formas principais, onde `\in` representa "pertence a" ($\in$):

- `f == [ x \in X, y \in Y |-> Op(x, y) ]`

- `f[ x \in X, y \in Y ] == Op(x, y)`

3.2.2 Expressões e estruturas de dados

Uma expressão é uma construção sintática com valor. Em geral, tudo que vem após == é uma expressão.

Aproveitando a gramática de TLA$^+$ exposta no capítulo 15 de (LAMPORT, 2002), mas utilizando itálicos para símbolos não terminais, e utilizando símbolos terminais `verbatim`; o símbolo $^?$ representa zero ou um elemento, o símbolo * representa zero ou mais elementos, o símbolo $^+$ representa um ou mais elementos, e o símbolo | representa opção. Listamos as seguintes expressões possíveis:

- *Number*
- *String*
- *GeneralIdentifier*
- *GeneralIdentifier* `(` *CommaList*(*Argument*) `)`
- *GeneralPrefixOp Expression*
- *Expression GeneralInfixOp Expression*
- *Expression GeneralPostfixOp*
- `(` *Expression* `)`
- (`\A`|`\E`) *CommaList*(*QuantifierBound*) : *Expression*
- (`\E`|`\A`|`\EE`|`\AA`) *CommaList*(*Identifier*) : *Expression*
- `CHOOSE` *IdentifierOrTuple* (`\in` *Expression*)$^?$: *Expression*
- `{` *CommaList*(*Expression*)$^?$ `}`
- `{` *IdentifierOrTuple* `\in` *Expression* : *Expression*`}`
- `{` *Expression* : *CommaList*(*G.QuantifierBound*) `}`
- *Expression* `[` *CommaList*(*Expression*) `]`
- `[` *CommaList*(*G.QuantifierBound*) `|->` *Expression* `]`
- `[` *Expression* `->` *Expression* `]`
- `[` *CommaList*(*Name*) `|->` *Expression* `]`
- `[` *CommaList*(*Name*) : *Expression* `]`

- `[` *Expression* `EXCEPT`
  CommaList(`!` (`.` *Name* | `[` *CommaList*(*Expression*) `]`)$^+$ `=` *Expression*) `]`
- `@`
- `<<` *CommaList*(*Expression*) `>>`
- *Expression* (`\X` *Expression*)$^+$
- `[` *Expression* `]_` *Expression*
- `<<` *Expression* `>>_` *Expression*
- `WF_` *Expression* `(` *Expression* `)`
- `SF_` *Expression* `(` *Expression* `)`
- `IF` *Expression* `THEN` *Expression* `ELSE` *Expression*
- `CASE` *Expression* `->` *Expression*
  (`[]` *Expression* `->` *Expression*)*
  (`[]` `OTHER` `->` *Expression*)$^?$
- `LET` (*OperatorDefinition*|*FunctionDefinition*|*ModuleDefinition*)$^+$ `IN` *Expression*
- (`/\` *Expression*)$^+$
- (`\/` *Expression*)$^+$

As expressões úteis para nosso trabalho serão detalhadas a seguir.

Subfórmulas

Subfórmulas são expressões escritas na forma `LET <Definições> IN <Expressões>` e nos permitem escrever definições que serão acessíveis somente em uma "fórmula-mãe".

```
\* Definição de X acessível em toda especificação.
X == ...

\* Definição de Z acessível em toda especificação.
Z == LET W == ...     \* Subfórmulas W e Y acessíveis
         Y == ...     \* somente dentro da definição de Z.
     IN ...
```

Estruturas condicionais

TLA$^+$ possui duas estruturas condicionais com sintaxe familiar. Um exemplo de ambas as estruturas com semântica equivalente:

```
IF cond1
THEN expr1
ELSE
  IF cond2
  THEN expr2
  ELSE exprPadrao
```

```
CASE cond1 -> expr1
[] cond2 -> expr2
[] OTHER -> exprPadrao
```

Como toda expressão, a avaliação de uma expressão condicional resulta em um valor. Por exemplo, se `cond1` for verdadeira, o valor dessas expressões será o valor da expressão `expr1`. Se `cond1` for falsa e `cond2` for verdadeira, o valor das expressões será o valor da expressão `expr2`. Se `cond1` e `cond2` forem falsas, então o valor das expressões será `exprPadrao`. As expressões `expr1`, `expr2` e `exprPadrao` podem ser outras estruturas condicionais aninhadas.

Lógica

Além de verdadeiro (`TRUE`), falso (`FALSE`) e do conjunto `BOOLEAN`, TLA$^+$ inclui os operadores booleanos 'negação' (`~` ou `\neg` ou `\lnot`), 'e' (`/\` ou `\land`), 'ou' (`\/` ou `\lor`), 'implica' (`=>`), 'equivale a' (`<=>` ou `\equiv`).

E duas expressões para quantificação lógica:

- `\E x1, x2 \in X, \y1, y2 \in Y: pred` : existem elementos `x1` e `x2` em `X` e `y1` e `y2` em `Y` tal que `pred` é verdade.
- `\A x1, x2 \in X, \y1, y2 \in Y: pred` : para todo `x1` e `x2` em `X` e `y1` e `y2` em `Y` tal que `pred` é verdade.

Nota-se que os elementos `x1` e `x2`, por exemplo, podem ser iguais.

Uma característica especial da sintaxe de TLA$^+$ é a escrita de conjunções ou disjunções de fórmulas aninhadas e verticalizadas como listagens, facilitando a leitura de fórmulas longas.

```
A /\ ((B /\ C) \/ (D => E)) /\ F
```

```
/\ A
/\ \/ B /\ C
   \/ D => E
/\ F
```

As fórmulas acima são equivalentes, mas o exemplo da direita evita parênteses. Nota-se que os símbolos `\/` e `/\` devem estar alinhados verticalmente para as fórmulas serem avaliadas corretamente.

Conjuntos

TLA$^+$ disponibiliza por padrão vários operadores sobre conjuntos: igualdade (`=`), desigualdade (`\=` ou `#`), "pertence a" (`\in`), "não pertence a" (`\notin`), união (`\cup` ou `\union`), interseção (`\cap` ou `\intersect`), "é subconjunto de" (`\subseteq`), diferença (`\`) e produto cartesiano (`\X` ou `\times`).

Além disso, temos algumas expressões e operadores especiais:

- `{e1, ... , en}` : conjunto formado pelos elementos `e1`, ..., `en`.
- `{s \in S: pred}` : conjunto de todos os elementos em `S` que satisfazem o predicado `pred`. Conhecido como `filter` em algumas linguagens de programação.
- `{expr: s \in S}` : conjunto de todos os elementos da forma `expr`, para todo elemento `s` em `S`. Conhecido como `map` em algumas linguagens de programação.
- `[ S -> T ]` : O conjunto de todas as funções de `S` em `T`.
- `SUBSET S` : O conjunto dos subconjuntos de `S`.
- `UNION S` : União dos conjuntos em `S`.

O módulo relacionado a conjuntos, `FiniteSets`, define apenas dois operadores:

- `Cardinality(S)` : Quantidade de elementos em um conjunto `S`, se `S` for finito.
- `IsFiniteSet(S)` : Verdade se, e apenas se `S` for finito.

Sequências

Uma sequência de N elementos (também chamada de tupla) é uma função de domínio $1..N$, então o acesso a seus elementos é feito utilizando a sintaxe de funções. Sequências são cercadas por `<<` e `>>`, e seus elementos são separados por vírgula.

Operações especiais sobre sequências requerem a extensão do módulo `Sequences`. Algumas delas são:

- `s \o t`: concatenação das sequências `s` e `t`.
- `Head(s)`: primeiro elemento de uma sequência `s` não vazia.
- `Tail(s)`: cauda de uma sequência `s` não vazia.
- `Len(s)`: tamanho da sequência `s`.

Strings

Strings são sequências de caracteres, apesar de TLA$^+$ não definir formalmente o que são caracteres. Seus elementos podem ser acessados individualmente assim como em sequências por meio de índices em $1..N$ onde N é o tamanho da sequência. Tudo que estiver entre aspas duplas (`"..."`) é uma *string*.

Dicionários

Dicionários (*records*, em inglês) são funções com domínios finitos de *strings*. Por exemplo, $\{$"k_1", "k_2", ..., "k_N"$\}$, onde "k_i" é uma chave i do dicionário e N é a quantidade de chaves no dicionário.

Definamos um dicionário `d` com chaves `"k1"`, `"k2"` e `"k3"`:

```
d == [ k1 |-> v1, k2 |-> v2, k3 |-> v3 ]
```

Acessamos `v1` – o valor da chave `k1` – de duas maneiras: `d["k1"]` ou `d.k1`.

Frequentemente precisamos definir todos os dicionários possíveis com certas chaves. Por exemplo, se o valor de `k1` pertence ao conjunto `S1`, o valor de `k2` pertence ao conjunto `S2` e o valor de `k3` pertence ao conjunto `S3`, definimos este conjunto de dicionários assim:

```
[ k1: S1, k2: S2, k3: S3]
```

Também costumamos modificar apenas determinado valor correspondente a uma chave. Suponhamos que queremos modificar os valores das chaves `k1` e `k2` de `d`, enquanto `k3` permanece igual. Temos dois modos de efetuar essa mudança:

```
[ d EXCEPT
      !.k1 = v1,
      !.k2 = v2 ]
```

```
[ d EXCEPT
      !["k1"] = v1,
      !["k2"] = v2 ]
```

Operador CHOOSE

O operador CHOOSE seleciona um valor que obedece determinada propriedade, contanto que essa propriedade seja verdadeira. Uma expressão com o operador CHOOSE tem a seguinte forma:

$$\texttt{CHOOSE}\ \mathit{IdentifierOrTuple}\ (\texttt{\textbackslash in}\ \mathit{Expression})^{?}\ :\ \mathit{Expression}$$

Por exemplo, definimos abaixo três operadores utilizando CHOOSE: um que seleciona um valor qualquer de um conjunto S, um que seleciona o maior valor de um conjunto S não vazio, e um que seleciona a raiz quadrada de qualquer número natural, respectivamente.

```
Escolhe(S)      == CHOOSE s: s \in S
```

```
Max(S)           == CHOOSE x: x \in S /\ \A y \in S: x >= y
RaizQuadrada(x) == CHOOSE y \in Nat: y * y = x
```

"Choose" que se traduz do inglês para "escolha" ou "escolher" é um operador determinístico: toda operação CHOOSE é sempre avaliado para o mesmo valor em todo estado de um comportamento. Isso pode ser expresso formalmente e de forma geral da seguinte maneira (LAMPORT, 2024a):

$$(\forall\, v : F \equiv G) \Rightarrow ((\text{CHOOSE } v : F) = (\text{CHOOSE } v : G))$$

Ou seja, toda fórmula equivalente F e G possui o mesmo valor de sua operação CHOOSE.

Seguindo o exemplo de (LAMPORT, 2002, p. 73), a especificação

$$(x = \text{CHOOSE } n : n \in \mathbb{N}) \wedge \Box[x' = \text{CHOOSE } n : n \in \mathbb{N}]_x$$

é satisfeita por somente um comportamento onde x sempre é igual a CHOOSE $n : n \in \mathbb{N}$, um número natural não especificado. Isso é diferente da especificação não determinística

$$(x \in \mathbb{N}) \wedge \Box[x' \in \mathbb{N}]_x$$

, que é satisfeita por todos os comportamentos onde x é um número natural qualquer, possivelmente diferente em cada estado.

3.3 TLC

O TLC é um programa feito para verificar especificações escritas em TLA$^+$ (YU; MANOLIOS; LAMPORT, 1999). O TLC verifica especificações gerando comportamentos que a satisfazem. Esta verificação pode ser feita de duas formas:

- **Verificação de modelo de estado explícito:** o TLC explora exaustivamente todos os estados descritos pela especificação;
- **Simulação:** o TLC explora um caminho aleatório finito ao longo do espaço de estados;

O objetivo da verificação é procurar comportamentos que violem propriedades de interesse. Tais comportamentos, finitos ou infinitos, são chamados de **contraexemplos**. É responsabilidade do usuário declarar quais propriedades em determinado módulo serão verificadas. Portanto, além de um módulo de TLA$^+$ (um arquivo de extensão `tla`), o TLC requer um arquivo de configuração (um arquivo de extensão `cfg`) para ser executado. Enquanto o módulo de interesse contém as fórmulas, variáveis e constantes, o arquivo de configuração contém as valorações de tais constantes, e declarações apontando quais

fórmulas do módulo de interesse a se verificar. Quando propriedades não são dadas, o TLC buscará erros semânticos neste módulo, e impasses nos comportamentos gerados. Na Seção 3.3.2 detalharemos como um arquivo de configuração é preenchido.

É possível executar o TLC de algumas formas: ou pela interface gráfica TLA$^+$ *Toolbox* (TLA+ Foundation, 2021), ou pela extensão de TLA$^+$ (LYGIN, 2024) disponível para o editor de texto *Visual Studio Code* (MICROSOFT, 2024), ou diretamente pelo terminal. Instruções para instalação e uso da ferramenta estão disponíveis em (LAMPORT, 2024b). Utilizaremos, ao longo deste trabalho, a versão 2.18 (rev: 6fb13e) (TLA+ Foundation, 2023). Para mais detalhes sobre TLC, ver (LAMPORT, 2002, cap. 14).

A seguir, listaremos algumas limitações de TLC e, em seguida, o conteúdo do arquivo de configurações.

3.3.1 Limitações de TLC

TLA$^+$ é uma linguagem poderosa, e Lamport imaginava ser impossível de verificar especificações escritas com ela. Para sua surpresa, Yuan Yu et al. criaram o verificador de modelos TLC, capaz de verificar um subconjunto significativo de especificações escritas em TLA$^+$.

Para que a avaliação de uma fórmula termine, espera-se que seu valor seja decidível. Porém, por questões de implementação e otimização, até expressões avaliadas para conjuntos finitos, por exemplo, podem não ser aceitas pelo TLC. As limitações relevantes são:

- Números: números naturais e inteiros permitidos estão no intervalo $-2^{31}..(2^{31}-1)$.
- Fórmulas ilimitadas: o TLC não avalia fórmulas do tipo

  $$(\texttt{\textbackslash E}|\texttt{\textbackslash A}|\texttt{CHOOSE})\ a : e$$

  – onde a é um identificador e e é uma expressão –, mesmo que tal fórmula ilimitada seja facilmente avaliada. Por exemplo, o TLC não aceita uma fórmula como `\E x: x \in {0, 1}`.
- CHOOSE: para que duas expressões CHOOSE equivalentes sejam avaliadas para o mesmo valor, elas também devem ser sintaticamente idênticas. Isso significa, por exemplo, que o TLC pode computar valores diferentes para as seguintes expressões equivalentes:

  ```
  CHOOSE x \in {1, 2, 3}:  x < 3          CHOOSE x \in {3, 2, 1}:  x < 3
  ```

- *Strings*: o TLC considera *strings* como valores primitivos, e não como funções (sequências de caracteres). Isso significa que uma expressão como `"abc"[1]` não é aceita pelo TLC, apesar de ser uma expressão legal em TLA$^+$.

3.3.2 Arquivo de configurações

O TLC não funciona sem um arquivo de configurações. Quando verificamos um módulo `M`, é obrigatório ter um arquivo de configurações que nos aponte a fórmula de especificação em `M` a se verificar, e o valor de todas as constantes declaradas em `M`. Opcionalmente, podemos declarar quais invariantes e propriedades de futuro gostaríamos de verificar, sob quais restrições de estado ou ação verificar a especificação, entre outras informações.

Ao todo, o arquivo de configurações pode conter declaração de qual especificação verificar, atribuições de constantes, substituições de símbolos, invariantes e propriedades temporais a se verificar, restrições de estado, restrições de ação, habilitação ou desabilitação da detecção de impasses, redução por simetria, *view* e *alias*. Não detalharemos as três últimas declarações.

3.3.2.1 Especificação

A sintaxe para declarar uma fórmula como a especificação de interesse é:

`SPECIFICATION` *Identificador*

onde *Identificador* é uma fórmula na forma padrão $Ini \wedge \Box[Prox]_v \wedge Vivacidade$.

3.3.2.2 Constantes

Todas as constantes declaradas no módulo que desejamos verificar precisam ter seus valores atribuídos no arquivo de configurações. Além disso, é possível substituir o valor de outros símbolos. Essas atribuições e substituições seguem a seguinte sintaxe:

`CONSTANT` *SeparadosPorEspaco*(*Atribuicao*|*Substituicao*)

As sintaxes para *Atribuicao* e *Substituicao* serão detalhadas a seguir.

Atribuição

A sintaxe de atribuição de constantes na declaração `CONSTANT` é a seguinte:

Identificador `=` *ValorDeTLC*

Um valor de TLC (*ValorDeTLC*) é um valor primitivo (booleano, inteiro, *string*, valor-modelo), conjunto finito de valores **comparáveis**, ou uma função f com domínio sendo um valor de TLC tal que $f[x]$ é um valor de TLC para todo x no domínio de f.

Valores comparáveis seguem as seguintes regras:

- Dois valores primitivos são comparáveis se e somente se tiverem o mesmo tipo de valor;

- Um valor-modelo é comparável com qualquer valor e só é igual a si mesmo;
- Dois conjuntos são comparáveis se têm quantidades diferentes de elementos, ou se têm a mesma quantidade de elementos e todos os elementos de um conjunto são comparáveis com todos os elementos do outro;
- Duas funções f e g são comparáveis se e somente (i) seus domínios são comparáveis, e (ii) se seus domínios são iguais, então $f[x]$ e $g[x]$ são comparáveis para todo elemento x de seus domínios;

Um valor-modelo é um valor abstrato, comparável com qualquer outro valor, mas só é igual a si próprio. Uma constante pode se tornar um valor-modelo ou um conjunto finito de valores-modelo. Suponhamos que uma constante `C` foi declarada no módulo de interesse. Se quisermos atribuir um valor-modelo a `C`, utilizamos seu próprio identificador, isto é, `CONSTANT C = C`. Se quisermos atribuir `C` como um conjunto de valores-modelo, usamos um conjunto de identificadores não presentes no módulo de interesse. Por exemplo, `CONSTANT C = {c1, c2, c3}`.

Uma intuição para valores-modelo são os valores `None`, `Nil` e `NULL` em linguagens de programação. O valor subjacente de tais símbolos não nos interessa. Por exemplo, apenas sabemos que `None = None` e que `None` é diferente de qualquer outro valor.

Substituição

A sintaxe de substituição de símbolos na declaração `CONSTANT` é a seguinte:

Identificador `<-` *Identificador*

Ambos identificadores são distintos e declarados no módulo de interesse ou em módulos estendidos.

A substituição pode acontecer por vários motivos. Um deles é de que o valor de determinado identificador é lento demais para ser calculado pelo TLC. Um exemplo disso dado em (LAMPORT, 2002, p. 235) é a operação de obter uma sequência de elementos de um conjunto `S` em ordem crescente:

```
Sort(S) ==
  CHOOSE s \in [1..Cardinality(S) -> S]:
    \A i, j \in DOMAIN s:
      (i < j) => (s[i] < s[j])
```

Para avaliar este operador, o TLC precisa calcular todas as n^n permutações de elementos no conjunto $[1..n \to S]$, o que pode ser extremamente ineficiente. Em vez de otimizar o operador `Sort` modificando a especificação original, dizemos ao TLC para substituir como este operador será avaliado. Neste caso, podemos substituir `Sort` pelo operador

`FastSort` definido no módulo-padrão `TLC`. Primeiro estendemos o módulo `TLC` no módulo de interesse, e então substituímos `Sort` por `FastSort` no arquivo de configurações assim:

```
CONSTANT Sort <- FastSort \* Substituição de Sort por FastSort
```

3.3.2.3 Invariantes e propriedades temporais

A sintaxe para declaração de invariantes ou propriedades temporais são similares:

(`INVARIANT`|`PROPERTY`) *SeparadosPorEspaco*(*Identificador*)

Nota-se que, ao declarar determinada propriedade P como invariante no arquivo de configuração – P é um predicado de estado –, o TLC verificará $\Box P$.

3.3.2.4 Restrições de estado e ação

Devido ao alto número de estados possíveis, é possível adicionar restrições ao espaço de estado, principalmente quando ele é infinito. O TLC possui restrições de estado e restrições de ação.

Restrições de estado são predicados que, se forem falsos, impedem que um estado seja avaliado. No arquivo de configuração, adicionamos este tipo de restrição com `CONSTRAINT` seguido por uma lista de identificadores separados por espaço, onde *Identificador* se refere a um predicado de estado.

`CONSTRAINT` *SeparadosPorEspaco*(*Identificador*)

Restrições de ação são análogas a restrições de estado, onde *Identificador* se refere a uma ação.

`ACTION_CONSTRAINT` *SeparadosPorEspaco*(*Identificador*)

É importante ressaltar que restrições de estado ou de ação podem impedir que o TLC verifique propriedades de vivacidade ao impedir que ele encontre comportamentos infinitos.

3.3.2.5 Detecção de impasses

Quando verificamos determinada especificação, o TLC detecta impasses por padrão. O TLC entende um impasse como a impossibilidade de dar novos passos exceto passos balbuciantes. Formalmente, um impasse acontece quando a propriedade $\Box$ENABLED $\langle Prox \rangle_v$ é violada, onde *Prox* é a relação de transições da especificação de interesse, e v são as variáveis declaradas no módulo de interesse. Podemos perceber que esta definição de impasse é bem branda, então ela não necessariamente aponta um problema.

4 EXEMPLOS DE MODELAGEM E VERIFICAÇÃO FORMAL

Neste capítulo especificaremos uma *lock* e um contador concorrente com uma variável compartilhada para ilustrar os conceitos da lógica temporal de ações na prática. Este processo se dará desde a escrita, modelagem e discussão de um sistema simples (a *lock*), até a escrita e modelagem incremental de um contador concorrente e os desafios que surgem no caminho.

4.1 *LOCK*

Nesta seção especificaremos um sistema cíclico não terminal onde uma *lock* é trancada e destrancada por uma quantidade arbitrária de agentes para descrever as operações de sua interface em alto nível. Quando mencionamos a palavra "interface", pode-se pensar em interfaces como as encontradas em programação orientada a objetos. Por exemplo, no Código 1 temos expressa na linguagem de programação Java uma interface `ILock` com dois métodos: `lock` (trancar) e `unlock` (destrancar).

Código 1 – Interface `ILock` na linguagem de programação Java

```
interface ILock {
    public void lock();
    public void unlock();
}
```

Este tipo de interface está presente de uma forma ou de outra em diversas linguagens de programação, e expressa pouquíssima informação, porque só sabemos a entrada e saída de seus métodos, e não a relação entre esses métodos. TLA$^+$ nos permite especificar a interface de determinado sistema, definir a relação entre os métodos dessa interface e ainda verificar propriedades sobre esse sistema.

4.1.1 Constantes e variáveis

O sistema contém uma quantidade arbitrária, finita e fixa de agentes. Não nos importa o que são agentes, apenas que os agentes diferem entre si. Isso nos dá liberdade para que uma futura implementação considere o que são agentes: *threads*, processos ou nós de um sistema distribuído, por exemplo. Declararemos uma constante chamada *Agentes* que representará este conjunto possivelmente vazio de agentes.

Uma *lock* é trancada e destrancada por algum agente. Declaramos uma variável *dono* que representará o agente que está acessando a seção crítica atualmente. Se nenhum agente estiver acessando a seção crítica, utilizaremos um valor *Ninguem*, declararemos

Ninguem como outra constante (um valor desconhecido) e esperamos que *Ninguem* não seja um agente do conjunto *Agentes*.

O trecho inicial do módulo `Lock` com as declarações de variáveis, constantes *Agentes* e *Ninguem*, e das premissas relacionadas se encontra no Código 2.

Código 2 – Primeiros passos na definição do módulo `Lock`

```
---- MODULE Lock ----
CONSTANTS Agentes, Ninguem
ASSUME Ninguem \notin Agentes  \* 'Ninguem' não é um agente.
VARIABLES dono
=====================
```

4.1.2 Estado inicial e relação de transições

A especificação deste módulo envolve a definição de ações, como também a de como o sistema evolui ao tomar essas ações ao longo do tempo. Por especificarmos não só as ações de uma *lock*, mas também o sistema cíclico não terminal no qual ela está inserida, temos de definir um conjunto de estados iniciais e uma relação de transições.

Começando pelos estados iniciais, especificaremos um sistema cuja *lock* sempre começa sem dono. Representamos este fato afirmando através do predicado *Ini* que o valor da variável *dono* é *Ninguem* no primeiro estado de todo comportamento.

Uma *lock* pode ser trancada e destrancada. Como queremos especificar uma *lock* em alto nível e alta granularidade, a relação de transições só conterá duas ações: trancar e destrancar. Imagina-se que uma *lock* especificada em menor granularidade de suas ações conteria múltiplas ações para trancar e destrancar.

A relação de transições unirá as ações possíveis deste sistema. Se tivéssemos dois agentes como o contador $SpecXY_{balb}$ do capítulo anterior, poderíamos definir ações como *Trancar1*, *Destrancar1*, *Trancar2*, *Destrancar2*, correspondendo ao primeiro e segundo agentes, respectivamente. A relação de transições *Prox* seria

$$Trancar1 \lor Trancar2 \lor Destrancar1 \lor Destrancar2$$

Para uma quantidade arbitrária de agentes é inviável utilizar disjunções dessa maneira. A solução será utilizarmos a quantificação existencial sobre o conjunto `Agentes` e colocarmos o agente responsável pela ação como um parâmetro das ações *Trancar* e *Destrancar*, como feito no Código 3. Ou seja, *Prox* expressa que, num determinado estado, existe um agente *ag* que tranca ou destranca a *lock*. A partir daí, poderemos definir as ações *Trancar* e *Destrancar* sabendo que elas têm um agente como argumento.

Código 3 – Relação de transições *Prox* do módulo `Lock`

```
Prox == \E ag \in Agentes: Trancar(ag) \/ Destrancar(ag)
```

Só se pode trancar uma *lock* quando ninguém for dono da *lock*. Definimos a ação *Trancar*(*agente*) no Código 4 como uma ação que acontece quando a *lock* não tem dono, ou seja, *dono* = *Ninguem*, e que muda a variável *dono* para *agente*.

Código 4 – Ação *Trancar* do módulo `Lock`

```
Trancar(agente) == dono = Ninguem /\ dono' = agente
```

Só se pode destrancar uma *lock* quando *agente* for dono da *lock*. Definimos a ação *Destrancar*(*agente*) no Código 5 como uma ação que acontece quando o dono da *lock* é *agente*, e que muda a variável *dono* para *Ninguem*.

Código 5 – Ação *Destrancar* do módulo `Lock`

```
Destrancar(agente) == dono = agente /\ dono' = Ninguem
```

Por último, definimos a fórmula *Spec* que une todas as fórmulas anteriores: *Ini* é o predicado de estados iniciais, *Prox* é a relação de transições. Suas definições expressas em TLA$^+$ estão disponíveis no Código 6.

Código 6 – Fórmulas *Ini*, *Prox* e *Spec* do módulo `Lock`

```
Ini  == dono = Ninguem
Prox == \E ag \in Agentes: Trancar(ag) \/ Destrancar(ag)
Spec == Ini /\ [][Prox]_dono
```

4.1.3 Definindo invariantes e propriedades temporais

Todas as fórmulas escritas sobre a *lock* até aqui se referem a seus comportamentos possíveis, o que **pode** acontecer. Acrescentar vivacidade à especificação permitirá expressarmos o que **tem que** acontecer. Porém, o que tem que acontecer está fortemente relacionado às propriedades que esperamos que o sistema satisfaça. Antes de declararmos a vivacidade de ações do sistema, é interessante definirmos formalmente quais propriedades o sistema deverá satisfazer. Definiremos duas propriedades: uma invariante e uma propriedade de futuro.

TLA e TLA$^+$ não possuem tipagem de variáveis. Para garantir a corretude sobre o domínio das variáveis da *lock*, definiremos uma **invariante** de tipos chamada *InvarianteDeTipos* afirmando que o valor *dono* ou é um agente, ou é *Ninguem*.

Como estamos num sistema cíclico não terminal, esperamos que todo agente acesse a *lock* um número infinito de vezes: todo agente acessa a *lock* infinitamente frequentemente. Caso contrário, um agente acessaria a *lock* um número finito de vezes, e nunca mais a acessaria, sofrendo **inanição**. Formalmente, definiremos a **propriedade de futuro** *AusenciaDeInanicao* afirmando que todo agente deverá ser dono da lock infinitamente

frequentemente ($\Box\Diamond$). As definições de *InvarianteDeTipos* e *AusenciaDeInanicao* estão presentes no Código 7.

Código 7 – Fórmulas *InvarianteDeTipos* e *AusenciaDeInanicao* do módulo `Lock`

```
InvarianteDeTipos == dono \in Agentes \cup { Ninguem }
AusenciaDeInanicao == \A agente \in Agentes: []<>(dono = agente)
```

4.1.4 Declarando justiça

Tendo definido as propriedades de interesse, podemos continuar com as declarações de vivacidade do sistema. Declararemos essa vivacidade através de justiça de modo que a especificação satisfaça as propriedades definidas no Código 7.

Se não há justiça para *Trancar*(*agente*), não há garantia de que a *lock* será adquirida por *agente*, mesmo que só *agente* esteja trabalhando. Se não há justiça para *Destrancar*(*agente*), não há garantia de que a *lock* será liberada por *agente*. Analisaremos as justiças para ambas as ações.

Analisando justiça de *Trancar*

Queremos garantir que *Trancar* aconteça repetidamente para todo agente *ag* acessar a *lock* repetidamente. Teremos que analisar as condições de justiça fraca e forte sobre *Trancar*.

$$\mathrm{WF}_{dono}(Trancar(ag)) \equiv \Diamond\Box Enabled\ \langle Trancar(ag)\rangle_{dono} \Rightarrow \Box\Diamond\langle Trancar(ag)\rangle_{dono}$$

A justiça fraca para *Trancar* é definida como na fórmula acima: garante-se que, dado um agente *ag*, se um passo *Trancar*(*ag*) for continuamente permitido, então *Trancar*(*ag*) acontece repetidamente. No contexto desta especificação, a condição de que *Trancar*(*ag*) deve ser continuamente permitida para acontecer significa que nenhum outro agente compete para trancar a *lock*. Se algum outro agente competir para trancar a *lock*, então a ação *Trancar*(*ag*) deixa de ser continuamente permitida. Ou seja, a condição de justiça fraca só garante ausência de inanição para um sistema com um único agente.

$$\mathrm{SF}_{dono}(Trancar(ag)) \equiv \Box\Diamond Enabled\ \langle Trancar(ag)\rangle_{dono} \Rightarrow \Box\Diamond\langle Trancar(ag)\rangle_{dono}$$

Evidentemente, sobrou a condição de justiça forte sobre *Trancar*(*ag*). A justiça forte para *Trancar*(*ag*) é definida como na fórmula acima: dado um agente *ag*, se um passo *Trancar*(*ag*) for permitido repetidamente, então *Trancar*(*ag*) acontece repetidamente. Condicionar um passo *Trancar*(*ag*) a ser permitido repetidamente (em vez de permitido

continuamente) é justamente o que precisamos, pois um passo *Trancar*(*ag*) nunca é permitido continuamente, e sim repetidamente, num sistema com uma quantidade arbitrária de agentes.

Concluímos que, para garantir ausência de inanição para uma quantidade arbitrária de agentes do sistema, precisamos da condição forte de justiça sobre *Trancar*.

Analisando justiça de *Destrancar*

A análise de condição de justiça para *Destrancar* é bem mais simples. Justiça fraca em *Destrancar*(*ag*) garante que, dado um agente *ag*, se a ação *Destrancar*(*ag*) for permitida continuamente, então *Destrancar*(*ag*) acontece repetidamente. Não existe competição para destrancar uma *lock*: uma vez que a *lock* foi trancada por um agente *ag*, a possibilidade de *ag* destrancá-la é **contínua** até que *ag* a destranque. Isso significa que a justiça fraca será suficiente para garantir que *Destrancar*(*ag*) aconteça repetidamente para todo agente *ag*.

Definindo fórmula de justiça

Finalmente, definimos a especificação *SpecJusta* como no Código 8 onde unimos a fórmula de justiça *Justica* à fórmula de especificação *Spec* previamente definida no Código 6.

Código 8 – Fórmulas *Justica* e *SpecJusta* do módulo `Lock`

```
Justica   == \A ag \in Agentes: /\ SF_dono(Trancar(ag))
                                /\ WF_dono(Destrancar(ag))
SpecJusta == Spec /\ Justica
```

O módulo `Lock` completo está disponível no Apêndice 6.

4.1.5 Verificando a especificação com TLC

Para verificarmos uma especificação, damos ao TLC um módulo de TLA$^+$ e um arquivo de configuração onde declaramos a fórmula de especificação a se verificar, os valores para constantes do módulo de interesse, além de quais invariantes e propriedades de futuro verificar. Mostramos no Código 9 um arquivo de configuração para nossa primeira verificação.

Este arquivo expressa o seguinte, nesta ordem: *SpecJusta* é a fórmula de especificação de interesse; as constantes *Agentes* e *Ninguem* e suas atribuições; declara *InvarianteDeTipos* como a invariante que queremos verificar, e *AusenciadeInanicao* como a propriedade de futuro que queremos verificar.

Executar o TLC com os arquivos `Lock.tla` e `Lock.cfg` nos retorna um resultado positivo, ou seja, não há erros sintáticos ou semânticos nos arquivos dados, assim como

Código 9 – Arquivo de configuração `Lock.cfg`

```
SPECIFICATION SpecJusta
CONSTANTS Agentes = {a1, a2, a3}  \* Conjunto de valores-modelo
          Ninguem = Ninguem       \* Valor-modelo

INVARIANT InvarianteDeTipos
PROPERTY AusenciaDeInanicao
```

não há contraexemplos para as propriedades declaradas após a verificação exaustiva do espaço de estados que a especificação descreve. O relatório completo disponível na Figura 15 afirma que o TLC, em menos de um segundo, verificou sete estados, onde quatro estados são distintos.

```
Model checking completed. No error has been found.
(...)
7 states generated, 4 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 2.
The average outdegree of the complete state graph is 1
(minimum is 0, the maximum 3 and the 95th percentile is 3).
```

Figura 15 – Trecho do relatório do TLC sobre verificação do módulo `Lock` e arquivo de configurações `Lock.cfg`

Já estamos confiantes em nossa especificação por sua simplicidade, mas, por precaução, podemos aumentar o número de agentes e verificar que a especificação continua satisfazendo as propriedades *InvarianteDeTipos* e *AusenciaDeInanicao*. Por exemplo, podemos criar um modelo com seis agentes em vez de três. Para isso, modificaremos a atribuição da constante *Agentes* do arquivo `Lock.cfg` atual, tal que *Agentes* será um conjunto de 6 valores-modelo: `{a1, a2, a3, a4, a5, a6}`. Executando o TLC, recebemos, em menos de um segundo, um resultado positivo após o TLC explorar treze estados, sendo sete deles distintos.

4.1.6 Contraexemplos

O módulo `Lock` é um exemplo de especificação simples de ser verificada, inclusive manualmente. Pelo interesse pedagógico, demonstraremos os contraexemplos retornados pelo TLC quando a especificação *SpecJusta* não possui ausência de inanição devido a condições de justiça incorretas.

Como nosso módulo tem duas ações, e cada ação pode (i) não ter justiça, (ii) ter justiça fraca, ou (iii) ter justiça forte, existem oito definições possíveis para a fórmula *Justica*. Vimos que somente duas dessas fórmulas garantem ausência de inanição:

- $\mathrm{SF}_{dono}(Trancar(ag)) \wedge \mathrm{WF}_{dono}(Destrancar(ag))$;

- $\text{SF}_{dono}(Trancar(ag)) \land \text{SF}_{dono}(Destrancar(ag))$.

Evidentemente, as outras seis fórmulas fazem com que *SpecJusta* não satisfaça a propriedade *AusenciaDeInanicao*. O TLC nos mostra que determinada propriedade foi insatisfeita através de um contraexemplo – uma sequência finita ou infinita de estados a partir de um estado inicial.

```
State 1: <Initial predicate>
/\ dono = Ninguem

State 2: Stuttering
```

(a) Contraexemplo para *lock* sem justiça

```
State 1: <Initial predicate>
/\ dono = Ninguem

State 2: <Trancar(a2)>
/\ dono = a2

State 3: <Destrancar(a2)>
Back to state 1
```

(b) Contraexemplo para *lock* com justiça fraca em ambas as ações

Figura 16 – Contraexemplos para a propriedade *AusenciaDeInanicao*

Executando os TLC com as seis configurações de justiça incorretas de cada vez (e o arquivo de configuração `Lock.cfg`, que permanece igual), obtivemos seis contraexemplos correspondentes, demonstrando que *AusenciaDeInanicao* foi insatisfeita por *SpecJusta*. Selecionamos dois desses seis contraexemplos e os apresentamos na Figura 16.

A Figura 16a apresenta um contraexemplo onde nenhuma ação da especificação tem justiça. Não se pode garantir que uma ação acontecerá se ela não tem justiça, portanto, é possível que o sistema fique parado, e nenhum agente tranque a *lock*. O contraexemplo demonstra essa situação como infinitos passos balbuciantes a partir o estado inicial.

A Figura 16b apresenta um contraexemplo onde ambas as ações da especificação têm justiça fraca. Como discutido na Seção 4.1.4, a justiça fraca para *Trancar* não garante ausência de inanição: é possível que determinado agente seja constantemente ultrapassado por outros agentes e nunca tranque a *lock*. O contraexemplo demonstra essa situação como determinado (a_2) trancando e destrancando a *lock* para sempre enquanto outros agentes nunca trancam a *lock*.

4.1.7 Discussão

Acabamos de especificar uma *lock* em alto nível de abstração, então as possibilidades de implementação são vastas. A partir de duas ações simples e algumas condições de justiça, obtivemos informações valiosas para uma futura implementação de uma *lock* num sistema cíclico não terminal sem possibilidade de inanição de seus agentes. Como mencionamos no início, por não sabermos o que são agentes, esta *lock* pode ser implementada localmente ou distribuidamente. Porém, implementar justiça para esta *lock* pode ser desafiador.

Por exemplo, para que a operação de trancar a *lock* seja fortemente justa, algum tipo de fila de prioridade que considere o tempo de espera deve estar presente, de modo que nenhum agente domine o uso da *lock* e acesse a seção crítica frequentemente. Ao mesmo tempo, para que a *lock* seja trancada por algum agente, ela também deve ser destrancada frequentemente por outros agentes, então deve-se garantir que, ao tomar a *lock* para si, todo agente a libere futuramente. Uma *lock* distribuída com justiça será ainda mais desafiadora, pois teremos de contar com falhas de outras categorias. Por exemplo, será necessário garantir que mensagens de trancar e destrancar cheguem em algum momento ao nó onde reside a *lock*.

4.2 CONTADOR COMPARTILHADO

No Capítulo 3 introduzimos especificações na lógica TLA com um exemplo de um par de contadores infinitos independentes. Na seção anterior vimos como definir uma *lock* num sistema concorrente com uma quantidade arbitrária de agentes, e as nuances ao definir propriedades de justiça sobre ações do sistema. Nesta seção uniremos os conceitos vistos previamente ao especificar um contador concorrente finito a partir de um contador infinito.

4.2.1 Contador infinito

Especificaremos a seguir um contador infinito. Declaramos uma variável, c, que representa uma variável contadora. O valor inicial de c é zero, e cada passo *MaisUm* acresce o valor de c em 1. A fórmula de especificação deste contador com justiça *SpecJusta* está definida no módulo `ContadorInfinito` no Código 10. A extensão do módulo-padrão `Naturals` é necessária devido à operação de soma entre números naturais.

Código 10 – Especificação de um contador infinito `ContadorInfinito`

```
---- MODULE ContadorInfinito ----
EXTENDS Naturals
VARIABLES c

Ini == c = 0
MaisUm == c' = c + 1
Prox == MaisUm
Justica == WF_c(MaisUm)
SpecJusta == Ini /\ [][Prox]_c /\ Justica
================================
```

Apresentamos o arquivo de configuração `ContadorInfinito.cfg` no Código 11. Este arquivo só declara qual especificação verificar: não há constantes a se atribuir ou propriedades a se verificar.

Código 11 – Arquivo de configuração `ContadorInfinito.cfg`

```
SPECIFICATION SpecJusta
```

Porém, o processo do TLC verificar esta especificação é interminável, porque o valor da variável c cresce sem limites, enquanto qualquer verificação de modelos só termina se o espaço de estados gerado pelo TLC for finito. Ao perceber que o espaço de estados cresce mais do que o esperado, o usuário terá de interromper manualmente o processo de verificação.

Demonstraremos uma solução deste problema no módulo `CI_Restricao`. Tal solução será informar ao TLC quais estados visitar através de uma restrição de estado. Restrições de estado e restrições de ação são soluções artificiais (ou superficiais), porque limitam o espaço de estados apenas para o TLC. A lógica da especificação permanece a mesma.

Uma restrição de estado é definida em TLA$^+$ através de um predicado de estado. O TLC ignora todo estado onde a restrição de estado é **falsa**. Queremos limitar o valor de c a um valor máximo *ValorMaximo*, então a fórmula para a restrição será $c \leq ValorMaximo$. Isso significa que todo estado onde $c > ValorMaximo$ será ignorado.

Na prática, faremos o seguinte. No módulo `CI_Restricao` (i) declararemos uma constante *ValorMaximo*; (ii) declararemos uma premissa onde *ValorMaximo* é um número natural qualquer; (iii) definiremos o predicado $Restricao \triangleq c \leq ValorMaximo$.

Para acompanhar essas mudanças, criaremos num novo arquivo de configurações `CI_Restricao.cfg`, contendo a atribuição de valor à constante *ValorMaximo*, e nova declaração de que a fórmula *Restricao* é uma restrição ao TLC. O módulo `CI_Restricao` e o arquivo de configuração que o acompanha estão disponíveis na íntegra nos Códigos 12 e 13, respectivamente.

Código 12 – Especificação de um contador infinito com restrição `CI_Restricao`

```
---- MODULE CI_Restricao ----
EXTENDS Naturals
CONSTANTS ValorMaximo
ASSUME ValorMaximo \in Nat
VARIABLES c

Ini == c = 0
MaisUm == c' = c + 1
Prox == MaisUm
Justica == WF_c(MaisUm)
SpecJusta == Ini /\ [][Prox]_c /\ Justica

Restricao == c <= ValorMaximo
=============================
```

Código 14 – Especificação de um contador finito `ContadorFinito`

```
---- MODULE ContadorFinito ----
EXTENDS Naturals

CONSTANTS ValorMaximo
ASSUME ValorMaximo \in Nat
VARIABLES c

Ini == c = 0
MaisUm == /\ c < ValorMaximo
          /\ c' = c + 1
Prox == MaisUm
Justica == WF_c(MaisUm)
SpecJusta == Ini /\ [][Prox]_c /\ Justica

InvarianteDeTipos == c \in 0..ValorMaximo
================================
```

Nota-se que, no arquivo de configuração, atribuímos um número natural arbitrário para *ValorMaximo* e que redefinimos `Nat`.

Código 13 – Arquivo de configurações para `CI_Restricao.cfg`

```
SPECIFICATION SpecJusta
CONSTANTS ValorMaximo = 3
          Nat = {0, 1, 2, 3}
CONSTRAINT Restricao
```

Verificando o par de módulo e arquivo de configuração para `CI_Restricao`, recebemos um resultado positivo, ou seja, nenhum erro foi apontado e nenhum contraexemplo gerado. Porém, é uma verificação simplória porque não definimos propriedades de interesse: os únicos erros detectáveis seriam erros sintáticos, semânticos ou impasse. Além disso, utilizar restrições para verificar especificações mais complexas pode limitar o espaço de estados de maneira imprevisível, esconder erros de interesse, e impedir a verificação de propriedades de futuro.

4.2.2 Contador finito

Um contador finito limitará o valor de sua variável contadora c a um valor máximo na própria lógica do sistema, em vez de através de restrições. Em `ContadorFinito` modificaremos a ação *MaisUm* condicionando seu acontecimento ao valor da variável c ser menor do que o da constante *ValorMaximo*.

Além disso, definiremos a propriedade *InvarianteDeTipos*, que afirma que o valor de c sempre pertence ao conjunto $0..$*ValorMaximo*. Apresentamos o módulo `ContadorFinito` no Código 14 e o arquivo de configuração que o acompanha no Código 15.

Código 15 – Arquivo de configuração `ContadorFinito.cfg`

```
SPECIFICATION SpecJusta
CONSTANTS ValorMaximo = 3
          Nat = {0, 1, 2, 3}
INVARIANT InvarianteDeTipos
```

Verificando o módulo `ContadorFinito` com o arquivo de configuração do Código 15, recebemos um contraexemplo (Figura 17) que pode parecer estranho: um impasse. Aqui o TLC nos mostra que um impasse acontece quando o valor de c alcança *ValorMaximo*.

```
Deadlock reached.
The behavior up to this point is:
State 1: <Initial predicate>
c = 0
State 2: <MaisUm>
c = 1
State 3: <MaisUm>
c = 2
State 4: <MaisUm>
c = 3
```

Figura 17 – Contraexemplo para *SpecJusta* do módulo `ContadorFinito`

O TLC entende impasse como violação da propriedade $\Box Enabled\ \langle Prox \rangle_v$. Essa violação significa que a relação de transições *Prox* parou de ser habilitada em algum momento. A única ação em *Prox* é *MaisUm*, e *MaisUm* só é habilitada se for possível avançar o contador. A restrição ao espaço de estados impediu o TLC de avançar o contador ao limitar o valor de c, portanto, o TLC detectou impasse na especificação quando o contador alcançou o estado que entendemos como terminal. Evidentemente, não consideramos que um sistema alcançar seu estado terminal seja um impasse: é um comportamento esperado.

Uma solução que evitaria mudar a especificação seria desativar a detecção de impasses no arquivo de configuração acrescentando a declaração `CHECK_DEADLOCK FALSE`. Porém, preferimos ser explícitos e escreveremos uma especificação com uma nova ação que manterá o sistema no estado terminal quando c alcançar *ValorMaximo* no módulo `CF_EstadoTerminal`. Esta nova ação, *Fim*, afirmará que, quando $c = ValorMaximo$, é esperado que c não mude. A ação *Fim* não requer justiça: como um impasse só é detectado pelo TLC quando nenhuma ação for *habilitada* – e não quando nenhuma ação *acontecer* –, então basta definir a ação *Fim* e acrescentá-la à relação de transições.

Para reforçar o uso de propriedades de futuro, neste novo módulo definiremos duas delas: (i) *Termina*, onde afirmamos que o valor de c alcança *ValorMaximo* e permanece assim para sempre, e (ii) *ContadorMuda*, onde afirmamos que c assume todos os valores entre 0 e *ValorMaximo* pelo menos uma vez e em qualquer ordem. O trecho relevante de

Código 16 – Trecho relevante do módulo `CF_EstadoTerminal`

```
Fim == /\ c = ValorMaximo
       /\ UNCHANGED c

Prox == MaisUm \/ Fim

Justica == WF_c(MaisUm)

Termina == <>[](c = ValorMaximo)
ContadorMuda == \A n \in 0..ValorMaximo: <>(c = n)
```

`CF_EstadoTerminal` contendo as modificações sobre `ContadorFinito` está disponível no Código 16.

O arquivo de configuração para a próxima verificação com TLC está apresentado no Código 17.

Código 17 – Arquivo de configuração `CF_EstadoTerminal.cfg`

```
SPECIFICATION SpecJusta
CONSTANTS ValorMaximo = 3
          Nat = {0, 1, 2, 3}
INVARIANT InvarianteDeTipos
PROPERTY Termina ContadorMuda
```

O resultado desta última verificação nos retornou, em menos de um segundo, um resultado positivo após visitar cinco estados, dos quais quatro são distintos.

4.2.3 Contador finito com duas ações atômicas sequenciais

O contador finito especificado no módulo `CF_EstadoTerminal` lê o valor da variável c e muda seu valor num único passo atômico através da ação *MaisUm*. Nesta seção criaremos um módulo `ContadorFinito2` para ilustrar ações em TLA$^+$ em menor granularidade: a ação *MaisUm* será dividida em duas ações, ou seja, dois passos atômicos em vez de um. Dividir uma ação em duas nos aproximará de uma implementação real, de modo que o comportamento desta especificação será similar ao observado no pseudocódigo do Código 18, ou seja, similar a um laço onde se testa o valor da variável c em um passo atômico, e, se c for menor do que *ValorMaximo*, outro passo atômico modifica o valor de c.

Código 18 – Pseudocódigo da especificação em `ContadorFinito2`

```
1: while c < ValorMaximo:
2:     c := c + 1
```

Quando lemos o pseudocódigo no Código 18, a ordem de execução dos comandos é evidente. Porém, diferente dos comandos em linguagens de programação, ações em lógica

temporal não têm ordem implícita. O primeiro passo para escrevermos uma especificação a partir de um pseudocódigo será explicitar tanto os saltos entre instruções quanto o estado terminal, como fizemos no Código 19. Assim, o `while` se tornará um `if` com saltos para instruções atômicas diferentes, e acrescentaremos uma pseudoinstrução `fim` para representar um estado terminal.

Nota-se que, como TLA$^+$ não é linguagem de programação, podemos modificar a variável *c* num único passo atômico. Se um programa implementasse esta especificação, ele provavelmente realizaria múltiplas operações intermediárias de leitura e escrita para modificar a variável *c*.

Código 19 – Pseudocódigo da especificação em `ContadorFinito2` com saltos explícitos entre instruções e estado terminal

```
1: if c < ValorMaximo goto 2
   else goto 3
2: c := c + 1; goto 1
3: fim
```

A partir daqui, traduzir o pseudocódigo do Código 19 para TLA$^+$ se tornou uma tarefa mais fácil porque cada instrução terá a uma ação correspondente em TLA$^+$. O próximo passo será ordenar as instruções. Modelaremos o mecanismo de ordenação de instruções em TLA$^+$ através de uma nova variável chamada *pc*, do inglês "*program counter*", um nome convencional. O valor de *pc* representará a próxima instrução a ser executada: uma ação, representando a execução atômica de uma instrução correspondente, estará condicionada ao valor atual da variável *pc*. Como *pc* representa a próxima instrução, e temos três instruções, *pc* terá três valores possíveis que precisamos modelar de alguma forma. Escolheremos estes três valores possíveis como as *strings* "1", "2" e "3".

Código 20 – Primeira metade do módulo `ContadorFinito2`

```
--- MODULE ContadorFinito2 ----
EXTENDS Naturals
CONSTANTS ValorMaximo
ASSUME ValorMaximo \in Nat

VARIABLES c, pc
vars == <<c, pc>>

\* 1: if (c < ValorMaximo)
\*       goto 2
\*    else
\*       goto 3
I1 == /\ pc = "1"
      /\ IF c < ValorMaximo
         THEN pc' = "2"
         ELSE pc' = "3"
      /\ UNCHANGED c

\* 2: c := c + 1; goto 1
I2 == /\ pc = "2"
      /\ c' = c + 1
      /\ pc' = "1"

\* 3: fim
Fim == /\ pc = "3"
       /\ UNCHANGED vars
```

Código 21 – Segunda metade de `ContadorFinito2`

```
Ini == /\ c = 0
       /\ pc = "1"

Prox == I1 \/ I2 \/ Fim

Justica == /\ WF_vars(I1)
           /\ WF_vars(I2)
SpecJusta == /\ Ini
             /\ [][Prox]_vars
             /\ Justica

(* ------------------------- *)

InvarianteDeTipos ==
  /\ c \in 0..ValorMaximo
  /\ pc \in {"1", "2", "3"}
ContadorMuda ==
  \A n \in 0..ValorMaximo:
    <>(c = n)
Termina ==
  <>[](/\ c = ValorMaximo
       /\ pc = "3")
================================
```

As três instruções do pseudocódigo serão representadas pelas ações *I1*, *I2* e *Fim*, respectivamente, como apresentamos na primeira metade da especificação (Código 20). Nota-se que definimos *vars* como a tupla de variáveis declaradas *c* e *pc* para utilizarmos *vars* em vez `<<c, pc>>` repetidamente nas fórmulas de relação de transições e de justiça. Nomear a tupla de variáveis como *vars* é outra convenção.

As fórmulas de estados iniciais, relação de transições, justiça e especificação com justiça são apresentadas nesta ordem no Código 21. O contador possui um único estado inicial onde *c* é zero e *pc* é "1". A relação de transições é a disjunção de *I1*, *I2* e *Fim*. A fórmula de justiça é composta somente de justiça fraca sobre as ações *I1* e *I2*: a ação *I1* só será habilitada até que aconteça, e o mesmo é verdade para *I2*. Nota-se novamente que uma condição de justiça para a ação *Fim* não é necessária. Outra diferença relevante se comparamos este módulo com `ContadorFinito` é que a fórmula *InvarianteDeTipos* inclui os valores possíveis para *pc*, e que *Termina* inclui o fato de *pc* alcançar "3" e assim ficar para sempre.

Verificaremos o módulo `ContadorFinito2` com o arquivo de configuração idêntico a `CF_EstadoTerminal.cfg` (Código 17). O TLC nos retornou um resultado positivo, então a especificação *SpecJusta* usando as constantes definidas no arquivo de configuração satisfaz a invariante *InvarianteDeTipos* e as propriedades de futuro *Termina* e *ContadorMuda*.

4.2.4 Contador finito concorrente

Nesta seção especificaremos um contador finito concorrente num módulo chamado `CFC`. Transformaremos o contador finito sequencial do módulo `ContadorFinito2` em um contador concorrente com uma quantidade arbitrária de agentes. Ao final demonstraremos como TLC detecta a violação de atomicidade causada quando variáveis são compartilhadas entre múltiplos agentes sem os devidos cuidados.

O módulo `CFC` conterá as mesmas variáveis do módulo `ContadorFinito2`: c e pc. Porém, agora c representa uma variável compartilhada por todos os agentes do sistema, enquanto pc representa uma variável local, ou seja, cada agente possui um valor de pc inacessível aos outros agentes.

Nota-se que a lógica temporal de ações não distingue entre variáveis locais e globais. A distinção entre uma variável ser local ou global depende de nossa interpretação sobre os valores das variáveis. Essa distância entre uma especificação e sua interpretação ressalta a importância de documentar especificações em detalhes, e destaca como formalismos ainda requerem documentação em linguagem natural – uma linguagem informal – para comunicação de informações cruciais.

Enquanto c continua um número natural, pc terá que mudar. Mencionamos que pc representará uma variável (i) local a cada agente e (ii) inacessível a outros agentes do sistema. Nota-se que:

(i) Para modelar uma variável **local** a cada agente ag, mapearemos ag ao valor de seu contador de instruções $pc[ag]$ utilizando uma função cujo domínio é o conjunto *Agentes* e cuja imagem é o conjunto de instruções {"1", "2", "3"}.

(ii) Para modelar uma variável **inacessível a outros agentes**, porém, dependemos somente de como definimos as ações do sistema: nada impede – logicamente falando – de especificarmos um sistema onde um agente de modificar uma variável destinada a outro agente. Por exemplo, nada impede a_1 de modificar o valor de $pc[a_2]$.

Modificar somente um ou alguns elementos de uma função é tão comum em TLA$^+$ que a linguagem possui a palavra-chave `EXCEPT` para nos ajudar nessas ocasiões e diminuir a possibilidade de erros de nossa parte. Por exemplo, suponhamos uma ação que modifica $pc[ag]$ para o valor de uma expressão exp. No Código 22 atribuímos a pc' o valor modificado de pc utilizando um mapeamento. No Código 23, expressamos que o pc' é idêntico a

pc, exceto (`EXCEPT`) $pc[ag]$, que é igual à avaliação da expressão *exp*. Usar `EXCEPT` deixa imediatamente claro que somente $pc[ag]$ mudou.

Código 22 – Exemplo de sintaxe sem `EXCEPT`

```
pc' = [ d \in DOMAIN Agentes |-> IF d = ag THEN exp ELSE pc[d] ]
```

Código 23 – Exemplo de sintaxe com `EXCEPT`

```
pc' = [ pc EXCEPT ![ag] = exp ]
```

Voltando à especificação do contador concorrente, os módulos `ContadorFinito2` e `CFC`, além de terem as mesmas variáveis, possuem definições de predicado de estados iniciais e da relação de transições similares. Apesar do pseudocódigo que representa nossa especificação não mudar (Código 19), nota-se que estamos especificando um sistema onde *múltiplos* agentes executam estas três instruções.

Portanto, este novo módulo muda devido a essa nova característica concorrente, apesar do pseudocódigo não mudar: teremos uma constante *Agentes*, e definiremos as ações desses agentes com os próprios agentes como argumento. As ações *I1*, *I2* e *Fim* do módulo `CFC` estão definidas no Código 24.

Código 24 – Trecho de `CFC` com as ações *I1*, *I2* e *Fim*

```
\* 1: if (c < ValorMaximo) goto 2
\*    else goto 3
I1(ag) == /\ pc[ag] = "1"
          /\ IF c < ValorMaximo
             THEN pc' = [ pc EXCEPT ![ag] = "2" ]
             ELSE pc' = [ pc EXCEPT ![ag] = "3" ]
          /\ UNCHANGED <<c>>

\* 2: c := c + 1; goto 1
I2(ag) == /\ pc[ag] = "2"
          /\ c' = c + 1
          /\ pc' = [ pc EXCEPT ![ag] = "1" ]

\* 3: fim
Fim == /\ \A ag \in Agentes: pc[ag] = "3"
       /\ UNCHANGED vars
```

Nota-se que a ação *Fim* não tem um agente como argumento. Esta ação representa o fim do programa, e não a execução de uma instrução por um agente, e o fim do programa acontece quando todos os agentes alcançarem a terceira pseudoinstrução.

As definições de *Prox*, *Justica* e *SpecJusta* do Código 25 já podem ser mais familiares. Destacamos as definições de *Ini*, *InvarianteDeTipos* e *Termina*:

- *Ini* afirma que o valor inicial de c é zero, e que a instrução inicial de todo agente ag é "1";
- *InvarianteDeTipos* afirma que c é um número entre 0 e *ValorMaximo*, e que pc é uma função do conjunto *Agentes* para o conjunto {"1", "2", "3"};
- *Termina* afirma que o valor de c alcança *ValorMaximo* e assim fica para sempre, e que, para todo agente ag, o valor de $pc[ag]$ se torna "3" e assim continua para sempre.

Código 25 – Trecho do módulo `CFC` incluindo especificação justa e propriedades temporais

```
Ini == /\ c = 0
       /\ pc = [ ag \in Agentes |-> "1" ]

Prox == \/ \E ag \in Agentes: I1(ag) \/ I2(ag)
        \/ Fim

Justica == \A ag \in Agentes: WF_vars(I1(ag)) /\ WF_vars(I2(ag))


SpecJusta == Ini /\ [][Prox]_vars /\ Justica

InvarianteDeTipos == /\ c \in 0..ValorMaximo
                     /\ pc \in [ Agentes -> {"1", "2", "3"} ]
ContadorMuda == \A n \in 0..ValorMaximo: <>(c = n)
Termina == <>[](/\ c = ValorMaximo
                /\ \A ag \in Agentes: pc[ag] = "3")
```

Para verificar as propriedades deste módulo, criaremos um arquivo de configurações `CFC.cfg` (Código 26): verificaremos se um sistema com dois agentes que conta até dois satisfaz a invariante *InvarianteDeTipos* e as propriedades *Termina* e *ContadorMuda*.

Código 26 – Arquivo de configurações `CFC.cfg` na íntegra

```
SPECIFICATION SpecJusta
CONSTANTS Agentes = {a1, a2}
          ValorMaximo = 2
          Nat = {0, 1, 2}
INVARIANT InvarianteDeTipos
PROPERTIES Termina ContadorMuda
```

Verificando o módulo `CFC` com o arquivo `CFC.cfg`, o TLC imediatamente nos retornou o contraexemplo, resumido da Figura 18a, afirmando que *InvarianteDeTipos* foi violada. Mais especificamente, a partir do estado n.º 4, o TLC nos mostra que é possível que múltiplos agentes leiam o valor de c como sendo $ValorMaximo - 1$, e avancem o valor de c

para além de *ValorMaximo*. Neste caso, o valor de c no estado n.º 7 é três, ultrapassando *ValorMaximo*, que é dois. No pior caso, o sistema que especificamos, quando for executado por N agentes ($N > 0$), poderá fazer o valor de c alcançar $N + ValorMaximo - 1$.

```
Invariant InvarianteDeTipos
is violated.
The behavior up to this
point is:

State 1: <Initial predicate>
/\ c = 0
/\ pc = (a1 :> "1" @@ a2 :> "1")
State 2: <I1(a1)>
/\ c = 0
/\ pc = (a1 :> "2" @@ a2 :> "1")
State 3: <I2(a1)>
/\ c = 1
/\ pc = (a1 :> "1" @@ a2 :> "1")
State 4: <I1(a1)>
/\ c = 1
/\ pc = (a1 :> "2" @@ a2 :> "1")
State 5: <I1(a2)>
/\ c = 1
/\ pc = (a1 :> "2" @@ a2 :> "2")
State 6: <I2(a1)>
/\ c = 2
/\ pc = (a1 :> "1" @@ a2 :> "2")
State 7: <I2(a2)>
/\ c = 3
/\ pc = (a1 :> "1" @@ a2 :> "1")
```

(a) Contraexemplo resumido para `InvarianteDeTipos`

```
Temporal properties were
violated.
The following behavior
constitutes a counter-example:

State 1: <Initial predicate>
/\ c = 0
/\ pc = (a1 :> "1" @@ a2 :> "1")
State 2: <I1(a2)>
/\ c = 0
/\ pc = (a1 :> "1" @@ a2 :> "2")
State 3: <I2(a2)>
/\ c = 1
/\ pc = (a1 :> "1" @@ a2 :> "1")
State 4: <I1(a2)>
/\ c = 1
/\ pc = (a1 :> "1" @@ a2 :> "2")
State 5: <I1(a1)>
/\ c = 1
/\ pc = (a1 :> "2" @@ a2 :> "2")
State 6: <I2(a1)>
/\ c = 2
/\ pc = (a1 :> "1" @@ a2 :> "2")
State 7: <I2(a2)>
/\ c = 3
/\ pc = (a1 :> "1" @@ a2 :> "1")
State 8: <I1(a1)>
/\ c = 3
/\ pc = (a1 :> "3" @@ a2 :> "1")
State 9: <I1(a2)>
/\ c = 3
/\ pc = (a1 :> "3" @@ a2 :> "3")
State 10: Stuttering
```

(b) Contraexemplo resumido para alguma propriedade de futuro

Figura 18 – Contraexemplos resumidos para o módulo `CFC`

Ter encontrado este contraexemplo não significa que as outras propriedades foram satisfeitas. Para buscarmos outros contraexemplos, comentaremos a linha `INVARIANT InvarianteDeTipos` do arquivo de configuração `CFC.cfg` e verificaremos novamente nossa especificação. Desta vez o TLC encontrou o contraexemplo da Figura 18b, mas não apontou qual das propriedades de futuro – `Termina` e `ContadorMuda` – foi violada, uma característica inconveniente do algoritmo de verificação de vivacidade. Geralmente falando, quando é difícil descobrir qual propriedade de futuro foi insatisfeita, teremos que verificar uma propriedade de futuro por vez. Este não será nosso caso. Resolvemos a dúvida sobre qual propriedade de futuro foi insatisfeita quando percebemos que ambos os agentes alcan-

çam o fim de seus programas (a instrução "3") com $c = 3$. A propriedade *ContadorMuda* afirma que c percorre todos os números naturais ($\{0, 1, 2\}$) em alguma ordem, o que é verdade para o contraexemplo da Figura 18b. Já a propriedade *Termina* afirma que os agentes terminam na terceira instrução com $c = ValorMaximo$, e *ValorMaximo* é 2. Em resumo, este comportamento de contraexemplo termina com infinitos passos balbuciantes demonstrando que a subfórmula $\Diamond\Box(c = ValorMaximo)$ da propriedade *Termina* nunca é satisfeita.

Com estes dois contraexemplos da Figura 18 e nossos conhecimentos de computação concorrente, podemos concluir que as propriedades definidas foram insatisfeitas pela especificação porque não existe sincronização de acesso à variável c para evitar que determinado agente leia um valor obsoleto de c.

4.2.5 Contador finito concorrente com *lock*

Nesta seção especificaremos um novo módulo `CFC_Lock` que resolverá o problema encontrado ao final da subseção anterior. Para evitar que um agente do sistema leia um valor obsoleto de c e avance c além de *ValorMaximo*, aplicaremos um mecanismo de sincronização de leitura e escrita à variável compartilhada c, garantindo exclusão mútua. O mecanismo de sincronização utilizado será uma *lock*. Veremos como podemos reaproveitar as definições das ações *Trancar* e *Destrancar* definidas no módulo `Lock` da Seção 4.1 através da instanciação de um módulo em outro.

Acrescentar uma *lock* ao pseudocódigo do Código 19 é relativamente simples, mas somente cercar a leitura e escrita de c com o respectivo trancamento e destrancamento da *lock*, como vemos no Código 27, não basta. Sabemos que esta solução é insuficiente porque a violação de atomicidade aparece devido à leitura de um valor obsoleto de c, enquanto aqui estamos garantindo somente a atomicidade de escrita.

Código 27 – Exemplo de pseudocódigo com *lock* incorreto

```
1: while c < ValorMaximo:
2:     Trancar()
3:     c := c + 1
4:     Destrancar()
5: fim
```

Em vez disso, devemos incluir a leitura do valor de c na seção crítica. O pseudocódigo do Código 28 mostra uma solução com laço `while` incondicional. Um agente acessa a seção crítica e, se $c = ValorMaximo$, ele destranca a *lock* e sai do laço utilizando o comando `break`; senão, ele avança c e destranca a *lock*, voltando ao começo do laço. A sua transformação para um formato mais fácil de traduzir para TLA$^+$ está contido no Código 29.

Código 28 – Pseudocódigo de nossa solução

```
1: while True:
2:     Trancar()
3:     if c = ValorMaximo:
4:         Destrancar(); break
5:     c := c + 1
6:     Destrancar()
```

Código 29 – Transformação do pseudocódigo de Código 28

```
1: Trancar(); goto 2
2: if c = ValorMaximo: goto 3
   else: goto 4
3: Destrancar(); goto 6
4: c := c + 1; goto 5
5: Destrancar(); goto 1
6: fim
```

Portanto, comecemos o processo de especificação do módulo `CFC_Lock`. Temos um módulo conceitualmente similar a `CFC`, mas reescreveremos suas ações tal que as propriedades `InvarianteDeTipos`, `Termina` e `ContadorMuda` sejam satisfeitas.

Mencionamos que aproveitaremos as ações *Trancar* e *Destrancar* do módulo `Lock`. Toda instanciação de um módulo remoto num módulo local requer um mapeamento entre variáveis e constantes do módulo local para as variáveis e constantes do módulo remoto. Quando há variáveis com nomes em comum entre ambos os módulos, por exemplo, ambas têm uma variável v, supõe-se um mapeamento $v \leftarrow v$ entre tais módulos, e esse mapeamento pode ser omitido na instanciação como discutido na Seção 3.2.1.

O módulo `Lock` possui a variável *dono* e o módulo `CFC_Lock` possui as variáveis *pc* e *c*. Não existe um mapeamento claro entre elas, então simplesmente redeclararemos *dono* no módulo `CFC_Lock`. Além disso, as ações *Trancar* e *Destrancar* utilizam a constante *Ninguem*. Precisamos redeclará-la aqui pelo mesmo motivo. Redeclarar a premissa sobre o valor da constante *Ninguem* não é obrigatório, mas é importante também.

O restante do módulo `CFC_Lock` segue um processo já discutido: traduzimos as instruções do pseudocódigo (Código 29) para ações em TLA$^+$; aplicamos justiça fraca sobre essas ações, exceto *Fim* por ser desnecessário; e definimos as propriedades relevantes. Evitando nos repetirmos, disponibilizamos o módulo `CFC_Lock` na íntegra no Apêndice 6, e focaremos nas propriedades *InvarianteDeTipos*, *Termina* e *ContadorMuda* disponíveis no Código 30.

Código 30 – Propriedades *InvarianteDeTipos*, *Termina* e *ContadorMuda* do módulo `CFC_Lock`

```
InvarianteDeTipos ==
  /\ c \in 0..ValorMaximo
  /\ pc \in [ Agentes -> {"1", "2", "3", "4", "5", "6"} ]
  /\ dono \in Agentes \cup { Ninguem }
Termina == <>[](/\ c = ValorMaximo
                /\ dono = Ninguem
                /\ \A ag \in Agentes: pc[ag] = "6")
ContadorMuda == \A n \in 0..ValorMaximo: <>(c = n)
```

Utilizando as configurações do Código 31, o TLC nos retornou um resultado positivo após visitar 30 estados, dos quais 26 são distintos.

Código 31 – Arquivo de configurações `CFC_Lock.cfg`

```
SPECIFICATION SpecJusta
CONSTANTS Agentes = {a1, a2}
          ValorMaximo = 2
          Nat = {0, 1, 2}
          Ninguem = Ninguem
INVARIANT InvarianteDeTipos
PROPERTY ContadorMuda Termina
```

Nota-se que verificação exaustiva não é prova matemática: escolhemos valores arbitrários no arquivo de configurações e devemos decidir se o resultado positivo do TLC nos satisfaz ou não. Todo processo de verificação depende de confiarmos que as propriedades foram definidas corretamente, e que elas continuam satisfazendo a especificação dados outros valores para constantes. Como esta é uma especificação de um sistema concorrente que envolve algumas instruções, este resultado positivo pode não ser tão evidente. Para ganharmos confiança em nossa especificação, podemos aumentar o número de agentes e o valor máximo, lembrando de expandir o conjunto de números naturais `Nat`. Criamos um novo arquivo de configuração `CFC_Lock2.cfg` acompanhando esta mudança (Código 32).

Código 32 – Arquivo de configurações `CFC_Lock2.cfg`

```
SPECIFICATION SpecJusta
CONSTANTS Agentes = {a1, a2, a3, a4, a5, a6}
          ValorMaximo = 6
          Nat = {0, 1, 2, 3, 4, 5, 6}
          Ninguem = Ninguem
INVARIANT InvarianteDeTipos
PROPERTY ContadorMuda Termina
```

Ao verificarmos este módulo do contador concorrente com esta nova configuração, o TLC novamente nos retorna um resultado positivo ao verificarmos um sistema com seis agentes que conta até seis. Ao longo da verificação, que durou dois segundos, o TLC explorou 722 estados, sendo 562 deles distintos.

Esta verificação com um número maior de estados nos trouxe mais confiança, mas podemos ir além: definiremos e verificaremos novas propriedades sobre esse contador concorrente. Por exemplo, podemos verificar que existe exclusão mútua no acesso à seção crítica (*ExclusaoMutua*), e que todo agente que desejar acessar a seção crítica a acessará futuramente (*AusenciaDeInanicao*). Apresentamos estas definições em TLA$^+$ no Código 33.

Código 33 – Propriedades *ExclusaoMutua* e *AusenciaDeInanicao* em TLA$^+$

```
EmSecaoCritica(ag) == pc[ag] \in {"2", "3", "4", "5"}
ExclusaoMutua ==
  \A ag1, ag2 \in Agentes:
    (ag1 # ag2) => ~(EmSecaoCritica(ag1) /\ EmSecaoCritica(ag2))

AusenciaDeInanicao ==
  \A ag \in Agentes: (pc[ag] = "1") ~> (pc[ag] = "2")
```

A propriedade *ExclusaoMutua* é uma invariante que afirma que, para todo par de agentes distintos ag_1 e ag_2, não acontece de ag_1 e ag_2 estarem na seção crítica ao mesmo tempo. Definimos seção crítica como o conjunto de instruções {"2", "3", "4", "5"}. A propriedade *AusenciaDeInanicao* é uma propriedade de futuro que afirma que todo agente *ag* alcança a instrução "2" após a instrução "1", ou seja, que todo agente entra na seção crítica em algum momento.

A próxima verificação do módulo `CFC_Lock` será acompanhada pelo arquivo `CFC_Lock3.cfg` idêntico senão pela adição da invariante *ExclusaoMutua* e da propriedade de futuro *AusenciaDeInanicao*. O Código 34 apresenta o trecho relevante deste arquivo.

Código 34 – Trecho relevante do arquivo de configuração `CFC_Lock3.cfg`

```
INVARIANT InvarianteDeTipos ExclusaoMutua
PROPERTY ContadorMuda Termina AusenciaDeInanicao
```

Verificando novamente a especificação utilizando o arquivo de configuração do Código 34, o TLC nos retornou um resultado positivo. Como a especificação não mudou, o verificador novamente visitou 722 estados, sendo 562 deles distintos.

Nota-se que a propriedade *AusenciaDeInanicao* é uma propriedade bastante fraca por dois motivos: (i) ela não afirma sobre o tempo que um agente deve esperar para acessar a seção crítica e (ii) ela não afirma sobre o acesso de todo agente à instrução de soma do contador (instrução "4").

Quanto ao motivo (i), afirmar sobre o tempo de espera requer definir algum tipo de relógio global, e um limite máximo de espera, além das restrições de tempo das outras instruções. Quanto ao motivo (ii), podemos dividi-lo em dois subproblemas: (a) quando o número de agentes é menor ou igual a *ValorMaximo* e (b) quando o número de agentes é maior do que *ValorMaximo*. A solução para o subproblema (a) pode ser alcançada definindo uma fila de prioridades. A solução para o subproblema (b) não existe: é impossível que todos os N agentes modifiquem o contador pelo menos uma vez quando $N >$ *ValorMaximo*. O detalhamento destas soluções foge do escopo deste trabalho.

Finalmente, verificaremos que a especificação do contador finito concorrente com *lock* implementa a especificação do contador finito da Seção 4.2.2. A palavra "implementa" neste contexto significa implicação lógica: a especificação do contador finito concorrente

implica no contador finito sequencial. Este tipo de verificação é feito no próprio módulo `CFC_Lock`, e se resume a verificar que *SpecJusta* do módulo `CFC_Lock` satisfaz a propriedade *SpecJusta* do módulo `ContadorFinito`.

Quando instanciamos um módulo, um mapeamento é necessário: mapearemos as variáveis e constantes do contador concorrente para as variáveis do contador sequencial. Lembremos que as variáveis do módulo `CFC_Lock` são c, pc e *dono*, enquanto o módulo `ContadorFinito` só possui a variável c. Portanto, o mapeamento será: $c \leftarrow c$, enquanto pc e *dono* são ignoradas. Como a variável c de um módulo é mapeada para c de outro, e ambas representam o mesmo conceito, podemos omiti-la da instanciação. O Código 35 contém o trecho de `CFC_Lock` onde instanciamos `ContadorFinito` e definimos a propriedade *ImplementaContadorFinito*. O símbolo `!` é utilizado para referenciar a fórmula *SpecJusta* da instância `CF` do módulo `ContadorFinito`.

Código 35 – Trecho do módulo `CFC_Lock` incluindo instanciação de *ContadorFinito*

```
CF == INSTANCE ContadorFinito
ImplementaContadorFinito == CF!SpecJusta
```

Para acompanhar esta última propriedade, e já confiantes que a especificação *SpecJusta* satisfaz todas as propriedades definidas, adicionaremos *ImplementaContadorFinito* à declaração de propriedades. O arquivo de configuração resultante se encontra no Apêndice 6.

O resultado do TLC é positivo ao verificar todas essas propriedades deste modelo em três segundos. Ou seja, agora também sabemos que este contador concorrente implementa o contador sequencial. Mais especificamente, essa chamada "implementação" significa que as mudanças do valor de c descritos por *SpecJusta* do módulo `CFC_Lock` satisfazem as mudanças do valor de c descritas pela especificação *SpecJusta* do módulo `ContadorFinito`: c é um valor que cresce em ordem crescente de 0 a *ValorMaximo* em ambas especificações.

4.2.6 Discussão

Nesta seção vimos o processo incremental de especificar um contador finito concorrente a partir de um contador infinito sequencial. Este processo envolveu a escrita de pseudocódigo, a tradução deste pseudocódigo para TLA$^+$, e a verificação de que a especificação resultante satisfazia determinadas propriedades, especialmente a propriedade de exclusão mútua. Ao final deste processo, conseguimos verificar que a especificação do contador finito concorrente finito possui exclusão mútua, que todo agente consegue acessar a seção crítica, e que a especificação do contador finito concorrente implica na especificação do contador finito sequencial.

Ao longo deste processo de especificação vimos como lidar com a falta de tipos em TLA$^+$, vimos que não existe distinção entre variáveis locais ou compartilhadas e que isso

depende de nossa interpretação sobre os valores das variáveis, e vimos reações possíveis aos resultados positivos ou contraexemplos do TLC. Um resultado positivo do TLC não significa ausência de erros da especificação, (i) porque é possível termos atribuído valores que convenientemente fazem a especificação satisfazer as propriedades verificadas, e (ii) porque temos de nos certificar que as propriedades definidas formalmente expressam o mesmo que as propriedades desejadas.

As etapas de tradução de pseudocódigo para TLA$^+$, incluindo o uso da variável `pc` não foram criadas por nós. Na realidade, algo similar acontece durante a tradução automática de PlusCal para TLA$^+$. PlusCal (LAMPORT, 2009) é uma linguagem do ecossistema de TLA$^+$ criada por Leslie Lamport. Um algoritmo é escrito em PlusCal dentro de um módulo de TLA$^+$, e um tradutor automático traduz este algoritmo para TLA$^+$ para este mesmo módulo. Como tanto o algoritmo em PlusCal quanto a especificação resultante em TLA$^+$ residem no mesmo módulo, o processo de especificação é transparente. Por PlusCal ser fundamentado na lógica TLA, e por ter sintaxe similar ao pseudocódigo, a linguagem tem semântica formal bem definida, nos permitindo escrever, verificar e provar pseudocódigos formalmente. A linguagem também contém etiquetas para grupos de instruções atômicas, e um mecanismo para chamadas de procedimentos, facilitando escrever determinado algoritmo em diferentes níveis de granularidade e abstração. Porém, PlusCal tem certas desvantagens: é uma linguagem menos poderosa do que TLA$^+$ para descrever sistemas concorrentes, sendo necessário acrescentar detalhes, principalmente de vivacidade, diretamente em TLA$^+$; além disso, pode ser necessário entender como um algoritmo em PlusCal será traduzido para TLA$^+$ antes de escrevê-lo.

5 DISCUSSÃO

Este TCC foi motivado pela dificuldade de depurar programas distribuídos. O objetivo do trabalho seria demonstrado ao especificarmos e verificarmos um algoritmo distribuído relativamente simples com TLA$^+$ e TLC, respectivamente. Selecionamos o algoritmo valentão para eleição de líder, baseado na descrição em (TANENBAUM; STEEN, 2017) devido a sua simplicidade. Escrever especificações deste algoritmo conforme as descrições do livro foi um processo longo e sem sucesso. De qualquer forma, essa experiência nos informou sobre quais elementos de TLA e TLA+ deveriam ser destacados no trabalho tal como é hoje, e pontos a se discutir sobre essa experiência. Nossa discussão será dividida em duas partes: TLA+ e TLC; e a experiência de aprender TLA+.

5.1 TLA+ E TLC

Nesta seção nos apoiaremos principalmente no artigo "Formal Specification: A Roadmap" de Axel van Lamsweerde (LAMSWEERDE, 2000) para discutir TLA$^+$ e TLC.

5.1.1 Utilidade de TLA+

Lamsweerde lista cinco critérios para avaliar a utilidade de uma técnica de métodos formais: poder expressivo; capacidade de construção, gerenciamento e evolução; poder e eficiência de análise; comunicabilidade; usabilidade. Uma avaliação supõe uma comparação com outros pontos de referência. Como só temos experiência com TLA$^+$, não podemos compará-la com outras técnicas. Assim, cada critério inspirará algum comentário sobre a TLA$^+$ e seu verificador de modelos. A lista dos critérios simplificados de Lamsweerde incluindo nossos comentários segue:

- **Poder expressivo:** O quão expressiva é a linguagem de especificação? Qual é seu viés semântico?

  TLA$^+$ é uma linguagem que prioriza conceitos lógicos e matemáticos em vez de conceitos de programação. Comparado com descrições em linguagens de programação, a lógica matemática nos permite descrever sistemas em altos níveis de abstração. Descrever sistemas em abstrato antes de implementá-los, principalmente sistemas complexos, evita que nos percamos em detalhes de implementação, facilitando a busca por erros fundamentais de projeto. A facilidade de encontrar erros na etapa de especificação diminui o risco de encontrar e corrigir erros de projeto durante sua implementação.

  O viés semântico de TLA$^+$ tende para especificações de sistemas concorrentes por facilitar a descrição de passos não determinísticos. Execuções de passos sequen-

ciais dependem de algum mecanismo de ordenação, por exemplo, através de uma variável auxiliar. De qualquer modo, nada nos impediria de usar a linguagem para especificar qualquer outro tipo de sistema discreto não concorrente. Em questão de expressividade, TLA+ tem certas limitações por ser baseada em lógica temporal linear. Por exemplo, usamos variáveis auxiliares para nos referir a algum estado passado ou futuro.

- **Capacidade de construção, gerenciamento e evolução:** Quais mecanismos de estruturação existem para escrita incremental de especificações complexas, tal que as mudanças no domínio do problema sejam facilmente refletidas nas mudanças da especificação do sistema?

  Módulos de TLA$^+$ nos permitem compartimentalizar especificações e a reutilizar fórmulas, mas as grandes contribuições em especificação construtiva são características da lógica TLA, isto é, passos balbuciantes, composição entre especificações ser conjunção lógica, e refinamento entre especificações ser implicação lógica.

- **Poder e eficiência de análise:** O quão poderosa e eficiente é a análise das ferramentas automatizadas? É possível verificar modelos, ou provar teoremas? Existe algum *feedback* automatizado que facilite o processo de especificação?

  TLA$^+$ é uma linguagem tão expressiva que certas fórmulas são impossíveis de serem verificadas exaustivamente. Ainda assim, o conjunto de fórmulas verificáveis é significativo. TLC é um verificador de modelos exaustivo, então consome memória rapidamente. Consequentemente, o número de estados gerados pelo modelo deve ser sempre considerado. Uma possível vantagem desse consumo de memória – o problema de explosão do espaço de estados – é incentivar a especificação de modelos bem concisos e mais fáceis de entender. O verificador Apalache (KONNOV; KUKOVEC; TRAN, 2019) é uma alternativa que promete uso econômico da memória por usar SMTs para verificar especificações em TLA$^+$. A desvantagem dessa abordagem é limitar a expressividade de TLA$^+$ por requerer anotações de tipos em variáveis e operadores. Em questão de *feedback*, sentimos falta de algum mecanismo de análise estática de especificações.

Finalmente, separamos os dois últimos critérios de Lamsweerde: comunicabilidade e usabilidade. Simplificando, o autor define esses termos como a facilidade de "pessoas razoavelmente bem treinadas" de lerem e escreverem "especificações de alta qualidade", respectivamente. Porém, são termos vagos. Deixaremos comentários sobre leitura e escrita de especificações para a Seção 5.2.

5.1.2 Fraquezas de TLA+

Lamsweerde lista sete fraquezas de métodos formais: escopo limitado, má separação de responsabilidades, uso de ontologias de baixo nível, isolamento na cadeia produtiva, pouca orientação, alto custo financeiro, *feedback* ruim de ferramentas.

Destas fraquezas, destacamos três relevantes no contexto de TLA$^+$ e as detalhamos a seguir:

- **Má separação de responsabilidades:** A abstração de TLA$^+$ faz com que não exista separação clara entre propriedades esperadas do sistema, suposições sobre o ambiente do sistema e propriedades do domínio de aplicação: tudo é descrito pela mudança de valores de variáveis conforme determinadas ações. O significado de cada variável do sistema depende de sua interpretação, e, portanto, da documentação da especificação.

- **Pouca orientação:** Além das ferramentas darem pouca orientação enquanto especificamos, faltam referências que ensinem modelagem e especificação de sistemas de modo "incremental, seguro e sistemático", como diz Lamsweerde. O livro de referência de TLA$^+$ – *Specifying Systems* – contém vários exemplos de modelagem incremental através de refinamento, mas são exemplos complexos que iniciantes teriam dificuldade de entender.

- **Isolamento na cadeia produtiva:** Especificações formais podem ficar isoladas vertical e horizontalmente de produtos de *software*. O isolamento vertical acontece quando uma especificação formal não se comunica com requisitos informais (*upstream*), nem com uma implementação real (*downstream*). O isolamento horizontal acontece quando uma especificação formal não se comunica com informações acompanhantes, como especificação informal, documentação de escolhas e informações do projeto. No contexto de TLA$^+$ há algumas tentativas de diminuir esse isolamento, seja através da geração de código (HACKETT et al., 2023), ou através da geração de testes (DAVIS; HIRSCHHORN; SCHVIMER, 2020).

5.1.3 Desafios para TLA+

Lamsweerde lista quinze desafios que métodos formais devem enfrentar para se tornarem um veículo essencial na engenharia de *software* de alta qualidade. Destes desafios, selecionamos o seguinte para TLA$^+$:

- **Integração:** Como nos asseguramos que uma especificação em TLA$^+$ satisfaz os requisitos informais, e que uma implementação satisfaz sua especificação formal?

- **Separação de responsabilidades:** Existe alguma forma estruturar uma linguagem derivada de TLA$^+$ que fosse igualmente poderosa, mas que separasse propriedades descritivas das propriedades prescritivas?
- **Especificação multiformato:** Como transformar uma especificação formal de TLA$^+$ num formato legível para pessoas sem familiaridade com o formalismo?

5.1.4 Sugestões para TLA+ e TLC

O projeto de TLA$^+$ reconhece alguns pontos de melhoria[1]. Dessas propostas, destacamos um algoritmo concorrente que verifique vivacidade e melhorias na qualidade de código e documentação do TLC. Além disso, temos algumas sugestões:

- **Módulos:** Um sistema de módulos que permita estender módulos de outros diretórios;
- **Interface da linha de comando**: Uma interface de linha de comando simplificada, e que permita a sobreposição de trechos de um arquivo de configuração, facilitando a execução de múltiplas verificações de modelos através de *scripts*;
- **Análise estática:** Um *parser* incremental que permita a análise estática de especificações;
- **Verificador de modelos:**
  - Melhores mensagens de erro sobre violação de refinamentos. As mensagens atuais são opacas. Sugerimos aplicar o dado mapeamento de refinamento às variáveis do contraexemplo para ilustrar como esta sequência de estados viola a propriedade de refinamento;
  - Mais operadores probabilísticos para uso em simulações probabilísticas. Por exemplo, distribuições exponencial e Poisson para simulação de sistemas de filas;
  - Uma implementação de TLC que compila especificações em vez de interpretá-las. A solução atual para otimizar a avaliação de expressões é aplicar manualmente os operadores `TLCEval` ou `TLCCache`. A ideia de um TLC que compila já foi considerada em (YU; MANOLIOS; LAMPORT, 1999);
  - Uma implementação de TLC para navegadores da *web*. Ferramentas complexas como Photoshop podem executadas diretamente por navegadores através de WebAssembly. A proximidade entre um usuário e a ferramenta através do navegador facilita a propagação de ideias. Este verificador executado pelo navegador poderia ser usado, por exemplo, em tutoriais interativos *online*;

[1] *List of projects to extend/improve TLA+, TLC, TLAPS or the Toolbox*: https://github.com/tlaplus/tlaplus/blob/564f2a6b256e2e524d73381a9973a2a1e6005329/general/docs/contributions.md

5.2 A EXPERIÊNCIA DE APRENDER TLA+

Os materiais principais de aprendizado de TLA$^+$ foram o artigo *The Temporal Logic of Actions* (LAMPORT, 1994), o livro *Specifying Systems* (LAMPORT, 2002), a série de vídeos de Leslie Lamport sobre TLA$^+$[2]. Materiais auxiliares foram o site `learntla.com`[3] e palestras da TLA+ Conf na internet[4]. Durante o aprendizado sempre surgem dúvidas e, felizmente, existe um fórum na plataforma Google Groups[5] onde membros da comunidade podem respondê-las.

Usar lógica matemática para descrever sistemas discretos é inicialmente desconfortável quando estamos mais acostumados a programá-los, então nossas primeiras especificações podem se parecer com programas. De qualquer modo, conseguimos escrever e verificar especificações em relativamente pouco tempo, mesmo que tais especificações sejam prolixas.

O maior benefício de aprender TLA$^+$ é perceber que o processo de especificar é, no mínimo, tão importante quanto a especificação resultante por nos fazer pensar profundamente sobre o problema. Percebemos que, como diz Hall em (HALL, 1990), após o processo de escrita e documentação da especificação formal, pode-se inclusive gerar uma especificação informal sem nenhuma matemática: uma especificação informal mais compreensível, mais precisa, e até mais curta do que uma especificação informal convencional.

Outro benefício de aprender TLA$^+$ (ou métodos formais em geral) é aprendermos a definir propriedades de interesse formalmente. Invariantes, por exemplo, são propriedades importantes de se afirmar inclusive sobre programas. Todo trecho de código, sejam laços de repetição, funções ou classes, têm invariantes que costumam ser implícitas para programadores. Explicitar essas invariantes facilita a depuração de código, mesmo que nenhuma verificação formal seja aplicada.

Não sentimos desvantagens nessa experiência, mas podemos questionar, por exemplo, se obteríamos os mesmos benefícios (ou mais) se aprendêssemos mais técnicas de métodos formais em menor profundidade.

Nossa maior dificuldade em aprender TLA$^+$ existiu devido ao autodidatismo, e não por alguma complexidade inerente da linguagem e sua lógica subjacente. Por exemplo, alguns detalhes importantes não receberam atenção adequada e só descobrimos nosso desentendimento durante reuniões de orientação deste TCC. Além disso, o principal material sobre TLA$^+$ – o livro *Specifying Systems* – não contém exercícios para testar nosso conhecimento teórico.

A segunda maior dificuldade nesse processo foi aprender a escrever boas especificações, principalmente bons modelos. Essa dificuldade provavelmente existe em todo método

[2] *TLA$^+$ Video Course*: https://lamport.azurewebsites.net/video/videos.html
[3] *Learn TLA*: https://www.learntla.com
[4] *TLA+ Conf*: https://conf.tlapl.us/home/
[5] `tlaplus` *Google Groups*: https://groups.google.com/g/tlaplus

formal. *Specifying Systems* contém diversos exemplos do que podemos chamar de boas especificações, mas seus modelos são raramente discutidos: uma especificação pronta só é usada para ilustrar conceitos de TLA$^+$. Por outro lado, Lamport separou um capítulo de dez páginas (Capítulo 7) para dar conselhos sobre como escrever boas especificações.

Mesmo não sendo o maior impeditivo, TLA tem suas dificuldades inerentes, mesmo que elas só apareçam raramente, ou para usuários mais avançados. Essas dificuldades são:

- Invariância sob balbuciação: a escrita de propriedades invariantes sob balbuciação pode dificultar a escrita de algumas propriedades mais sutis;
- Refinamento de especificações: é difícil encontrar mapeamentos de refinamento adequado, e, às vezes, esse mapeamento não existe;
- Composição de especificações: como garantir que, compartilhamento de estado entre especificações, que suas ações devem ou não entrelaçar (LAMPORT, 2002, Cap. 10).

Este trabalho não compara TLA$^+$ com outras linguagens de especificação de sistemas concorrentes, então não há motivo para preferirmos TLA$^+$ a formalismos alternativos para a especificação de sistemas concorrentes. Ainda assim, acreditamos que aprender TLA$^+$ foi uma experiência positiva, ou seja, não temos arrependimentos. Porém, afirmarmos que foi uma experiência *benéfica* seria mais subjetivo. Pensar sobre determinado problema e modelá-lo, explicitar suposições, e obter melhor intuição e maior confiança sobre a solução do problema são características que todos os métodos formais costumam trazer. Talvez tenhamos fortalecido essas características, mas seria um julgamento pouco modesto.

Sem a formação dada pelo curso de Ciência da Computação, aprender TLA$^+$ teria sido impraticável. Disciplinas como Lógica, Inteligência Artificial, Arquitetura de Computadores, Sistemas Operacionais, Computação Concorrente e Sistemas Distribuídos contribuíram para o entendimento dos exemplos dados em *Specifying Systems*. Porém, a falta de formalização de algoritmos (inclusive concorrentes) no curso é notável. Ao longo de nossa formação provamos propriedades sobre números, autômatos e grafos, enquanto algoritmos são intocados. O benefício de provar propriedades sobre algoritmos, por exemplo, através de invariantes, pré-condições e pós-condições, não é somente maior "rigor acadêmico" e sim rigor prático que facilita projetar, escrever e depurar programas.

Pelo interesse de explorarmos caminhos em direção à corretude e formalização de algoritmos, faremos uma análise breve de sugestões de referências bibliográficas para o currículo atual de nosso curso. O objetivo dessa análise é encontrar referências que fortaleçam tanto o rigor formal dos alunos quanto intuições úteis de desenvolvimento de programas em contextos acadêmicos e industriais. Nossa avaliação se resume a encontrar livros para disciplinas obrigatórias – exceto Sistemas Distribuídos – que satisfaçam determinados requisitos.

Nossos requisitos para referências são:

- o ensino de múltiplas técnicas de provas (prova direta, contraexemplo, contradição, contrapositivo, indução, etc.);
- a preferência pelo ensino do método de provar em vez de focar em provas específicas, evitando que alunos decorem tais provas;
- a preferência pelo uso de definições matemáticas em vez de definições em linguagem natural;
- o ensino de lógica temporal em suas diversas apresentações (LTL, CTL). Essa exposição inclui a prática de definir e verificar automaticamente tais fórmulas lógicas (verificação de modelos, provadores automáticos, SAT *solvers*, etc.);
- o ensino das práticas de verificação e formalização de algoritmos (pré-condições, pós-condições, invariantes, etc.);
- especificamente sobre computação concorrente e distribuída, a modelagem de sistemas, definição formal de propriedades em alguma lógica temporal, e verificação automática de que tal sistema satisfaz tais propriedades;

As disciplinas Introdução ao Pensamento Dedutivo e Matemática Discreta não existiam no currículo anterior. Suas bibliografias contêm livros que satisfazem alguns de nossos requisitos. Estes livros são:

- "A Transition to Advanced Mathematics" (SMITH; EGGEN; ANDRE, 2014), que contém um capítulo dedicado a diversas técnicas de prova, e um capítulo que se aprofunda em provas por indução. Porém, ambos capítulos só contêm provas sobre números naturais, números inteiros e conjuntos;
- "Mathematics for Computer Science" (LEHMAN; LEIGHTON; MEYER, 2017), que contém um capítulo sobre provas por indução, e outro sobre máquinas de estado – inclusive invariantes, corretude parcial e término de programas.

Em quesito de lógicas temporais e verificação de programas, "Logic in Computer Science" (HUTH; RYAN, 2004) e "Mathematical Logic for Computer Science" (BEN-ARI, 2001) parecem excelentes referências: ambos vão além da lógica de primeira ordem quando abordam lógicas temporais e verificação de programas sequenciais e concorrentes.

Em quesito de propriedades de algoritmos sequenciais, o clássico "Algoritmos - Teoria e Prática" (*"Introduction to Algorithms"*) (CORMEN, 2012) parece bom o bastante por conter centenas de provas de teoremas sobre corretude de determinados algoritmos, inclusive sobre invariantes. O autor menciona invariantes de laços (*loop invariants*) explicitamente, mas alguns teoremas são implicitamente sobre invariantes de outros tipos.

Nesse contexto, seria interessante focar em definições matemáticas de estruturas de dados e propriedades formais, e, em sequência, ilustrar como programas implementam tais estruturas ou satisfazem tais propriedades.

Em quesito de formalizar algoritmos concorrentes, consideramos o foco somente em programas como um foco insuficiente, mesmo que o livro contenha algumas provas informais. Claro que ter implementações e testes executáveis de programas é essencial, pois precisamos verificar que o produto final – o programa – satisfaz sua especificação. Porém (i) esse foco único em programas nos distrai de problemas mais fundamentais em concorrência, que são os problemas lógicos; (ii) testes de programas concorrentes, que também são essenciais na exposição do assunto, não verificam diversas propriedades interessantes, tal como esperamos ter exposto ao longo deste trabalho.

Assim, sugerimos um livro com viés mais formal: "Principles of Concurrent and Distributed Programming" (BEN-ARI, 2006). Este livro que além de conter quase tantas provas informais de algoritmos quanto "The Art of Multiprocessor Programming" (HERLIHY; SHAVIT, 2012), também contém implementações de algoritmos concorrentes em Java, mas dedica três capítulos a algoritmos distribuídos e um capítulo à verificação de modelos de algoritmos concorrentes com lógica temporal linear.

Focando somente em sistemas distribuídos com abordagem mais formal, destacamos "Distributed Computing: Fundamentals, Simulations and Advanced Topics" (ATTIYA; WELCH, 2004), "Design and Analysis of Distributed Algorithms" (SANTORO, 2006), e o clássico "Distributed Algorithms" (LYNCH, 1996). Infelizmente esses três livros não contêm nem a verificação de modelos de seus algoritmos. Um livro que ensine sistemas distribuídos com abordagem totalmente formal talvez não exista. Supondo que os alunos tenham sido expostos à lógica temporal e a verificação de modelos em alguma disciplina obrigatória, essa ausência não deveria ser um problema aqui. Essa ausência pode ser uma oportunidade de alunos exercitarem a verificação formal.

6 CONCLUSÃO

Este trabalho foi escrito sob a perspectiva de um bacharelando em Ciência da Computação que descobriu métodos formais por acidente na forma da linguagem de especificação TLA$^+$, sua lógica subjacente TLA e seu verificador de modelos TLC. Alguns anos se passaram antes de percebermos a dificuldade de depurar programas distribuídos e sentirmos motivação para nos aprofundarmos na teoria de TLA$^+$.

Nosso objetivo foi demonstrar que métodos formais (i) facilitam o projeto de sistemas concorrentes ao nos permitir abstraí-los em diferentes níveis de complexidade e granularidade e (ii) garantem que os projetos de tais sistemas satisfaçam propriedades que linguagens e técnicas de teste raramente conseguem garantir.

Buscamos argumentar sobre os pontos (i) e (ii) através de uma escrita com tom didático, exemplos (Cap. 4) e discussão (Cap. 5) da nossa experiência, visando leitores sem familiaridade com métodos formais, que é um tema relativamente complexo. Esperamos ter demonstrado que TLA$^+$ nos permite descrever sistemas em alto nível de abstração, e que essa abstração facilita a compreensão e desenvolvimento de sistemas concorrentes.

No entanto, precisamos apontar nossas próprias limitações. O ponto (i) requer uma referência antes de afirmarmos que tal técnica *facilita* o projeto de um sistema. Tal referência não existe aqui. Espera-se que leitores tenham alguma experiência com o projeto de sistemas concorrentes para perceberem que uma especificação, comparada a suas possíveis implementações, facilita o projeto desses sistemas. O ponto (ii) afirma sobre propriedades que linguagens e técnicas de teste raramente conseguem garantir. Dizemos "raramente" porque é impossível estudarmos todas as linguagens e técnicas existentes. Espera-se que leitores notem, por exemplo, que afirmar sobre o estado global de um sistema, ou sobre estados futuros através de vivacidade é essencialmente impossível ao nível de implementação: é necessário um modelo, e, portanto, métodos formais.

Métodos formais requerem modelos matemáticos da realidade, e esses modelos ruins podem remover detalhes importantes do domínio do problema. Além disso, métodos formais só são formais até dependermos da linguagem natural – uma linguagem informal – para comunicarmos o que cada símbolo da especificação representa no mundo real (LAMSWEERDE, 2000, p. 151). Por sua vez, a verificação de modelos não verifica sistemas reais, somente modelos; está sujeita à qualidade desses modelos; está sujeita ao tamanho do espaço de estados gerado por tais modelos, podendo ultrapassar a memória disponível do sistema; é limitada no tipo de problema que pode ser verificado: sistemas com infinitos estados precisam de restrições possivelmente artificiais; não garante corretude, porque está sujeita a defeitos de *software*; não garante completude, porque não se pode julgar sobre propriedades não verificadas; não é uma prova formal, então não permite verificar generalizações sobre dados parâmetros (BAIER; KATOEN, 2008, p. 14,

15).

Mesmo com todos os desafios discutidas durante o trabalho, reforçamos (i) a importância de especificar algoritmos por nos fazer pensar sobre o problema em mãos, inclusive gerando especificações informais mais efetivas, (ii) a importância de provar algoritmos por ser o parâmetro mais alto de confiança, e (iii) que algum formalismo adequado, incluindo TLA^+, deve ser usado para projetar sistemas concorrentes corretos, principalmente porque certas propriedades são impossíveis de serem definidas e verificadas sem métodos formais. Por sua vez, a verificação de modelos é o mínimo que se deve fazer para confiarmos que determinado algoritmo concorrente esteja correto, e a prova formal seria o "padrão-ouro" nesse sentido ao confirmarmos que o algoritmo concorrente satisfaz determinadas propriedades utilizando quaisquer parâmetros de entrada.

Isso não significa que métodos formais são uma panaceia – considerando as desvantagens discutidas –, que métodos formais devem ser usados em todo tipo de projeto – deve-se comparar o custo de verificar com o custo de errar –, ou que testes de implementação são desnecessários – a implementação do sistema ainda requer o uso de linguagens adequadas e ferramentas de análise estática, testagem e simulações determinísticas e não determinísticas.

Sobre trabalhos futuros, mencionamos no início do capítulo de discussão (Cap. 5) que tentamos especificar o algoritmo valentão para eleição de líder descrito informalmente no livro *"Distributed Systems"* (TANENBAUM; STEEN, 2017). Sabemos que a descrição do livro foge da descrição do artigo que originou o algoritmo valentão (e que foi citado no livro), isto é, (GARCIA-MOLINA, 1982). Suspeitamos que a descrição de Garcia-Molina esteja correta, enquanto a descrição de Tanenbaum e Steen, não: mais de um coordenador poderia ser eleito, o que é um resultado indesejado.

Assim, temos dois algoritmos com descrições informais – o livro usa texto e diagramas; o artigo usa texto, pseudocódigo e provas informais – que poderiam ser formalizados e verificados. Entre os dois, destacamos o algoritmo de Tanenbaum e Steen, porque é um algoritmo simples, então tem sua utilidade pedagógica, e porque é popular, já que uma versão similar aparece em (COULOURIS et al., 2011), nos sugerindo que este algoritmo seja uma espécie de "folclore" em Sistemas Distribuídos. Em resumo, sugerimos (i) especificar o algoritmo do livro formalmente; (ii) demonstrar um contraexemplo onde mais de um coordenador é eleito; e (iii) tentar corrigir este algoritmo, verificando formalmente que ele satisfaz as propriedades do livro, ou até as propriedades em (GARCIA-MOLINA, 1982), e sob quais suposições de ambiente.

REFERÊNCIAS

ALPERN, B.; SCHNEIDER, F. B. Defining liveness. **Information Processing Letters**, v. 21, n. 4, p. 181–185, 1985. ISSN 0020-0190. Disponível em: https://www.sciencedirect.com/science/article/pii/0020019085900560.

ARMANDO, A.; MANTOVANI, J.; PLATANIA, L. Bounded model checking of software using smt solvers instead of sat solvers. In: VALMARI, A. (Ed.). **Model Checking Software**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 146–162. ISBN 978-3-540-33103-2.

ARMSTRONG, J. **Making reliable distributed systems in the presence of software errors**. Tese (Doutorado) — Swedish Institute of Computer Science, 2003. Disponível em: https://urn.kb.se/resolve?urn=urn:nbn:se:ri:diva-22455.

ATTIYA, H.; WELCH, J. **Distributed Computing: Fundamentals, Simulations, and Advanced Topics**. Wiley, 2004. (Wiley Series on Parallel and Distributed Computing). ISBN 9780471453246. Disponível em: https://books.google.com.br/books?id=3xfhhRjLUJEC.

BAIER, C.; KATOEN, J.-P. **Principles of model checking**. Cambridge, Mass.: MIT Press, 2008. ISBN 9780262267564 026226756X 9781435643277 1435643275. Disponível em: http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=226091.

BEN-ARI, M. **Mathematical Logic for Computer Science**. Springer, 2001. (Prentice-Hall international series in computer science). ISBN 9781852333195. Disponível em: https://books.google.com.br/books?id=5wj91wfmQ_gC.

BEN-ARI, M. **Principles of Concurrent and Distributed Programming**. Addison-Wesley, 2006. (Prentice Hall international series in computer science). ISBN 9780321312839. Disponível em: https://books.google.com.br/books?id=oP-2hpMEdb8C.

BIERE, A. et al. Symbolic model checking without bdds. In: CLEAVELAND, W. R. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. p. 193–207. ISBN 978-3-540-49059-3.

BURCH, J. et al. Symbolic model checking: 1020 states and beyond. **Information and Computation**, v. 98, n. 2, p. 142–170, 1992. ISSN 0890-5401. Disponível em: https://www.sciencedirect.com/science/article/pii/089054019290017A.

CHAUDHURI, K. et al. Verifying safety properties with the tla+ proof system. In: **Automated Reasoning**. Springer Berlin Heidelberg, 2010. p. 142–148. Disponível em: https://doi.org/10.1007%2F978-3-642-14203-1_12.

CLARKE, E. M.; EMERSON, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In: KOZEN, D. (Ed.). **Logics of Programs**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982. p. 52–71. ISBN 978-3-540-39047-3.

CLEBSH, S. **Pony programming language**. 2015. Acesso em 20/07/2023. Disponível em: https://www.ponylang.io/.

CORMEN, T. **Algoritmos - Teoria e Prática**. GEN LTC, 2012. ISBN 9788535236996. Disponível em: https://books.google.com.br/books?id=6iA4LgEACAAJ.

COULOURIS, G. et al. **Distributed Systems: Concepts and Design**. Pearson Education, 2011. ISBN 9780133001372. Disponível em: https://books.google.com.br/books?id=3ZouAAAAQBAJ.

DAVIS, A. J. J.; HIRSCHHORN, M.; SCHVIMER, J. Extreme modelling in practice. **Proc. VLDB Endow.**, VLDB Endowment, v. 13, n. 9, p. 1346–1358, may 2020. ISSN 2150-8097. Disponível em: https://doi.org/10.14778/3397230.3397233.

DAY, N.; BANDALI, A. A Comprehensive Study of Declarative Behavioural Modelling Languages. 1 2022. Disponível em: https://www.techrxiv.org/articles/preprint/A_Comprehensive_Study_of_Declarative_Behavioural_Modelling_Languages/18551255.

GARCIA-MOLINA, H. Elections in a distributed computing system. **IEEE Transactions on Computers**, C-31, p. 48–59, 1982.

GOOGLE. **Go programming language**. 2009. Acesso em 20/07/2023. Disponível em: https://go.dev/.

GOOGLE. **ThreadSanitizer**. 2015. Acesso em 20/07/2023. Disponível em: https://github.com/google/sanitizers.

HACKETT, F. et al. Compiling distributed system models with pgo. In: **Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2**. New York, NY, USA: Association for Computing Machinery, 2023. (ASPLOS 2023), p. 159–175. ISBN 9781450399166. Disponível em: https://doi.org/10.1145/3575693.3575695.

HALL, A. Seven myths of formal methods. **IEEE Software**, v. 7, n. 5, p. 11–19, 1990.

HERLIHY, M.; SHAVIT, N. **The Art of Multiprocessor Programming, Revised Reprint**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN 9780123973375.

HICKEY, R. **Clojure programming language**. 2007. Acesso em 20/07/2023. Disponível em: https://clojure.org/.

HUTH, M.; RYAN, M. **Logic in Computer Science: Modelling and Reasoning about Systems (2nd edn.)**. [S.l.]: Cambridge University Press, 2004. ISBN 9780521543101.

KONNOV, I.; KUKOVEC, J.; TRAN, T.-H. Tla+ model checking made symbolic. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 3, n. OOPSLA, oct 2019. Disponível em: https://doi.org/10.1145/3360549.

LAMPORT, L. **The Temporal Logic of Actions**. Acm transactions on programming languages and systems 16. [S.l.], 1994. ACM Transactions on Programming Languages and Systems 16. Disponível em: https://www.microsoft.com/en-us/research/publication/the-temporal-logic-of-actions/.

LAMPORT, L. **Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers**. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 032114306X.

LAMPORT, L. The pluscal algorithm language. **Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36-60.**, January 2009. Disponível em: https://www.microsoft.com/en-us/research/publication/pluscal-algorithm-language/.

LAMPORT, L. **Industrial Use of TLA+**. 2022. Acesso em 08/06/2022. Disponível em: https://lamport.azurewebsites.net/tla/industrial-use.html.

LAMPORT, L. Science of concurrent programs. Acesso em 05/03/2024. 2024.

LAMPORT, L. **TLA+ Tools**. 2024. Acesso em 07/02/2024. Disponível em: https://lamport.azurewebsites.net/tla/tools.html.

LAMSWEERDE, A. v. Formal specification: A roadmap. In: **Proceedings of the Conference on The Future of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2000. (ICSE '00), p. 147–159. ISBN 1581132530. Disponível em: https://doi.org/10.1145/336512.336546.

LEHMAN, E.; LEIGHTON, F.; MEYER, A. **Mathematics for Computer Science**. 12th Media Services, 2017. ISBN 9781680921229. Disponível em: https://books.google.com.br/books?id=sWNdtAEACAAJ.

LYGIN, A. **TLA+ for Visual Studio Code**. 2024. Acesso em 04/03/2024. Disponível em: https://marketplace.visualstudio.com/items?itemName=alygin.vscode-tlaplus.

LYNCH, N. **Distributed Algorithms**. Morgan Kaufmann, 1996. (The Morgan Kaufmann Data Manag). ISBN 9781558603486. Disponível em: https://books.google.com.br/books?id=7C7oIV48RQQC.

MATSAKIS, N. D.; KLOCK, F. S. The rust language. **Ada Lett.**, Association for Computing Machinery, New York, NY, USA, v. 34, n. 3, p. 103–104, oct 2014. ISSN 1094-3641. Disponível em: https://doi.org/10.1145/2692956.2663188.

MAZZANTI, F.; FERRARI, A. Ten diverse formal models for a CBTC automatic train supervision system. **Electronic Proceedings in Theoretical Computer Science**, Open Publishing Association, v. 268, p. 104–149, mar 2018. Disponível em: https://doi.org/10.4204/eptcs.268.4.

MICROSOFT. **Visual Studio Code**. 2024. Acesso em 04/03/2024. Disponível em: https://code.visualstudio.com/.

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

PNUELI, A. The temporal logic of programs. In: **18th Annual Symposium on Foundations of Computer Science (sfcs 1977)**. [S.l.: s.n.], 1977. p. 46–57.

SANTORO, N. **Design and Analysis of Distributed Algorithms**. Wiley, 2006. (Wiley Series on Parallel and Distributed Computing). ISBN 9780470072639. Disponível em: https://books.google.com.br/books?id=iXlBscntcUgC.

SHAVIT, N.; TOUITOU, D. Software transactional memory. In: **Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing**. [S.l.: s.n.], 1995. p. 204–213.

SMITH, D.; EGGEN, M.; ANDRE, R. **A Transition to Advanced Mathematics**. Cengage Learning, 2014. ISBN 9781285463261. Disponível em: https://books.google.com.br/books?id=DOUbCgAAQBAJ.

SOMMERVILLE, I. **Software Engineering**. Tenth. Pearson, 2016. ISBN 978-1-292-09613-1. Disponível em: https://www.pearson.com/us/higher-education/program/PGM35255.html.

TANENBAUM, A.; STEEN, M. van. **Distributed Systems**. CreateSpace Independent Publishing Platform, 2017. ISBN 9781543057386. Disponível em: https://books.google.com.br/books?id=c77GAQAACAAJ.

TLA+ Foundation. **TLA+ Toolbox**. 2021. Acesso em 04/03/2024. Disponível em: https://lamport.azurewebsites.net/tla/toolbox.html.

TLA+ Foundation. **Repositório GitHub de TLC e TLA+ Toolbox**. 2023. Acesso em 04/03/2024. Disponível em: https://www.github.com/tlaplus/tlaplus.

VALGRIND™. **DRD: a thread error detector**. 2007. Acesso em 20/07/2023. Disponível em: https://valgrind.org/docs/manual/drd-manual.html.

VALGRIND™. **Helgrind: a thread error detector**. 2007. Acesso em 20/07/2023. Disponível em: https://valgrind.org/docs/manual/hg-manual.html.

VALIM, J. **Elixir programming language**. 2012. Acesso em 20/07/2023. Disponível em: https://elixir-lang.org.

YU, Y.; MANOLIOS, P.; LAMPORT, L. Model checking tla+ specifications. In: **In Correct Hardware Design and Verification Methods (CHARME '99), Laurence Pierre and Thomas Kropf editors. Lecture Notes in Computer Science, Springer-Verlag.** [s.n.], 1999. v. 1703, p. 54–66. Disponível em: https://www.microsoft.com/en-us/research/publication/model-checking-tla-specifications/.

GLOSSÁRIO

time-to-market Intervalo de tempo entre ideação e lançamento de determinado produto no mercado.

produção Código executado no mundo-real, executado em servidores, executado por clientes, etc. Em oposição a protótipo, ou código executado em privado.

APÊNDICE A – MÓDULO LOCK

Código 36 – Módulo Lock

```
---------------------- MODULE Lock ----------------------
CONSTANTS Agentes, Ninguem

ASSUME Agentes # {}
ASSUME Ninguem \notin Agentes

VARIABLES dono

Trancar(agente)    == dono = Ninguem /\ dono' = agente
Destrancar(agente) == dono = agente  /\ dono' = Ninguem

Ini  == dono = Ninguem
Prox == \E ag \in Agentes: Trancar(ag) \/ Destrancar(ag)
Spec == Ini /\ [][Prox]_dono

Justica   == \A ag \in Agentes: /\ SF_dono(Trancar(ag))
                                /\ WF_dono(Destrancar(ag))

SpecJusta == Spec /\ Justica

InvarianteDeTipos  == dono \in Agentes \cup {Ninguem}
AusenciaDeInanicao == \A ag \in Agentes: []<>(dono = ag)
=============================================================
```

Código 37 – Arquivo de configuração Lock.cfg

```
SPECIFICATION SpecJusta
CONSTANTS Agentes = {a1, a2, a3}  \* Conjunto de valores-modelo
          Ninguem = Ninguem       \* Valor-modelo

INVARIANT InvarianteDeTipos
PROPERTY AusenciaDeInanicao
```

APÊNDICE B – MÓDULO CFC_LOCK E ARQUIVO DE CONFIGURAÇÃO CFC_LOCK.CFG

Código 38 – Módulo CFC_Lock

```
------------ MODULE CFC_Lock ------------
EXTENDS Naturals

CONSTANTS Agentes, ValorMaximo, Ninguem
ASSUME ValorMaximo \in Nat
ASSUME Ninguem \notin Agentes

VARIABLES c, pc, dono
vars == <<c, pc, dono>>

L == INSTANCE Lock

Ini == /\ c = 0
       /\ pc = [ ag \in Agentes |-> "1" ]
       /\ dono = Ninguem

\* 1: Trancar(); goto 2
I1(ag) == /\ pc[ag] = "1"
          /\ L!Trancar(ag)
          /\ pc' = [ pc EXCEPT ![ag] = "2" ]
          /\ UNCHANGED c

\* 2: if (c >= ValorMaximo) goto 3
\*    else goto 4
I2(ag) == /\ pc[ag] = "2"
          /\ IF c >= ValorMaximo
             THEN pc' = [ pc EXCEPT ![ag] = "3" ]
             ELSE pc' = [ pc EXCEPT ![ag] = "4" ]
          /\ UNCHANGED <<c, dono>>

\* 3: Destrancar(); goto 6
I3(ag) == /\ pc[ag] = "3"
          /\ L!Destrancar(ag)
          /\ pc' = [ pc EXCEPT ![ag] = "6" ]
          /\ UNCHANGED c

\* 4: c := c + 1; goto 5
I4(ag) == /\ pc[ag] = "4"
          /\ c' = c + 1
```

```
           /\ pc' = [ pc EXCEPT ![ag] = "5" ]
           /\ UNCHANGED dono

\* 5: Destrancar(); goto 1
I5(ag) == /\ pc[ag] = "5"
           /\ L!Destrancar(ag)
           /\ pc' = [ pc EXCEPT ![ag] = "1" ]
           /\ UNCHANGED c

\* 6: fim
Fim == /\ \A ag \in Agentes: pc[ag] = "6"
       /\ UNCHANGED vars

Prox == \/ \E ag \in Agentes: \/ I1(ag) \/ I2(ag) \/ I3(ag)
                              \/ I4(ag) \/ I5(ag)
        \/ Fim

Justica == \A ag \in Agentes: /\ WF_vars(I1(ag)) /\ WF_vars(I2(ag))
                              /\ WF_vars(I3(ag)) /\ WF_vars(I4(ag))
                              /\ WF_vars(I5(ag))

SpecJusta == Ini /\ [][Prox]_vars /\ Justica
-----------------------------------------
InvarianteDeTipos ==
  /\ c \in 0..ValorMaximo
  /\ pc \in [ Agentes ->  {"1", "2", "3", "4", "5", "6"} ]
  /\ dono \in Agentes \cup { Ninguem }

Termina == <>[](
  /\ c = ValorMaximo
  /\ dono = Ninguem
  /\ \A ag \in Agentes: pc[ag] = "6"
)
ContadorMuda == \A n \in 0..ValorMaximo: <>(c = n)

EmSecaoCritica(ag) == pc[ag] \in {"2", "3", "4", "5"}
ExclusaoMutua ==
  \A a1, a2 \in Agentes:
    (a1 # a2) => ~(EmSecaoCritica(a1) /\ EmSecaoCritica(a2))

AusenciaDeInanicao ==
  \A ag \in Agentes: (pc[ag] = "1") ~> (pc[ag] = "2")

CF == INSTANCE ContadorSomaAtomicaFinito
ImplementaCF == CF!Spec
=========================================
```

Código 39 – Arquivo de configuração `CFC_Lock.cfg`

```
SPECIFICATION SpecJusta
CONSTANT ValorMaximo = 6
         Agentes = {a1, a2, a3, a4, a5, a6}
         Nat = {0, 1, 2, 3, 4, 5, 6}
         Ninguem = Ninguem
INVARIANT
    InvarianteDeTipos
    ExclusaoMutua
PROPERTIES
    Termina
    ContadorMuda
    AusenciaDeInanicao
    ImplementaContadorFinito
```

ANEXO A – SÍMBOLOS ASCII DE TLA+

LÓGICA

Notação original	Notação em ASCII
$\wedge$	/\ ou \land
$\vee$	\/ ou \lor
$\neg$	~ ou \neg ou \lnot
$\rightarrow$	->

Notação original	Notação em ASCII
$\equiv$	<=> ou \equiv
$\forall$	\A
$\exists$	\E
$\Rightarrow$	=>

CONJUNTOS

Notação original	Notação em ASCII
$=$	=
$\neq$	\= ou #
$\in$	\in
$\notin$	\notin

Notação original	Notação em ASCII
$\cup$	\cup ou \union
$\cap$	\cap ou \intersect
$\subseteq$	\subseteq
$\setminus$	\

FUNÇÕES

Notação original	Notação em ASCII
$\mapsto$	|->

TUPLAS

Notação original	Notação em ASCII
$\langle 1, 2, 3 \rangle$	<<1, 2, 3>>
$\times$	\X
$\circ$	\o ou \circ

DIVERSOS

Notação original	Notação em ASCII
$\triangleq$	==
"s"	"s"

OPERADORES DE AÇÃO

Notação original	Notação em ASCII
e'	`e'`
$[A]_e$	`[A]_e`
$\langle A \rangle_e$	`<<A>>_e`

OPERADORES TEMPORAIS

Notação original	Notação em ASCII
$\Box F$	`[]F`
$\Diamond F$	`<>F`
$\mathrm{WF}_e(A)$	`WF_e(A)`
$\mathrm{SF}_e(A)$	`SF_e(A)`
$F \rightsquigarrow G$	`F ~> G`

MATEMÁTICA

Notação original	Notação em ASCII
$+$	`+`
$-$	`-`
$*$	`*`
$\div$	`\div`
$\%$	`%`

Notação original	Notação em ASCII
x^y	`x^y`
$<$	`<`
$\leq$	`<=` ou `=<` ou `\leq`
$>$	`>`
$\geq$	`>=` ou `\geq`

ANEXO B – NOTAÇÃO DE TLA

SINTAXE SIMPLIFICADA

$$\begin{aligned} \textit{fórmula} &\triangleq \textit{predicado} \mid \Box\,[\textit{ação}]_{\textit{função de estado}} \mid \neg \textit{fórmula} \\ &\mid \textit{fórmula} \wedge \textit{fórmula} \mid \Box \textit{fórmula} \\ \textit{ação} &\triangleq \text{predicado contendo constantes, variáveis e variáveis-linha} \\ \textit{predicado} &\triangleq \textit{ação} \text{ sem variáveis-linha} \mid \textit{Enabled ação} \\ \textit{função de estado} &\triangleq \text{expressão não-booleana que contém variáveis e símbolos constantes} \end{aligned}$$

NOTAÇÃO ADICIONAL

$$\begin{aligned} p' &\triangleq p\ (\forall\, v : v'/v) & \Diamond F &\triangleq \neg\Box\neg F \\ [\mathcal{A}]_f &\triangleq \mathcal{A} \vee (f' = f) & F \rightsquigarrow G &\triangleq \Box(F \Rightarrow \Diamond G) \\ \langle \mathcal{A} \rangle_f &\triangleq \mathcal{A} \wedge (f' \neq f) & \mathrm{WF}_f(\mathcal{A}) &\triangleq \Box\Diamond\langle \mathcal{A} \rangle_f \vee \Box\Diamond\neg \textit{Enabled}\ \langle \mathcal{A} \rangle_f \\ \textit{Unchanged}\ f &\triangleq f' = f & \mathrm{SF}_f(\mathcal{A}) &\triangleq \Box\Diamond\langle \mathcal{A} \rangle_f \vee \Diamond\Box\neg \textit{Enabled}\ \langle \mathcal{A} \rangle_f \end{aligned}$$

onde f é uma *função de estado*
$\mathcal{A}$ é uma *ação*
F e G são *fórmulas*
p é uma *função de estado* ou *predicado*

QUANTIFICAÇÃO EM TLA

$$\begin{aligned} \textit{fórmula geral} &\triangleq \textit{fórmula} \mid \exists\!\!\exists\ \textit{variável} : \textit{fórmula geral} \\ &\mid \exists\ \textit{variável rígida} : \textit{fórmula geral} \\ &\mid \textit{fórmula geral} \wedge \textit{fórmula geral} \\ &\mid \neg \textit{fórmula geral} \end{aligned}$$