UNIVERSIDADE FEDERAL FLUMINENSE

DANIEL ARENA TOLEDO

Formalização de conectores Reo híbridos com aplicações a Consenso Bizantino

NITERÓI

2021

Daniel Arena Toledo

Formalização de conectores Reo híbridos com aplicações a Consenso Bizantino

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação

Orientador:
Prof. Bruno Lopes

Co-orientador:
Prof. Igor Machado Coelho

Niterói
2021

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

T649f Toledo, Daniel Arena
Formalização de conectores Reo híbridos com aplicações a Consenso Bizantino / Daniel Arena Toledo ; Bruno Lopes, orientador ; Igor Machado Coelho, coorientador. Niterói, 2021.
71 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2021.

DOI: http://dx.doi.org/10.22409/PGC.2021.m.17291265703

1. Lógica. 2. Métodos formais. 3. Reo. 4. Consenso Bizantino. 5. Produção intelectual. I. Lopes, Bruno, orientador. II. Coelho, Igor Machado, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

Bibliotecário responsável: Debora do Nascimento - CRB7/6368

Daniel Arena Toledo

Formalização de conectores Reo híbridos com aplicações a Consenso Bizantino

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Ciência da Computação

Aprovada em agosto de 2021.

BANCA EXAMINADORA

Prof. Bruno Lopes - Orientador, UFF

Prof. Igor Machado Coelho - Co-orientador, UFF

Vitor Nazário Coelho - Seiva Research

Prof. Mario Benevides - UFF

Prof. Daniel Cardoso Moraes de Oliverira - UFF

Prof. Edward Hermann Haeusler - PUC-Rio

Niterói

2021

Agradecimentos

Primeiramente, gostaria de agradecer a minha família pelo suporte incondicional a minha longa caminhada até aqui.

Ao meu orientador Bruno Lopes, que no tempo que trabalhamos neste projeto se tornou um amigo, pelas dicas e conselhos que me ofereceu.

Pelo co-orientador Igor, que apesar de ter chegado meio em cima da hora neste projeto, ofereceu ajuda inestimável em entender o funcionamento do consenso.

Ao meu grande amigo Erick, que não só me ajudou nesta jornada, como também disponibilizou algumas das imagens usadas neste projeto.

À banca examinadora, que ofereceu seu tempo e apresentou melhorias indispensáveis a este trabalho.

Aos meus amigos, que sem eles eu não teria chegado aqui.

A todos que me ajudaram neste projeto.

Resumo

Reo é uma linguagem gráfica de modelagem baseada em coordenações que busca capturar e modelar a interação entre diferentes componentes de um sistema usando estruturas conhecidas como canais. Reo tem sido usado na modelagem de várias situações do mundo real e várias pesquisas foram feitas buscando formalizar e verificar propriedades desses circuitos usando *Constraint Automata*. Ao adicionar comportamento dinâmico a um canal Reo, é possível a modelagem de interações contínuas como em sistemas ciber-físicos e comunicações de rede dependente de tempo não determinísticas, mas para que seja possível analisar esses comportamentos dinâmicos, é necessário um formalismo mais expressivo. *Hybrid Constraint Automata* (HCA) é uma semântica formal para Reo onde comportamentos dinâmicos e discretos coexistem e interagem entre si. Neste trabalho, é apresentado três novos canais Reo que modelam comportamento dinâmico e também suas formalizações em HCA, também é apresentada uma tradução automática deles para o verificador de modelos nuXmv, que conta com operação de composição e uma ferramenta que automatiza o processo. Essa estrutura permite a modelagem do algoritmo de consenso bizantino estado-da-arte *Delegated Byzantine Fault Tolerance* (dBFT), assim como analisar sua estrutura interna do quórum incerto. É apresentado uma formalização do dBFT usando esse novos canais para modelar suas interações dependentes de tempo.

Abstract

Reo is a graphics-based coordination modeling language that aims to capture and model the interaction between different components of systems using structures known as channels. Reo has been used to model various real-world situations and many researches focus on formalizing and verifying properties of its circuits relying on Constraint Automata. By adding dynamic behavior to Reo channels, it is possible to model continuous interactions as in cyber-physical systems and nondeterministic time-reliant network communications, but in order to reason about those dynamic behaviors, it is necessary a more expressive formalism. Hybrid Constraint Automata (HCA) provides a formal semantic for Reo where discrete and continuous dynamics co-exist and interact with each other. In this work, we make use of three Reo channels that model dynamic behavior and their formalizations as HCAs, and present automatic translations of them to the model checker nuXmv, with a composition operation and a tool to automate the process. This framework enables modeling of a state-of-the-art byzantine consensus algorithm Delegated Byzantine Fault Tolerance (dBFT), as well as reasoning on its internal structures of uncertain quorums. We present a formalization of dBFT and we use those new channels to model its time-dependent interactions.

Lista de Figuras

2.1 Modelo de um sequenciador em Reo 10

2.2 Representação gráfica de um CA FIFO de dois dados sem restrições 11

2.3 Um fragmento do conector sequenciador 16

2.4 O CA do fragmento do conector do sequenciador 17

2.5 Autômato híbrido para um sistema de controle de trem simplificado 18

2.6 Exemplo para ilustrar um HCA (timer) 21

2.7 Exemplo do produto de HCA entre o lossySync e o timer 22

4.1 Canal Reo para o timer e seu HCA 29

4.2 Exemplo do HCA do timer com parâmetros para um contador de 30 segundos 29

4.3 Canal Reo para o timedDelay e seu HCA 30

4.4 HCA do timedDelay com tempo mínimo de 10 segundo e máximo de 50 segundos 30

4.5 Canal Reo para o timedTransform e seu HCA 31

4.6 HCA do timedTransform para simular um termômetro 31

5.1 Exemplo de um sistema de replicação de máquinas de estado onde uma réplica é bizantina 47

5.2 Exemplo do caso de operação comum para o PBFT com uma réplica falha 49

5.3 Uma *blockchain* onde os blocos em vermelho foram gerados de *forks* ignorados 49

6.1 Modelo Reo para uma réplica de consenso do dBFT 2.0 53

6.2 Quorum incerto de $2f + 1$ para o dBFT 2.0 (com $f = 1$), onde a réplica C é falha e o estado Ac ainda é alcançável por A, B e D. 53

Lista de Tabelas

2.1 Tabela contendo os conectores Reo dados por [3] e seus comportamentos . 9

2.2 Tabela contendo os conectores Reo canônicos e seus respectivos CA [22]. . 15

Conteúdo

1 Introdução **1**

2 Definições básicas **3**

2.1 Lógica Clássica Proposicional 3

2.2 Lógica Modal 4

2.3 *Computation Tree Logic* 5

2.4 Reo 8

2.5 Constraint Automata 10

2.6 Sistemas Dinâmicos 17

2.7 Hybrid Constraint Automata 19

2.8 nuXmv 21

3 Trabalhos Relacionados **26**

4 Compilador **28**

4.1 Novos Canais 28

4.2 HCA no nuXmv 31

4.3 *Product Automata* 36

4.4 *Timed Data Streams* 43

5 Consenso **45**

6 Modelagem do mecanismo de consenso **52**

7 Conclusões e trabalhos futuros **56**

7.1 Trabalhos Futuros 57

Referências **58**

Chapter 1

Introdução

Sistemas ciber-físicos (CPSs) são sistemas que integram computação com controle de entidades físicas, como por exemplo automóveis autônomos, monitoramento médico ou outros, o que leva a modelos com comportamentos dinâmicos. O advento desses CPSs gera interesse em modelá-los para que possam ser analisados formalmente.

Nesses sistemas, uma importante funcionalidade é garantir a integridade e o consenso entre seus diferentes participantes. De uma perspectiva de aplicação, alcançar consenso por replicação de máquinas de estado tem sido muito discutidos nos últimos anos. Após o sucesso do *Practical Byzantine Fault Tolerance* (PBFT) [10] em 1999, por Miguel Castro e Barbara Liskov, diversos projetos tentaram melhorar o PBFT, mas poucos conseguiram avanços com um sólido formalismo matemático.

Reo [3] é uma linguagem gráfica baseada em coordenações e canais, usada na modelagem de sistemas e que toma vantagem de características de sistemas distribuídos como a chamada remota de funções e troca de mensagem assíncrona. Seu principal objetivo é prover um modelo de código que trata como as diferentes partes heterogêneas de um sistema interagem entre si. A modelagem composicional de sistemas é reforçada pela presença de conectores canônicos e uma operação de composição para gerar canais complexos a partir de canais mais simples.

Constraint Automata (CA) [8] é uma semântica formal para Reo proposta pelo seu criador. Mas CA tem suas falhas quando tratamos de interações dependentes de tempo, como é o caso dos sistemas dinâmicos. Um outro formalismo, o *Hybrid Constraint Automata* (HCA) [14], é uma formalização semântica para Reo onde dinâmicas discretas e contínuas coexistem e interagem entre si.

Sistemas formais compõem um background teórico que pode ser usado para raciocinar

acerca desses sistemas críticos, permitindo a garantia matemática de que os requerimentos são cumpridos e que o sistema se comporta como esperado.

A Airbus [9] usa métodos formais para certificar seus sistemas eletrônicos das famílias de aeronaves A318 e A340-500/60. Além disso, a Linha 1 do Metrô de Paris [20] também foi validada formalmente, levando a um sistema totalmente automatizado.

Portanto, essa necessidade de validar sistemas trouxe consigo a necessidade por ferramentas para auxiliar nessa tarefa, que é tipicamente árdua, complexa e às vezes combinatorial. O objetivo dessas ferramentas é justamente reduzir a complexidade desse processo de validação, além de agilizá-lo. Dentre as várias ferramentas disponíveis, estão os verificadores de modelos, que são amplamente utilizados.

Esses verificadores são ferramentas capazes de verificar propriedades acerca de um modelo, normalmente modeladas por meio de uma lógica. Como exemplo de propriedades que podem ser verificadas, temos a verificação da presença de *deadlocks* e a alcançabilidade ou não de algum estado modelado [7]. Como exemplo de verificadores de modelos temos o PRISM [31] e o nuXmv [11], que é o verificador usado neste projeto.

Nesse trabalho, é estendido nosso trabalho anterior [23], onde é proposta uma tradução Reo $\xrightarrow{SMV} \mathcal{R}'$ onde $\mathcal{R}'$ é um modelo no nuXmv e Reo é formalizado usando CA. No trabalho atual, a tradução que foi feita é migrada de CA para HCA, desta forma é apresentado um modelo formal em nuXmv para HCA, onde para o melhor de nosso conhecimento, nunca antes feito. Também é apresentado três novos canais para facilitar a adição de comportamento dinâmico ao Reo:

- *timer*: que funciona como um contador;
- *timedDelay*: que simula uma comunicação com atraso mínimo e máximo não determinístico;
- *timedTransform*: onde o dado recebido vai se transformando com o passar do tempo.

Este trabalho foi organizado da seguinte maneira. O Capítulo 2 apresenta uma explicação a respeito dos conceitos usados na ferramenta, o Capítulo 3 apresenta alguns trabalhos relacionados, o Capítulo 4 apresenta a ferramenta que foi implementada neste projeto, discutindo suas principais funções. O Capítulo 5 apresenta noções de consenso e o Capítulo 6 exemplifica a motivação desse trabalho, o porquê de estender nossa ferramenta para sistemas híbridos. Por último Capítulo 7 conclui este projeto, apontando para algumas direções futuras.

Chapter 2

Definições básicas

Este projeto usa uma abordagem lógica com Reo, portanto neste capítulo será apresentada a base teórica usada neste trabalho.

2.1 Lógica Clássica Proposicional

Lógica Proposicional é um modelo matemático que permite raciocinar sobre sentenças lógicas (proposições) que contém um valor de verdade. Essas proposições podem ser combinadas para produzir sentenças complexas a partir de sentenças mais simples [1]. Lógica Proposicional Clássica é um ramo da lógica que estuda métodos para raciocinar acerca das proposições e de como compor proposições de outras proposições.

Proposições nessa lógica são separadas em duas categorias: proposições atômicas, aquelas que não tem nenhum conectivo lógico, podem ser por exemplo "Marcos é professor"e "Luana é professora"; e proposições moleculares, que são compostas de outras proposições por meio de conectivos, por exemplo "Marcos e Luana são professores". Conectivos lógicos podem ser encontrados na língua falada como conjunções: "e", "ou"e "não".

Para raciocinar formalmente acerca dessas proposições, um sistema formal é definido onde as sentenças escritas são convertidas para sentenças proposicionais, e regras de inferência são estabelecidas, para que se possa discutir a respeito dessas proposições.

Na Lógica Clássica Proposicional, ambos os tipos de proposições devem denotar um valor-verdade: verdadeiro ou falso, a qualquer momento uma proposição deve ser ou verdadeira ou falsa, não podendo ter os dois valores simultaneamente.

A linguagem da Lógica Proposicional pode ser definida formalmente como

Definição 2.1.1 (Linguagem da Lógica Proposicional)**.** *A linguagem da Lógica Proposicional pode ser descrita por*

um conjunto Φ *enumerável de símbolos proposicionais;*

um conjunto de conectores dos quais são interpretados como símbolos de operadores, compondo sentenças moleculares de outras proposições atômicas ou moleculares, propriamente os conectivos $\wedge$ *(e),* $\vee$ *(ou),* $\rightarrow$ *(implica),* $\leftrightarrow$ *(bicondição) e* $\neg$ *(não).*

Em notação BNF, uma fórmula pode ser definida como

$$\varphi ::= p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \mid \neg\varphi$$

2.2 Lógica Modal

Lógica Modal [13] é uma lógica que estende a Lógica Clássica Proposicional com a ideia de modalidade. Na lógica modal uma simples afirmação "Amanhã fará Sol" pode receber uma noção de modalidade como "Possivelmente, amanhã fará Sol", que indica que existe a possibilidade de fazer Sol amanhã.

Lógicas modais expressam o conceito de modalidade por meio de operadores modais, a Lógica Modal Clássica tem os operadores de possibilidade $\diamond$ e necessidade $\Box$. Mas existem outras lógicas modais com outros conceitos de modalidade, como modalidades de tempo ("sempre é o caso de φ", "foi o caso de φ", "será o caso de φ"), modalidades deônticas ("é obrigatório φ", "é permitido φ"), lógicas epistemológicas ("é conhecido φ") entre outras.

Para dar semântica à lógica modal, é usado o que chamamos de semântica relacional ou semântica de Kripke. Nesta semântica, é definido uma estrutura de Kripke como um par $\langle W, R\rangle$ onde W é um conjunto, onde seus elementos são chamados de mundos e R é uma relação binária sobre W conhecida como relação de acessibilidade, que controla qual mundo é acessível por qual mundo. Por exemplo, wRu diz que o mundo u é um possível mundo acessível de w.

Um modelo de Kripe é então uma tripla $\mathcal{M} = \langle W, R, V\rangle$, onde:

$\boldsymbol{W}$ é um conjunto não-vazio de estados,

$\boldsymbol{R}$ é uma relação binária sobre $W \times W$ e

$\boldsymbol{V : W \to 2^{\Phi}}$ é uma função de valoração que associa estados de W a conjuntos de proposições atômicas válidas, onde Φ é o conjunto de todas as proposições atômicas.

Seja $\mathcal{M} = \langle W, R, V \rangle$ um modelo, a noção de satisfação de uma fórmula φ em $\mathcal{M}$ no mundo w, notação $\mathcal{M}, w \models \varphi$, é definida indutivamente da seguinte forma.

- $\mathcal{M}, w \models p$ sse $p \in V(w)$;
- $\mathcal{M}, w \models \top$ sempre;
- $\mathcal{M}, w \models \neg\varphi$ sse $\mathcal{M}, w \not\models \varphi$;
- $\mathcal{M}, w \models \varphi_1 \wedge \varphi_2$ sse $\mathcal{M}, w \models \varphi_1$ e $\mathcal{M}, w \models \varphi_2$;
- $\mathcal{M}, w \models \varphi_1 \vee \varphi_2$ sse $\mathcal{M}, w \models \varphi_1$ ou $\mathcal{M}, w \models \varphi_2$;
- $\mathcal{M}, w \models \varphi_1 \to \varphi_2$ sse $\mathcal{M}, w \not\models \varphi_1$ ou $\mathcal{M}, w \models \varphi_2$;
- $\mathcal{M}, w \models \varphi_1 \leftrightarrow \varphi_2$ sse $\mathcal{M}, w \models \varphi_1$ e $\mathcal{M}, w \models \varphi_2$ ou $\mathcal{M}, w \not\models \varphi_1$ e $\mathcal{M}, w \not\models \varphi_2$;
- $\mathcal{M}, w \models \diamond\varphi$ sse existe $w' \in W$ tal que wRw' e $\mathcal{M}, w' \models \varphi$;
- $\mathcal{M}, w \models \Box\varphi$ sse para todo $w' \in W$ tal que wRw' e $\mathcal{M}, w' \models \varphi$.

Como a lógica modal tem muitas variações, elas são classificadas em sistemas axiomáticos, onde a lógica modal mais básica pertence ao sistema K (em homenagem a Saul Kripke) onde dois axiomas valem:

N Regra da necessitação: Se φ é teorema em K, então $\Box\varphi$ também é.

K Regra da distribuição: $\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$

Outros sistemas da lógica modal podem ser obtidos adicionando outros axiomas ao sistema K, por exemplo, o sistema T é o sistema K mais o axioma **T**:$\Box\varphi \to \varphi$ (Axioma da reflexividade).

2.3 Computation Tree Logic

Computation Tree Logic(CTL) [27] é uma lógica onde o tempo é representado como ramificações, ou seja, o modelo de tempo funciona como uma estrutura de árvore. CTL é uma lógica modal temporal que pode representar sentenças como "Hoje eu estou triste, mas um dia estarei feliz, nem que seja por um dia."

Definição 2.3.1 (BNF da CTL)**.** *A gramática da CTL pode ser dada como*

$\varphi ::=$

p

$\mid (\varphi)$

$\mid \neg\varphi$

$\mid \varphi \wedge \varphi$

$\mid \varphi \vee \varphi$

$\mid \varphi \rightarrow \varphi$

$\mid \varphi \leftrightarrow \varphi$

$\mid \boldsymbol{EG}\ \varphi$ – φ *existe globalmente*

$\mid \boldsymbol{EX}\ \varphi$ – φ *existe em algum próximo estado*

$\mid \boldsymbol{EF}\ \varphi$ – φ *existe finalmente*

$\mid \boldsymbol{AG}\ \varphi$ – φ *vale para todos os estados globalmente*

$\mid \boldsymbol{AX}\ \varphi$ – φ *vale para todos os próximos estados*

$\mid \boldsymbol{AF}\ \varphi$ – φ *vale para todos os estados finalmente*

$\mid \boldsymbol{E}\ [\varphi\ \boldsymbol{U}\ \psi]$ – φ *existe até* ψ

$\mid \boldsymbol{A}\ [\varphi\ \boldsymbol{U}\ \psi]$ – φ *vale para todos os estados até* ψ

Tomando "estou triste" como ψ, podemos modelar "Hoje eu estou triste, mas um dia estarei feliz, nem que seja por um dia." como $\psi \wedge \mathbf{EF}\psi$.

CTL é interpretado sobre um sistema de transições, como uma tripla $\mathcal{M} = (S, \rightarrow, L)$, onde:

$\boldsymbol{S}$ é um conjunto não-vazio de estados,

$\rightarrow \subseteq \boldsymbol{S} \times \boldsymbol{S}$ é a relação de transição e

$\boldsymbol{L}$ é uma função de rotulação, rotulando letras proposicionais a estados.

Tome $\mathcal{M} = (S, \rightarrow, L)$ como tal modelo de transição com $s \in S, \psi \in F$, onde F é um conjunto de fórmulas sobre $\mathcal{M}$, então a relação de semântica $(\mathcal{M}, s \models \psi)$ é definida por indução em ψ:

- $(\mathcal{M}, s) \models \top$;
- $(\mathcal{M}, s) \not\models \bot$;
- $(\mathcal{M}, s) \models p$ sse $p \in L(s)$;
- $(\mathcal{M}, s) \models \neg\psi$ sse $(\mathcal{M}, s) \not\models \psi$ $(\mathcal{M}, s) \models \psi_1 \wedge (\mathcal{M}, s) \models \psi_2$;
- $(\mathcal{M}, s) \models \psi_1 \vee \psi_2$ sse $(\mathcal{M}, s) \models \psi_1 \vee (\mathcal{M}, s) \models \psi_2$;
- $(\mathcal{M}, s) \models \psi_1 \rightarrow \psi_2$ sse $(\mathcal{M}, s) \not\models \psi_1 \vee (\mathcal{M}, s) \models \psi_2$;
- $((\mathcal{M}, s) \models \psi_1 \leftrightarrow \psi_2)$ sse $((\mathcal{M}, s) \models \psi_1 \wedge (\mathcal{M}, s) \models \psi_2) \vee (\neg(\mathcal{M}, s) \models \psi_1 \wedge \neg(\mathcal{M}, s) \models \psi_2)$;
- $(\mathcal{M}, s) \models \mathbf{AX}\psi$ sse para todo s_1 tal que $s \rightarrow s_1$ tem-se $(\mathcal{M}, s) \models \psi$;
- $(\mathcal{M}, s) \models \mathbf{EX}\psi$ sse para algum s_1 tal que $s \rightarrow s_1$ tem-se $(\mathcal{M}, s) \models \psi$;
- $(\mathcal{M}, s) \models \mathbf{AG}\psi$ sse para todos os caminhos $s_1 \rightarrow s_2 \rightarrow \ldots$ onde $s = s_1$ e para todos s_i ao longo do caminho tem-se $(\mathcal{M}, s_i) \models \psi$;
- $(\mathcal{M}, s) \models \mathbf{EG}\psi$ sse existe um caminho $s_1 \rightarrow s_2 \rightarrow \ldots$ onde $s = s_1$ e para todos s_i ao longo do caminho tem-se $(\mathcal{M}, s_i) \models \psi$;
- $(\mathcal{M}, s) \models \mathbf{AF}\psi$ sse para todos os caminhos $s_1 \rightarrow s_2 \rightarrow \ldots$ onde $s = s_1$, existe um s_i ao longo do caminho tal que $(\mathcal{M}, s_i) \models \psi$;
- $(\mathcal{M}, s) \models \mathbf{EF}\psi$ sse existe um caminho $s_1 \rightarrow s_2 \rightarrow \ldots$ onde $s = s_1$, existe um s_i ao longo do caminho tal que $(\mathcal{M}, s_i) \models \psi$;
- $(\mathcal{M}, s) \models \mathbf{A}[\psi_1 \mathbf{U} \psi_2]$ sse para todos os caminhos $s_1 \rightarrow s_2 \rightarrow \ldots$ onde $s = s_1$, esse caminho satisfaz $\psi_1 \mathbf{U} \psi_2$, ou seja, existe algum s_i ao longo desse caminho, tal que $(\mathcal{M}, s_i) \models \psi_2$ e, para cada $j < i$, tem-se $(\mathcal{M}, s_j) \models \psi_1$;
- $(\mathcal{M}, s) \models \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ sse existe um caminho $s_1 \rightarrow s_2 \rightarrow \ldots$ onde $s = s_1$, e esse caminho satisfaz $\psi_1 \mathbf{U} \psi_2$, ou seja, existe algum s_i ao longo desse caminho, tal que $(\mathcal{M}, s_i) \models \psi_2$ e, para cada $j < i$, tem-se $(\mathcal{M}, s_j) \models \psi_1$.

2.4 Reo

Desde a década de noventa, desenvolvedores de software têm pesquisado novas formas de como produzir *software*. Surgiram novas técnicas como a computação orientada a serviços [37] e o desenvolvimento orientado a modelos [5], onde o primeiro promove a ideia de compor softwares de outros softwares e o segundo tem a ideia de desenvolver baseando-se em modelos prévios.

Reo [3] é uma linguagem gráfica baseada em coordenações composta por canais onde coordenadores complexos são composicionalmente construídos a partir dos mais simples. Tendo o principal objetivo de funcionar como uma "linguagem de cola"onde ele conecta instâncias de diferente componentes para agirem juntos num único sistema baseado em componentes.

Os canais de Reo podem ser vistos como primitivas para modelar sistemas concorrentes, implementando nativamente propriedades desses sistemas, como chamadas remotas de funções, troca de mensagens e compartilhamento de memória. Portanto, Reo foca nesses conectores, suas composições e como eles se comportam, sem se importar com as entidades nas quais ele está conectado. Logo essas entidades podem ser qualquer componente de software, como serviços web, módulos de código sequencial, objetos, agentes, processos entre outros. Essas entidades são conhecidas como instância de componente no Reo.

Na computação orientada a serviços, é esperado que cada componente seja independente do outro e mais adaptado ao ambiente em que ele deve ser executado. Portanto, se a integração desses serviços for considerada como os meios em que os dados são trocados por esses componentes, essa definição deve ser exterior a definição de cada componente. Logo, é necessária a construção de um "código de cola" para cada software, Reo provê conceitos e ferramentas para desenvolvimento desse "código de cola" que age como o coordenador dessa integração entre os serviços.

Um sistema em Reo é composto por instâncias de componentes que interagem entre si por meio de conectores. Instâncias de componentes são definidas como um conjunto não vazio P que denota o conjunto de entidades envolvidas numa instância e um conjunto predefinido de operações I/O associadas à essas entidade, onde o único meio de realizar tais operações é por meio dos fins dos canais conectados a esse conjunto. Vale ressaltar que esses componentes de software são entidades abstratas que descrevem o comportamento de suas instâncias.

Canais em Reo são definidos como ligações ponto a ponto entre dois nós distintos, onde cada canal tem seu comportamento único predeterminado. Canais são usados para compor conectores mais complexos, que podem ser combinados entre si ou combinados com os conectores canônicos dados por [3] vistos na Tabela 2.1 com uma breve explicação de seus comportamentos. Além disso é possível a construção de conectores a partir das definições de canais criadas por um usuário.

Tabela 2.1: Tabela contendo os conectores Reo dados por [3] e seus comportamentos

Nome	**Imagem**	**Comportamento**
Sync	$A \longrightarrow B$	O dado vai da porta A para a porta B.
LossySync	A - - - - - - B	O dado vai da porta A para a B com possibilidade de perda.
FIFO	A —□→ B	O dado da porta A é armazenado no *buffer* até que possa ser escrito na B.
SyncDrain	A ⊢——⊣ B	As duas portas devem ter algum dado.
AsyncDrain	A ⊢—‖—⊣ B	As portas não podem ter dado ao mesmo tempo.
Filter	A ∿∿→ B	O dado da porta A vai para a B caso uma propriedade parametrizada seja verdadeira.
Transform	A —▶→ B	O dado da porta A vai para a B após sofrer uma transformação parametrizada.
Merger	B, A ⋟—• C	A porta C recebe o dado da porta A ou B de forma não-determinística.
Replicator	A —•⋞ C, B	O dado da porta A vai para as portas B e C.

Um nó em Reo é definido como a organização lógica que denota a estrutura de como os fins dos canais estão conectados entre si num conector, ou seja, um nó é um ponto onde os fins dos canais coincidem. Um nó em Reo pode ser: nó de origem, aqueles que aceitam dado para dentro do canal; nós de fim, da onde o dado sai de um canal; ou nós mistos, que funcionam tanto como origem quanto fim, mas não simultaneamente.

Um fim do canal pode ser usado por qualquer entidade para enviar ou receber dados, dado que essa entidade pertença a uma instância que conhece esses fins. A ligação entre um fim de canal e uma instância é uma conexão lógica que independe da localização de

tal entidades, portanto tanto o canal quanto o componente podem ser móveis, levando a possibilidade de configurações de conectores Reo dinâmicos.

Como um exemplo de modelo Reo, a Figura 2.1 apresenta um sequenciador[1]. Esse conector modela o sequenciamento dos processos que estão interconectados pelos meios desse conector. O dado chega no conector e é armazenado no primeiro *buffer*, então se possível, o dado vai ser escrito na porta de saída B, dando início ao processo conectado à porta B, e também será armazenado no segundo *buffer*, e novamente, se possível, ele será escrito na porta C e armazenado no terceiro *buffer*, e por último, o dado será escrito na porta D e então toda sequência será reiniciada, desta forma é garantida a sequência dos processos B, C e D. Portanto propriedades como a forma em que os dados fluem entre as entidades conectadas podem ser declaradas e provadas usando seu respectivo *contraint automaton*.

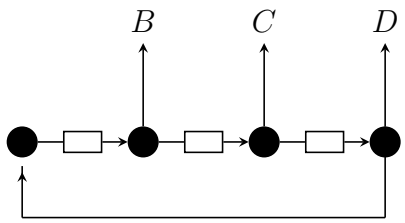

Figura 2.1: Modelo de um sequenciador em Reo

2.5 Constraint Automata

Constraint Automata são definidos como os modelos operacionais mais básicos para Reo [8], apesar de existirem várias outras semânticas formais para Reo [28]. Este trabalho foca nos *Constraint Automata* como foi proposto pelos criadores de Reo e por ser um dos formalismos mais proeminentes para se raciocinar a respeito de Reo.

CA podem ser visto como uma variação de Autômatos Finitos onde as transições são influenciadas por portas que contêm dados e por restrições dos dados nessas portas, ao invés de depender somente do valor dado na entrada. Essas portas são os fins dos canais de Reo, e a ideia de estabelecer restrições sobre o fluxo de dados permite o enriquecimento dos cenários de modelagem, servindo muito bem para o funcionamento de Reo.

Ao se usar *Constraint Automata* como semântica formal para Reo, temos que os estados do autômato representam as possíveis configurações de um canal (por exemplo o dado dentro do conector num dado tempo), enquanto que as transições do autômato

[1] http://reo.project.cwi.nl/v2/#examples-of-complex-connectors

denotam como os dados no conector fluem e como esse fluxo muda a configuração do autômato.

Formalmente, *Constraint Automata* é definido como

Definição 2.5.1 (Constraint Automata)**.** *Um Constraint Automaton (CA) é uma tupla* $\mathcal{A} = (Q, \mathcal{N}ames, \rightarrow, Q_0)$ *onde*

$\boldsymbol{Q}$ *é um conjunto finito de estados,*

$\boldsymbol{\mathcal{N}ames}$ *é um conjunto finito de nomes,*

$\boldsymbol{\rightarrow : Q \times 2^{\mathcal{N}ames} \times DC \times Q}$ *é a relação de transição onde DC é um conjunto de Data Constraints (em lógica proposicional) , e*

$\boldsymbol{Q_0 \subseteq Q}$ *é o conjunto dos estados iniciais.*

A Figura 2.2 mostra graficamente o CA de um FIFO de dois dados sem restrições. A ideia desse autômato é verificar se dado fluxo de dados descreve o comportamento de uma fila com exatamente dois itens de dados (sem restrições sobre esse fluxo), isto é, os dados estão fluindo da porta A para a porta B num conector Reo.

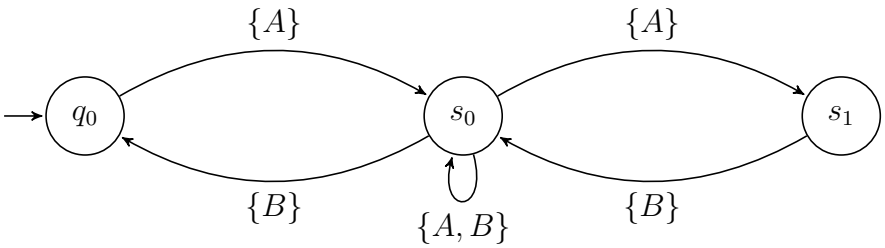

Figura 2.2: Representação gráfica de um CA FIFO de dois dados sem restrições

Streams são definidas como um conjunto A^ω contendo todas as sequências infinitas sobre A, onde A pode ser qualquer conjunto. Então, $A^\omega = \{\alpha \mid \alpha \colon \{0, 1, 2, \dots\} \rightarrow A\}$.

Definição 2.5.2 (*Stream*)**.** *Streams individuais são descritas como* $\alpha = \alpha(0), \alpha(1), \alpha(2), \dots$ *e a derivada de uma stream* α *é denotada como a stream começando no próximo valor,* $\alpha' = \alpha(1), \alpha(2), \alpha(3), \dots$ *onde* $\alpha^{(i)}$ *denota a i-ésima derivada, tal que* $\alpha'(k) = \alpha(k+1)$ *e* $\alpha^{(i)}(k) = \alpha(i+k), \forall i, \forall k > 0$.

Timed Data Streams são compostas por duas *streams* (α, a), onde $\alpha)$ denota os itens de dados no domínio de um autômato e a denota o instante de tempo que um elemento de dado está numa porta. *Constraint Automata* são vistos como aceitadores de TDS.

Definição 2.5.3 (*Timed Data Streams*)**.** *Uma Timed Data Streams é definida como um par de funções* (α, a) *como segue.*
$TDS = \{\ (\alpha, a) \in Data^{\omega} \times \mathbb{R}_{+}^{\omega} \colon \forall k \geq 0 \colon a(k) \leq a(k+1)\ e\ \lim_{k\to\infty} a(k) = \infty\ \}$

Portanto, TDS são compostas por uma *stream* $\alpha \in Data^{\omega}$ onde *Data* é um conjunto não vazio finito e uma *stream* de tempo $a \in \mathbb{R}_{+}^{\omega}$, de crescentes números reais positivos.

Para formalizar o conceito do comportamento de entrado/saída do *Constraint Automata* por meio de TDS, um conjunto de nomes *Names* é usado, onde *Names* é um conjunto finito de nomes $\mathrm{A}_1, \mathrm{A}_2, \ldots, \mathrm{A}_n$ usado para identificar as portas que conectam os diferentes componentes. Para cada porta $A_i \in \mathcal{N}ames$, um TDS é definida. Portanto temos $\mathrm{TDS}^{\mathcal{N}ames}$ como o conjunto de todas as tuplas TDS definidas para cada porta $A_i \in \mathcal{N}ames$.

Definição 2.5.4 ($\mathrm{TDS}^{\boldsymbol{\mathcal{N}ames}}$)**.** $TDS^{\mathcal{N}ames}$ *é um conjunto contendo uma TDS para cada porta* $A_i \in \mathcal{N}ames$ *como*
$TDS^{\mathcal{N}ames} = \{((\alpha_1, a_1), (\alpha_2, a_2), \ldots, (\alpha_n, a_n)) \colon (\alpha_i, a_i) \in TDS, i = 1, 2, \ldots, n,\ onde\ n = |\mathcal{N}ames|.\ \}$

Atribuição de Dados denota qual dado está em cada porta que pertence a um conjunto não vazio de portas $N \subseteq \mathcal{N}ames$.

Definição 2.5.5 (Atribuição de Dados)**.** *Uma Atribuição de Dados* δ *descreve a atribuição de algum item de dado* $\delta_A \in Data$ *para uma porta* $A \in \mathcal{N}ames$*, ou seja,* $\delta = [A \mapsto \delta_A : A \in N]$.

Tomando $\theta = ((\alpha_1, a_1), (\alpha_2, a_2), \ldots, (\alpha_n, a_n) : (\alpha_i, a_i)) \in TDS^{\mathcal{N}ames}$, temos $\theta.time$ definido como a *time stream* obtida juntando todas *timed streams* $a_1, a_2, \ldots, a_n$. A cada iteração, o valor de $\theta.time$ é recalculado como o menor valor de tempo de tal junção, considerando θ' como a derivada de θ.

Definição 2.5.6 ($\boldsymbol{\theta.time(k)}$)**.** *A junção de* time streams *em ordem crescente denota* $\theta.time(k)$ *como:*

- $\theta.time(0) = min\{a_i(0) : i = 1, 2, \ldots, n\}$,
- $\theta.time(m+1) = min\{a_i(k) : a_i(k) > \theta.time(m), i \in 1, 2, \ldots, n\ e\ k \in 1, 2, \ldots.$

Com $\theta.time(k)$ como o k-ésimo tempo mínimo onde dado começa a fluir em uma porta, $\theta.N = \theta.N(0), \theta.N(1), \theta.N(2), \ldots$ captura a noção de selecionar todas as portas que estão em $\theta.time(k)$ como uma *stream* sobre $2^{\mathcal{N}ames}$.

Definição 2.5.7 ($\boldsymbol{\theta.N(k)}$)**.** *$\theta.N(k)$ denota todas as portas que contêm dado em um instante de tempo $\theta.time(k)$:*

$$\theta.N(k) = \{A_i \in \mathcal{N}a, es : a_1(l) = \theta.time(k) \text{ para algum } l \in \{0, 1, 2, \dots\}, i = 1, 2, \dots, n\}$$

Seguindo a mesma ideia, o conceito de uma *stream* sobre o fluxo de dados em portas em $\theta.time$ é definido como $\theta.\delta = \theta.\delta(0), \theta.\delta(1), \theta.\delta(2), \dots$ como uma *stream* sobre o conjunto de atribuições de dados para cada porta $A_i \in \theta.N$. Intuitivamente, $\theta.\delta(k)$ tem todo fluxo de dados observado no instante de tempo $\theta.time(k)$.

Definição 2.5.8 ($\boldsymbol{\theta.\delta(k)}$)**.** *A* stream *$\theta.\delta(k)$ sobre o conjunto de Atribuições de Dados é definido como*

$$\theta.\delta(k) = [A_i \mapsto \alpha_i(l_i) : A_i \in \theta.N(k)] \text{ onde } l_i \in [0, 1, \dots] \text{ é o índice único com}$$

$$a_i(l_i) = \theta.time(k)$$

Como definido pela Definição 2.5.1, transições de *Constraint Automata* são etiquetadas por pares contendo um subconjunto não vazio $N \subseteq \mathcal{N}ames$ e uma *data constraint* g. Uma DC é vista como uma representação de qual dado pode ser observado numa dada porta, onde sua fórmula proposicional é construída de proposições atômicas como $d_A = d$, significando que o dado na porta A deve ser d, onde $A \in Names$ e $d \in Data$.

Definição 2.5.9 (Data constraints)**.** *Uma data constraint (DC) g é formalmente definida pela seguinte gramática:*

$$g ::= true \mid d_a = d \mid g_1 \vee g_2 \mid \neg g.$$

Para facilitar a leitura, uma transição é denotada como $q \xrightarrow{N,g} p$ ao invés de $(q, N, g, p) \in \rightarrow$, onde $\rightarrow$ é a relação de transição definida na Definição 2.5.1, $q, p \in Q$ são estados do autômato, g uma DC e $N \subseteq \mathcal{N}ames$. Para cada transição, é necessário que $N \neq \emptyset$ e que g seja satisfatível por fluxo de dados da $\text{TDS}^{\mathcal{N}ames}$ no instante k $\theta.\delta(k)$ naquele instante de tempo k.

Como uma transição pode ser vista como a mudança na configuração de um conector num único passo, uma transição $q \xrightarrow{N,g} p$ significa que o autômato se encontra na configuração q, tem dados nas portas $A \in N$ e esses dados satisfazem a DC g, enquanto que nas portas $N \setminus \mathcal{N}ames$ não tem dados, então no próximo instante o autômato se encontrará no estado p.

Portanto uma execução de um autômato $\mathcal{A}$ começa em um dos seus estados iniciais $q_0 \in Q_0$, e desse estado $\mathcal{A}$ vai esperar por um dado em ao menos uma porta $A_i \in \mathcal{N}ames$. Então de q_0, o autômato verifica se existe ao menos uma transição que parte de q_0 nos quais as portas com dados satisfazem a DC g dessa transição e todas as outras portas $N \setminus \mathcal{N}ames$ não tenham dados.

Definição 2.5.10 (Execuções de *Constraint Automata*)**.** *Dado uma tupla TDS $\theta \in TDS^{\mathcal{N}ames}$ como entrada, o conjunto de infinitas execuções de um CA $\mathcal{A}$ é denotado pelo maior conjunto de streams $q = q_0, q_1, q_2, \ldots$ sobre Q onde:*

(i) Existe uma transição $q_0 \xrightarrow{N,g} q_1$;

(ii) $N = \theta.N(0)$;

(iii) $\theta.\delta(0) \models g$;

(iv) q' (uma stream infinita obtida do estado resultante de (i)) significa infinitas q_1-execuções sobre θ' em $\mathcal{A}$;

onde (i) denota que é necessário que exista ao menos uma transição que possa ser executada do estado atual na execução, (ii) significando que nenhuma outra porta além das portas envolvidas na execução tem dados, (iii) diz que os dados nessas portas devem satisfazer g e (iv) representa que o mesmo comportamento é esperado no resto da execução.

A Tabela 2.2 oferece o CA de cada um dos conectores canônicos da Tabela 2.1, de acordo com seus respectivos comportamentos. Os nomes entre $\{\}$ são as portas que podem ter dados na transição, enquanto que o rótulo abaixo representa a *data constraint* da transição. Caso o rótulo de baixo esteja omitido, significa que a transição não tem restrição para seus dados.

A ideia de composicionalmente construir conectores Reo mais complexos a partir de conectores mais simples é unir os nós de origem em Reo com os outros nós por meio de um produto entre autômatos. Portanto, a união natural das linguagens $\mathcal{L}_1$ e $\mathcal{L}_2$, respectivamente dos autômatos $\mathcal{A}_1$ e $\mathcal{A}_2$ é análogo a mesma operação para bancos de dados relacionais [4].

Explorando essa ideia, tome dois circuitos Reo com portas denotadas por $\mathcal{N}ames_1$ e $\mathcal{N}ames_2$. A ideia da operação de produto é juntar todas as portas comuns $B \in \mathcal{N}ames_1 \cap \mathcal{N}ames_2$ sem perder as portas $A \in \mathcal{N}ames_1$ e $C \in \mathcal{N}ames_2$, onde $A, C \notin \mathcal{N}ames_1 \cap \mathcal{N}ames_2$. Recuperando o exemplo definido em [4], tome $L_1(A, B)$ e $L_2(B, C)$, onde a

Tabela 2.2: Tabela contendo os conectores Reo canônicos e seus respectivos CA [22].

Conector	**Reo**	***Constraint automaton***
Sync	$A \longrightarrow B$	q_0 $\{A,B\}$ $d_A = d_B$
LossySync	$A \dashrightarrow B$	$\{A\}$ q_0 $\{A,B\}$ $d_A = d_B$
FIFO	$A \longrightarrow\!\!\square\!\!\longrightarrow B$	$\{B\}$ $d_B = 1$ $\{B\}$ $d_B = 0$ p_1 q_0 p_0 $\{A\}$ $d_A = 1$ $\{A\}$ $d_A = 0$
SyncDrain	$A \longrightarrow\!\!\dashv\!\!\longleftarrow B$	q_0 $\{A,B\}$
AsyncDrain	$A \longrightarrow\!\!\dashv\!\!\vdash\!\!\longleftarrow B$	$\{B\}$ q_0 $\{A\}$
Filter	$A \rightsquigarrow B$	$\{A,B\}$ $P(A) \wedge d_A = d_B$ q_0 $\{A\}$ $\neg P(d_A)$
Transform	$A \longrightarrow\!\!\blacktriangleright\!\!\longrightarrow B$	q_0 $\{A,B\}$ $f(d_A) = d_B$
Merger	B A C	$\{A,B\}$ $d_A = d_B$ q_0 $\{A,C\}$ $d_A = d_C$
Replicator	A C B	q_0 $\{A,B,C\}$ $d_A = d_B \wedge d_A = d_C$

notação $L_i(X)$ significa "L_i é a linguagem TDS para o conjunto de nome X". O produto de $\mathcal{N}ames_1 \cap \mathcal{N}ames_2$ é a linguagem TDS resultante $L_1 \cap L_2(A, B, C)$, formalizada como

$$\boldsymbol{L_1 \cap L_2} = \{ \ (\ (\alpha, a), \ (\beta, b), \ (\gamma, c) \ ) : ((\alpha, a), \ (\beta, b)) \in L_1 \text{ e } ((\beta, b), \ (\gamma, c)) \in L_2 \ \}.$$

O autômato do produto que encapsula o aceitador dessa linguagem TDS resultante é definido formalmente como

Definição 2.5.11 (*Product Automata [8]*)**.** *Dado dois Constraint Automata* $A_1 = (Q_1, \mathcal{N}ames_1, \rightarrow_1, Q_{0,1}$ *e* $A_2 = (Q_2, \mathcal{N}ames_2, \rightarrow_2, Q_{0,2}$*, o Product Automaton* $A_1 \bowtie A_2$ *é formalmente defi-*

nido como $A_1 \bowtie A_2 = (Q_1 \times Q_2, \mathcal{N}ames_1 \cup \mathcal{N}ames_2, \rightarrow, Q_{0,1} \times Q_{0,2})$*, onde* $\rightarrow$ *é a relação de transição resultante, definida como*

$$(i) \quad \frac{q_1 \xrightarrow{N_1,g_1} p_1, q_2 \xrightarrow{N_2,g_2} p_2, N_1 \cap \mathcal{N}ames_2 = N_2 \cap \mathcal{N}ames_1}{(q_1, q_2) \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} (p_1, p_2)}$$

$$(ii) \quad \frac{q_1 \xrightarrow{N,g} p_1, N \cap \mathcal{N}ames_2 = \emptyset}{(q_1, q_2) \xrightarrow{N,g} (p_1, q_2)}$$

$$(iii) \quad \frac{q_2 \xrightarrow{N,g} p_2, N \cap \mathcal{N}ames_1 = \emptyset}{(q_1, q_2) \xrightarrow{N,g} (q_1, p_2)}$$

Intuitivamente, as regras para construir as transições do autômato resultante do produto são

regra (*i*) significa unir as transições que podem ocorrer simultaneamente. $N_1 \cap \mathcal{N}ames_2$ ser igual a $N_2 \cap \mathcal{N}ames_1$ indica que as duas interseções podem ocorrer ao mesmo tempo, portanto o autômato resultante terá uma transição que é a união de ambas transições.

regras (*ii*) e (*iii*) denotam a ideia de preservar as transições de um autômato que não se relacionam com o outro autômato, isso é observado ao ver que a interseção das portas da transição com as portas do outro autômato é vazia.

A Figura 2.3 apresenta um fragmento do conector apresentado na Figura 2.1. Esse conector é composto por um canal FIFO e um canal *Sync*, o autômato resultante desse produto está representado na Figura 2.4, neles podemos ver que as transições do FIFO $q_0 \xrightarrow{\{A\},d_A=0} p_0$ e $q_0 \xrightarrow{\{A\},d_A=1} p_1$ foram preservadas. Enquanto que a transição do *Sync* $q_0 \xrightarrow{\{B,C\},d_B=d_C} q_0$ foi unida com a transição $p_0 \xrightarrow{\{C\},d_C=0} q_0$ e com a transição $p_1 \xrightarrow{\{C\},d_C=1} q_0$ do FIFO.

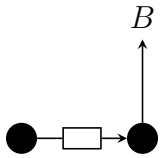

Figura 2.3: Um fragmento do conector sequenciador

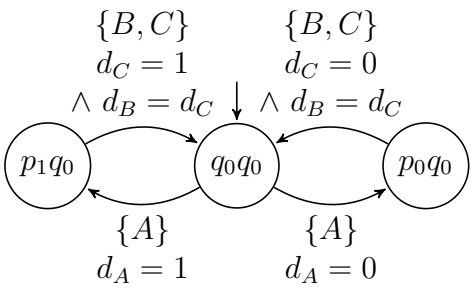

Figura 2.4: O CA do fragmento do conector do sequenciador

2.6 Sistemas Dinâmicos

Sistemas dinâmicos são sistemas que são capazes de modelar comportamentos contínuos. Esses comportamentos são comumente observados em interações com o mundo físico, como por exemplo dispositivos médicos, redes de comunicação, sistemas de defesas, etc. Desta forma, sistemas dinâmicos são muito úteis em modelar sistemas ciber-físicos (CPSs), onde a computação e comunicação é integrado com o monitoramento e controle de entidades físicas.

Um exemplo de um cenário onde é adequado o uso de um sistema dinâmico seria na modelagem do movimento de um trem, onde ele tem um movimento contínuo pelo trilho, mas ao ser acionado o freio sua aceleração é subitamente reduzida.

Considere a seguinte equação diferencial:

$$\dot{\xi} = f(\xi)$$

onde a variável pontilhada é a primeira derivada durante a mudança contínua e $f : \mathcal{R}^n \to \mathcal{R}^n$ é uma função infimamente diferenciável. Pela trajetória previamente definida, com condição inicial $x \in \mathcal{R}^n$, uma curva suave é

$$\xi : [0, \tau) \to \mathcal{R}^n$$

satisfazendo:

- $\tau > 0$;
- $\xi(0) = x$;
- $\dot{\xi}(t) = f(\xi(t))$ para cada $t \in (o, \tau)$.

Logo, um sistema dinâmico n-dimensional pode ser definido como $\Sigma = (\mathcal{R}^n, f)$, onde

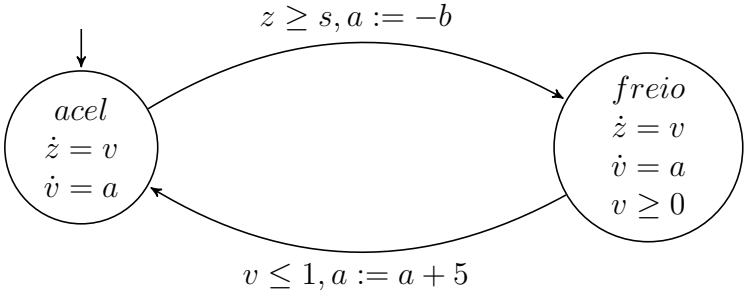

Figura 2.5: Autômato híbrido para um sistema de controle de trem simplificado

$\mathcal{R}^n$ é o espaço real equipado com a equação diferencial dado pelo mapeamento suave $f : \mathcal{R}^n \to \mathcal{R}^n$.

Intuitivamente, um sistema dinâmico n-dimensional $\Sigma = (\mathcal{R}^n, f)$ descreve como um ponto P flui no espaço $\mathcal{R}^n$ baseado nas regras dadas pelas equações diferenciais. Faz sentido que se existem duas trajetórias ξ_1 com duração τ_1 e ξ_2 com duração τ_2 compartilhando a mesma condição inicial, então $\xi_1(t) = \xi_2(t)$ para cada $t \in [0, min(\tau_1, \tau_2)]$.

A Figura 2.5 [38] apresenta um autômato híbrido representando um sistema de controle de trem simplificado, onde cada nó corresponde a um sistema dinâmico. No nó *acel* as equações diferencias $\dot{z} = v$ e $\dot{v} = a$ descrevem o movimento do trem enquanto acelera. As aresta descrevem o comportamento discreto alterando entre os comportamentos contínuos do trem, elas apresentam guardas que devem ser válidas e transformações discretas que ocorrem instantaneamente quando o sistema segue uma aresta. Na aresta que sai do nó *acel*, quando o trem z ultrapassar o ponto s, o trem vai frear transformando a aceleração para $a := -b$ e entrar no modo $freio$. Já nesse nó, as equações diferencias $\dot{z} = v$ e $\dot{v} = a$ se aplicam somente sobre a região da invariante $v \geq 0$, o trem não anda para trás ao frear. Enquanto o trem está freando, quando sua velocidade v for menor ou igual a 1, então ele vai entrar no modo de acelerar com $a := a + 5$.

Uma restrição espacial φ para um sistema dinâmico n-dimensional Σ é definido como um predicado sobre as variáveis livres $\#_1, \#_2, ..., \#_n$ onde $\#_i$ é a i-ésima coordenado de um ponto $x \in \mathcal{R}^n$ para cada $i \in 1, 2, ..., n$. Se $n = 1$, então $\#_1$ é abreviado para $\#$. Para denotar que o ponto x no espaço $\mathcal{R}^n$ satisfaz a restrição espacial φ, $x \models \varphi$ é usado.

Para ilustrar, tome x_1 e x_2 como dois pontos no espaço $\mathcal{R}^3$ com coordenadas

$$x_1 = (1, 1, 1) \text{ and } x_2 = (1, 2, 3)$$

e φ_1 e φ_2 duas restrições espaciais definidas como

$$\varphi_1 = (\#_1 = \#_2) \text{ and } \varphi_2 = (\#_1 + \#_2 \geq \#_3)$$

então:

$$x_1 \models \varphi_1 \text{ and } x_1 \not\models \varphi_2$$

$$x_2 \not\models \varphi_1 \text{ and } x_2 \models \varphi_2$$

2.7 Hybrid Constraint Automata

Hybrid Constraint Automata (HCA) [14] é uma extensão de Constraint Automata para modelar sistemas dinâmicos. É uma tupla $\Gamma(S, \mathcal{R}^n, \mathcal{N}, \delta, IS, \{In_s\}_{s \in S}, \{f_s\}_{s \in S}, \{re_t\}_{t \in \delta})$, onde:

- S é um conjunto finito de estados e IS é o conjunto de estados iniciais, tal que $IS \subseteq S$;
- $\mathcal{R}^n$ é o espaço do sistema dinâmico;
- $\mathcal{N}$ é o conjunto finito de portas;
- $\Sigma_s = (In_s, f_s)$ é o sistema dinâmico n-dimensional para cada estado $s \in S$, onde In_s é a invariante do sistema no estado s e $f_s : \mathcal{R}^n \to \mathcal{R}^n$ é o mapeamento suave para o sistema em s;
- δ é conjunto de transições entre os estados, como um subconjunto $S \times 2^{\mathcal{N}} \times DC \times SC \times S$;
- a função de reset $re_{(s,N,g,\varphi,\bar{s})} : Date^N \times IN_s \to In_{\bar{s}}$.

Intuitivamente, HCA opera da seguinte forma: Γ começa em um dos estados iniciais $s_0 \in IS$ e se comporta como o sistema dinâmico Σ_{s_0}, então, se Γ está em um estado $s \in S$ com $x \in In_s$, ele:

- deve escolher uma transição $t = (s, N, g, \varphi, \bar{s})$ de δ tal que a atribuição de dados $\varepsilon \models g$ e $z \models \varphi$ e se as operações de I/O especificadas em ε ocorrem exatamente nos nós presentes em N. Se t é aceita, as operações de I/O são executadas, o modelo move para o estado $\bar{s}$ e se comporta de acordo com sistema $\Sigma_{\bar{s}}$ com a condição inicial dada por $re_t(\varepsilon, x) \in In_{\bar{s}}$;

- deve escolher uma transição $t = (s, N, g, \varphi, \bar{s})$ de δ onde $N = \emptyset$ e $g = []$ tal que $x \models \varphi$, e Γ viola a invariante In_s e nenhuma operação de I/O está acontecendo no momento. Se t é aceita, o modelo vai para o estado $\bar{s}$ e se comporta de acordo com o sistema $\Sigma_{\bar{s}}$ com condição inicial dada por $re_t([], x) \in In_{\bar{s}}$;

- continua em s e se comporta seguindo o sistema dinâmico Σ_s, desde que não tenha que se mover para um novo estado.

Uma transição no HCA é classificada como:

- (*Flow*) $s = \bar{s}, N = \emptyset, \varepsilon = [], \tau > 0$ e existe uma trajetória ξ com duração τ e condição inicial x qua vai para $\bar{x} = \lim_{t \to \tau^-} \xi(t)$;

- (*External interaction*) $s \xrightarrow[re]{N,g,\varphi} \bar{s}, N \neq \emptyset, \varepsilon \in DA(N), \varepsilon \models g, x \models \phi, \tau = 0$ and $\bar{x} = re(\varepsilon, x)$;

- (*Internal jump*) $s \xrightarrow[re]{N,g,\varphi} \bar{s}, N = \emptyset, \varepsilon = [], g = true, x \models \phi, \tau = 0$ e $\bar{x} = re([], x)$.

Uma excução no HCA é uma sequência de transições que alternam entre transições contínuas (*flows*) e ações discretas (*interactions* ou *jumps*)

$$q_0 \xrightarrow{N_0,g_0,\varphi_0} q_1 \xrightarrow{N_1,g_1,\varphi_1} \dots$$

Essa alternância deve ocorrer pois se uma execução possui duas transições contínuas seguidas, elas podem ser colapsadas em uma transição só sem perda de comportamento. Já se existirem duas transições discretas seguidas, então essas ações estão sendo executadas simultaneamente, o que viola a ideia do constraint automata onde as transições que podem ser simultâneas são colapsadas em uma única transição.

Em relação a Constraint automata, um autômato modelado em CA não sofre alterações quando é convertido para HCA, ele basicamente é um HCA cujo sistemas dinâmicos são estáticos.

A figura 2.6 ilustra um HCA que representa um contador, onde ao receber um dado, esse dado vai ser escrito após um determinado tempo. No estado inicial o sistema dinâmico é estático, representa que o autômato está aguardando uma entrada. A transição $s \xrightarrow{\{A\},d:=d_A,\#:=5} p$ representa que a porta de entrada A tem dado, então esse dado vai ser armazenado em d, a variável do sistema dinâmico # vai receber 5 e vai mover para o estado de variável lida p. Enquanto está no estado p, o sistema dinâmico decrementa

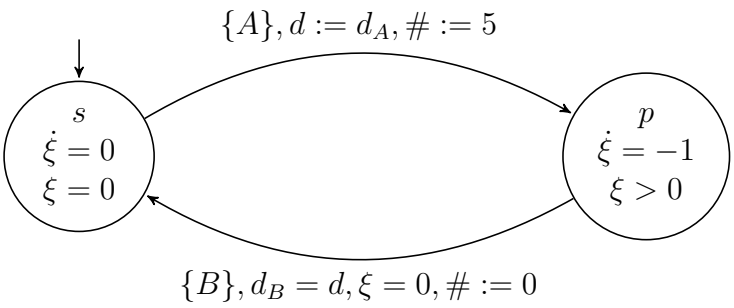

Figura 2.6: Exemplo para ilustrar um HCA (timer)

sua variável em 1 a cada instante enquanto sua invariável $\xi > 0$ valer. A outra transição $p \xrightarrow{\{B\}, d=d_B, \xi=0, \#:=0} s$ é executada se a porta de escrita B tem dado, essa dado é igual ao dado armazenado e se a variável do sistema dinâmico chegou em 0 então o autômato volta para o estado inicial s.

O produto entre um HCA $\Gamma_1(S_1, \mathcal{R}^{n_1}, \mathcal{N}_1, \delta_1, IS_1, In_{1s s \in S_1}, f_{1s s \in S_1}, re_{1t t \in \delta_1})$ e outro HCA $\Gamma_2(S_2, \mathcal{R}^{n_2}, \mathcal{N}_2, \delta_2, IS_2, In_{2s s \in S_2}, f_{2s s \in S_2}, re_{2t t \in \delta_2})$ funciona de forma similar ao produto de dois CA, a diferença está em como os sistemas dinâmicos vão se unir:

- o espaço do sistema dinâmico será $\mathcal{R}^n$ onde $n = n_1 + n_2$
- o sistema n-dimensional $\Sigma_s = (In_s, f_s)$ para cada $s = \langle s_1, s_2 \rangle \in S$ onde $In_s = In_{1s} \times In_{2s}$, e $f_s : In_s \to \mathcal{R}^n$ é uma função definida como $f_s(x_1, x_2) = (f_{1s}(x_1), f_{2s}(x_2))$ para cada $x_1 \in In_{1s}$ and $x_2 \in In_{2s}$;
- a função de reset $re_t : Data^N \times In_s \to In_{\bar{s}}$ para cada $t = (s, N, g, \varphi, \bar{s}) \in \delta$;

A figura 2.7 ilustra um exemplo do produto entre um timer e o lossySync, demonstrando como um CA interage de forma simples com o HCA. É possível observar que a transição do lossySync que ocorre independentemente permanece inalterada enquanto que a outra transição é incorporada nas transições to timer funcionando com o sistema dinâmico.

2.8 nuXmv

Verificadores de modelos são ferramentas usadas na verificação de propriedades caracterizadas acerca de sistemas previamente modelados formalmente. Essa verificação se dá explorando todos os possíveis estados do modelo [7], ou seja, o verificador examina cada

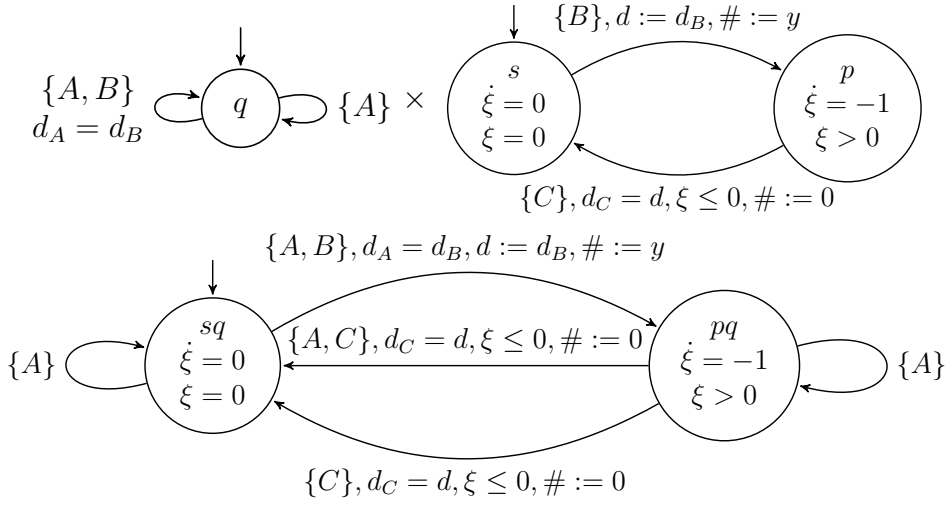

Figura 2.7: Exemplo do produto de HCA entre o lossySync e o timer

possível cenário do sistema de forma sistemática para mostrar que o modelo realmente satisfaz dada propriedade.

Essas propriedades a serem verificadas podem representar características de segurança, ou seja, propriedades que devem sempre valer para que algo indesejado nunca aconteça, como por exemplo a ausência de *deadlock*. Também podem representar características relacionadas a *liveness*, que são propriedades para garantir a evolução do programa, ou seja, elas definem comportamentos que o sistema deve ter para sua execução; além de outras.

Ao se verificar uma propriedade, existem três possíveis resultados: a propriedade é dada como válida; a propriedade é dada como falsa; ou não foi possível verificar a propriedade, seja por falha na modelagem, ou porque o modelo explodiu, ou seja, se tornou muito grande para ser avaliado. Além disso, uma propriedade pode ser do tipo universal (ela vale para todos os estados do sistema), então em caso de falha o verificador apresentará um contra-exemplo; ou do tipo existencial (há algum estado do sistema em que ela vale), nesse caso o avaliador apresentará um exemplo no caso em que ela se dá como verdadeira.

Existem vários verificadores de modelos como o PRISM [31], um verificador probabilístico usado na modelagem e análise de sistemas que apresentam um comportamento aleatório ou probabilístico. Existe o MCRL2 [24], usado na modelagem, validação e verificação de sistemas concorrentes e protocolos, o Vereofy[2], um verificador usado para verificar sistemas baseados em componentes. O nuXmv é usado neste projeto por caraca-

[2] http://www.vereofy.de

teísticas de modularização e algoritmos para verificação que se adequam naturalmente à semântica de Reo. Ele é um verificador simbólico usado tanto na verificação de sistemas síncronos finitos quanto infinitos.

O nuXmv é uma ferramenta que estende o NuSMV [15] adquirindo todas as suas funcionalidades. O NuSMV é um verificador que combina verificação baseada em *Binary Decision Diagram*(BDD) com a verificação baseada em *Boolean satisfability problem*(SAT). nuXmv estende o NuSMV em duas principais direções, na verificação de modelos finitos ele apresenta um forte mecanismo de verificação baseado em algoritmos SAT estados da arte; nos casos da verificação de modelos infinitos, ele apresenta técnicas de verificações baseadas em *Satisfability Modulo Theory*(SMT).

Como um verificador, o nuXmv foi usado em vários projetos como o AUTOGEF [2], com o objetivo de validar sistemas críticos aeroespaciais, o EuRailCheck [12] foi um projeto que desenvolveu uma ferramenta para formalizar e validar as especificações para *The European Train Control System*(ETCS).

Uma das formas que o nuXmv verifica um modelo é baseada no SAT. Esse problema consiste em determinar se existe alguma interpretação que satisfaça uma dada fórmula booleana. A noção de se usar *SATsolvers* no problema de verificação de modelos [16] para atacar o problema da explosão de estados existe desde os anos 2000. Como o problema SAT é NP-completo, existem várias implementações que resolvem o problema de forma diferente buscando minimizar seu custo.

Um programa nuXmv é estruturado como uma lista de **`MODULE`**`s` onde um deles será o `main`, que é de onde o modelo começará a ser montado. Uma declaração de um **`MODULE`** então é onde serão declaradas as variáveis, as transições, as especificações e qualquer outra declaração. Essas declarações seguem uma estrutura de blocos, dos quais alguns são:

`MODULE` encapsula as outra declarações, é declarado como **`MODULE`** `identifier` **`(`** `module_parameters`**`)`**, onde `identifier` é o identificador do **`MODULE`** e `module_parameters` são os parâmetros opcionais do **`MODULE`**.

`VAR` encapsula lista das variáveis pertencentes ao **`MODULE`** que ele se encontra. Uma variável é declarada como `identifier : type_specifier;`, onde `identifier` é seu identificador e `type_specifier` é seu tipo, um **`MODULE`** é considerado como um tipo de variável, dessa forma, é possível instanciar um **`MODULE`** como uma variável.

`FROZENVAR` encapsula uma lista de variáveis que são declaradas como acima, a diferença

é que uma variável declarada neste bloco retêm o primeiro valor atribuído a ela.

ASSIGN encapsula uma lista de atribuições, uma atribuição pode ser do tipo **init**(identifier) := simple_expr, onde a variável com o identifier tem o valor inicial da simple_expr; e do tipo **next**(identifier) := simple_expr, onde identifier terá o valor no próximo estado.

TRANS encapsula uma lista de transições, essas transições são declaradas como expressões booleanas que devem conter a atribuição **next()**.

INVAR encapsula uma lista de expressões booleanas que restringem os estados do modelo.

CTLSPEC encapsula uma expressão CTL a ser verificada no modelo.

LTLSPEC encapsula uma expressão LTL a ser verificada no modelo.

O Listing 2.1 apresenta um exemplo de um autômato modelado no nuXmv, o **MODULE** main tem um bloco **VAR** onde são declaradas as variáveis cs (estado atual) com possíveis valores {s,e}, val (valor da entrada) com possíveis valores {0,1} e input como uma instância de input onde a variável val é passada como parâmetro. O **MODULE** input vai receber uma variável como parâmetro e a iniciará com o valor 0 em seu bloco **ASSIGN** e depois irá alternar o valor dessa variável entre 0 e 1 no bloco **TRANS**. Já no main, em seu bloco **ASSIGN** a variável cs é iniciada com o valor s. O bloco **TRANS** lida com as transições do autômato, onde para cada valor de cs e val ele atribui o próximo valor de cs.

```
1  MODULE main
2  VAR
3     cs: {s,e};
4     val: {0,1};
5     input: input(val);
6  ASSIGN
7     init(cs) := s;
8  TRANS
9     (cs = s & val = 0 -> next(cs) = e) & (cs = s & val = 1 -> next(cs)
         = s) &
10    (cs = e & val = 0 -> next(cs) = e) & (cs = e & val = 1 -> next(cs)
         = s);
11
12 MODULE input(a)
13 ASSIGN
14    init(a) := 0;
15 TRANS
16    (a = 0 <-> next(a) = 1) & (a = 1 <-> next(a) = 0);
```

Algoritmo 2.1: Exemplo de autômato modelado no nuXmv

Chapter 3

Trabalhos Relacionados

Como Reo pode ser usado para modelar várias situações do mundo real, vários trabalhos foram dedicados para formalizar meios de verificar propriedades acerca desses sistemas, além da construção de várias outras semânticas formais para a linguagem [28]. Klein et al. [29] propõem uma plataforma para raciocinar sobre modelos Reo usando o Vereofy[1], um verificador para sistemas baseados em componentes, já Pourvatan et al. [39] usa *Constraint Automata* para formalizar Reo e analisar por meio de execução simbólica. Kokash et al. [30] codifica Reo no verificador mCRL2 [24] usando *Constraint Automata* e suas principais variações. Mouzavi et al. [35] tem uma proposta baseada em Maude[2] para verificar modelos Reo e Li et al. [34] propõe uma extensão em tempo real para o Reo, implementando novos canais usando *Stochastic Timed Automata*(STA) como formalismo para Reo, fornecendo uma tradução de STA para o PRISM.

No uso de HCA, Chen et al. [14] formalizam modelos Reo com HCA para funcionamento em sistemas ciber-físicos, apresentando as provas tanto para o formalismo do HCA quanto para o produto, no qual este trabalho é baseado. Mas até onde sabemos, não existe uma formalização de HCA em uma ferramenta de verificação.

No campo de verificadores de modelo, UPPAL [18] é uma caixa de ferramentas para verificação de sistemas de tempo real. Baseado na teoria de *timed automata*, é apropriado para sistemas que podem ser modelados com uma coleção de processos não-determinísticos com uma estrutura de controle finita e relógios com valores reais. HyTech [25] foi concebido com o objetivo de verificar autômatos híbridos, primariamente aplicações de controle. Usando verificação simbólica, ele adapta a teoria de *timed automata* para autômatos híbridos lineares, além da análise de alcançabilidade padrão, HyTech também oferece

[1] http://www.vereofy.de
[2] http://maude.cs.illinois.edu/w/index.php/The_Maude_System

uma análise de quantificadores, podendo determinar condições necessárias e suficientes sobre seus parâmetros.

Na análise formal de consenso para *blockchain*, LibBFT [36] é uma biblioteca de alta performance criada para tratar de mecanismos de consenso tolerantes a falhas bizantinas. Também inspirado pela teoria de *timed automata*, ela permite modelar algoritmos de consenso como uma máquina de estados temporal de forma direta, desde que o comportamento esperado seja conhecido.

Chapter 4

Compilador

Esse projeto expandiu nossa ferramenta em C para compilar um modelo Reo para um modelo nuXmv chamada Reo2nuXmv [23], essa expansão para comportar HCA é encontrada em `https://github.com/frame-lab/ReoXplore/tree/hybridAutomata`. Ela recebe um circuito Reo como entrada, e automaticamente gera o modelo nuXmv que simula o comportamento desse circuito permitindo a verificação de propriedades acerca desse modelo.

4.1 Novos Canais

Além dos canais canônicos apresentados na Tabela 2.1, são propostos três novos conectores assíncronos no intuito de simplificar a adição do comportamento dinâmico ao Reo. Esta seção vai abordar e explicar cada um desses três conectores.

O primeiro canal, o timer, visto na Figura 4.1, representa um contador onde a intuição é que depois do dado ser lido, após uma determinada quantidade de tempo decorrida o dado vai ser escrito. Observando seu HCA, ele aguarda no estado inicial s, com um sistema dinâmico estático, até que a porta de entrada A tenha dado. Recebido esse dado, ele é armazenado e a variável do sistema dinâmico $\#$ recebe o valor parametrizado para contagem y, então o autômato vai para o estado de contagem p. Nesse estado, o autômato vai esperar de acordo com a função de contagem parametrizada $f(\#)$ enquanto que o valor da variável for maior que 0, então o dado lido vai ser escrito na porta B, reiniciando o autômato para o estado inicial.

Apesar de simples, esse conector já adiciona comportamento dinâmico a um circuito Reo, podendo ser usado por exemplo como um cronômetro, para indicar que após o

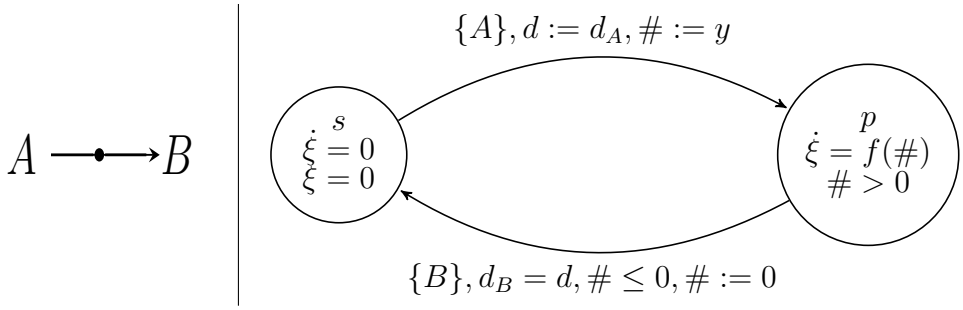

Figura 4.1: Canal Reo para o timer e seu HCA

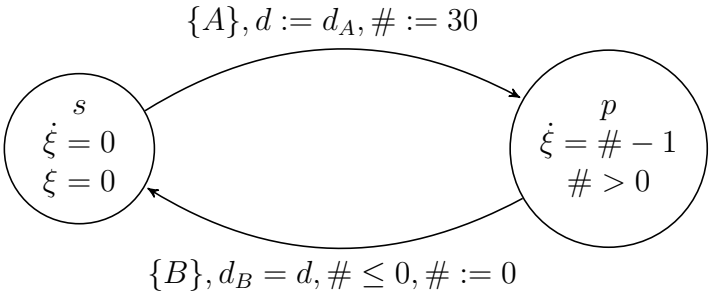

Figura 4.2: Exemplo do HCA do timer com parâmetros para um contador de 30 segundos

recebimento do dado, ele deve ser escrito após y segundos. A Figura 4.2 apresenta um cronômetro que contará 30 segundos, pois a função do sistema decrementa # em 1 a cada instante, para escrever o dado recebido.

O timedDelay visto na Figura 4.3 é um canal onde a intuição é que depois do dado ser lido, ele vai esperar um tempo mínimo até que o dado possa ser escrito enquanto que também tem um tempo máximo para que o dado seja escrito, dessa forma, ele representa um atraso limitado não-determinístico. Observando seu HCA, seu estado inicial é semelhante ao do timer, assim como a transição de leitura, mas variável do sistema dinâmico # vai receber o valor inicial 0. Já no estado p, a variável vai incrementar de acordo com a função parametrizada $f(\#)$ e seu invariante é que # deve ser menor que o atraso máximo parametrizado y. A escrita do dado lido, vai se dar desde que o atraso mínimo parametrizado x tenha contado, mas deve acontecer antes que o atraso máximo y tenha passado.

Esse conector foi criado para simular a comunicação entre dois nós de uma rede, onde sua comunicação leva um tempo mínimo e é garantida para um certo limite de tempo. Com esse conector é possível representar em um circuito Reo a comunicação dos componentes de uma rede levando em consideração a natureza dessa comunicação no mundo real, onde não é possível ter um controle exato do tempo de transmissão. A Figura 4.4 apresenta o HCA desse canal para um limite mínimo de 10 e limite máximo de 50 segundos, onde a

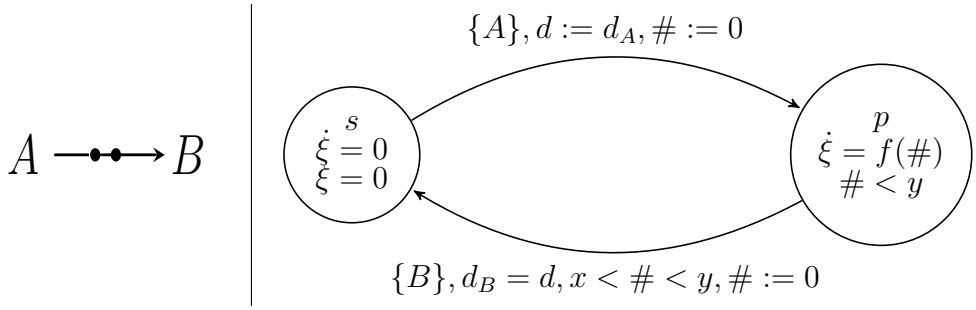

Figura 4.3: Canal Reo para o timedDelay e seu HCA

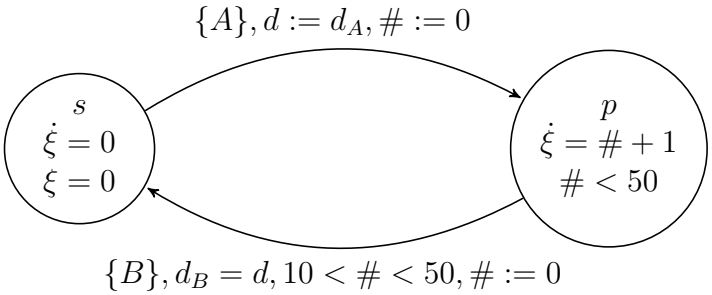

Figura 4.4: HCA do timedDelay com tempo mínimo de 10 segundo e máximo de 50 segundos

função vai incrementando # em 1, dessa forma, o dado será escrito entre 10 e 50 segundos após lido.

O timedTransform visto na Figura 4.5 é um canal onde a intuição é que após o dado ser lido, ele vai sofrer transformações até uma determinada condição, e depois o dado transformado é escrito. Observado seu HCA, o estado inicial s aguardo a chegado do dado, e por isso seu sistema dinâmico é estático. Tendo dado na porta de leitura A, esse dado vai ser armazenado na variável do sistema dinâmico # e vai para o estado de dado lido p. Nesse estado, # vai sofrer transformações de acordo com a função do sistema dinâmico parametrizada $f(\#)$ até que a condição também parametrizada q não seja mais verdadeira. No momento que essa condição não vale mais, dado por $!q$, essa dado transformado vai ser escrito na porta de escrita B e o autômato é reiniciado para o estado inicial s.

Esse conector é diferente dos outros porque apesar de adicionar comportamento dinâmico ao circuito Reo, ele é dependente diretamente do dado que está sendo lido. A figura 4.6 apresenta um HCA do timedTransform para representar um termômetro, onde o dado recebido seria a temperatura atual, e sua função de transformação vai reduzindo a temperatura até o valor determinado de 20, então a temperatura atual é escrita.

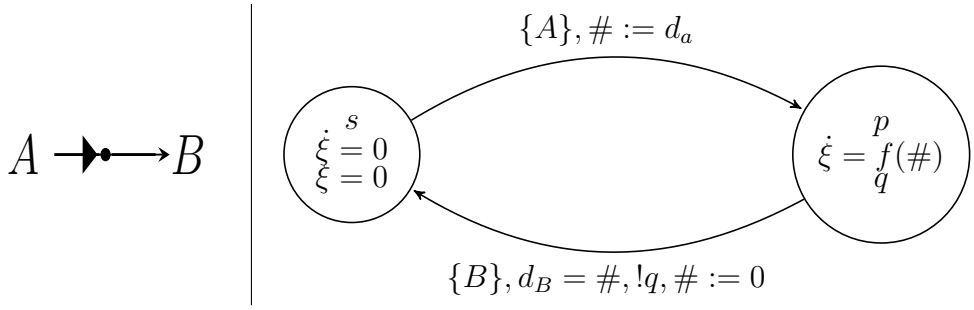

Figura 4.5: Canal Reo para o timedTransform e seu HCA

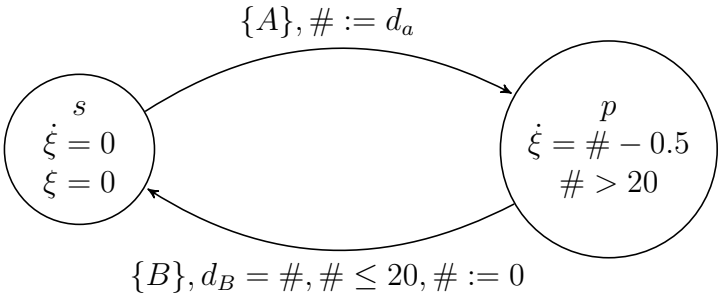

Figura 4.6: HCA do timedTransform para simular um termômetro

4.2 HCA no nuXmv

Hybrid Constraint Automata são modelados no nuXmv seguindo uma representação direta onde um HCA será um **MODULE** no modelo, onde seus estados, transições e sistema dinâmico estarão representados. O Algoritmo 1 constrói essa conversão de um HCA para um **MODULE** nuXmv.

Para isso, é preciso garantir que a semântica do modelo será equivalente ao autômato. Portanto, é necessário que o modelo tenha uma representação dos estados, de suas transições e de sua entrada. Com relação a saída do algoritmo, o **`MODULE`** gerado tem um parâmetro `time`, essa variável é usada para sincronizar todos os conectores modelados.

Invariante 1: Estados É importante distinguir a diferença entre um estado do nuXmv e o do autômato. Um estado no nuXmv representa um estado da verificação, ou seja, o valor de todas as variáveis do modelo num instante. Enquanto que o estado do autômato está representado pela variável *cs* do modelo, e representa uma configuração do conector. É nessa variável que se estará representado o estado do autômato que está sendo verificado.

Inicialização `cs` será inicializado com os valores dos estados iniciais do autômato (linha 3).

Algorithm 1: Geração modelo nuXmv de um autômato

Data: $A = (Q, \mathcal{N}ames, \rightarrow, Q_0, \Sigma_s, re_{\rightarrow})$
Result: MODULE A(time)

1 **begin**
2 VAR $\leftarrow$ cs : Q; ports : portsModule; var: real; data: {NULL, 0, 1};
3 ASSIGN $\leftarrow$ init(cs) := Q_0; init(var) := 0; init(data) := NULL;
4 TRANS $\leftarrow \emptyset$;
5 **foreach** $q \in Q$ **do**
6 $Q_{inalc}(q) = Q \setminus \{q\}$;
7 **foreach** $q \xrightarrow{N,g,\varphi} p \in \rightarrow$ **do**
8 TRANS $\leftarrow$ TRANS $\cup \{(cs = q) \wedge (ports[N] = g) \wedge (ports[\mathcal{N}ames \setminus N] =$ NULL $\wedge\ \varphi \wedge var = re_{q \xrightarrow{N,g,\varphi} p}) \rightarrow \text{next}(cs) = p\}$;
9 $Q_{inalc}(q) = Q_{inalc}(q) \setminus \{p\}$;
10 **end foreach**
11 TRANS $\leftarrow$ TRANS $\cup \{cs = q \rightarrow \text{next}(cs) \neq Q_{inalc}(q)\}$;
12 TRANS $\leftarrow$ TRANS $\cup \{cs = q \wedge In_q \rightarrow var = f_q\}$;
13 **end foreach**
14 **end**

Manutenção os possíveis valores de `cs` são iguais ao conjunto de estados do autômato (linha 2), ou seja, todo estado representado terá uma correspondência no autômato e todo estado do autômato terá uma representação.

Finalização Os estados são finitos pois seus valores são iguais aos estados do autômato.

entrada A entrada do autômato está representada por um outro módulo no nuXmv, o `portsModule`, explicado na seção 4.4, que está sendo instanciado na variável `ports` (linha 2).

Invariante 2: Transições Todas as transições do autômato têm que estar representadas no modelo, e, além disso, é preciso dizer os estados inalcançáveis a partir de cada estado do autômato.

Inicialização Para uma transição estar representada é necessário dizer que o estado atual é o estado de início da transição (`cs=q`), qual a condição da transição (`ports[N]=g`), que as portas que não participam devem estar vazias ($ports[\mathcal{N}ames \setminus N] =$ `NULL`), que a condição do sistema dinâmico vale (φ), qual o valor vai ser inicializado na variável do sistema dinâmico ($var = re_{q \xrightarrow{N,g,\varphi}}$) e qual o estado de destino (`next(cs)=p`). A linha 8 então diz: se o estado atual é q, a entrada respeita a condição g, as portas que não estão em N estão

vazias e a variável respeita a condição do sistema dinâmico então a variável vai ser inicializada conforme a transição e o próximo estado é p.

Para representar os estados inalcançáveis, é preciso primeiro saber quais são eles para cada estado do autômato. Para isso, cada estado destino de uma transição é removido do conjunto de estados inalcançáveis de um estado (linha 9), também é removido o próprio estado (linha 6). Depois, é preciso dizer no modelo que se o estado atual é q então o próximo estado não deve pertencer ao conjunto de estados inalcançáveis de q (linha 11).

Manutenção Uma transição só estará representada no modelo se ela existir no automato, e todas as transições representadas no modelo existem no automato.

Finalização O número de transições é limitado pela quantidade de transições do autômato.

Invariante 3: Sistema dinâmico O sistema dinâmico deve estar representado no modelo, e além disso suas invariantes devem ser respeitadas.

Inicialização Para representar o sistema dinâmico no modelo, é necessário representá-lo para cada estado do autômato, qual é sua invariante e como a variável se comporta nele. Portanto se o autômato está no estado q (linha 12), então o invariante do sistema em q deve valer (In_q), e o próximo valor da variável vai respeitar a função do sistema em q (f_q).

Manutenção O sistema dinâmico do autômato estará representado para cada um de seus estados e o sistema representado será dado pelo autômato.

Finalização A representação do sistema dinâmico é dado para cada estado do autômato.

O modelo nuXmv do timer visto na Figura 4.1 é apresentado no Algoritmo 4.1, na linha 3 a variável `cs` é declarada com dois possíveis valores `q0 e p0`, onde `q0` é o estado inicial e `p0` é o estado onde o dado está armazenado. A linha 4 declara a variável `var`, que vai ser a variável do sistema dinâmico, onde vai contar o tempo decorrido, na linha 5, `data` é declarado e é onde o dado lido vai ser armazenado. A linha 11 apresenta a transição onde o dado é recebido, então o estado deve ser `q0`, a porta de escrita deve estar vazia (`ports.b[time] = NULL`) e a porta de leitura deve ter algum dado (`ports.a[time] != NULL`), além disso, o dado nessa porta vai ser armazenado (`(next(data) = ports.a[time])`) e a variável de contagem vai receber o tempo determinado (`next(var) = 5`), então, se e somente se essas condições são verdadeiras,

então o estado atual vai para o estado de dado armazenado (`next(cs) = p0`). A linha 12 apresenta o dado sendo escrito quando o tempo determinado decorreu, para isso, o estado atual deve ser o de dado armazenado (`cs = p0`), a porta de escrita deve estar vazia (`ports.a[time] = NULL`), a porta de escrita deve ter o mesmo dado que foi armazenado (`ports.b[time] = data`) e o contador deve estar zerado (`var = 0`), além disso, o dado armazenado vai ser resetado (`next(data) = NULL`), e se e somente se essas condições forem verdadeiras, o estado atual vai para o inicial (`next(cs) = q0`). Por último, a linha 13 modela o sistema dinâmico em si, onde o contador vai decrementar em cada ciclo da verificação, então, se o contador e maior que zero (`var > 0`), ele vai decrementar de acordo com a função parametrizada (`next(var) = (var - 1)`).

```
1  MODULE timer(time, ports)
2  VAR
3      cs: {q0,p0};
4      var: real;
5      data: {NULL, 0, 1};
6  ASSIGN
7      init(cs) := {q0};
8      init(var) := 0;
9      init(data) := NULL;
10 TRANS
11     ((cs = q0 & ports.b[time] = NULL & ports.a[time] != NULL & next(
           data) = ports.a[time] & next(var) = 5)) <-> (next(cs) = p0) &
12     ((cs = p0 & ports.a[time] = NULL & ports.b[time] = data & var = 0
            & next(data) = NULL)) <-> (next(cs) = q0) &
13     (var > 0 <-> next(var) = (var - 1));
```

Algoritmo 4.1: **MODULE** nuXmv para o timer

O modelo nuXmv do timedDelay visto na Figura 4.3 é apresentado no Algoritmo 4.2, na linha 3 a variável `cs` é declarada com dois possíveis valores `q0 e p0`, onde `q0` é o estado inicial e `p0` é o estado onde o dado está armazenado. A linha 4 declara a variável `var`, que vai ser a variável do sistema dinâmico, onde vai ser contado o tempo decorrido, na linha 5 `data` é declarado, é onde o dado lido vai ser armazenado. A linha 11 representa a transição onde o dado é recebido, então o estado atual deve ser `q0`, a porta de escrita deve esta vazia (`ports.b[time] = NULL`) e a porta de leitura deve ter algum dado (`ports.a[time] != NULL`), além disso, o dado nessa porta vai ser armazenado (`(next(data) = ports.a[time])`) e a variável de contagem vai ser

zerada (`next(var) = 0`), então, se e somente se essas condições são verdadeiras, então o estado atual vai para o de dado armazenado (`next(cs) = p0`). Lihae 13 apresenta o dado sendo escrito quando o contador ultrapassou o atraso mínimo passado, para isso, o estado atual deve estar no de dado armazenado (`cs = p0`), a porta de leitura deve estar vazia (`ports.a[time] = NULL`), a porta de escrita deve ter o mesmo dado que foi armazenado (`ports.b[time] = data`) e o contador deve estar entre os atrasos mínimo e máximo determinados (`50 < var < 100`), além disso, o dado armazenado vai ser resetado (`next(data) = NULL`), então se e somente se essas condições são verdeiras, o estado atual vai para o estado inicial (`next(cs) = q0`). Por último, a linha 13 modela o sistema dinâmico em si, onde o contador vai incrementar a cada ciclo da verificação, então, se o contador é menor que o atraso máximo (`var < 100`), ele vai incrementar em um (`next(var) = (var + 1)`).

```
1  MODULE timedDelay(time, ports)
2  VAR
3      cs: {q0,p0};
4      var: real;
5      data: {NULL, 0, 1};
6  ASSIGN
7      init(cs) := {q0};
8      init(var) := 0;
9      init(data) := NULL;
10 TRANS
11     ((cs = q0 & ports.b[time] = NULL & ports.a[time] != NULL & next(
           data) = ports.a[time] & next(var) = 0)) <-> (next(cs) = p0) &
12     ((cs = p0 & ports.a[time] = NULL & ports.b[time] = data & 50 <
           var < 100  & next(data) = NULL)) <-> (next(cs) = q0) &
13     (var < 100 <-> next(var) = (var + 1));
```

Algoritmo 4.2: **MODULE** nuXmv para o timedDelay

O Algoritmo 4.3 apresenta o modelo nuXmv do canal timedTransform visto na Figura 4.5, a linha 3 declara a variável `cs` com dois possíveis valores, `q0 and p0`, onde `q0` é o estado incial e `p0` é o estado onde o dado está armazenado. A linha 4 declara a variável `var`, ela representa a variável do sistema dinâmico que vai sofrer transformações. A linha 9 apresenta a transição onde o dado é lido, então o estado atual deve ser o inicial (`cs = q0`), a porta de escrita deve estar vazia (`ports.b[time] = NULL`) e a porta de leitura deve ter algum dado (texttttports.a[time] != NULL), além disso, esse dado vai ser

armazenado (`next(var) = ports.a[time]`), então se e somente se essas condições forem verdadeiras, o estado atual vai mudar para o estado de dado lido (`next(cs) = p0`). Na linha 10 tem a transição onde o dado é escrito após ter sido transformado para que a condição passada seja verdadeira, para isso, o estado atual deve ser o estado de dado lido (`cs = p0`), a porta de leitura deve estar vazia (`ports.a[time] = NULL`), a porta de escrita deve ter o mesmo valor do dado transformado (`ports.b[time] = var`), a condição determinada deve ser verdadeira (`var < 1`) e o dado armazenado vai ser zerado (`next(var) = 0`), então se e somente se essas condições forem verdadeiras, o estudo atual vai para o estado inicial (`next(cs) = q0`). Por último, a linha 11 modela o sistema dinâmico, que vai transformar o dado armazenado em cada instante da verificação, então se a variável é maior que zero (`var > 0`), ela vai ser transformado de acordo com a função passada (`next(var) = (var - var*0.75)`).

```
1  MODULE timedTransformer(time, ports)
2  VAR
3      cs: {q0,p0};
4      var: real;
5  ASSIGN
6      init(cs) := {q0};
7      init(var) := 0;
8  TRANS
9      ((cs = q0 & ports.b[time] = NULL & ports.a[time] != NULL & next(
           var) = ports.a[time])) <-> (next(cs) = p0) &
10     ((cs = p0 & ports.a[time] = NULL & ports.b[time] = var & var <1 &
            next(var) = 0)) <-> (next(cs) = q0 ) &
11     (var > 0 <-> next(var) = (var - var*0.75));
```

Algoritmo 4.3: **MODULE** nuXmv for timedTransform

4.3 Product Automata

Também é criado um modelo para o autômato gerado pela operação de produto definida na Definição 2.5.11 adaptado ao HCA. Esse modelo vai usar os **MODULEs** dos autômatos que compõem a operação de composição.

O Algoritmo 2 recebe dois autômatos já modelados no nuXmv e gera os conjuntos necessários para criar o modelo do produto deles, seguindo as regras definidas por [3]. As linhas 2 a 6 do algoritmo inicializam os conjuntos de saída com conjuntos vazios.

Algorithm 2: Geração do modelo do produto a partir de dois autômatos

Data: $\mathcal{A}_1 = (Q_1, \mathcal{N}ames_1, \rightarrow_1, Q_{0,1}, \Sigma_{1_s}, re_{1\rightarrow})$ and
$\mathcal{A}_2 = (Q_2, \mathcal{N}ames_2, \rightarrow_2, Q_{0,2}, \Sigma_{2_s}, re_{2\rightarrow})$

Result: Q_p: Estados do resultado do produto; $Q_{0,p}$: Estados iniciais do autômato do produto; $Trans_p$: Novas transições criadas seguindo as regras de transições do produto; $Q_{inalc}(q_1q_2)$: Estados inalcançáveis para cada estado q_1q_2 do autômato do produto; Σ_s: Sistema dinâmico do produto para cada estado; $re_{\rightarrow}$: funções de reset para cada nova transição do produto;

1 **begin**
2 $\quad Q_p \leftarrow \emptyset$;
3 $\quad Q_{0,p} \leftarrow \emptyset$;
4 $\quad Trasn_p \leftarrow \emptyset$;
5 $\quad \Sigma_s \leftarrow \emptyset$;
6 $\quad re_{\rightarrow} \leftarrow \emptyset$;
7 $\quad$ **foreach** $q_1 \in Q_1$ **do**
8 $\quad\quad$ **foreach** $q_2 \in Q_2$ **do**
9 $\quad\quad\quad Q_p \leftarrow Q_p \cup \{q_1q_2\}$;
10 $\quad\quad\quad \Sigma_s \leftarrow \Sigma_s \cup \{\Sigma_1 q_1 \cup \Sigma_2 q_2\}$;
11 $\quad\quad\quad$ **if** $(q_1 \in Q_{0,1}) \wedge (q_2 \in Q_{0,2})$ **then** $Q_{0,p} \leftarrow Q_{0,p} \cup \{q_1q_2\}$;
12 $\quad\quad$ **end foreach**
13 $\quad$ **end foreach**
14 $\quad$ **foreach** $q_1q_2 \in Q_p$ **do**
15 $\quad\quad Q_{inalc}(q_1q_2) \leftarrow Q_p \setminus \{q_1q_2\}$;
16 $\quad\quad$ **foreach** $q_1 \xrightarrow{N_1,g_1,\varphi_1} p_1 \in \rightarrow_1$ **do**
17 $\quad\quad\quad$ **foreach** $q_2 \xrightarrow{N_2,g_2,\varphi_2} p_2 \in \rightarrow_2$ **do**
18 $\quad\quad\quad\quad$ **if** $(N_1 \cap \mathcal{N}ames_2) = (N_2 \cap \mathcal{N}ames_1)$ **then**
19 $\quad\quad\quad\quad\quad Q_{inalc}(q_1q_2) \leftarrow Q_{inalc}(q_1q_2) \setminus \{p_1p_2\}$;
20 $\quad\quad\quad\quad\quad Trans_p \leftarrow Trans_p \cup q_1q_2 \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2, \varphi_1 \wedge \varphi_2} p_1p_2$;
21 $\quad\quad\quad\quad\quad re_{\rightarrow} \leftarrow re_{\rightarrow} \cup \{re_1 \underset{q_1 \xrightarrow{N_1,g_1,\varphi_1} p_1}{} \cup re_2 \underset{q_2 \xrightarrow{N_2,g_2,\varphi_2} p_2}{} \}$
22 $\quad\quad\quad\quad$ **end if**
23 $\quad\quad\quad$ **end foreach**
24 $\quad\quad\quad$ **if** $(N_1 \cap \mathcal{N}ames_2 = \emptyset)$ **then** $Q_{inalc}(q_1q_2) \leftarrow Q_{inalc}(q_1q_2) \setminus \{p_1p_2\}$;
25 $\quad\quad\quad$ **else** $g_1 \leftarrow g_1 \wedge \text{FALSE}$;
26 $\quad\quad$ **end foreach**
27 $\quad\quad$ **foreach** $q_2 \xrightarrow{N_2,g_2} p_2 \in \rightarrow_2$ **do**
28 $\quad\quad\quad$ **if** $(N_2 \cap \mathcal{N}ames_1) = \emptyset)$ **then** $Q_{inalc}(q_1q_2) \leftarrow Q_{inalc}(q_1q_2) \setminus \{q_1p_2\}$;
29 $\quad\quad\quad$ **else** $g_2 \leftarrow g_2 \wedge \text{FALSE}$;
30 $\quad\quad$ **end foreach**
31 $\quad$ **end foreach**
32 **end**

A linha 7 tem um loop que itera sobre os estados do primeiro autômato e a linha 8 itera sobre os estados do segundo autômato. Essas iterações são usadas para gerar os estados do produto que pela regra é dado por $Q_1 \times Q_2$ e a linha 10 gera os sistemas dinâmicos de cada estado do produto. A linha 11 checa se os estados atuais são estados

iniciais em seus respectivos autômatos, se sim, então esse estado do produto também será adicionado ao conjunto de estados iniciais.

A linha 14 tem um loop que itera sobre todos os estados do produto criados, a linha 15 inicializa o conjunto de estados inalcançáveis do estado atual do loop com todos os estados do produto menos o estado atual. A linha 16 é um loop que itera sobre cada transição do autômato $\mathcal{A}_1$ que parta da primeira parte do estado do produto. Similarmente, a linha 17 itera sobre cada transição do autômato $\mathcal{A}_2$ que parta da segunda parte do estado do produto.

Na linha 18 tem a verificação feita para a primeira regra da geração das transições do produto, e se for verdade, a união dos estados finais de ambas as transições vai ser removida do conjunto de estado inalcançáveis do estado atual, linha 19. Na linha 20, segundo a primeira regra, cria uma nova transição que é a união das duas transições e a adiciona ao conjunto de novas transições e a linha 21 gera a função de reset para essa nova transição unindo as duas funções de reset das componentes do produto.

A linha 24 verifica a segunda regra da geração das transições do produto, e se for verdade, então o estado em que essa transição chega unido com a segunda parte do estado atual vai ser removido do conjunto de estados inalcançáveis. Note que a transição não é adicionada ao conjunto de novas transições, isso se deve ao fato que o produto vai usar a transição já modelada no autômato que compõe o produto. Caso essa condição seja falsa, então essa transição vai ser bloqueada no modelo do componente do produto.

A linha 27 itera novamente sobre as transições do segundo autômato $\mathcal{A}_2$ que comecem pela segunda parte do produto; a linha 28 faz a verificação da terceira regra da geração das transições do produto, e similarmente à segunda regra, o estado que essa transição chega unido a primeira parte do estado atual é removida do conjunto de estados inalcançáveis. E caso essa verificação não seja verdadeira, essa transição é bloqueada no componente.

O Algoritmo 2 gera então um conjunto que representa os estados do produto, um conjunto dos estados iniciais do produto, um conjunto de novas transições a ser modeladas no **`MODULE`** do produto, um conjunto de conjuntos, que representa os estados inalcançáveis para cada estado do produto, um conjunto de conjuntos que representa o sistema dinâmico para cada estado do produto e um outro conjunto de conjuntos que representa as função de reset de cada nova transição a ser modelada.

O Algoritmo 3 recebe a saída do Algoritmo 2 e gera o **`MODULE`** do autômato do produto. Para isso, ele também recebe o nome dos **`MODULEs`** de cada um dos componentes

do autômato. Na linha 2 as variáveis são declaradas, `prod1` e `prod2` serão as instâncias dos **`MODULEs`** dos componentes do produto e `cs` receberá os estados gerados do produto Q_p.

A linha 3 inicializa o bloco `ASSIGN` com a inicialização da variável `cs` com o conjunto de estados iniciais $Q_{0,p}$. A linha 4 inicializa o bloco de transições com as novas transições geradas $Trans_p$ e a linha 5 inicializa o bloco de invariantes com o conjunto vazio.

O loop na linha 6 itera sobre cada estado do produto, na linha 7 ele adiciona as transições que representam os estados inalcançáveis de cada estado e na linha 8 ele insere uma invariável que garante a sincronia entre o estado do produto e os estados de seus componentes.

Algorithm 3: Geração modelo nuXmv do produto

Data: Q_p, $Q_{0,p}$, $Trans_p$, $Q_{inalc}(q_1q_2)$, $re_{\rightarrow}$, nome prod1 do **`MODULE`** de $\mathcal{A}_1$ e nome prod2 do **`MODULE`** de $\mathcal{A}_2$

Result: **`MODULE`** prod(time)

1 **begin**
2 VAR $\leftarrow$ prod1: prod1(time); prod2: prod2(time); cs : Q_p; ports : portsModule;
3 ASSIGN $\leftarrow$ init(cs) := $Q_{0,p}$;
4 TRANS $\leftarrow Trans_p \wedge re_{\rightarrow}$;
5 INVAR $\leftarrow \emptyset$;
6 **foreach** $q_1q_2 \in Q_p$ **do**
7 TRANS $\leftarrow$ TRANS $\cup$ $\{cs = q \rightarrow \text{next}(cs) \neq Q_{inalc}(q_1q_2)\}$;
8 INVAR $\leftarrow$ INVAR $\cup$ (prod1.cs = q_1) & (prod2.cs = q_2) $\leftrightarrow$ cs = q_1q_2;
9 **end foreach**
10 **end**

Invariante 1: Estados Os estados do modelo do produto devem ser iguais aos estados definidos pela regra do produto, que diz que os estados do produto serão $Q_1 \times Q_2$, esse conjunto é calculado pelo Algoritmo 2 e é atribuído a variável `cs` do produto.

Inicialização `cs` será inicializado com os valores dos estados iniciais do produto calculados pelo Algoritmo 2 (linha 3).

Manutenção os possíveis valores de `cs` são iguais ao conjunto de estados do produto (linha 2), ou seja, todo estado representado terá uma correspondência no produto e todo estado do produto terá uma representação.

Finalização Os estados são finitos pois seus valores são iguais aos estados do autômato do produto.

Invariante 2: Transições Todas as transições dadas pelas regras de geração de transições do produto do autômato têm que estar representadas no modelo, e, além disso, é preciso dizer os estados inalcançáveis a partir de cada estado do produto.

Inicialização As transições geradas pela primeira regra da geração de transição são geradas pelo Algoritmo 2 e elas são inseridas no bloco de transições na linha 4. Já as transições das regras 2 e 3 não estarão explicitamente representadas no modelo do produto, o produto usará essas transições já modeladas nos seus componentes. Para isso, o bloco `INVAR` garante a sincronia entre o produto e seus componentes (linha 9), ou seja, se uma transição no componente for executada, o estado alterado nesse componente também mudará o estado do produto e se uma transição for executada no produto, o estado alterado mudará o estado do componente. Por isso o Algoritmo 2 bloqueia as transições que não seguem essas regras, para que somente as transições que foram preservadas no produto possam ser executadas. A linha 7 trata de inserir os estados inalcançáveis de cada estado calculados pelo Algoritmo 2 no bloco de transições do produto;

Manutenção Uma transição só estará representada no modelo do produto se ela existir no autômato do produto, e todas as transições representadas no modelo existem no autômato.

Finalização O número de transições é limitado pela quantidade de transições do autômato.

Invariante 3: Sistema dinâmico O sistema dinâmico deve estar representado no modelo.

Inicialização O sistema dinâmico do produto vai usar o sistema já modelado nos componentes, tendo sua sincronia garantida da mesma forma que é garantida nas transições. Para isso, no `MODULE` do produto, as variáveis do sistema dinâmico verificadas vão ser as variáveis instanciadas nos componentes.

Manutenção O sistema dinâmico do autômato do produto será o mesmo de seus componentes.

Finalização A representação do sistema dinâmico é dado para cada estado do autômato de seus componentes.

O Algoritmo 4.4 exemplifica um modelo no nuXmv do produto da Figura 2.7 com seus componentes modelados criado à partir do compilador. O primeiro componente do produto é o **`MODULE`** `lossySync1` na linha 2, podemos observar que ele é o modelo do canal

lossySync, veja que uma de suas transições foi removida, essa é a transição bloqueada no algoritmo do produto; já o segundo componente é o **MODULE** `timer2` na linha 9, um modelo de um canal *timer*. O **MODULE** `finalAutomata` é o modelo do produto em si, nas linhas 23 e 24 ele instancia seus componentes, e na linha 25 ele declara o `cs` com os estados do produto. As linhas 29 e 30 têm as novas transições, observe que as variáveis `var` e `data` estão sendo prefixadas pelo componente de onde elas vêm, dessa forma, o sistema dinâmico usado no produto é o mesmo do componente. A linha 31 apresenta o bloco **INVAR** que sincroniza o produto com seus componentes, assim o produto tem as transições dos componentes já modeladas.

```
1  --Channel from line 3 on the input file
2  MODULE lossySync1(time, ports)
3  VAR
4   cs: {q0};
5  TRANS
6   ((cs = q0 & ports.b[time] = NULL & ports.d[time] = NULL &  ports.a[
       time] != NULL) <-> next(cs) = q0);
7
8  --Channel from line 4 on the input file
9  MODULE timer2(time, ports)
10 VAR
11  cs: {q0,p0};
12  var: real;
13  data: {0,1,NULL};
14 ASSIGN
15  init(cs) := {q0};
16 TRANS
17  ((cs = p0 & ports.a[time] = NULL & ports.b[time] = NULL &  ports.d[
       time] = data & var = 0 & next( data ) = NULL) <-> next(cs) = q0)
18  & ( cs = p0 & var > 0 <-> next( var ) = (var - 1) )
19
20 --Final product, corresponding to the full circuit
21 MODULE finalAutomata(time, ports)
22 VAR
23  prod1: lossySync1(time);
24  prod2: timer2(time);
25  cs: {q0q0,q0p0};
26 ASSIGN
```

```
27   init(cs) := {q0q0};
28 TRANS
29   ((cs = q0p0 & ports.b[time] = NULL &  ports.a[time] != NULL & ports.
        d[time] = data & prod2.var = 0 & next( prod2.data ) = NULL ) <->
        next(cs) = q0q0) &
30   ((cs = q0q0 & ports.d[time] = NULL &  ports.a[time] != NULL & ports.
        a[time] = ports.b[time] & ports.b[time] != NULL & next( prod2.
        data ) = ports.b[time] & next( prod2.var ) = 5 ) <-> next(cs) =
        q0p0);
31 INVAR
32   (((prod1.cs = q0) & (prod2.cs = q0)) <-> (cs = q0q0)) &
33   (((prod1.cs = q0) & (prod2.cs = p0)) <-> (cs = q0p0));
```

Algoritmo 4.4: Modelo do produto da Figura 2.7

Além do produto usando os componentes, o compilador também gera um modelo do produto do autômato que é gerado pela operação de composição. Esse modelo será mais compacto que o modelo usando os componentes, e sua performance é tipicamente melhor em relação ao espaço de estados e ao tempo para fazer a verificação. O Algoritmo 4.5 exemplifica esse modelo, ele modela o mesmo produto que o Algoritmo 4.4 e podemos observar que ele é consideravelmente menor. Este modelo apresenta tantos a novas transições, quanto as transições dos componentes que foram preservadas.

Apesar do modelo compacto representar todos os estados e transições do produto, como no modelo que usa os componentes, caso um erro ocorra durante a verificação, será difícil encontrar qual componente do produto o causou. Enquanto que no modelo que usa componentes, o *tracer* informará diretamente qual o componente que está causando o erro.

```
1  --Final product, corresponding to the full circuit
2  MODULE finalAutomata(time, ports)
3  VAR
4    cs: {q0q0,q0p0};
5    var: real;
6    data: {0,1,NULL};
7  ASSIGN
8    init(cs) := {q0q0};
9  TRANS
10   ((cs = q0p0 & ports.b[time] = NULL &  ports.a[time] != NULL & ports.
```

```
    d[time] = data & var = 0 & next( data ) = NULL )
11 | (cs = q0p0 & ports.a[time] = NULL & ports.b[time] = NULL & ports.
    d[time] = data & var = 0 & next( data ) = NULL )
12 | (cs = q0q0 & ports.b[time] = NULL & ports.d[time] = NULL & ports.
    a[time] != NULL) <-> next(cs) = q0q0) &
13 ((cs = q0p0 & ports.b[time] = NULL & ports.d[time] = NULL & ports.a
    [time] != NULL)
14 | (cs = q0q0 & ports.d[time] = NULL & ports.a[time] != NULL & ports
    .a[time] = ports.b[time] & ports.b[time] != NULL & next( data ) =
     ports.b[time] & next( var ) = 5 ) <-> next(cs) = q0p0)
15 & ( cs = p0 & var > 0 <-> next( var ) = (var - 1) );
```

Algoritmo 4.5: Modelo compacto do produto da Figura 2.7

O compilador usa como entrada um arquivo chamado "input", onde cada canal que compõe o conector é passado numa linha, e as portas usadas pelo canal entre parêntesis. O Algoritmo 4.6 exemplifica esse arquivo para gerar o modelo da Figura 2.7. O compilador fará as duas formas de produto e vai gerar dois arquivos de saída, o "nuxmv", onde estará o modelo usando os componentes, e o "nuxmv2"onde estará o modelo compacto.

```
1 lossySync(a,b)
2 timer(b,d)[5, var - 1;]
```

Algoritmo 4.6: Entrada do compilador para gerar o modelo da Figura 2.7

4.4 Timed Data Streams

Como forma de simular o comportamento de uma TDS, foi criado um **MODULE** chamado `portsModule`, exemplificado no Algoritmo 4.7. Nesse **MODULE** a TDS é discretizada e o valor de dados de uma porta é representado como um vetor, onde o valor de cada posição dele representa o dado na porta num instante de tempo. Caso esse valor seja `NULL`, aquela porta não tem dado naquele instante.

Além disso, uma variável chamada `time` é declarada no **MODULE** `main` do modelo, que é passada para cada outro **MODULE** que modele um canal. O objetivo dessa variável é garantir que todos os canais estejam sincronizados no mesmo instante da TDS. Logo, caso um canal precise verificar o valor de uma porta, por exemplo a porta `a`, ele consultará o valor da porta `a` naquele instante `time` com `ports.a[time]`.

```
1  MODULE portsModule
2  FROZENVAR
3   a : array 0..3 of {NULL, 0, 1};
4   b : array 0..3 of {NULL, 0, 1};
5  ASSIGN
6   init(a[0]) := 1;
7   init(a[1]) := 1;
8   init(a[2]) := 0;
9   init(a[3]) := 1;
10  init(b[0]) := NULL;
11  init(b[1]) := 1;
12  init(b[2]) := 0;
13  init(b[3]) := 0;
14
15 MODULE main
16 VAR
17  time: 0..3;
18  ports: portsModule();
19  finalAutomata: finalAutomata(time, ports);
20 ASSIGN
21  init(time) := 0;
22  next(time) := case
23   time < 3: time + 1;
24   TRUE: time;
25 esac;
```

Algoritmo 4.7: PortsModule para representar a entrada

Os valores das portas podem ser passados para o compilador, assim o `portsModule` atribuirá às portas os valores passados por meio do arquivo "ports.txt", exemplificado no Algoritmo 4.8. Neste arquivo, a primeira linha indica a forma em que as portas estão sendo passadas: list. E cada linha posterior será os valores de uma porta separados por vírgula, onde o primeiro elemento é o nome da porta. Caso esse arquivo não seja passado, o compilador gerará cinco valores para cada porta usada no modelo de forma aleatória.

```
1 list
2 a,1,1,0,1
3 b,NULL,1,0,0
```

Algoritmo 4.8: Arquivo de entrada para passar valores às portas

Chapter 5

Consenso

O Problema dos Generais Bizantinos, apresentado em 1982 [33], é um problema que gera muito interesse de estudo. Imagine diversas divisões do exército Bizantino acampadas em torno de uma cidade inimiga, onde cada divisão é comandada por um general. Esses generais podem comunicar entre si somente por mensageiros, e após observar o inimigo, eles devem decidir um plano de ação comum entre eles. Porém, algum desses generais podem ser traidores, cujo objetivo é prevenir que os generais leais cheguem em um acordo. É necessário que os generais tenham uma protocolo para garantir que:

A. Todos os generais leais decidam pelo mesmo plano de ação;

B. Uma quantidade pequena de traidores não podem fazer com que os generais leais decidam por um plano ruim.

Os generais leais vão todos fazer o que o protocolo mandar, mas traidores podem fazer o que desejarem, mas independente do que os traidores façam, o protocolo deve garantir a condição A. Já a condição B é mais difícil de formalizar, e por simplicidade, não é definido exatamente o que é um plano ruim. Ao invés disso, é visto como os generais alcançam uma decisão, considerando que cada general observa o inimigo e comunica suas observações; cada general vai usar um método para combinar essas observações e decidir em um plano de ação. Então a condição A é alcançada tendo que todos os generais usem o mesmo método para combinar informações, e a condição B é alcançada garantindo que esse método é robusto.

Por exemplo, tomando entre atacar e recuar como únicas decisões, cada general vai comunicar qual decisão é a melhor, e a decisão final pode ser alcançada com um voto majoritário dentre as decisões comunicadas. Dessa forma, uma pequena quantidade de

traidores só podem influenciar a decisão final caso a quantidade de generais leais seja quase a mesma, caso qual nenhuma decisão poderia ser considerada ruim.

O problema dos Generais Bizantinos também é conhecido como o problema do consenso, onde a concordância de um número de agentes é necessária para um dado valor, onde alguns desses agentes podem ser falhos ou não confiáveis, portanto algoritmos de consenso devem ser tolerantes a falhas. Para que o consenso seja alcançado, os agentes devem declarar seus valores, comunicar entre si e chegar em um único valor de consenso.

Definição 5.0.1. *Resolver um problema de consenso implica em garantir que as três propriedades seguintes sejam verdadeiras [21]:*

- *Concordância: dois agentes corretos não podem decidir em diferentes valores;*
- *Terminação: todos agentes corretos eventualmente decidem um valor;*
- *Validade: o valor decidido é um valor proposto por um dos agentes.*

Além disso, é conhecido que o consenso não pode ser alcançado em redes completamente assíncronas [19], ou seja, redes onde não são assumidas o tempo relativo de cada processo ou o atraso para a entrega de uma mensagem, até mesmo quando somente um agente é falho. Portanto, algoritmos que propõem uma solução para esse problema assumem algumas garantias sobre essas redes.

Uma importante aplicação onde é necessário o consenso é na replicação de máquinas estado, que é um método geral para implementar um serviço tolerante a falhas, onde o prestador de serviços (servidor) é copiado em diversas máquinas diferentes (réplicas) por meio de um sistema distribuído tipicamente assíncrono. Nesse sistema, uma requisição do cliente deve ser coordenada entre as diversas réplicas que devem alcançar um consenso para responder à requisição, garantindo que todas as máquinas estão de acordo. Na Figura 5.1 é apresentado um exemplo de sistema de replicação de máquinas de estado, onde cada réplica é capaz de comunicar entre si, com exceção da réplica C que apresenta alguma falha arbitrária (seja maliciosa ou nó defeituoso), ou seja, é uma réplica bizantina (no caso do problema dos Generais Bizantinos, seria um general traidor).

Quando consideramos que nenhuma das réplicas desse sistema distribuído pode ser bizantina, ainda é preciso considerar que o sistema é assíncrono. Nesse cenário, uma importante solução é o algoritmo de Paxos [32].

No algoritmo de Paxos, uma réplica, conhecida como líder, recebe a requisição do cliente e prepara a mensagem a ser distribuída para as outras réplicas atribuindo a ela

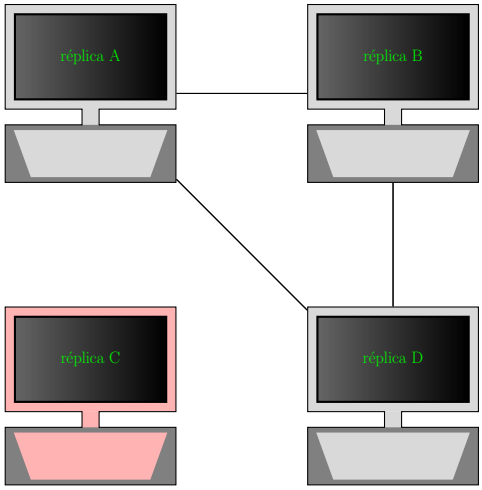

Figura 5.1: Exemplo de um sistema de replicação de máquinas de estado onde uma réplica é bizantina

um índice n maior que qualquer outra mensagem já preparada. Então essa mensagem é enviada às outras réplicas na rede, conhecidas como aceitadoras, que vão decidir se vão prometer responder a essa mensagem caso n seja maior que qualquer índice recebido, caso contrário a mensagem é ignorada. Se o líder receber uma maioria de promessas das outras réplicas, ele vai atribuir um valor de resposta a mensagem e vai reenviar às réplicas, que por sua vez vão armazenar a resposta recebida. Caso o líder não receba uma maioria de promessas, ele vai tentar novamente com um índice n maior. Quando uma réplica aceitadora recebe uma mensagem com valor de resposta, essa mensagem também é enviada a todas as outras réplicas conhecidas como aprendiz (que podem ser iguais a líder) que vão enviar a resposta ao cliente.

No cenário onde devem ser consideradas réplicas bizantinas, um algoritmo de consenso importante nesse cenário foi o PBFT que é capaz de alcançar consenso tolerando falhas bizantinas em sistema assíncrono, para isso o PBFT assume que no máximo $f = \frac{n-1}{3}$ de n réplicas são simultaneamente falhas ou bizantinas, ou seja, o número de réplicas do sistema $n = 3f + 1$. Dessa forma, como a resposta de cada réplica pode ser recebida em seu tempo, não sendo necessariamente simultânea, o algoritmo é capaz de funcionar em sistemas assíncronos como a Internet.

Para um sistema com $3f + 1$ réplicas, onde f é a quantidade máxima de réplicas falhas. Essas réplicas seguem uma sequência de configurações chamadas de *view*, onde uma réplica é primária e as outras são *backups*, a réplica primária de uma *view* é a réplica p tal que $p = v \, mod \, n$, onde v é o índice da *view*. Caso o primária apareça falhar, ocorre uma mudança de *view*. O algoritmo funciona simplificadamente como:

1 O cliente envia a requisição de operação para a réplica primária;

2 O primário envia a requisição para todos os *backups*;

3 As réplicas executam a requisição e enviam a resposta para o cliente;

4 O cliente espera por $f + 1$ respostas iguais de diferentes réplicas, essa resposta vai ser o resultado da operação.

Para que o algoritmo funcione, são impostos dois requerimentos para as réplicas: elas devem ser determinísticas (no sentido que uma operação em um dado estado com a mesma entrada sempre tem o mesmo resultado) e todas devem começar no mesmo estado.

A Figura 5.2 apresenta uma sequência das operações realizadas pela réplicas em uma situação comum onde a réplica 3 é bizantina, $f = 1$ e $n = 4$. Nesta figura, a réplica 0 é a primária e C é o cliente.

1. *request*: O cliente C faz a requisição para a réplica primária 0;
2. *pre-prepare*: A réplica primária atribui um índice n a requisição e envia a todos as réplicas *backup*;
3. *prepare*: Se o *backup* aceita o *pre-prepare* baseado no seu índice n, ele envia *prepare* a todos as outras réplicas;
4. *commit*: Se uma réplica receber $2f$ mensagens de *prepare* de outras réplicas, ela aceita a requisição e envia *commit* a todas as outras réplicas;
5. *reply*: Se uma réplica aceitar $2f + 1$ mensagens de *commit* (incluindo sua própria), então a resposta da requisição *reply* é enviada para o cliente. Se o cliente receber $f + 1$ mensagens de *reply* de diferentes réplicas e com o mesmo resultado, ele aceita essa resposta.

Das diversas tentativas que tentaram melhorar o PBFT, algumas se sobressaem como a *Redundant Byzantine Fault Tolerance* [6], cujo objetivo é aumentar a disponibilidade do sistema e reduzir a superfície de ataque Bizantino adicionando processamento redundante em todas as réplicas. Outro exemplo é o *Spinning Byzantine Fault Tolerance*(SBFT) [40], que adiciona um processamento *round-robin* sobre a seleção de nó primário do PBFT.

Sistemas *blockchain* se assemelham a máquinas de estado replicado, esse sistema é formado por uma cadeia de blocos, onde cada bloco contém um lote de transações e uma

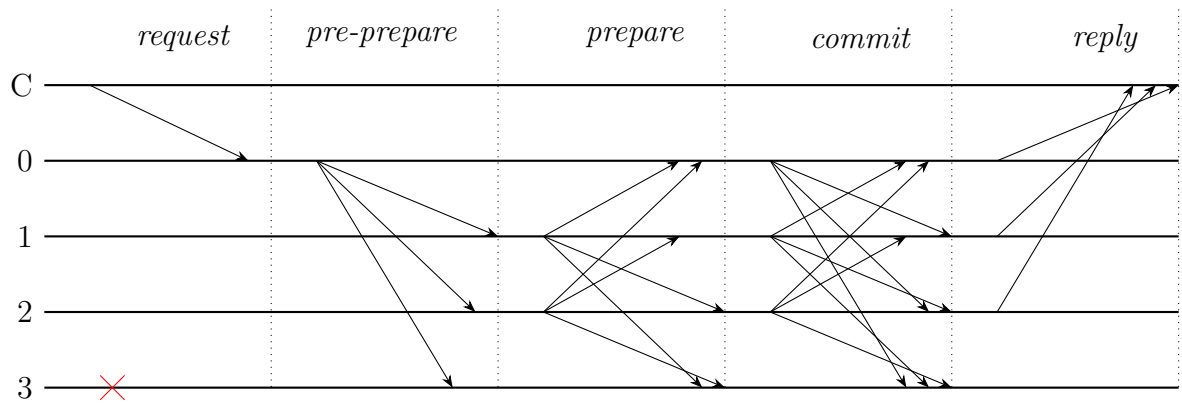

Figura 5.2: Exemplo do caso de operação comum para o PBFT com uma réplica falha

hash criptográfica do bloco anterior, conectando o bloco atual e seu bloco anterior. A cadeia dos blocos e formada por essas conexões recursivas, confirmando a integridade do bloco anterior até o bloco inicial.

Quando um novo bloco está sendo criado, é possível, por conta de diversas características como o atraso da rede, que dois blocos diferentes sejam criados como próximo integrante da cadeia, criando assim uma bifurcação na cadeia conhecida como *fork*. Algoritmos de *blockchain* diferentes tratam esse problema de forma diferente, mas tipicamente a ideia é preservar a cadeia mais relevante usando um sistema de pontuação, onde a cadeia usada é a que obtiver a maior pontuação, e a outra é ignorada. A Figura 5.3 ilustra um exemplo de uma cadeia, onde a pontuação é dada pela bloco de maior altura, dessa forma, os blocos marcados em vermelho são ignorados.

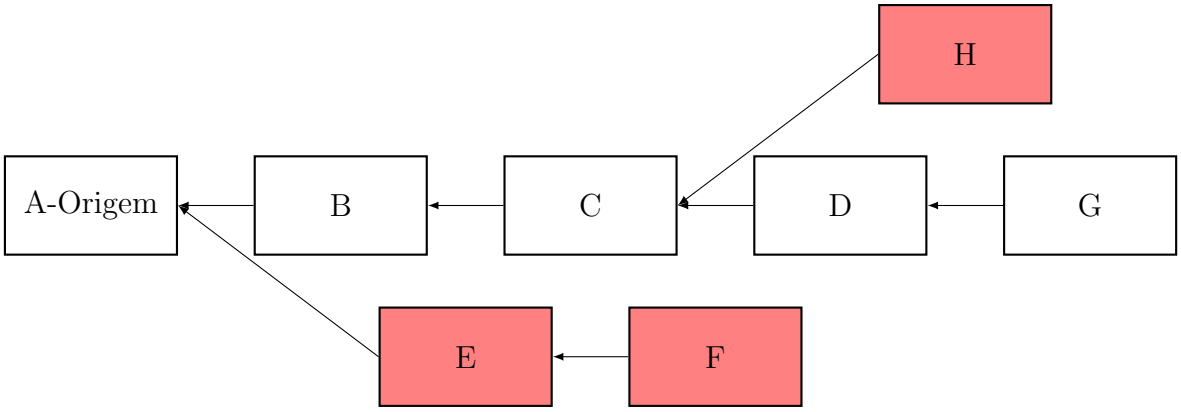

Figura 5.3: Uma *blockchain* onde os blocos em vermelho foram gerados de *forks* ignorados

Sistemas *blockchain* também buscam resolver o problema de consenso, para que dado um índice de bloco, todos os processos devem concordar com o mesmo bloco nesse índice [21].

Definição 5.0.2. *O Consenso Bizantino em* Blockchain *implica em garantir que as três*

propriedades seguintes sejam verdadeiras para um dado índice:

- *Concordância: dois processos corretos não podem decidir em diferentes blocos;*
- *Terminação: todos os processos corretos eventualmente decidem um bloco;*
- *Validade: o bloco decidido é válido, se ele satisfaz um predicado* valid *pré-definido.*

A diferença entre o problema de consenso no *blockchain* está na propriedade de validade, onde ele faz uso da validação inerente em um sistema *blockchain*. O predicado *valid* garante que um bloco possa ser executado, e assumindo que um processo só possa propor um bloco válido, então o predicado *valid* deve valer no bloco decidido.

É importante ressaltar que o predicado *valid*, é tipicamente não trivial. Pois para um dado índice, podem ser forjados diversos blocos com conteúdo diferente, logo o consenso deve ser dado para um bloco específico para aquele índice. Mas como as possíveis transações que compõem um bloco são naturalmente não-determinísticas, o predicado deve lidar com o fato que talvez alguma informação legítima do bloco não esteja presente (tendo que atualizar o bloco para tê-la), ou que ela nem exista (como no ataque de um nó bizantino, onde é usado uma transação falsa). Portanto a adição desse predicado complica consideravelmente o problema do consenso nesse cenário.

Um algoritmo que é aplicado em sistemas *blockchain* baseado em PBFT que herdou a característica do SBFT é o *Delegated Byzantine Fault Tolerance* (dBFT) [26] aplicado em 2015 no Neo Blockchain, para alcançar consenso em um sistema blockchain de estado-global e posteriormente o dBFT 2.0, onde é adicionada uma fase a mais ao algoritmo para resolver um problema de *forks* de um único bloco encontrado na versão anterior.

Uma característica compartilhada entre sistemas de consenso modernos inspirados pelo PBFT [17] é tomar decisões baseadas em um mecanismo de votação, que por simplicidade é nomeado nesse trabalho como Quorum Incerto:

Definição 5.0.3. *O Quorum Incerto é um mecanismo de votação onde* $2f + 1$ *réplicas determinísticas dentre as* $n = 3f + 1$ *réplicas, onde* f *é o número de réplicas consideradas não confiáveis, entram em acordo para uma determinada informação oriunda de uma rede de troca de mensagens não-determinística, como por exemplo em canais* peer-to-peer *usados em projetos de* blockchains *públicos.*

O dBFT 2.0 é um algoritmo de três fases, onde uma réplica vai ser escolhida como a primária usando uma estratégia *round-robin* (onde a réplica primária vai alternando pelo

índice da operação atual), sendo responsável por propor o pacote de informações (Fase I), enquanto que as outras réplicas (conhecidas como *backup*) vão validar tal informação (Fase II), e por último, farão o *commit* dessa informação, assim que $2f + 1$ respostas válidas existem (Fase III). Após o *commit*, a réplica é incapaz de reverter sua própria decisão, então a lógica do algoritmo garante que após uma decisão de *commit* local, sendo feita por uma réplica honesta, vai representar o *commit* global após passar pelo limiar de $2f + 1$ do quorum incerto. Se uma quantidade de tempo suficiente passa sem que o estado do sistema mude (assumindo que a réplica primária possa ter falhado), então uma condição é acionada que vai resetar todo o processo e recomeçar com o próxima réplica primária (chamado *view change*).

Enquanto consideramos incerteza sobre a camada de comunicação *peer-to-peer*, este Quorum Incerto é o mecanismo central para algoritmos modernos inspirados pelo PBFT, o que motiva esse trabalho para a modelagem do Quorum Incerto por meio de Reo estendido para sistemas híbridos.

Chapter 6

Modelagem do mecanismo de consenso

Neste capítulo é exemplificado a modelagem do Quorum Incerto por meio de Reo alcançada por este trabalho, a Figura 6.1 apresenta o circuito Reo para uma réplica do algoritmo estado da arte dBFT 2.0 [26]. Ele é um algoritmo *round-robin* inspirado no PBFT que usa replicação de máquinas de estado, com $n = 3f + 1$ réplicas, onde no máximo f réplicas podem ser falhas ou apresenta comportamento Bizantino. Devido ao atraso inerente de das redes, é necessário um Quorum Incerto de $2f + 1$ réplicas para decidir sobre a validade de uma informação de entrada.

No circuito Reo da Figura 6.1, nó A' representa a entrada do circuito, e então, em um esquema de *round-robin*, vai ser filtrado entre o nó primário P ou o nó de *backup* B. Se está no nó P, após um determinando tempo, as transações são liberadas para o nó de *request* R. No nó de *request*, têm-se dois nós, um que vai esgotar após um determinado tempo, enviando a réplica para o nó de *change view* V'; o outro nó, se a condição para a validade do bloco for válida, então vai mover para o nó de *commit* C' (observe o asyncDrain garantido que não seja possível estar em V' e C' ao mesmo tempo). Se é uma réplica de *backup*, então está no nó B, que após um determinado tempo vai para o nó de *change view* V' (assumindo que a réplica primária falhou).

O nó R' representa as outras réplicas na rede, onde o seu dado vai ser as solicitações liberadas pelas outras réplicas de consenso, onde elas chegam com um atraso limitado não determinístico. Usando essas solicitações, a réplica de *backup*, vai decidir pelo *commit* indo para o nó C' ou se vei esgotar o tempo e ir para o nó de *change view* V'.

Analisando em termos da réplica de consenso, A' e R' são entradas, onde A' vem da entrada da rede e R' é a entrada que vem do nó R das outras réplicas de dentro da rede. Nós V' e C' são saídas no sentido que esses nós vão transmitir para todas os outras réplicas

da rede que vai fazer *commit* (C') ou vai para *change view* (V'). Note que nesse exemplo, por simplicidade, não foi modelada como é feita a comunicação entre as diferentes réplicas da rede.

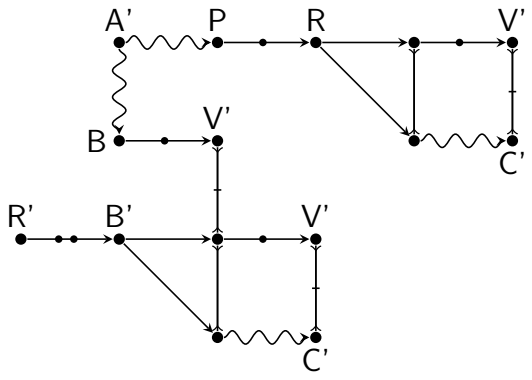

Figura 6.1: Modelo Reo para uma réplica de consenso do dBFT 2.0

Uma parte da comunicação entre as diferentes réplicas de consenso na rede para que o consenso seja alcançado é modelado na Figura 6.2 para uma rede de quatro nós, representando quatro réplicas ($n = 3f+1 = 4$, com $f = 1$). Nesse exemplo, são modeladas as solicitações que a réplica A deve receber da rede para que seja decidido o `commit` (Fase III), onde ela deve receber duas outras respostas (para alcançar o limiar do quorum incerto de $2f+1 = 3$, com $f = 1$). Também é modelado uma réplica falha C onde sua solicitação pode ser perdida (simulando comportamento Bizantino). O estado de *commit* Ac ainda é alcançável independentemente de qualquer falha na réplica C, enquanto que o estado Av indica que a réplica a decidiu por *view change* por conta de um atraso muito grande na rede.

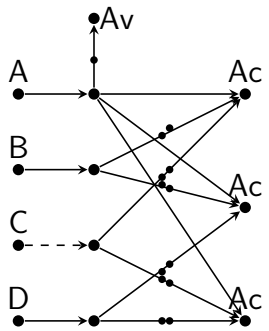

Figura 6.2: Quorum incerto de $2f + 1$ para o dBFT 2.0 (com $f = 1$), onde a réplica C é falha e o estado Ac ainda é alcançável por A, B e D.

O exemplo da Figura 6.2 foi modelado no nuXmv usando o compilador para gerar tanto o modelo com componentes quanto o compacto usando o arquivo 6.1 como entrada. No modelo de componentes foram gerados foram gerados 29 `MODULE`s, dos quais 14 são

referentes aos canais básicos e 15 são resultado da composição dois a dois, totalizando 1424 linhas, em comparação o modelo compacto teve 733 linhas ao todo, ambos os modelos podem ser encontrados em https://github.com/frame-lab/ReoXplore/tree/hybridAutomata/Examples.

```
1  main() {
2   //Channels
3   sync(a, b)
4   sync(c, d)
5   lossysync(e, f)
6   sync(g, h)
7   timer(b, j)[15, var - 1;]
8   timeddelay(f, t)[5, 10;]
9   timeddelay(d, t)[8, 14;]
10  sync(b, t)
11  timeddelay(h, z)[4, 12;]
12  timeddelay(d, z)[5, 10;]
13  sync(b, z)
14  timeddelay(h, i)[5, 14;]
15  timeddelay(f, i)[2, 8;]
16  sync(b, i)
17 }
```

Algoritmo 6.1: Entrada do compilador para gerar o modelo da Figura 6.2

Buscando verificar que é possível que as réplicas decidam pelo *commit* antes do *timeout* da votação foi criada uma propriedade em CTL para ser verificada. Para isso, a propriedade pode ser dividida em duas partes, uma que diz que foi decidido pelo *commit* e outra que diz que *timeout* não ocorreu.

Olhando para o autômato, para representar que o *timeout* não ocorreu, significa que o *timer* que leva ao *view change* ainda está com seu dado armazenado, ou seja, sua variável ainda está contando. Para isso, o autômato final, aquele que representa o produto como um todo, deve estar no estado onde o *timer* ainda está contando: `finalAutomata.cs = q0q0q0q0p0q0q0q0q0q0q0q0q0q0`.

Para a última parte, basta dizer que as portas que representam o *commit* iniciam vazias, e pelo menos alguma delas vai receber algum dado. Para representar que iniciam vazias, essas portas no instante inicial devem estar vazias: `ports.t[0] = NULL & ports.z[0] = NULL & ports.i[0] = NULL`. Para representar que pelo menos

alguma delas vai ter dado, foi feita uma disjunção que diz que em algum momento essas portas vai ter algum dado: `(EX(ports.t[15] = 1 | ports.t[15] = 0)) | (EX(ports.z[15] = 1 | ports.z[15] = 0)) | (EX(ports.i[15] = 1 | ports.i[15] = 0))`.

Para ter a propriedade como um todo, basta juntar todas as partes com conjunções como no Algoritmo 6.2. Resumindo essa propriedade nos diz que o autômato está no estado onde o *timer* para o *view change* ainda está contando, ou seja, ele não chegou no estado *Av*. As portas do *commit* `t, z` e `i` iniciam vazias mas em algum momento, pelo menos uma delas vai ter dado, representando que alcançou o estado de *commit Ac*.

```
1 CTLSPEC finalAutomata.cs = q0q0q0q0p0q0q0q0q0q0q0q0q0q0q0 & ports.t[0]
    = NULL & ports.z[0] = NULL & ports.i[0] = NULL & ((EX(ports.t[15]
    = 1 | ports.t[15] = 0)) | (EX(ports.z[15] = 1 | ports.z[15] = 0))
    | (EX(ports.i[15] = 1 | ports.i[15] = 0)));
```

Algoritmo 6.2: CTLSpec verificada para o quorum mínimo.

Chapter 7

Conclusões e trabalhos futuros

Neste trabalho foram formalizados três novos canais Reo para que sua expressividade possa ser estendida para melhor capturar o comportamento de sistemas dinâmicos, são eles o *timer*, que funciona como um contador; o *timedDelay*, que representa a comunicação com atraso não determinístico de uma rede; e o *timedTransorm*, onde o dado é transformado com o passar do tempo. Foi usado HCA como semântica formal para definir esses componentes e estender nosso trabalho anterior, modelando eles no verificador de modelos nuXmv.

Também foi estendido nosso compilador para lidar com essa nova formalização. O compilador agora apoia todos os canais canônicos e os três novos canais apresentados em HCA. Ele é capaz de automaticamente modelar o produto desses canais criando modelos pré e pós-processados. A versão pré-processada oferece um modelo nuXmv menor (onde só o autômato final é convertido para o nuXmv); já o pós-processado cria um modelo detalhado com todos os autômatos e composições modeladas, permitindo uma melhor rastreabilidade.

Por último, foi apresentada a motivação desse trabalho modelando uma réplica no algoritmo de consenso estado da arte dBFT 2.0 em Reo e também foi apresentado um modelo Reo para o quorum incerto de uma réplica em uma rede de quatro réplicas com uma bizantina. Esse modelo foi passado para o compilador, que gerou o modelo nuXmv onde foi verificada a condição apresentada.

7.1 Trabalhos Futuros

Para o compilador, uma melhoria seria implementar uma ferramenta que possa mapear o modelo Reo a cada autômato para facilitar a construção das verificações, podendo até ser capaz de mapear o autômato para o modelo nuXmv também. Além disso, o nuXmv apresentou alguns desafios na modelagem do HCA, como por exemplo em tratar o tempo real, então é interessante explorar modelar o HCA em outros verificadores como o UPPAL.

Em termos da análise de consenso, os modelos apresentados foram pouco usados, então é interessante investigar integrações e possíveis falhas desses componentes, para explorar as propriedades do dBFT 2.0, e até possivelmente ataques Bizantinos desconhecidos. Uma outra parte que não foi explorada foi em como as diversas réplicas da rede interagem entre si, e como essa comunicação pode ser modelada.

Referências

[1] Aho, A. V., Ullman, J. D. *Foundations of computer science.* Computer Science Press, Inc., 1992.

[2] Alaña, E., Naranjo, H., Yushtein, Y., Bozzano, M., Cimatti, A., Gario, M., De Ferluc, R., Gérald, G. Automated generation of fdir for the compass integrated toolset (autogef). *European Space Agency, (Special Publication) ESA SP 701* (01 2012).

[3] Arbab, F. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science 14*, 3 (2004), 329–366.

[4] Arbab, F. Coordination for component composition. *Electronic Notes in Theoretical Computer Science 160* (2006), 15 – 40. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).

[5] Atkinson, C., Kuhne, T. Model-driven development: a metamodeling foundation. *IEEE software 20*, 5 (2003), 36–41.

[6] Aublin, P.-L., Mokhtar, S. B., Quéma, V. Rbft: Redundant byzantine fault tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems* (2013), IEEE, p. 297–306.

[7] Baier, C., Katoen, J.-P. *Principles of Model Checking.* The MIT Press, 2008.

[8] Baier, C., Sirjani, M., Arbab, F., Rutten, J. Modeling component connectors in reo by constraint automata. *Science of computer programming 61*, 2 (2006), 75–113.

[9] Bochot, T., Virelizier, P., Waeselynck, H., Wiels, V. Model checking flight control systems: The airbus experience. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on* (2009), IEEE, p. 18–27.

[10] Castro, M., Liskov, B., others. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, p. 173–186.

[11] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S. The nuxmv symbolic model checker. In *CAV* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, p. 334–342.

[12] Cavada, R., Cimatti, A., Mariotti, A., Mattarei, C., Micheli, A., Mover, S., Pensallorto, M., Roveri, M., Susi, A., Tonetta, S. Supporting

requirements validation: The eurailcheck tool. In *2009 IEEE/ACM International Conference on Automated Software Engineering* (Nov 2009), p. 665–667.

[13] Chellas, B. F. *Modal Logic: An Introduction*. Cambridge University Press, 1980.

[14] Chen, X., Sun, J., Sun, M. A hybrid model of connectors in cyber-physical systems. In *Formal Methods and Software Engineering* (Cham, 2014), S. Merz and J. Pang, Eds., Springer International Publishing, p. 59–74.

[15] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification* (Berlin, Heidelberg, 2002), E. Brinksma and K. G. Larsen, Eds., Springer Berlin Heidelberg, p. 359–364.

[16] Clarke, E., Biere, A., Raimi, R., Zhu, Y. Bounded model checking using satisfiability solving. *Formal Methods in System Design 19*, 1 (Jul 2001), 7–34.

[17] Coelho, I. M., Coelho, V. N., Araujo, R. P., Yong Qiang, W., Rhodes, B. D. Challenges of pbft-inspired consensus for blockchain and enhancements over neo dbft. *Future Internet 12*, 8 (2020).

[18] David, A., Larsen, K. A tutorial on uppaal 4.0.

[19] Fischer, M. J., Lynch, N. A., Paterson, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (abril de 1985), 374–382.

[20] Gerhart, S., Craigen, D., Ralston, T. Case study: Paris metro signaling system. *IEEE Software 11*, 1 (1994), 32–28.

[21] Gramoli, V. From blockchain consensus back to byzantine consensus. *Future Generation Computer Systems 107* (09 2017).

[22] Grilo, E. Compiling certifieed reo code. Universidade Federal Fluminense, 2018.

[23] Grilo, E., Toledo, D., Lopes, B. A logical framework to reason about reo circuits. In *Journal of Applied Logics - IfCoLog Journal of Logics and their Applications (FLAP)* (in press).

[24] Groote, J. F., Mousavi, M. R. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[25] Henzinger, T. A., Ho, P.-H., Wong-Toi, H. Hytech: A model checker for hybrid systems. In *Computer Aided Verification* (Berlin, Heidelberg, 1997), O. Grumberg, Ed., Springer Berlin Heidelberg, p. 460–463.

[26] Hongfei, Da and Zhang, Erik. Neo: A distributed network for the smart economy. Relatório Técnico, NEO Foundation, 2015.

[27] Huth, M., Ryan, M. *Logic in Computer Science: Modelling and Reasoning about Systems*, 2 ed. Cambridge University Press, 2004.

[28] Jongmans, S.-S. T., Arbab, F. Overview of thirty semantic formalisms for reo. *Scientific Annals of Computer Science 22*, 1 (2012).

[29] Klein, J., Klüppelholz, S., Stam, A., Baier, C. Hierarchical modeling and formal verification. an industrial case study using reo and vereofy. In *International Workshop on Formal Methods for Industrial Critical Systems* (2011), Springer, p. 228–243.

[30] Kokash, N., Krause, C., De Vink, E. Reo+ mcrl2: A framework for model-checking dataflow in service compositions. *Formal Aspects of Computing 24*, 2 (2012), 187–216.

[31] Kwiatkowska, M., Norman, G., Parker, D. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)* (2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *LNCS*, Springer, p. 585–591.

[32] Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (maio de 1998), 133–169.

[33] Lamport, L., Shostak, R., Pease, M. The byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (julho de 1982), 382–401.

[34] Li, Y., Zhang, X., Ji, Y., Sun, M. A formal framework capturing real-time and stochastic behavior in connectors. *Science of Computer Programming* (2019).

[35] Mousavi, M. R., Sirjani, M., Arbab, F. Formal semantics and analysis of component connectors in reo. *Electronic Notes in Theoretical Computer Science 154*, 1 (2006), 83–99.

[36] Nazário Coelho, V., Pereira Araújo, R., Gambini Santos, H., Yong Qiang, W., Machado Coelho, I. A milp model for a byzantine fault tolerant blockchain consensus. *Future Internet 12*, 11 (2020).

[37] Papazoglou, M. P. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on* (2003), IEEE, p. 3–12.

[38] Platzer, A. Differential dynamic logic for hybrid systems. *J. Autom. Reas. 41*, 2 (2008), 143–189.

[39] Pourvatan, B., Sirjani, M., Hojjat, H., Arbab, F. Automated analysis of reo circuits using symbolic execution. *Electronic Notes in Theoretical Computer Science 255* (2009), 137–158.

[40] Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems* (2009), IEEE, p. 135–144.