



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

UnB-DALi: Biblioteca Para Transformação De Modelos Em Análise De Dependabilidade

Abílio Esteves Calegário de Oliveira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.^a Dr.^a Genaína Nunes Rodrigues

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Homero Luiz Piccolo

Banca examinadora composta por:

Prof.^a Dr.^a Genaína Nunes Rodrigues (Orientadora) — CIC/UnB
Prof. Dr. Flávio Leonardo Cavalcanti de Moura — CIC/UnB
Prof. Dr. Vander Ramos Alves — CIC/UnB

CIP — Catalogação Internacional na Publicação

Esteves Calegário de Oliveira, Abílio.

UnB-DALi: Biblioteca Para Transformação De Modelos Em Análise De Dependabilidade / Abílio Esteves Calegário de Oliveira. Brasília : UnB, 2016.

95 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. dependabilidade, 2. transformação de modelos, 3. verificação de modelos, 4. AGG, 5. UML, 6. PRISM

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília—DF — Brasil



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

UnB-DALi: Biblioteca Para Transformação De Modelos Em Análise De Dependabilidade

Abílio Esteves Calegário de Oliveira

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.^a Dr.^a Genaína Nunes Rodrigues (Orientadora)
CIC/UnB

Prof. Dr. Flávio Leonardo Cavalcanti de Moura Prof. Dr. Vander Ramos Alves
CIC/UnB CIC/UnB

Prof. Dr. Homero Luiz Piccolo
Coordenador do Bacharelado em Ciência da Computação

Brasília, de fevereiro de 2016

Dedicatória

Este trabalho é dedicado a todos aqueles que, por falta de oportunidade, infelizmente não puderam mostrar ao mundo seu verdadeiro potencial. Dedico esse trabalho também a todos os cientistas que contribuem, incondicionalmente, com vigor e humildade para o avanço do conhecimento humano.

Agradecimentos

Numa analogia a Teoria do Caos, todos os indivíduos com quem tive contato em minha vida contribuiram, de forma negativa ou positiva, para a concretização desse trabalho. Por motivos óbvios, apenas alguns poucos se destacaram.

Eternamente agradecido serei pelas orientações sábias, racionio lógico afiado e humor irreparável de meu pai, Dimas Rodrigues de Oliveira, pessoa justíssima que merece o respeito de todos. Sem seus ensinamentos e supervisão creio que não estaria onde estou. Agradeço encarecidamente também a minha mãe, Francisca Calegário, que, mesmo distante, sempre tentou cuidar de mim da maneira que pode e sempre torceu para que tudo desse certo em minha vida.

Aos amigos do peito George Rabelo, Pedro Torres, Márcio Rodrigues e Hugo Paixão, agradeço pela força, conselhos, conversas e amizade verdadeira. Aos amigos mais recentes Michael Rodrigues, Tarcísio Junior, Henrique Vieira e Breno Teixeira, agradeço por tornar a rotina do dia-a-dia um pouco mais tolerável.

Aos professores Bruno Macciavello e Flávio Moura minha eterna gratidão pela recomendação sem questionamentos para o processo seletivo do Ciência sem Fronteiras, definitivamente uma experiência que me transformou como pessoa e me capacitou como profissional, marcando para sempre minha vida.

Em especial, gostaria de agradecer a professora Genaina Nunes Rodrigues que, além de me garantir uma posição no Decal Lab da UC Davis, onde tive a felicidade de publicar um artigo Qualis A1, acreditou em mim do início ao fim nessa jornada de mais de 1 ano e meio para conclusão desse trabalho. Sem seus conselhos e entusiasmo seria impossível atingir esse feito. De verdade, muito obrigado por tudo!

Algumas menções honrosas: Leandro Batista de Silva, Osvaldo Andrade, Alexandre Swioklo, Flávio Manacey, João Junior, Denis Aguilar, Victor Barbosa, Erick Moreno, Prof. Claudia Nalon, Prof. Marcelo Ladeira, Caio Yuri, e Elizabete Cerqueira; à todos vocês, meu muito obrigado!

Não poderia deixar de mencionar também Julia Souto, pessoa com quem compartilhei minha vida durante a maior parte dessa aventura que é a UnB e que me ajudou incondicionalmente sempre que pode. Muito obrigado por tudo!

Resumo

Métodos para a engenharia e análise de Sistemas Críticos precisam lidar constantemente com Transformações de Modelos, dado que a maioria desses sistemas são desenvolvidos seguindo uma abordagem modelo-dirigida. Essa asserção é especialmente válida para uma análise de dependabilidade desses sistemas via Verificação de Modelos UML, onde modelos UML precisam ser transformados em modelos passíveis de *Model Checking* por uma ferramenta apropriada. Várias são as técnicas existentes que automatizam tais transformações, mas a maioria delas carece de interoperabilidade, reusabilidade e rigorosidade matemática. Dito isso, nessa monografia é apresentado e elaborado a arquitetura e implementação do *UnB-DALi*, uma biblioteca Java que conduz a transformação de modelos comportamentais UML anotados para modelos DTMCs na notação da ferramenta PRISM. As transformações são baseadas na sintaxe abstrata inerente a esses modelos, seus grafos subjacentes, construção possível via formalismo matemático de Transformação de Grafos Tipados e Atributados, apropriadamente implementado pela ferramenta AGG. Ao final desta monografia, experimentos foram conduzidos para atestar parcialmente a validade do que foi implementado.

Palavras-chave: dependabilidade, transformação de modelos, verificação de modelos, AGG, UML, PRISM

Abstract

Methods for the engineering and analysis of Safety-Critical Systems need to constantly deal with Model Transformations, since most of these Systems are developed following a Model-driven approach. This assertion is specially true for the dependability analysis of such systems via UML Model Checking, where UML models need to be transformed into a model that can be understood by a Model Checking tool. Several are the existent techniques to automate such transformations, but the majority of them lack interoperability, reusability and mathematical rigor. That said, in this monography we present the design and construction of the *UnB-DALi*, a Java library for the model transformation of behavioral UML annotated models into PRISM's DTMC model notation. The transformations are based on the models' inherent abstract syntax, their underlying graphs, made possible via the algebraic Typed Attributed Graph Transformation formalism, appropriately implemented by the AGG tool. At the end of this monography, experiments were conducted to partially verify the validity of what was implemented.

Keywords: dependability, model transformation, model checking, AGG, UML, PRISM

Sumário

1	Introdução	1
1.1	Objetivo Geral	3
1.2	Objetivos Específicos	3
1.3	Estrutura do Documento	4
2	Dependabilidade	5
2.1	O Termo	5
2.2	Verificação de Modelos Probabilísticos	7
2.2.1	Modelos de Especificação de Sistemas	8
2.2.2	Especificação das Propriedades para Análise	10
2.2.3	A Ferramenta PRISM	11
2.3	Análise	13
2.3.1	Metodologia de Transformação de Modelos UML em PRISM	14
2.3.2	Aplicabilidade do Método - Os problemas	18
2.4	Considerações Finais	19
3	Transformação de Modelos	20
3.1	Noções Gerais	21
3.2	Taxonomia	22
3.2.1	Características dos Modelos	22
3.2.2	Características da Transformação	24
3.2.3	Características das Linguagens ou Ferramentas	24
3.3	Transformação de Grafos	27
3.3.1	O Método de Transformação de Grafos	28
3.3.2	A Ferramenta AGG	30
3.4	Considerações Finais	34
4	Trabalhos Relacionados	35
4.1	<i>U-MarMo</i>	35
4.2	Grønmo <i>et al.</i>	36

4.3 Considerações Finais	37
5 UnB-DALi	38
5.1 Requisitos	39
5.2 Arquitetura e Implementação	39
5.2.1 Arquitetura	40
5.2.2 Implementação	42
5.3 Modelos Comportamentais para DTMC	45
5.3.1 Metamodelos de Entrada e Saída	46
5.3.2 Transformações	53
5.4 Validação	57
5.4.1 $AD \rightarrow PRISM(DTMC)$	58
5.4.2 $SD \rightarrow PRISM(DTMC)$	61
5.5 Discussão	63
5.6 Considerações Finais	64
6 Conclusão e Trabalhos Futuros	66
Referências	68
A Restrições Atômicas	72
A.1 Diagrama de Atividades (AD)	72
A.2 Diagrama de Sequência (SD)	74
A.3 Cadeias de Markov de Tempo Discreto (DTMC)	75
B Regras de Transformação	76
B.1 $AD \rightarrow DTMC$	76
B.2 $SD \rightarrow DTMC$	82

Listas de Figuras

1.1 Lançamento do foguete Ariane-5 em 4 de Junho de 1996 [3].	1
2.1 Visão geral da metodologia do <i>Model Checking</i> [3].	7
2.2 Três instâncias de ambientes do PRISM. Da esquerda para a direita: ambiente de modelagem, ambiente de especificação e <i>plot</i> , ambiente de simulação.	12
2.3 Metodologia de análise de dependabilidade proposto por Rodrigues <i>et al.</i> [12].	14
3.1 Visão geral de uma transformação de modelos [21].	21
3.2 Processo de transformação de grafos e seus tipos. [19]	29
3.3 Visão geral do ambiente de programação do AGG.	32
4.1 Metamodelo minimalista de um Diagrama de Sequência proposto por Grønmo <i>et al.</i>	36
5.1 Visão geral do comportamento esperado da biblioteca <i>UnB-DALi</i>	41
5.2 Relacionamento estrutural entre o <i>UnB-DALi</i> e o motor do AGG para transformação de grafos.	42
5.3 Diagrama de Atividades: (a): Grafo-Tipo do AGG. (b): Diagrama de Classes do UnB-DALi.	47
5.4 Condição de consistência que governa a boa formação de um grafo subjacente a um diagrama de atividades no AGG.	48
5.5 Diagrama de Sequência: (a): Grafo-Tipo do AGG. (b): Diagrama de Classes do UnB-DALi.	49
5.6 Condição de consistência que governa a boa formação de um grafo subjacente a um diagrama de sequências no AGG.	50
5.7 Cadeias de Markov de Tempo Discreto (DTMC): (a): Grafo-Tipo do AGG. (b): Diagrama de Classes do UnB-DALi.	52
5.8 Condição de consistência que governa a boa formação de um grafo subjacente a um DTMC no AGG.	53

5.9 Grafo-tipo <i>AD2DTMC</i> do GTS de um diagrama de atividades para um DTMC.	54
5.10 Grafo-tipo <i>SD2DTMC</i> do GTS de um diagrama de sequência para um DTMC	55
5.11 <i>Workflow</i> da transformação de um AD ou SD para um DTMC no formato PRISM.	56

Listas de Tabelas

2.1	Principais técnicas de análise de dependabilidade IEC-60300-3-1 (2003) [35].	13
3.1	Ortogonalidade entre as dimensões <i>edógenas vs. exógenas e verticais vs. horizontais</i> .	23
3.2	Taxonomia da Ferramenta AGG proposta por Mens <i>et al.</i> em [38].	33
4.1	Classificação da Ferramenta <i>U-MarMo</i> conforme taxonomia proposta por Mens <i>et al.</i>	36
4.2	Classificação do trabalho de Grønmo <i>et al.</i> conforme taxonomia proposta por Mens <i>et al.</i>	37
5.1	Testes conduzidos sobre a transformação de um AD para um DTMC no formato PRISM.	58
5.2	Testes conduzidos sobre a transformação de um SD para um DTMC no formato PRISM.	61
A.1	Restrições atômicas aplicadas a grafos-instância do grafo-tipo de um AD.	72
A.2	Restrições atômicas aplicadas a grafos-instância do grafo-tipo de um SD.	74
A.3	Restrições atômicas aplicadas a grafos-instância do grafo-tipo de um DTMC.	75
B.1	Regras para transformação <i>AD</i> → <i>DTMC</i> , organizadas por camadas. Camada 0: conecta diretamente todos os possíveis nós alcançáveis dos nós de decisão iniciais. Camadas 1 – 3 inicializa o contador, o nó inicial e os nós finais. Camada 5 resolve os casos intermediários. Camadas 6 e 7 limpam o grafo resultante.	76
B.2	Regras da transformação <i>SD</i> → <i>DTMC</i> , organizadas por camada. Camada de nível 0 inicializa as <i>lifelines</i> do SD em transformação. Camada de nível 1 transforma os padrões associados com arestas do tipo <i>First</i> . Camada de nível 2 lida com padrões de trocas de mensagens envolvendo apenas arestas do tipo <i>Next</i> . E por fim, a camada de nível 5 limpa o grafo resultante.	82

Capítulo 1

Introdução

Diversos são os tipos de software sendo desenvolvidos hoje no mercado e na academia, variando desde aplicativos para celular até sistemas de controle para propulsores de um foguete tripulado para a estação espacial internacional. É de conhecimento geral, entretanto, que os fatores governantes em qualquer advento na área de desenvolvimento de sistemas computacionais limitam-se a apenas dois: (1) custo e (2) segurança. Como garantir que um sistema de troca de informações pela Internet irá atender em nível aceitável à demanda de requisições e ao mesmo tempo garantir a rentabilidade do mesmo? Como garantir que um sistema de freio ABS irá sempre ser acionado em velocidade suficiente para que os efeitos de um possível acidente não sejam fatais, a um nível aceitável?

Não há respostas simples para essas questões. Por muitos anos cientistas da computação do mundo inteiro vêm estudando formas de garantir que acidentes como o do foguete Ariane-5 não venham ocorrer novamente por erros de software (ver Figura 1.1). Nesta ocasião, um foguete que custou a ESA (*European Space Agency*) um valor de aproximadamente \$7 bilhões de dólares para ser construído, e que carregava material avaliado em \$500 milhões de dólares, entrou em estado de auto-destruição após uma falha de *overflow*.

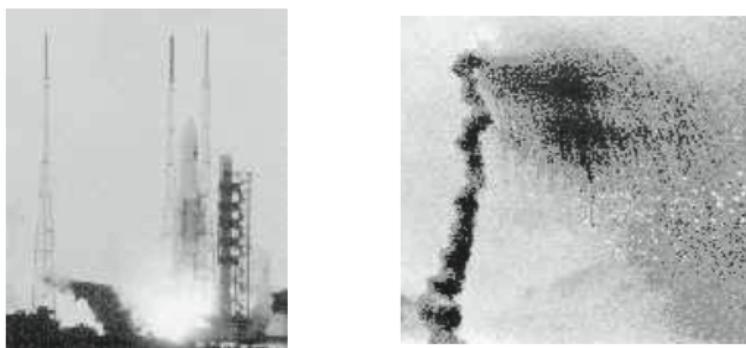


Figura 1.1: Lançamento do foguete Ariane-5 em 4 de Junho de 1996 [3].

na conversão de um valor de ponto flutuante de 64 bits para um inteiro de 16 bits. Isso passou despercebido pelos engenheiros do foguete quando dois módulos distintos, um novo com ULA¹ de 32 bits e outro antigo do predecessor Ariane-4 com ULA de 16 bits, foram inadvertidamente integrados. Tal relapso acarretou o que hoje se entende pelo mais caro *bug* da história [17] [36].

Para determinados tipos de software, portanto, é necessário dar garantias de que certos requisitos serão alcançados. Em se tratando de softwares considerados críticos - aqueles em que vidas dependem do bom funcionamento do sistema - é necessário um rigor extra na obtenção desses tipos de garantias. Várias são as metodologias e processos propostos para obter-se conclusões acerca da conformabilidade ou não de um sistema de software para com seus requisitos. Um método que vem ganhando bastante atenção no meio acadêmico (e industrial) ultimamente é aquele que se entende por *Model Checking* (ou Verificação de Modelos), onde requisitos descritos em lógica temporal são verificados contra um *Transition System*, um grafo dirigido acíclico (DAG) representando todo o espectro de estados alcançáveis na execução de um sistema (onde nós são *estados* e arestas são *transições* entre estados) [3]. Uma vantagem desse método em relação a outros é sua perspectiva de obtenção de mecanismos automáticos para verificação formal de um sistema de software (*push-button analysis*), algo que outros métodos como Provadores de Teoremas (PVS, COQ, etc) não oferecem. Em contrapartida, esse método é adequado apenas para sistemas que possuam um número finito de estados, fornecendo apenas garantias parciais com relação aos estados alcançáveis, ao passo que provadores de teoremas fornecem garantias totais.

A área de estudos em Dependabilidade de Sistemas Computacionais, tema central desta monografia, estuda então quais os requisitos precisam ser minimamente satisfeitos por sistemas críticos como o utilizado no Ariane-5; estuda também como verificar que tais sistemas realmente satisfazem seus requisitos. Consequentemente, dependabilidade trata de tentar elevar de forma justificada, a partir de **métodos formais**, o grau em que um usuário possa *depender* de um sistema [1]. Rodrigues *et al.*, no artigo intitulado *Dependability analysis in the Ambient Assisted Living Domain: An exploratory case study* [12], propõe uma metodologia semi-automática para a verificação das propriedades de dependabilidade a partir do *Model Checking* de modelos comportamentais UML, mais especificamente os conhecidos como *Interaction Overview Diagrams* (IOD). Um empecilho para a condução de análises de dependabilidade a partir desse método é a etapa de conversão de um IOD para um modelo que possa ser utilizado pela ferramenta que conduz o *model checking* propriamente dito. Até o momento da escrita desta monografia ainda não foi estabelecido dentro do grupo de pesquisa LaDeCiC [10] um ferramental **reusável**

¹Unidade Lógica e Aritmética

para automatizar essa etapa tediosa e repetitiva. Portanto, com intuito de instituir em definitivo uma automatização desta importante etapa do método em questão, é que este trabalho se apresenta.

Vale observar que (1) não necessariamente os indivíduos interessados no que aqui for discutido utilizam a mesma ferramenta UML no seu dia-a-dia e (2) não há garantias que modelos UML descritos em duas ferramentas distintas sejam **intercambiáveis** entre si, ou seja, que a sintaxe concreta de um modelo descrito numa ferramenta *A* seja passível de leitura por uma ferramenta *B*. Dessa forma, é interessante do ponto de vista de implementação da biblioteca prever mecanismos para conduzir as transformações num âmbito abstrato, omitindo detalhes da sintaxe de descrição de modelos UML, o XMI.

Para superar essas adversidades - formalização matemática, reusabilidade e interoperabilidade - este trabalho elabora essas transformações com base na sintaxe abstrata inerente a qualquer modelo: seu grafo subjacente. Para isso utilizamos o AGG [39], um ambiente de programação visual baseado em regras construído sobre o formalismo de Transformação de Grafos e implementado em Java.

1.1 Objetivo Geral

Propor, implementar e validar uma biblioteca Java que provê um arcabouço de classes e métodos para a automatização do processo de conversão de modelos comportamentais de um sistema de software, não especificamente restritos a UML, para modelos factíveis de análise por parte das ferramentas de *model checking*, a partir da técnica de transformação de grafos implementada pela ferramenta AGG.

1.2 Objetivos Específicos

- Revisar os conceitos de Dependabilidade, *Model Checking* e *Model Transformations*;
- Estudar as soluções existentes na literatura para o mesmo problema;
- Propor uma arquitetura escalável e manutenível para a biblioteca;
- Implementar a biblioteca;
- Validar parcialmente a biblioteca; e
- Documentar as APIs que a biblioteca fornece, dando exemplos de como usar e de como estendê-la;

1.3 Estrutura do Documento

O Capítulo 2 consiste de uma revisão dos conceitos de Dependabilidade e de Verificação de Modelos; é no Capítulo 2 também que o método proposto por Rodrigues *et al.* é apresentado. O Capítulo 3 foca na revisão dos conceitos sobre Transformação de Modelos, mais especificamente os conceitos de *Typed Attributed Graph Transformation Systems* e da ferramenta AGG, método e ferramenta adotados para a automatização da transformação entre modelos; No Capítulo 4 são discutidos os trabalhos relacionados e seus contrastes com a proposta aqui em destaque. O Capítulo 5 introduz a biblioteca desenvolvida, apresentando seus princípios arquiteturais e detalhes de sua implementação, validando parcialmente a transformação de modelos comportamentais UML para DTMC na linguagem PRISM conforme os conceitos de AGG implementados no UnB-DALi. O Capítulo 6 conclui esta monografia provendo considerações sobre o tema e indicações sobre possíveis trabalhos futuros.

Capítulo 2

Dependabilidade em Sistemas Computacionais

Todo projeto de um sistema de software, de forma resumida, consiste na concretização de seus *requisitos funcionais* e seus requisitos *não funcionais*. Os primeiros denotam *características e funcionalidades* que o sistema deve exibir e os últimos denotam *condições de qualidade* sobre o comportamento do sistema. Por exemplo: suponha um sistema controlador de freios ABS; um possível requisito funcional seria dizer: “quando acionado o freio pelo condutor, o controlador deve ativar o atuador”; um não funcional por sua vez seria dizer: “o intervalo de tempo entre o acionamento do freio pelo condutor e o ativação do atuador pelo controlador não deve ser maior que 0,005 segundos”.

Para qualquer tipo de software, especialmente os classificados como críticos, existe a necessidade de assegurar que o sistema cumpra seus requisitos. Para atestar essa adequação do sistema, técnicas de testes como os de Caixa Preta ou de Unidade são adotadas para verificar os requisitos funcionais, por exemplo; requisitos não funcionais, porém, exigem a aplicação de técnicas de verificação formal para este mesmo fim. É justamente com propriedades não funcionais (PNF) que a área de estudos em Dependabilidade atua [34] e se preocupa, provendo métodos e formalismos para a verificação da conformabilidade entre um sistema de software e seus devidos requisitos. Portanto, com o intuito de esclarecer o termo Dependabilidade, de explorar formalismos relacionados e de apresentar a técnica de análise base deste trabalho, é que este capítulo se apresenta.

2.1 O Termo Dependabilidade

Várias definições já foram dadas ao termo *Dependabilidade*. Neste trabalho seguiremos a linha de pensamento de Avizienis *et al.* [1] que sugere que todo sistema fidedigno seja um sistema que se possa *justificadamente* confiar, ou seja, que *demonstre* possuir habilidade

de evitar falhas que são mais frequentes e severas que o aceitável. Dessa forma, temos o próprio termo como uma propriedade não funcional de um sistema de software onde a área de pesquisa que o engloba busca por métodos e ferramentas para poder-se mensurar *quantitativa* e *qualitativamente* o nível de dependabilidade associado. Importante notar que a área não somente atua em software, dado que dependabilidade é uma propriedade almejada por qualquer tipo de sistema; tais outros casos, porém, fogem do escopo deste trabalho.

Dependabilidade, ao longo do tempo, foi desenvolvido como um conceito que trabalha e se compõe de outros dois elementos distintos acerca de um sistema: (1) *Atributos* de um sistema e (2) *Ameaças* a um sistema. Por *Atributos* entende-se todos os conceitos capazes de fornecer métricas para se aferir dependabilidade. Tais conceitos seriam:

- **Disponibilidade:** prontidão na entrega correta de um serviço¹;
- **Confiabilidade:** continuidade da entrega correta de um serviço;
- **Segurança Operacional**²: ausência de consequências catastróficas para o usuário ou ao meio;
- **Integridade:** ausência de modificações impróprias ao sistema;
- **Manutenabilidade:** habilidade do sistema submeter-se a reparos e alterações.

Ameaças englobam todos aqueles eventos e artefatos que possam afetar, de forma negativa, a dependabilidade de um sistema. Estão classificados em três categorias:

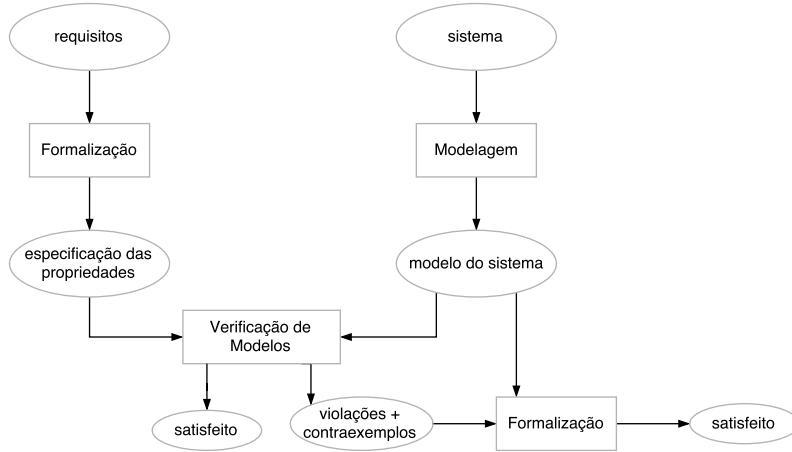
- **Defeitos:** eventos os quais desviaram um comportamento correto de um serviço do sistema para um comportamento incorreto do mesmo;
- **Erros:** estados do sistema onde desvios de comportamento ocorreram, não necessariamente causando falhas;
- **Falhas:** a julgada ou hipotetizada causa de um erro.

Meios foram, ao longo dos últimos 60 anos, desenvolvidos e colocados em prática com o intuito de garantir um maior grau de dependabilidade ao se desenvolver um sistema. Dado isso, *Avizienis et al.* os consideram como elementos integrantes do que se define por dependabilidade. Tais meios se classificam entre quatro categorias:

- **Prevenção de Falhas:** representando meios para a prevenção da ocorrência ou introdução de falhas;

¹Serviço: um comportamento específico do sistema como percebido pelo usuário [1]

²*Safety*, em inglês

Figura 2.1: Visão geral da metodologia do *Model Checking* [3].

- **Tolerância a Falhas:** indicando meios para evitar defeitos de serviço na presença de falhas;
- **Remoção de Falhas:** significando meios para se reduzir o número e a severidade de falhas;
- **Previsão de Falhas:** englobando meios para se estimar o atual número, a futura incidência, e as possíveis consequências de falhas.

Assim sendo, percebe-se que o termo Dependabilidade abrange um amplo campo de pesquisa, sendo considerado por alguns [20] [27] [3] uma propriedade extremamente importante e necessária para o futuro da Engenharia de Software. O método de análise utilizado como base para a formulação da biblioteca desenvolvida por este trabalho pode ser classificado como uma metodologia de auxílio para *Previsão de Falhas*, que não abrange nem *Manutenibilidade* e nem *Integridade* e apenas detecta *Falhas* [12] da arquitetura de um sistema. Na próxima seção, o formalismo utilizado pelo método em destaque é abordado.

2.2 Verificação de Modelos Probabilísticos

Verificação de Modelos, ou *Model Checking* no termo em inglês (Figura 2.1), consiste na modelagem e análise matematicamente rigorosa de um Sistema de Transição: um grafo acíclico dirigido (ou DAG) das transições entre todos os possíveis estados do objeto de análise, seja este software, hardware ou híbrido. As técnicas existentes realizam de forma sistemática uma varredura do modelo de entrada em busca de padrões que verifiquem a adequação ou não do sistema a um ou mais requisitos especificados em Lógica Temporal [3], que permite a captura de comportamentos que variam de acordo com tempo, a citar:

- *safety*: adversidades (situações indesejadas) que têm impacto negativo ao ocorrer durante o tempo de execução do sistema em análise, *e.g.*: situações de *deadlock*, onde processos concorrentes aguardam de forma mútua um ao outro progredir [3];
- *liveness*: características positivas que devem ser exibidas durante todo o tempo de execução do sistema em análise, *e.g.*: *starvation freedom*, onde é garantido que eventuais *deadlocks* entre processos concorrentes nunca irão ocorrer durante a execução do sistema [3].

Verificação de Modelos Probabilísticos (PMC) estende as capacidades da Verificação de Modelos “pura” (esclarecido acima) por levar em consideração índices de confiabilidade dos componentes do sistema coletados por meio de observação empírica ou reuso de componentes. Nessa categoria, os Sistemas de Transição então se tornam abrangentes o suficiente para poder representar sistemas que apresentem comportamentos aleatórios (sistemas tolerante a falhas, protocolos de comunicação, controladores, etc) ou que necessitem cumprir requisitos de natureza estocástica (*e.g.* “O serviço S1 do sistema A necessita estar 99% do tempo ativo”), tornando factível a verificação formal dos mesmos. Muito frequentemente, no processo de análise utilizando as técnicas de PMC, os índices citados anteriormente são parametrizados. Isso se torna um artifício útil no âmbito da análise de dependabilidade já que possibilita a condução de um estudo de sensibilidade dos componentes por meio da geração de gráficos e métricas estatísticas [11].

As ferramentas que conduzem este tipo de análise são chamadas de *Model Checkers* [40] [28] [37]. São elas as responsáveis por avaliar todos os possíveis cenários de execução de um sistema de forma a garantir que este *justificadamente* satisfaça as propriedades de interesse. Resumidamente, ferramentas desse tipo destinam-se a encontrar estados no Sistema de Transição que violem os requisitos declarados, fornecendo, na forma de um *trace*³ de execução, um contraexemplo que indique como este sistema pode alcançar tal comportamento contraditório. Neste trabalho lidaremos especificamente com a ferramenta PRISM [25] (ver Seção 2.2.3) que atualmente fornece meios para análise de seis tipos de modelos probabilísticos distintos: DTMCs, CTMCs, MDPs, PAs, PTAs, e PPTAs (vide Seção 2.2.1); todos devidamente acompanhados de suas Lógicas Temporais de especificação de propriedades (Seção 2.2.2).

2.2.1 Modelos de Especificação de Sistemas

No contexto de Verificação de Modelos, tem-se que um modelo que represente o sistema alvo com acurácia (1) e sem ambiguidades (2) são condições primordiais para se conduzir uma verificação do mesmo utilizando-se da técnica de Verificação de Modelos. Em [3],

³Sequência de estados a partir do estado inicial [3]

temos que o formalismo escolhido para modelagem de um sistema que satisfaça (1) e (2) é dado pelo conceito de um Sistema de Transição, definido abaixo:

Definição 1 (Sistema de Transição). *Um Sistema de Transição TS é a 6-tupla $(S, Act, \rightarrow, I, AP, L)$, onde:*

- S é um conjunto de estados,
- Act é um conjunto de ações,
- $\rightarrow \subseteq S \times Act \times S$ denota uma relação de transição,
- $I \subseteq S$ é um conjunto de estados iniciais,
- AP é um conjunto de proposições atômicas, e
- $L : S \rightarrow 2^{AP}$ é uma função rotuladora.

Em se tratando de *Probabilistic Model Checking*, temos que tais modelos precisam ser enriquecidos com informações de probabilidade. Isso é devido por *Transition Systems* “puros”, como os acima, não serem capazes de capturar completamente o comportamento de um sistema real, ou por serem rígidos demais no sentido de exigir a correção de propriedades *qualitativas*, inherentemente mais custosas, e assim acabar por coibir uma verificação satisfatória do sistema (dado preocupações como o *problema da explosão de estados*, por exemplo) [24]. Esse enriquecimento pode ser concretizado das mais diversas maneiras, mas é mais comumente feito pelo uso de Cadeias de Markov de Tempo Discreto (Definição 2), modelo probabilístico mais simples disponível na literatura.

Definição 2 (Cadeias de Markov de Tempo Discreto). *Uma Cadeia de Markov de Tempo Discreto, ou simplesmente DTMC, é uma 5-tupla da forma $(S, \bar{s}, \mathbf{P}, AP, L)$ [24], onde*

- S é um conjunto de estados,
- $\bar{s} \in S$ é um estado inicial,
- $P : S \times S \rightarrow [0, 1]$ é a matriz de probabilidades de transições, satisfazendo:

$$\forall s, \sum_{s' \in S} P(s, s') = 1,$$

- AP é o conjunto de Proposições Atômicas e,
- $L : S \rightarrow 2^{AP}$ é a função rotuladora;

Um maior poder de expressividade pode ser exigido por parte do Analista para que a modelagem possa capturar comportamentos mais complexos do objeto em estudo. Tal expressividade é obtida trabalhando-se os modelos em termos de continuidade de tempo e transições não-determinísticas. A ferramenta de *Model Checking* aqui utilizada, em sua mais nova versão (4.0) [25], suporta os seguintes tipos de modelos probabilísticos de

sistema:

- DTMC: discreto e determinístico,
- CTMC: contínuo e determinístico,
- MDP: discreto e não-determinístico,
- PA: discreto e determinístico,
- PTA: contínuo e não-determinístico,
- PPTA: contínuo e não-determinítico.

Para os objetivos deste trabalho, porém, o foco é em torno de DTMCs. Iremos, todavia, fornecer meios para facilitar que, num futuro próximo, a biblioteca dê suporte aos tipos de *Probabilistic Transition Systems* citados acima.

2.2.2 Especificação das Propriedades para Análise

A metodologia de análise de Verificação de Modelos (Figura 2.1) prevê como entrada uma formalização dos requisitos do sistema em estudo, o que geralmente é feito pelo uso de lógicas temporais. Quando os sistemas são descritos em DTMC, tem-se que a lógica mais comumente utilizada é PCTL (*Probabilistic Computational Tree Logic*), uma extensão probabilística de CTL [8].

Definição 3 (Gramática PCTL). *A gramática de PCTL é definida da seguinte forma:*

$$\Phi ::= \text{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid P_{\sim p}[\phi]$$

$$\phi ::= X\Phi \mid \Phi \cup^{\leq k} \Phi \mid \Phi \cup \Phi$$

onde a é uma proposição atômica, $\sim \in \{<, \leq, \geq, >\}$, $p \in [0, 1]$ e $k \in \mathbb{N}$.

Fórmulas PCTL seguem a gramática dada na Definição 3 e podem ser divididas entre duas categorias: *fórmulas sobre caminhos* (ϕ) e *fórmulas sobre estados* (Φ). Caminhos se definem por *fragmentos de caminhos iniciais e maximais* [3] (sequências finitas de estados onde o primeiro estado é um estado inicial e o último estado é um estado final em sistemas de transição finitos) e uma fórmula PCTL é dita *bem formada* toda vez que o operador mais externo da mesma pertencer a Φ , por exemplo:

$$P_{\leq 0.15}[\neg fail_A \cup fail_B],$$

denotando que “a probabilidade do componente B de falhar antes que o componente A falhe é de no máximo 0.15”. Uma fórmula mal formada seria, por exemplo, $true \cup \Phi$ (“eventualmente Φ é satisfeita”), pois não há nenhum quantificador sobre estados.

As fórmulas PCTL, como nota-se na definição acima, são construídas por diversos operadores; desde os oriundos da lógica proposicional ($true$, $\Phi \wedge \Psi$, $\Phi \vee \Psi$, $\neg\Phi$) aos quantificadores sobre caminhos ($X\Phi$, $\Phi \cup \Psi$, $P_{\sim p}[\phi]$). De forma resumida, em relação a um estado s :

- $X\Phi$ denota que o *próximo* estado tem que satisfazer Φ para que s satisfaça a fórmula;
- $\Phi_1 \cup \Phi_2$ denota que existe um caminho a partir de s o qual sempre satisfaz Φ_1 até que Φ_2 seja satisfeita;
- $P_{\sim p}[\phi]$ que significa que a probabilidade de uma *fórmula de caminho* ϕ ser verdadeira para o estado s satisfaz o limite $\sim p$.

Tem-se portanto que fórmulas PCTL são interpretadas sobre os estados de um DTMC e é dito que um estado $s \in S$ *satisfaz* uma fórmula Φ , denotado por $s \models \Phi$, se Φ é verdadeiro para o estado s .

Para se realizar a Verificação de Modelos de um requisito definido em PCTL sobre um DTMC, técnicas numéricas e algoritmos sobre grafos são utilizados. A solução de problemas comuns, como computar as probabilidades de uma fórmula $\Phi \cup \Psi$ de ser satisfeita, são geralmente reduzidos a resoluções de sistemas de equações lineares por Gauss-Seidel, ao invés de métodos diretos como a Eliminação Gaussian, dados diversos problemas de escalabilidade do processo de verificação.

Por fim, existem lógicas mais robustas que o PCTL para formalizar requisitos sobre um DTMC como o PCTL* [3], por exemplo; essa robustez vem, porém, ao custo de uma verificação mais cara em termos computacionais (tanto de tempo quanto de espaço).

2.2.3 A Ferramenta PRISM

Para se conduzir uma verificação de software pelo método de PMC, ferramentas devem checar sistematicamente os requisitos descritos em lógica temporal contra a *execução* do modelo (DTMC, MDP, PTA, etc) do sistema em estudo. Considerado o estado-da-arte da Verificação de Modelos Probabilísticos, a ferramenta *PRISM* [23] fornece todo o arcabouço necessário para a verificação formal de um software ou hardware. Esse arcabouço é caracterizado tanto por implementações otimizadas dos algoritmos de Verificação de Modelos quanto por um ambiente completo, amigável ao usuário, para a realização da análise.

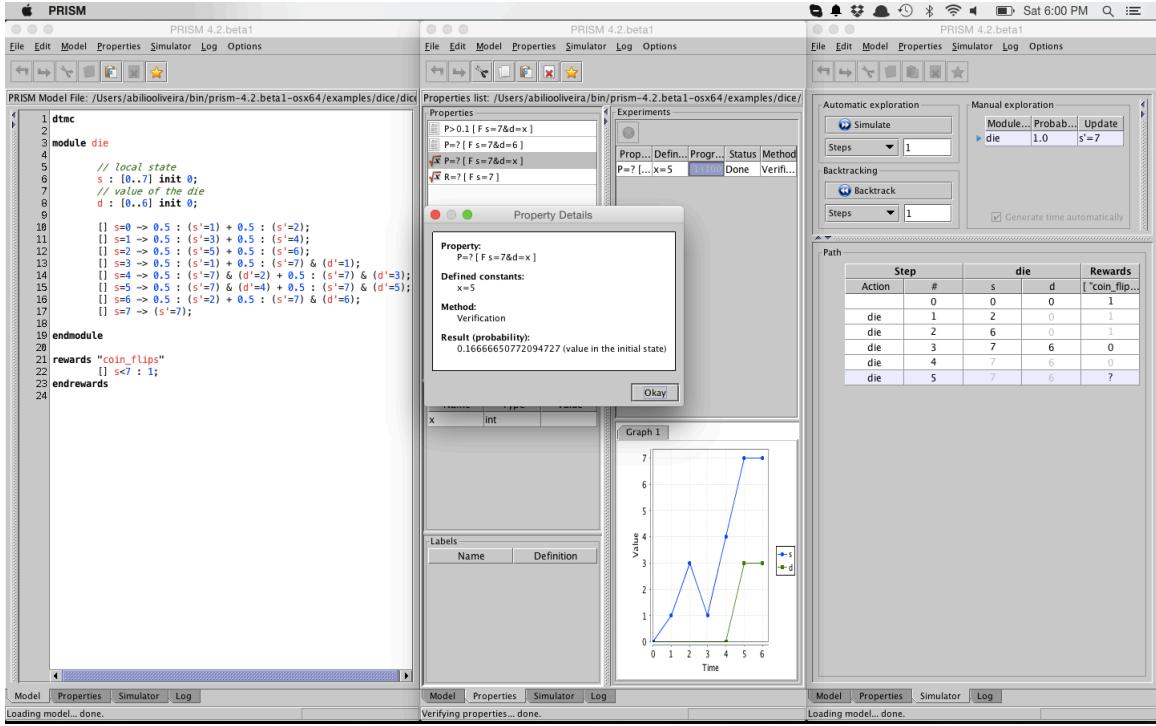


Figura 2.2: Três instâncias de ambientes do PRISM. Da esquerda para a direita: ambiente de modelagem, ambiente de especificação e *plot*, ambiente de simulação.

A interface do PRISM se divide em (1) um ambiente para a modelagem do sistema por meio da linguagem de descrição de modelos PRISM, (2) um simulador que possibilita o *debug* do modelo e (3) um ambiente de especificação de propriedades e plotagem gráfica para a condução de análises de sensibilidade dos componentes do sistema modelado. Em particular, no ambiente de propriedades, é possível conduzir experimentos com intuito de elucidar padrões no comportamento do sistema a medida que os parâmetros de probabilidade das transições do modelo são variados (ver Figura 2.2).

Como citado acima, a ferramenta PRISM fornece uma linguagem própria para a descrição dos modelos. Na linguagem PRISM, um modelo é composto basicamente de módulos; módulos por sua vez são compostos por um número finito de variáveis, que denotam seu conjunto de possíveis estados, e compostos também por um conjunto finito de comandos condicionais, denotando as transições entre seus estados. Um DTMC descrito na linguagem PRISM apresenta o seguinte *template* para seus comando condicionais:

$$[action]<guard>\rightarrow<probability>:<update>;$$

onde *guard* é um predicado sobre todas as variáveis do modelo, o qual quando satisfeito faz com que o módulo transite de seu estado atual para o estado de *update* com probabilidade *probability*⁴; e onde *action* é usado para sincronizar comandos entre si (sem o rótulo

⁴ $0 \leq probability \geq 1$, onde *probability* $\in \mathbb{R}$

action, o comando será executado assincronamente). Uma vez que a modelagem em PRISM é feita, o analista pode simular o modelo por meio do ambiente de simulação e então constatar a existência ou não de estados de *deadlock* por exemplo, ou verificar a satisfabilidade do modelo com relação a propriedades especificadas em lógica temporal por meio do ambiente de especificação.

PRISM, inicialmente desenvolvido na *University of Birmingham* e agora sendo mantido pela *University of Oxford*, é multiplataforma, código-livre, licenciado sobre GPL e um ambiente de PMC em constante atualização com suporte para os mais diversos tipos de modelos e lógicas temporais. Maiores detalhes podem ser obtidos via o website da aplicação [40].

2.3 Análise de Dependabilidade

Uma análise de dependabilidade tem por objetivo primário a verificação dos atributos de dependabilidade discutidos na Seção 2.1 de forma sistemática, o que implica dizer que tanto o sistema quanto os atributos precisam ser formalizados de maneira compatível entre si, permitindo assim a criação de algoritmos e ferramentas que possam checar se o sistema alvo satisfaz as propriedades formalizadas ou não.

Tabela 2.1: Principais técnicas de análise de dependabilidade IEC-60300-3-1 (2003) [35].

Técnicas
Event Tree Analysis (ETA)
Failure Mode and Effect Analysis (FMEA)
Failure Mode, Effect, and Criticality Analysis (FMECA)
Fault Trees Analysis (FTA)
Functional Failure Analysis (FFA)
Hazard and Operability studies (HAZOP)
Markov analysis
Petri Net analysis (PN)
Preliminary Hazard Analysis (PHA)
Reliability Block Diagrams analysis (RBD)
Truth table

Dado que a verificação completa de um sistema é sempre deveras custoso em qualquer técnica de verificação formal [3], geralmente análises deste tipo são conduzidas sobre *modelos* do objeto em análise, onde recai a máxima: “*Qualquer técnica de verificação baseada em modelos é tão somente boa quanto o modelo do sistema em si*” [3]. Assim, a condução do que se entende por uma análise de dependabilidade pode ser feita das mais diversas maneiras, onde o principal fator que varia é o modelo para representação do

sistema. Outros fatores também variam, como por exemplo o nível de abstração adotado (estrutural ou comportamental) e a fase do ciclo de desenvolvimento do sistema onde a técnica deve ser aplicada (início, durante todo o ciclo, ou somente no final). Um compilado das principais categorias de técnicas de análise de dependabilidade levantadas pelo padrão IEC-60300-3-1 estão destacadas na Tabela 2.1.

A técnica que a biblioteca aqui se propõe a dar suporte é um derivado do que se entende por *Markov Analysis*, utilizando-se da metodologia proposta por *Probabilistic Model Checking*, onde o modelo formalizado de um sistema é um DTMC e propriedades são formalizadas por meio de PCTL, vide Seção 2.2; é baseada em modelos comportamentais do sistema em análise e se foca nos estágios iniciais do ciclo de vida de seu desenvolvimento.

2.3.1 Metodologia de Transformação de Modelos UML em PRISM

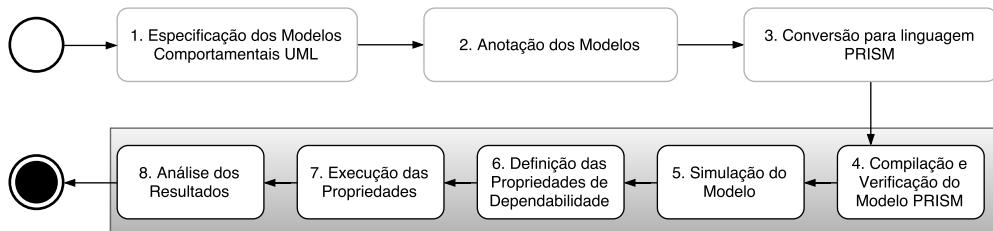


Figura 2.3: Metodologia de análise de dependabilidade proposto por Rodrigues *et al.* [12].

De forma resumida, todo método que envolve o uso de *Probabilistic Model Checking* requer uma formalização de um modelo comportamental de um sistema que carregue tanto informações probabilísticas sobre as transições entre os casos de uso como também informações sobre quanto um pode confiar num determinado componente do mesmo. É interessante, portanto, definir uma metodologia para condução de uma análise como essas que possa abranger o maior número de *workflows* de engenharia orientada a modelos possível. Em [12] é sugerido que tal objetivo possa ser alcançado ao adotar UML como padrão para modelagem de sistemas, dado seu alto índice de uso em projetos industriais e acadêmicos; Rodrigues *et al.* definem então uma metodologia para condução de uma Análise de Dependabilidade sobre modelos comportamentais UML (IOD), utilizando a ferramenta PRISM como *framework* para realização do PMC sobre a formalização em DTMC de tais modelos. Tal metodologia, a qual a biblioteca aqui sendo elaborada dará suporte, é elucidada pela Figura 2.3. Há, portanto, duas bem definidas macro etapas:

1. Modelagem UML, consistindo das atividades de 1 a 3;
2. PMC do modelo PRISM, consistindo das atividades de 4 a 8;

Modelagem UML

A primeira atividade do processo de análise consiste na especificação de modelos UML do sistema. Como o objeto da Análise de Dependabilidade aqui explorado são os estágios iniciais do ciclo de desenvolvimento, temos que o foco do processo é em cima de uma visão mais alto nível dos componentes do sistema. A técnica usufrui, portanto, dos Diagramas de Atividade UML (ADs) - que definem tanto o fluxo de controle do sistema como também seus *cenários*, por meio de *transições* e *nós de ação e decisão* - em conjunto com Diagramas de Sequência UML (SDs) - utilizados para elucidar como os *componentes* do sistema interagem entre si, dado *entidades* e *troca de mensagens* entre as mesmas.

Essa modelagem é conduzida nos termos do princípio proposto pelos Diagramas Comportamentais UML (ou *Interaction Overview Diagrams*, em inglês), onde cada nó de ação do AD é **especificado** por um SD, o que torna possível obter uma abstração bastante consistente de como o sistema deve se comportar e como a comunicação entre seus componentes deve ocorrer, justificando assim a sua adoção para se conduzir uma análise de dependabilidade efetiva.

Na segunda atividade temos que os diagramas UML já especificados e revisados são então *anotados* com informações sobre a confiabilidade dos componentes R_C e com informações sobre a probabilidade de transição PT_{ij} entre os cenários S_i e S_j ⁵. As probabilidades PT_{ij} , como são diretamente relacionadas a transições entre cenários, são anotadas nas correspondentes arestas das mesmas no AD do sistema e os índices de confiabilidade R_C , por sua vez, anotados nos componentes dos respectivos SDs. Vale citar que em [12], valores unitários - *coarse-grained* - são adotados para os índices de confiabilidade R_C de cada componente; em geral, porém, seria possível associar tais índices às mensagens individuais e/ou segmentos das *timelines* dos componentes em seus respectivos SDs, de forma a fazer com que a confiabilidade do componente seja então uma função da confiabilidade de seus elementos, aumentando assim o poder de expressividade e conclusão da análise. Isto porém não impede os autores de generalizar os resultados obtidos, dado que valores unitários para os índices não ocorre em perda de generalidade.

Temos que três importantes suposições subjazem as anotações de confiabilidade dos componentes:

1. Um serviço oferecido por C e invocado por C' é denotado por uma mensagem de C' a C . A confiabilidade associada ao serviço é igual a confiabilidade do componente C , ou seja, R_C . Concomitantemente, assumimos que o tempo de execução da invocação é não impactante na confiabilidade do componente.

⁵Tais dados são normalmente obtidos a partir de uma análise dos perfis de operação do sistema.

2. Assumimos que a transferência de controle entre os componentes do Diagrama de Sequência segue a premissa de um DTMC, ou seja, o estado de execução atual somente é afetado pelo estado de execução imediatamente anterior.
3. Falhas são consideradas eventos independentes entre as transições.

Tais suposições são cruciais, e alteram de maneira definitiva como um IOD é traduzido em um modelo PRISM (etapa 3).

Na terceira etapa, é dado então um IOD de um sistema construído nos termos das etapas 1 e 2, que será mapeado para seu respectivo modelo PRISM. O processo de conversão consiste em primeiramente obter um modelo de máquinas de estado para cada nó do Diagrama de Atividades e posteriormente convertê-los, um a um, para seus respectivos módulos no modelo PRISM.

Tais máquinas de estado são modeladas com base em seus respectivos SDs (com exceção dos nós de decisão, pois não possuem SDs associados), onde cada execução de um serviço modelado pelo Diagrama de Sequência é representado por um estado nesse modelo intermediário, e cada mensagem trocada entre os componentes do SD são representados por transições rotuladas entre os estados do mesmo. Dessa forma, para cada estado da máquina de estados resultante há em contrapartida um componente C processando um serviço. Por essa razão, cada estado está associado com uma confiabilidade do componente correspondente R_C , que aqui denotamos como a probabilidade de sucesso na execução do serviço, assim como associado com uma probabilidade de falha ($1 - R_C$), representado por uma transição para o estado de erro E . No contexto a que se insere essa análise, entende-se por falha toda e qualquer propagação de erro para a interface entre o serviço e seu consumidor (seja este último um componente do sistema ou uma pessoa).

Tais máquinas de estados são então mapeadas a seus respectivos módulos PRISM. Para ilustrar esse mapeamento, segue então um exemplo de uma ação simples do PRISM traduzida a partir da máquina de estados do serviço *notify* de um componente de um SD anotado com seu respectivo índice de confiabilidade R_C :

$$[notify]s = 0 \rightarrow R_C : (s' = 1) + (1 - R_C) : (s' = 2)$$

que estabelece que, se o estado após a chamada ao serviço é $s = 0$, então ocorre uma transição para $s = 1$, com probabilidade R_C , ou ocorre uma transição para $s = 2$ (estado de erro), com probabilidade $1 - R_C$. Além disso, observe que o rótulo do comando *notify* é usado para sincronizar este comando com outros comandos rotulados da mesma forma, em conformidade com as regras do *Communicating Sequential Process* (CSP) [6], o qual sincroniza os comandos rotulados com a mesma *ação* quando suas respectivas condições de guarda forem atendidas.

Finalmente, cada *nó de decisão* do AD também é associado com um *module* em PRISM, contendo apenas uma ação delegando qual o próximo estado a ser transitado para, com probabilidade PT_{ij} , dado que as respectivas condições de guarda foram satisfeitas. Vale citar que na técnica em questão, decidiu-se por sempre conduzir a análise sobre modelos DTMC, dado que seu mapeamento a partir de máquinas de estados é “direto”, ou seja: “cada estado e cada transição da máquina de estados são representados, no PRISM DTMC, por um único comando” [12]. Futuramente, planeja-se também criar mapeamentos para os outros tipos de modelos (ver Seção 2.2.1).

PMC do modelo PRISM

Após a geração do modelo PRISM DTMC, o método determina que seja feito então a compilação e verificação do mesmo (etapa 4). A verificação determina se o modelo está bem construído, dadas as regras de sintaxe da linguagem PRISM; já a compilação consiste basicamente de uma síntese do modelo para uma representação Markoviana interna considerando as regras de sincronização do CSP e as composições probabilísticas elucidadas factíveis de serem feitas, obtidas a partir do modelo PRISM gerado.

Na etapa 5, o modelo então é executado em modo de simulação por meio do ambiente de simulação do PRISM (Seção 2.2.3). O objetivo dessa etapa é verificar a ausência de propriedades simples de concorrência no modelo, como *deadlock* por exemplo. A etapa 6 se utiliza do ambiente de especificação do PRISM para formalizar as propriedades de dependabilidade (Seção 2.1) em PCTL, assim como outros requisitos do sistema. Para se conduzir uma análise de dependabilidade por meio de PMC é sempre de interesse do analista checar, para cada requisito Φ do sistema, a validade das seguintes três propriedades de dependabilidade:

- qual a probabilidade de que o sistema alcance um estado que satisfaça o requisito (*reachability*) - “ $P =? [F(\Phi)]$ ” em PCTL;
- garantia de execução livre de erros e *deadlocks* até atingir estado que satisfaça o requisito (*safety*): “ $init \Rightarrow P \geq 1[G(\Phi)]$ ” em PCTL; e
- garantia de que a partir do estado inicial, por mais tempo que possa demorar, o sistema irá satisfazer o requisito (*liveness*): “ $init \Rightarrow P \geq 1[F(\Phi)]$ ”, em PCTL.

Após formalizadas as propriedades desejadas, a etapa 7 determina que o DTMC do sistema seja então executado pelo PRISM, o qual irá procurar por violações das propriedades formalizadas e informar ao analista se para aquelas propriedades o sistema como modelado está correto ou não. Uma vez que tais execuções terminem, o processo definido por Rodrigues *et al.* estabelece que uma análise de domínio seja realizada em cima dos

resultados computados para auxiliar na tomada de decisão de como conduzir o desenvolvimento do sistema em questão, finalizando assim o que se entende por uma análise de dependabilidade.

Observe que a análise como estruturada pode ser então utilizada como base para uma melhor alocação de recursos (dinheiro, tempo e atenção) para o desenvolvimento e teste daqueles módulos identificados como mais críticos dentro do sistema. Importante notar que este método não se trata de um meio para atingir *Prevenção de Falhas* ou *Tolerância a Falhas* no sistema em desenvolvimento, mas pode ajudar a identificar os módulos do sistema aos quais tais meios deveriam ser aplicados. Trata-se portanto de um método para a *Previsão de falhas* de um sistema de software.

2.3.2 Aplicabilidade do Método - Os problemas

A metodologia proposta por Rodrigues *et al.* se demonstrou flexível o suficiente para poder abranger os mais diversos tipos de sistemas existentes, dadas as decisões como o uso do UML para modelagem dos sistemas e o uso de modelos baseados em máquinas de estados probabilísticas Markovianas para condução da análise de dependabilidade, por exemplo [12]. De acordo com os autores, em contrapartida, investigações empíricas ainda precisam ser realizadas para confirmar tal afirmação. Há, inevitavelmente, dois fatores predominantes que podem intervir no sucesso da aplicabilidade do método:

1. *Separação Semântica Modelo-Sistema*: não pode-se assumir que o processo de implementação do sistema somente irá ocorrer *após* a análise do modelo ser concluída; é de se esperar, portanto, que a análise ocorra **enquanto** o sistema esteja sendo desenvolvido. Logo, o método não provê nenhuma garantia de que o modelo sendo usado para a condução da análise esteja compatível/sincronizado, a um nível aceitável, com o sistema propriamente dito;
2. *Conversão UML-PRISM*: não há garantias de que o modelo PRISM resultante seja compável com o modelo UML desenhado; concomitantemente, tal rotina é deveras custosa em termos de tempo e nenhum mecanismo para automatizar essa etapa (reduzindo assim erro humano na conversão) é providenciado, agregando impactos desconhecidos na validade das conclusões das análises realizadas.

Percebe-se então dois níveis de separação entre o que a ferramenta de PMC utiliza para análise e o que o sistema realmente é. Este trabalho vem com o intuito, portanto, de dar o pontapé inicial para solucionar em definitivo essas ameaças a validade do processo proposto por Rodrigues *et al.* onde, **primeiramente**, será fornecida toda uma infraestrutura *reusável, interoperável e matematicamente rigorosa* para que a etapa de conversão

UML-PRISM seja feita de forma automatizada e sem erros sintáticos, resolvendo assim o fator 2 enumerado acima.

2.4 Considerações Finais

Nesse capítulo foram esclarecidos os conceitos que cercam uma análise de dependabilidade da forma como proposta por Rodrigues *et al.*. Assim, os conceitos básicos de dependabilidade foram expostos, a teoria e as ferramentas de *Probabilistic Model Checking* foram revisados e a metodologia em si foi abordada com detalhes.

No próximo capítulo os conceitos de Transformação de Modelos, importante para a implementação da conversão *UML-PRISM* de forma automática (tema desta monografia), serão abordados com detalhes.

Capítulo 3

Transformação de Modelos

Sistemas de Software, em termos gerais, têm como objetivo principal a automatização de uma determinada classe de tarefas [38]. Nesse contexto, modelos de um sistema - sejam estruturais, de requisitos ou comportamentais - ajudam primordialmente (1) na compreensão das partes envolvidas sobre como a automatização de uma determinada tarefa pode ser implementada e (2) na manutenibilidade dessas automatizações. Observou-se ao longo do tempo, porém, que modelos podem assumir um papel muito mais importante no ciclo de vida de desenvolvimento de um sistema: artefatos diversos - páginas web de documentação, modelos de análise [12], códigos-fonte da solução, etc - podem ser automaticamente construídos a partir de um modelo, facilitando assim o reuso de informação e, a longo prazo, diminuindo o tempo de manutenção do sistema ou de construção de sistemas correlatos.

Por definição, transformação de modelos trata-se da “automática construção de um ou mais modelos *alvo* a partir de um ou mais modelos *fonte*, dada uma *descrição* da transformação” (Mens *et al.* [38], numa tradução livre). No contexto de desenvolvimento de software, por exemplo, um código fonte - seja em linguagem C, Java, Python, etc - trata-se de uma representação simplificada, legível por humanos das estruturas de baixo nível de uma linguagem de máquina; logo, códigos-fonte podem ser interpretados imprevedivelmente como modelos. Um código de máquina relocável também pode ser interpretado como um modelo, dado que apenas orienta a ordem das operações que devem ser executadas pelo processador, não possuindo impacto no mundo real por si só. Dessa forma, o passo de compilação - a tradução de um código de uma linguagem de programação qualquer para um código de máquina relocável, a partir de uma gramática - é interpretado na literatura como um caso específico de transformação de modelos, especificamente *transformação de programas* [38], onde: o código-fonte é o *modelo fonte*; o código de máquina relocável é o *modelo alvo*; e a gramática é a *descrição* da transformação.

Transformação de Modelos (ou *Model Transformation* no termo em inglês) é a área da

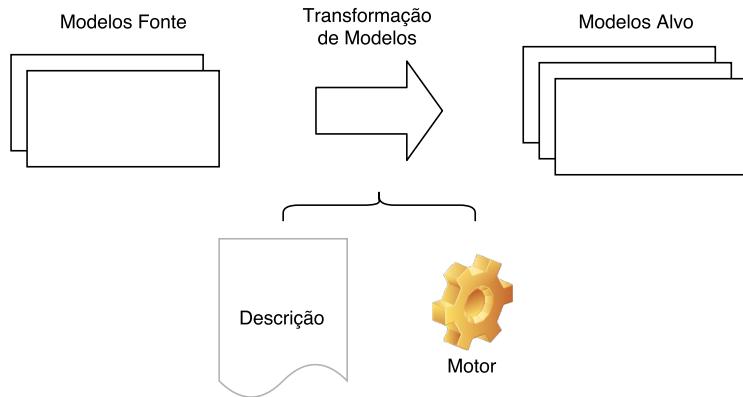


Figura 3.1: Visão geral de uma transformação de modelos [21].

Engenharia de Software que estuda as técnicas e provê as ferramentas para que a geração de diversos artefatos possa ser feita a partir dos modelos de um sistema [18]. Dado esse contexto, este capítulo tem como objetivos (1) dar uma visão global da literatura sobre transformações de modelos (Seções 3.1 e 3.2), (2) abordar os detalhes de *Transformação de Grafos* (Seção 3.3), técnica para transformação de modelos adotada, e (3) apresentar a ferramenta AGG [15] (Seção 3.3.2), cujo *motor* de transformação de grafos é utilizado na implementação do *UnB-DALi*.

3.1 Noções Gerais

De forma geral, toda transformação de modelos é composta dos seguintes elementos (Figura 3.1):

- Modelos fonte (ou de entrada): modelos de onde as informações importantes para conduzir a transformação serão extraídas.
- Modelos alvo (ou de saída): modelos construídos após a transformação.
- Descrição: conjunto de regras de como um ou mais modelos fonte serão transformados em um ou mais modelos alvo, escritas na linguagem adotada pela transformação.
- Motor (ou *engine*, no termo em inglês): arcabouço tecnológico capaz de conduzir a transformação.

Modelos de entrada e saída possuem suas estruturas fundamentadas em forma de metamodelos: modelos que descrevem a estrutura e regras da linguagem na qual modelos são expressos. Metamodelos por sua vez possuem sua estrutura formalmente fundamentada

em meta-metamodelos, modelos os quais definem a estrutura de seu espaço tecnológico ¹ em termos dele próprio e.g. EBNF [33], MOF [29], etc.

Essa fundamentação sólida também se aplica à descrição da transformação, que deve sempre respeitar a estrutura sintática e semântica da linguagem adotada para a transformação. Todo motor deve então ser capaz de verificar se os modelos de entrada e saída estão conforme o que foi definido em seus respectivos metamodelos, ao mesmo tempo devendo ser capaz de ler a descrição da transformação e executar as regras de acordo com padrões identificados nos modelos de entrada.

3.2 Taxonomia para Transformação de Modelos

As técnicas existentes na literatura para a criação de um compilador e para a construção de um gerador de código, por exemplo, são inherentemente diferentes entre si, mesmo se tratando fundamentalmente do mesmo conceito (transformação de modelos). Como é extensa a lista de técnicas e ferramentas disponíveis para conduzir essas transformações [21], inconsistências na hora de implementar uma transformação de modelos podem facilmente advir. Logo, para um melhor entendimento dessa área importante da Engenharia de Software, uma classificação mais adequada das técnicas existentes de transformação de modelos se tornou necessária.

O artigo intitulado *Taxonomy of Model Transformation* de Mens *et al.* [38] classifica as técnicas existentes de transformação de modelos de acordo com três tópicos: (1) características dos modelos fonte e alvo; (2) características da transformação propriamente dita; e (3) características das linguagens ou ferramentas disponíveis, os quais detalhamos a seguir.

3.2.1 Características dos Modelos

As características específicas dos modelos de entrada e saída influenciam como uma transformação entre modelos deve ser conduzida. Nesse contexto, Mens *et al.* levam em consideração quatro dimensões:

- **Número de modelos fonte e alvo:** temos que as transformações podem ser classificadas de acordo com o número de modelos de entrada e o número de modelos de saída. Uma transformação de *merge* ², por exemplo, teria vários modelos de entrada e um modelo de saída.

¹tecnologias usadas para a representação de um modelo: formato de arquivos, estrutura de dados, gramática, etc

²combinação de modelos desenvolvidos em paralelo [38]

Tabela 3.1: Ortogonalidade entre as dimensões *edógenas vs. exógenas* e *verticais vs. horizontais*.

	horizontal	vertical
endógeno	<i>Refatoração</i>	<i>Refinamento Formal</i>
exógeno	<i>Migração</i>	<i>Geração de Código</i>

- **Espaço tecnológico:** os modelos fonte e alvo podem pertencer a um mesmo espectro tecnológico ou não. Ou seja, a fundamentação tecnologia por trás de como os modelos de entrada são estruturados não necessariamente precisa se repetir nos modelos de saída. Dessa forma, se a transformação a ser executada é entre espaços tecnológicos diferentes, a ferramenta precisa prover mecanismos de importação e exportação adequados para interligar esses espaços tecnológicos. Um exemplo de espaço tecnológico seria o *Meta Object Facility* (MOF) da OMG [29], por meio do qual metamodelos que constituem o UML são formalmente estruturados.
- **Transformações endógenas vs. exógenas:** as linguagens sobre as quais os modelos de entrada e saída estão descritos, se são as mesmas ou diferentes, denotam dois tipos de transformação diferentes. Quando as linguagens são a mesma, temos que a transformação é chamada de *endógena*; quando as linguagens são diferentes, temos que a transformação é dita *exógena*. Note que transformações endógenas terão sempre o mesmo espaço tecnológico. Transformações exógenas, por sua vez, podem ser conduzidas tanto no mesmo espaço tecnológico quanto em espaços tecnológicos diferentes. Um exemplo do primeiro seria migrar um código escrito em C++ para um código escrito em C pois, apesar de serem linguagens diferentes, possuem um mesmo contexto tecnológico: EBNF. Um exemplo do segundo seria a transformação de um Diagrama de Atividades UML para um modelo passível por Model Checking (ver Capítulo 5).
- **Transformações verticais vs. horizontais:** o nível de abstração em que os modelos se situam caracterizam dois tipos diferentes de transformações. Quando o nível é o mesmo, temos uma transformação *horizontal*; já quando o nível é diferente, temos uma transformação *vertical*. Exemplo clássico de uma transformação horizontal é refatoração de modelos, onde os objetivos podem variar desde eliminação de redundância à melhorias na legibilidade dos mesmos. Uma transformação vertical, por sua vez, se caracteriza mais comumente pela geração de código fonte a partir de modelos de alto nível.

A Tabela 3.1 ilustra a ortogonalidade que existe entre as dimensões *edógenas vs. exógenas* e *verticais vs. horizontais*. Observe que *Refinamento Formal* pode denotar a

operação de um refinamento gradual sobre especificações de software descritas em lógica proposicional de primeira ordem ou teoria de conjuntos (adicionando-se mais axiomas, por exemplo).

3.2.2 Características da Transformação

A transformação em si possui características específicas que podem e devem ser levadas em consideração na classificação das técnicas de transformação de modelos. Elas são:

- **Nível de automação:** transformações de modelos podem ser conduzidas de forma totalmente autônoma, de forma totalmente manual ou com uma certa quantidade de intervenções manuais. Transformações que não podem ser feitas de forma automática geralmente tem em comum uma necessidade por heurísticas semânticas que somente um humano envolvido nesse processo poderia fornecer; a transformação de um documento de requisitos para um modelo de análise é um exemplo desse último cenário.
- **Complexidade da transformação:** o volume de trabalho e esforço computacional envolvido numa transformação de modelos pode ser pequeno (*e.g.* refatoração) ou grande (*e.g.* compilação). Quando essa diferença é muito grande, o conjunto de técnicas aplicáveis a cada caso é completamente diferente.
- **Preservação:** tipos diferentes de preservação de informação entre os modelos fonte e alvo se aplicam a tipos diferentes de transformação. Por exemplo, o comportamento do modelo é o que precisa ser preservado em um refatoramento. Já em um refinamento, temos que a correção é que precisa ser preservada.

3.2.3 Características das Linguagens ou Ferramentas

As linguagens e ferramentas também possuem características próprias que afetam a decisão de um desenvolvedor no momento da implementação de uma transformação entre modelos. Mens *et al.* subdividiram essas características em três macro tópicos: (1) critérios de sucesso, (2) requisitos de qualidade e (3) mecanismos de transformação.

Critérios de Sucesso para uma Linguagem ou Ferramenta de Transformação de Modelos

As propriedades funcionais das ferramentas ou linguagens de transformação de modelos mudam a maneira com que essas transformações são conduzidas. Abaixo encontram-se as características com as quais um pode classificar uma ferramenta ou linguagem:

- Capacidade para criar, ler, atualizar ou excluir transformações.
- Habilidade para sugerir quando uma transformação deve ser aplicada.
- Capacidade de se customizar ou reusar transformações.
- Existência de meios de conduzir uma verificação formal e assim garantir a correção (sintática e/ou semântica) das transformações.
- Existência de meios para o teste e validação das transformações.
- Capacidade para lidar com modelos inconsistentes ou incompletos.
- Habilidade de agrupar, compor e decompor transformações, com o intuito de melhorar a legibilidade, a modularidade e a manutenibilidade das regras de uma transformação.
- Ser capaz de especificar transformações genéricas e de alta ordem. Ou seja, ser capaz de especificar transformações que aceitam transformações como modelos de entrada ou saída, permitindo assim aplicações como refatoramento e otimização de transformações.
- Suporte a transformações bidirecionais, ou seja, para um transformação T_{\rightarrow} com conjunto de modelos de entrada X e conjunto de modelos de saída Y , existe uma transformação T_{\leftarrow} tal que dado Y como entrada, temos X como saída.
- Suporte para rastreabilidade e propagação de mudanças. Ou seja, capacidade para manter uma relação de como exatamente um modelo alvo foi obtido a partir de um modelo fonte.

Requisitos de Qualidade para uma Linguagem ou Ferramenta de Transformação de Modelos

Propriedades não-funcionais das ferramentas ou linguagens também influenciam na classificação de uma transformação de modelos. Elas são:

- **Usabilidade e utilidade:** qualquer ferramenta de transformação de modelos necessita ser *usável* e *útil*, ou seja, a experiência do usuário precisa ser eficiente e intuitiva assim como a ferramenta precisa servir a um propósito prático.
- **Verbosidade e concisão:** uma ferramenta ou linguagem de transformação pode ser classificada de acordo com seus níveis de verbosidade ou concisão. Uma linguagem ou ferramenta concisa fornece um número mínimo de elementos sintáticos, o que pode trazer impactos negativos na construção de transformações mais complexas, por exemplo. Já uma linguagem ou ferramenta verbosa introduz símbolos e

operadores sintáticos extra (*syntactic sugar*, no termo em inglês), podendo dificultar o entendimento da transformação por parte de terceiros, por exemplo.

- **Desempenho e escalabilidade:** é esperado que toda ferramenta de transformação de modelos precise responder bem, em termos de desempenho, a modelos grandes e complexos.
- **Extensibilidade:** mecanismos de extensão (*e.g. plugins*) podem ou não ser fornecidos pela ferramenta.
- **Fundamentação matemática:** caso a ferramenta de transformação possua uma fundamentação matemática, provas diversas a respeito da transformação podem ser conduzidas.
- **Público alvo:** divergências podem surgir, entre os pontos de vista teórico e pragmático, sobre qual a melhor ferramenta ou linguagem de transformação de modelos para determinada audiência. Ferramentas podem então ser classificadas de acordo com seu público alvo. Nesse contexto, não se recomendaria, por exemplo, uma linguagem para transformação em linguagem funcional para um público especializado em linguagens orientadas a objetos.
- **Padronização:** ferramentas/linguagens de transformação de modelos podem ser classificadas de acordo com o fornecimento de suporte ou não aos padrões industriais conhecidos (*e.g. UML [30]*).

Mecanismos para Transformação de Modelos

Todo motor de transformação de modelos é adepto de um determinado mecanismo de transformação. De forma geral, os mecanismos existentes se dividem em uma abordagem *declarativa* ou em uma abordagem *imperativa*.

Mecanismos declarativos forçam a especificação da transformação de maneira estática. Ou seja, o foco é em declarar de forma explícita o que exatamente precisa ser transformado em que, estabelecendo assim uma relação estrutural entre os modelos de entrada e saída. Esse tipo de mecanismo possui a capacidade de prover serviços de rastreabilidade de forma muito natural, dado a existência de um motor de transformação subjacente. Esse motor consegue manter registro de quais fragmentos do modelo de saída resultaram a partir da aplicação das regras associadas a padrões identificados nos modelos de entrada; dependendo da complexidade da transformação, porém, mecanismos declarativos podem ter seu desempenho comprometido.

Mecanismo imperativos, por sua vez, forçam a especificação da transformação num paradigma dinâmico. Nessa categoria, a construção dos modelos de saída é feita a partir

do estabelecimento de uma iteração bem definida de passos a serem executados sobre os modelos de entrada. Esse tipo de mecanismo permite, portanto, um maior controle sobre qual é exatamente a ordem de execução dos passos de transformação, o que pode ser importante para lidar com transformações de modelos muito complexas; nessa categoria, porém, o desenvolvedor pode se sentir desencorajado a implementar serviços de rastreabilidade, por exemplo, já que não há o suporte de um subjacente motor de transformação.

3.3 Transformação de Grafos

Como visto no capítulo anterior, o método desenvolvido por Rodrigues *et al.* é altamente dependente da transformação de um modelo que descreve o comportamento de um sistema (*UML Interaction Overview Diagram* [30]) para um modelo que seja passível de análise por uma ferramenta de *Model Checking* (PRISM). Vimos na seção anterior que na literatura sobre *Transformação de Modelos* é possível encontrar demasiadas formas de se alcançar esse objetivo; não serão todas, porém, que se encaixarão em possíveis soluções para o problema que aqui queremos resolver, pois:

1. Queremos facilitar a condução futura de uma verificação formal da correção sintática das transformações definidas pela biblioteca, contribuindo para uma elevada confiança no resultado de um processo de análise por meio deste método;
2. Não queremos nos limitar à sintaxe concreta de modelos UML, o XMI, dado que não foram encontradas ferramentas que implementem em completo a especificação da UML e não há garantia alguma de que tais ferramentas implementem corretamente essa especificação. Um simples teste demonstra isso: a ferramenta Modelio [26], dita compatível com UML 2.5, não conseguiu em um de nossos testes importar corretamente um modelo UML exportado pela própria ferramenta;
3. Precisamos garantir que a biblioteca *UnB-DALi* seja altamente reusável e leve, ou seja, não podemos *limitar* seu funcionamento a arquitetura de apenas uma ferramenta de modelagem;

É com base na restrição 1 acima que escolhemos o método de *Transformação de Grafos* [15] como linguagem para a transformação *UML* → *PRISM*; nesse contexto, escolhemos a ferramenta *AGG* [9] como arcabouço tecnológico para a implementação do *UnB-DALi*, cumprindo assim os requisitos 2 e 3.

3.3.1 O Método de Transformação de Grafos

Um dos grandes argumentos em prol do uso de transformação de grafos para transformação de modelos consiste na observação de que todo modelo é naturalmente representado como um grafo (Definição 4) [38]. Dessa forma, em uma transformação de modelos por meio da técnica em questão, os modelos fonte e os modelos alvo (Seção 3.1) são dados em termos dos seus grafos subjacentes. Resumidamente, esse método consiste em três passos:

1. obter um grafo G_s subjacente a um modelo de entrada;
2. transformar G_s de acordo com certas regras de transformação, obtendo assim G_t ;
3. obter o modelo de saída a partir de G_t ;

Explorando em maiores detalhes o passo 2 acima, temos que uma transformação de modelos pode ser definida de forma precisa por um sistema de transformação de grafos $GTS = (T, R)$, onde T é um *grafo-tipo* (Definição 5) e R um conjunto de regras de transformação (Definição 7). Temos também que os grafos G_s e G_t são ditos *tipados*, respectivamente, sobre os grafos-tipo T_S e T_T , onde: (1) $\mathbf{T}_S \subseteq \mathbf{T} \supseteq \mathbf{T}_T$ e (2) $\mathbf{T}_S \cup \mathbf{T}_T \subseteq \mathbf{T}$.

Definição 4 (Grafo [16]). *Um Grafo $G = (V, E, s, t)$, consiste de:*

- V , um conjunto de vértices;
- E , um conjunto de arestas: pares ordenados entre dois vértices;
- $s : E \rightarrow V$, função que recupera o vértice origem de uma aresta;
- $t : E \rightarrow V$, função que recupera o vértice destino de uma aresta;

Definição 5 (Grafo-tipo [16]). *Um Grafo-Tipo T trata-se de um grafo especial onde V define os **tipos de vértices** e E os **tipos de arestas** que grafos instância podem assumir (Definição 6). As arestas têm o papel auxiliar de elucidar quais relacionamentos são permitidos entre os vértices dos grafos tipados sobre T .*

Definição 6 (Grafo Tipado [16]). *Um Grafo Tipado, ou de Instância, G^T é uma tupla da forma $(G, type)$, onde G é um grafo, $type : G \rightarrow T$ é um morfismo entre grafos (Definição 8), e T é um grafo-tipo.*

Definição 7 (Regra de Transformação [19]). *Uma produção, ou regra de transformação, $r : L \rightarrow R$ consiste de um par de grafos (L, R) tipados sobre um grafo-tipo T , de forma que $L \cup R$ esteja sintaticamente bem definido em T . O lado esquerdo, ou LHS, L representa as pré-condições da aplicação da regra, enquanto o lado direito, ou RHS, R representa as pós-condições da aplicação da regra. Dessa forma, $L \setminus (L \cap R)$ define o subgrafo que será deletado quando da aplicação de r ; $R \setminus (L \cap R)$ define o subgrafo que será criado após a aplicação da regra; e $L \cap R$ define um subgrafo invariante dentro do contexto dessa regra.*

Definição 8 (Morfismo entre Grafos [16]). *Dado dois Grafos tipados $G_i^T = (G_i, type_i)_{i \in \{1,2\}}$, um morfismo entre grafos $f : G_1 \rightarrow G_2$, $f = (f_V, f_E)$ consiste de duas funções $f_V : V_1 \rightarrow V_2$ e $f_E : E_1 \rightarrow E_2$ que preserva as funções de origem e destino de um grafo, i.e. $f_V \circ s_1 = s_2 \circ f_E$ e $f_V \circ t_1 = t_2 \circ f_E$.*

Temos então, que o processo de transformação entre grafos (ver Figura 3.2) é caracterizado pela aplicação sequencial, não-determinística, de regras r_i apropriadas a partir de um grafo fonte $G_S^{T_S}$ ($T_S \subseteq T$). Grafos intermediários G_i^T surgem como resultado da aplicação dessas regras. Caso não existam mais regras aplicáveis a G_i^T e caso exista $type : G_i^T \rightarrow T_T$, um morfismo entre o grafo intermediário G_i com o grafo-tipo T_T (onde $T_T \subseteq T$), temos que a transformação é encerrada, dita sintaticamente correta e $G_i = G_T^{T_T}$ [19].

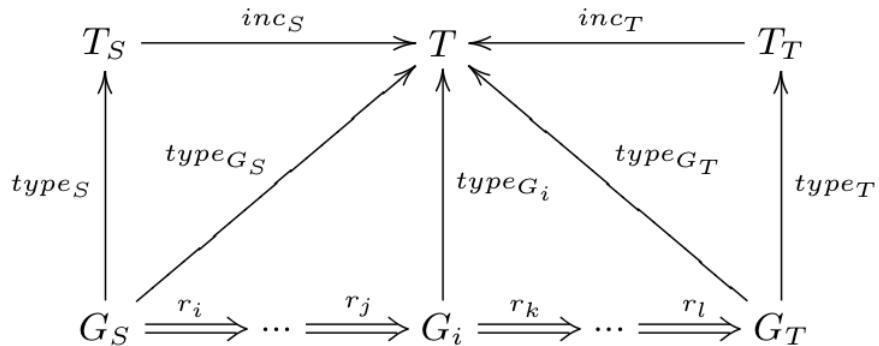


Figura 3.2: Processo de transformação de grafos e seus tipos. [19]

Para uma regra $r_i : L^T \rightarrow R^T$ poder ser aplicada a um determinado grafo G^T , é preciso que um morfismo $m : L^T \rightarrow G^T$ (correspondência, ou *match* no termo em inglês) exista entre L^T e G^T [15]. Esse procedimento é correlato ao *pattern-matching* inerente a linguagens de programação funcional. É possível, portanto, que várias regras possam ser aplicadas ao mesmo tempo. Geralmente é dado preferência à primeira regra que

for verificado que m existe; técnicas como regras prioritárias ou aplicação de regras em camadas existem para melhorar o controle sobre esse tipo de não-determinismo [19].

Regras também são permitidas ter *condições negativas de aplicação* (NACs³) dado por um morfismo $nac : L \rightarrow N$. Nesse caso, uma regra r somente pode ser aplicada a partir de uma correspondência $m : L \rightarrow G$ se não existirem morfismos injetivos da forma $q : N \rightarrow G$ tais que $q \circ nac = m$. Ou seja, NACs são morfismos associados ao LHS de uma regra r que proibem a aplicação dessa regra em G caso a imagem do morfismo, N , seja subgrafo de G . NACs permitem assim uma maior flexibilidade e controle na definição de regras de transformação por parte do desenvolvedor.

Algumas observações são pertinentes:

- Os grafos e grafo-tipos aqui trabalhados assumem, implicitamente, a presença de atributos em seus vértices e arestas (da mesma forma que classes em Java). As versões das definições dadas acima que trabalham explicitamente com atributos exigem o conhecimento de estruturas algébricas mais avançadas⁴, extrapolando assim o escopo deste trabalho; maiores detalhes podem ser obtidos em [15].
- O conceito de aplicação de regras adotado nesse trabalho trata-se do *single-pushout approach* (SPO), onde não há a presença de um *grafo-cola* entre o LHS e o RHS de uma regra de transformação. A teoria de transformação de grafos de Taentzer *et al.* [15], no entanto, é elaborada com base no *double-pushout approach* (DPO) que, por meio da presença de um *grafo-cola* em suas produções, permite a invertibilidade da aplicação de uma regra; exceto por essa propriedade, tudo que pode ser feito com DPO pode ser feito através de SPO.

Concluindo, temos que um sistema de transformação de grafos (GTS) possui sua teoria matemática fundamentada na área de categorias algébricas, fazendo parte das categorias *HLR adesivas* [15]. Para um GTS qualquer valem (1) o Teorema da Confluência Local e (2) o Teorema do Paralelismo e Concorrência [15], caracterizando assim um GTS como um sistema de reescrita. Por fim, a verificação de uma confluência global⁵ pode ser feita a partir de uma análise de pares críticos da forma como estabelecida em [32].

3.3.2 A Ferramenta AGG

Na Seção 3.3, foram relatados algumas restrições para a condução da transformação de modelos que queremos realizar neste trabalho. Temos que a ferramenta AGG atende adequadamente às restrições impostas, pois:

³do termo em inglês *negative application conditions*

⁴assinaturas algébricas, teoria de tipos, entre outros

⁵a ordem dos fatores não altera o resultado

1. é rigorosamente fundamentada sobre o formalismo de transformação de grafos explorado na Seção 3.3.1, o que possibilitou, entre outros detalhes, a implementação dos mecanismos previstos na literatura para a verificação da correção sintática de um sistema de transformação de grafos (por meio da análise de pares críticos) [19].
2. é independente de ferramentas de modelagem ou de desenvolvimento integrado (IDEs), aderindo ao *espaço-tecnológico* (conforme taxonomia da Seção 3.2) do GXL (padrão *quasi*-definitivo para o intercâmbio de grafos) [9].
3. disponibiliza seu motor Java de transformação de grafos (~ 1.6 MB) para a comunidade, permitindo assim o reuso de sua infraestrutura para a condução de transformações de modelos por meio do formalismo de transformações de grafos.

Visão Geral da Ferramenta

Sistemas de transformação de grafos, da forma como discutido na Seção 3.3.1, podem ser descritos e manipulados por meio do ambiente de programação do AGG. Grafos no AGG são definidos conforme um *grafo-tipo*, podendo seus nós e vértices possuir qualquer tipo de atributo Java (tanto os tipos fornecidos como *default* pelo Java, quanto tipos fornecidos pelo usuário). Os tipos permitidos de atributos para um nó ou arco precisam ser explicitamente definidos no *grafo-tipo* em questão. Dessa forma, as regras de uma transformação de grafos podem vir equipadas com computações arbitrárias nesses objetos Java, fornecendo portanto toda a flexibilidade de uma linguagem de programação usual.

No contexto da linguagem AGG [4], temos que:

- **Programa:** consiste de uma *grafo-gramática* (do termo em inglês *graph-grammar*) e um conjunto de classes Java definidas pelo usuário.
- **Grafo-gramática:** estrutura contendo um *grafo-tipo*, um conjunto de *grafos* (representando modelos de entrada ou de saída) e um conjunto de *regras*, podendo conter NACs 3.3.1 ou não.
- **Grafo:** dois conjuntos disjuntos de *nós* e *arestas*, também chamados de *objetos* de um grafo.
 - cada *objeto* possui um rótulo único, onde esse rótulo é chamado de tipo desse objeto pelo motor do AGG. Dessa forma, se um objeto o possui rótulo l , dizemos que o é do tipo l .
 - toda aresta possui sua própria identidade, permitindo assim a existência de vários arcos, do mesmo tipo, entre nós.

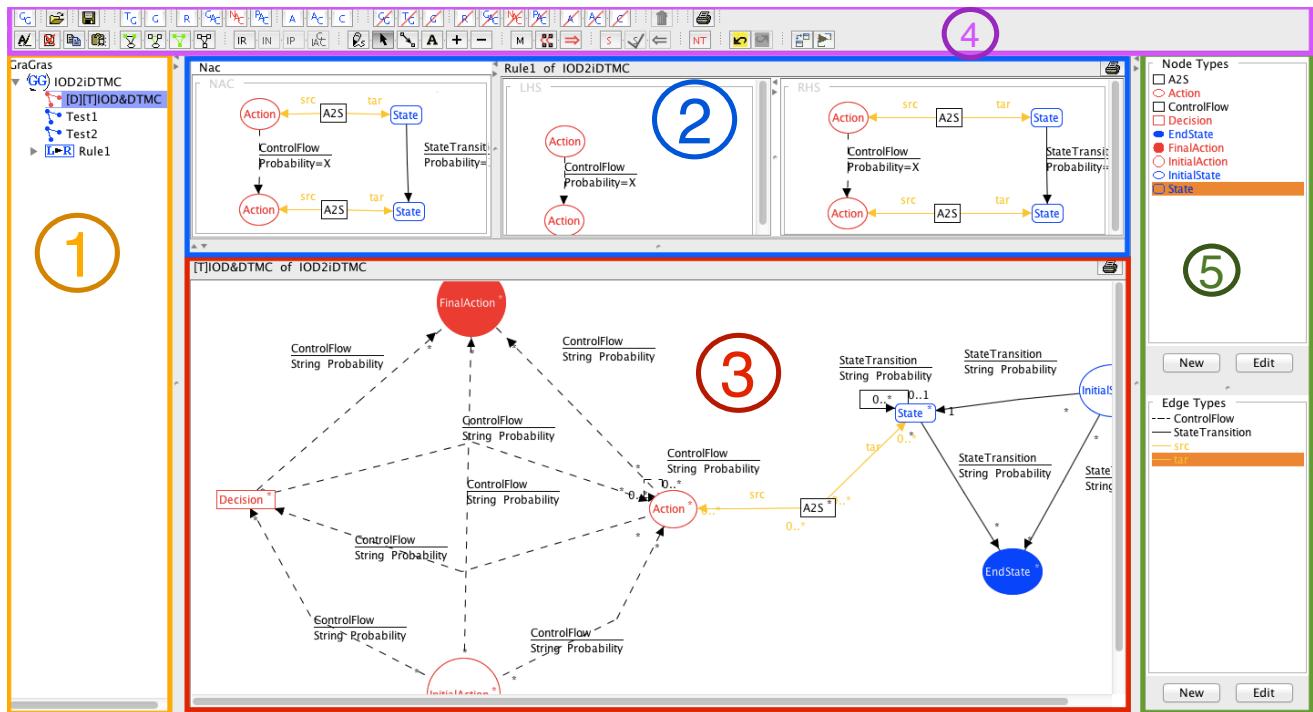


Figura 3.3: Visão geral do ambiente de programação do AGG.

- **Atributo:** assim como qualquer outro atributo de uma classe Java, possui um **nome**, um **tipo** e um **valor**, esse último opcional para grafos-tipo e obrigatório para grafos instância.
- **Ação:** um par de dois grafos (LHS e RHS) tipados sobre o grafo-tipo da gramática que modelam os estados anterior e posterior a um passo de transformação.

Editor Gráfico

Considerando a Figura 3.3 e fazendo um paralelo com a metodologia geral de transformação de modelos definida na Seção 3.1, temos que:

- o *modelo fonte* é o grafo que por último for selecionado na região de seleção (número 1 na figura). Dentro da região 1, grafos são identificados por .
- a *descrição* de uma transformação é formada pelo conjunto de regras **habilitadas** na região de seleção (regras podem ser desabilitadas a revelia do desenvolvedor). Tais regras de transformação são definidas e manipuladas de forma visual na região 2 da figura, onde as manipulações dos atributos dos elementos de um grafo por expressões Java é feito de forma textual também na região 2. Regras são identificadas por .

- os metamodelos dos modelos de entrada e saída são definidos por um *type-graph* (grafo-tipo, conforme Definição 5) identificados dentro do ambiente gráfico pelo ícone
- um *modelo alvo* é obtido após a aplicação de uma transformação em um modelo fonte.
- o motor de transformação é invisível ao usuário da interface gráfica. Esse é porém *open source*, distribuído como uma biblioteca Java e disponível através do site do AGG [39]. Sua documentação no estilo *Javadoc*, que acompanha seu arquivo de distribuição, é a única fonte de informação para o uso adequado do mesmo.

Ainda considerando a Figura 3.3, temos que:

- a região 3 é onde os grafos-tipo e os grafos de instância são desenhados. Quando um grafo ou grafo-tipo é selecionado na região 1, a região 3 é sensibilizada para mostrar a estrutura desse grafo.
- a região 4 é composta pelos vários comandos existentes para a manipulação de objetos pertencentes as diferentes regiões.
- os tipos existentes de vértices e arestas são definidos na região 5.

Tabela 3.2: Taxonomia da Ferramenta AGG proposta por Mens *et al.* em [38].

critério	classificação
número de modelos fonte e alvo	um para um
tipo da transformação	endógeno
espaço tecnológico	GXL
verificação e validação	análise de pares críticos
usabilidade	mediana
extensibilidade	inexistente
aceitação	acadêmica
padronização	GXL
interoperabilidade	alta

Finalizando, temos que o ambiente de programação do AGG possui tanto componentes visuais quanto textuais (vide expressões Java para manipulação de atributos); dessa forma, é considerado um ambiente de programação híbrido, o tornando assim mais flexível e reusável que outras ferramentas de transformação de grafos. AGG [39] é inteiramente implementado em Java, desenvolvido e mantido pelo grupo de pesquisas em

Grafo-gramáticas da *Technical University of Berlin* (TU Berlin) e supervisionado pela Prof. Dra. Gabriele Taentzer (agora na Universidade de Marbugo). Por fim, uma classificação de acordo com a taxonomia definida na Seção 3.2 é dada à ferramenta AGG pela Tabela 3.2.

3.4 Considerações Finais

Neste capítulo, uma visão geral da literatura sobre transformação de modelos foi dada ao leitor, abordando desde o que se entende por uma transformação de modelos, numa perspectiva global, à taxonomia existente na literatura para sistemas de transformação de modelos. O mecanismo de transformação de grafos foi abordado com maiores detalhes e foi introduzido ao leitor o AGG, ferramenta que implementa de forma apropriada esse formalismo.

No próximo capítulo, um resumo de dois trabalhos relacionados será dado com o intuito de elucidar o contexto tecnológico ao qual o UnB-DALi está inserido.

Capítulo 4

Trabalhos Relacionados

Com vistas a explorar a literatura tecnológica na qual a biblioteca *UnB-DALi* está inserida, ou seja, ferramentas e conceitos existentes para a transformação de modelos, seja para análise de dependabilidade ou não, temos que este capítulo introduz ao leitor dois trabalhos relacionados: a ferramenta *U-MarMo* e o trabalho de Grønmo *et al.*

4.1 *U-MarMo*

U-MarMo [5] (*UML to Markov Models*¹) consiste de um passo de transformação de modelos UML anotados com informações quantitativas para modelos passíveis de *Model Checking*, com o objetivo de verificação formal de requisitos não-funcionais. Na versão atual, apenas Diagramas de Sequência desenhados na ferramenta *MagicDraw* são suportados [30]. Os modelos de saída possíveis são DTMCs para verificações de confiabilidade, e CTMCs para verificações de performance.

O motor de transformação do *U-MarMo* é construído com base em um mecanismo imperativo, implementado em Java, que busca padrões de conversão diretamente no XMI [13] do modelo de entrada. Esse *pattern-matching* ocorre de forma que primeiramente obtém-se uma lista ordenada das mensagens trocadas entre as linhas de vida de um SD por meio do método *RetrievePerformableActions*. Após a obtenção dessa lista, o algoritmo de transformação é mais facilmente aplicável tendo a certeza de que as transições do DTMC ou CTMC resultante ocorrerão na ordem correta. A transformação proposta por essa ferramenta lida com a presença de fragmentos nas linhas de vida, além de ter suporte para mensagens síncronas e assíncronas.

Com base nos critérios de classificação de Mens *et al.*, poderíamos classificar essa ferramenta da forma como estabelecidade na Tabela 4.1.

¹UML para Modelos Markovianos, numa tradução livre

Tabela 4.1: Classificação da Ferramenta *U-MarMo* conforme taxonomia proposta por Mens *et al.*

critério	classificação
número de modelos fonte e alvo	um para um
tipo da transformação	exógeno
espaço tecnológico	XMI e PRISM
nível de automação	completamente autônomo
verificação e validação	inexistente
usabilidade	baixa
extensibilidade	inexistente
aceitação	baixa
padronização	MagicDraw
interoperabilidade	baixa

4.2 Grønmo *et al.*

O trabalho produzido por Grønmo *et al.* propõe a transformação de SDs para máquinas de estado por meio da técnica de transformações de grafos. A ferramenta usada para a implementação das transformações foi o AGG [39].

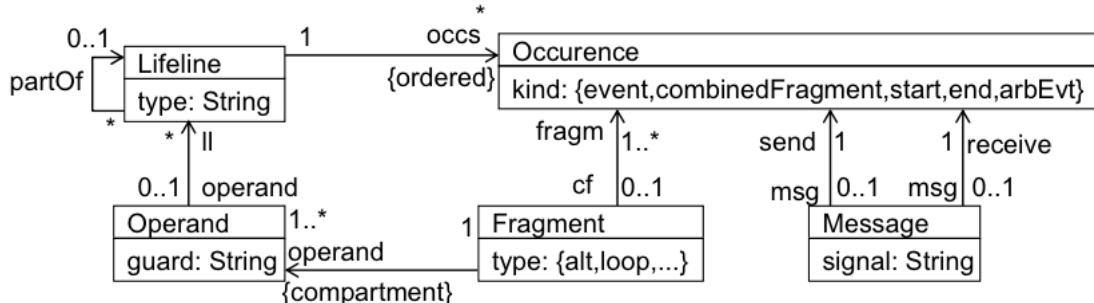


Figura 4.1: Metamodelo minimalistico de um Diagrama de Sequência proposto por Grønmo *et al.*

De forma resumida, o trabalho primeiramente propõe um metamodelo minimalistico de um SD, compatível com uma representação de grafos-tipo do AGG, onde é possível determinar uma troca de mensagens assíncronas entre linhas de vida além de determinar a presença de fragmentos nas mesmas (Figura 4.1). Numa perspectiva compatível com os conceitos trabalhados no Capítulo 3, temos que a representação de um SD em conformidade com o metamodelo acima é considerado pelos autores do trabalho como a sintaxe abstrata desse mesmo SD. Nessa perspectiva, entende-se como sintaxe concreta a notação de um SD conforme o que determina a especificação do UML.

Dessa forma, o trabalho prossegue por propor uma ferramenta que mapeia regras de transformações definidas sobre a sintaxe concreta de um SD para regras de transformações definidas sobre a sintaxe abstrata do mesmo. É argumentado ser mais didático e de fácil evolução a proposição dessas regras de transformação com base na sintaxe concreta. Observamos, porém, que isso vem ao custo de restringir toda a transformação a um específico arquivo de descrição de modelos em formato XMI. Posteriormente, esse mapeamento é formalizado.

O trabalho desenvolvido por Grønmo *et al.* garante dessa forma rigorosidade matemática na descrição das transformações. Peca em interoperabilidade porém, dado que ainda é preciso para cada ferramenta de modelagem reescrever completamente a transformação. A Tabela 4.2 classifica esse trabalho conforme a taxonomia proposta por Mens. *et al.*

Tabela 4.2: Classificação do trabalho de Grønmo *et al.* conforme taxonomia proposta por Mens *et al.*

critério	classificação
número de modelos fonte e alvo	um para um
tipo da transformação	exógeno
espaço tecnológico	XMI
verificação e validação	análise de pares críticos
nível de automação	completamente autônomo
aceitação	acadêmica
padronização	MagicDraw
interoperabilidade	baixa

4.3 Considerações Finais

Neste capítulo foram brevemente retratados dois trabalhos relacionados ao tema desta monografia, que se utilizam de duas metodologias diferentes para a condução de uma transformação de modelos: uma imperativa e outra declarativa.

No próximo capítulo abordaremos com detalhes a arquitetura e implementação do *UnB-DALi*.

Capítulo 5

UnB-DALi

Como visto nos capítulos anteriores, uma análise de dependabilidade como proposta por Rodrigues *et al.* necessita de uma automatização do processo de transformação de um modelo comportamental UML para um modelo DTMC próprio à condução de um *model checking*. Esforços [5] [35] foram conduzidos ao longo dos anos para resolver problemas diversos de transformação de modelos em análises de dependabilidade. A maioria dessas soluções, porém, foram desenvolvidas em escopo fechado, atendendo somente a situações específicas do problema em resolução. Isso se deu, muitas vezes, a escolha do espaço tecnológico dos modelos de entrada para aplicação das rotinas de transformação de modelos, que muitas vezes impõe restrições tecnológicas complicadas de superar caso não se dê o tempo necessário para ponderar sobre possíveis alternativas de solução.

Mais comumente, o espaço tecnológico escolhido para os modelos de entrada numa transformação de modelos é o XMI [13] (*XML Metadata Interchange*) ¹, uma especificação da OMG [31] para o intercâmbio ² de modelos (diagramas de classes, diagramas de sequência, etc) aderentes a um metamodelo em conformidade com MOF [29] (UML, SysML, entre outros). Com essa especificação, a OMG esperava que fosse possível o reaproveitamento de modelos entre ferramentas diferentes de modelagem (ou mesmo versões diferentes da mesma ferramenta), permitindo assim liberdade na escolha do ambiente de modelagem por parte do arquiteto e do time de desenvolvimento de um sistema. É sabido, porém, que ainda hoje essas ferramentas não se adequaram a esse padrão num patamar minimamente aceitável, dado diversas ambiguidades e inconsistências na especificação em questão [22]. A partir disso, observamos que talvez seja ingênuo implementar as rotinas de transformação de modelos com base na organização do arquivo XMI, dado que:

1. não há garantias de que essa organização se manterá válida por tempo aceitável;

¹XML Intercâmbio de Metadados, em português

²Intercâmbio por meio de arquivos de persistência descritos em XML [41], padrão da W3C para o representação em texto de dados estruturados

2. não há garantias de que essas rotinas sejam de fácil *reaproveitamento* por outras ferramentas de modelagem.

Com isso em vista, esse capítulo elabora em detalhes os requisitos, a arquitetura e a implementação do *UnB-DALi*, biblioteca que implementa a automatização do passo de transformação UML-PRISM de forma desacoplada de arquivos de persistência de modelos (como XMI), permitindo assim uma maior flexibilidade para a evolução e reuso das transformações de modelos implementadas. Testes para validação da biblioteca e discussões sobre a experiência com as ferramentas utilizadas e sobre os testes conduzidos são dados ao final deste capítulo.

5.1 Requisitos

A partir das observações do *caput* deste capítulo e com base no domínio tema desta monografia (análise de dependabilidade) precisamos desenvolver uma biblioteca que atenda primordialmente aos seguintes requisitos funcionais:

1. Demonstre rigor matemático na descrição e condução das transformações;
2. Facilite a condução de provas formais sobre a correção sintática das transformações a serem realizadas; e
3. Trabalhe as transformações de forma **independente** de arquivos de descrição de modelos: *xmi*, *xml*, etc.

É preciso também que a biblioteca seja desenvolvida de forma que os seguintes requisitos não-funcionais sejam atendidos:

1. Possua nível alto de manutenibilidade, reusabilidade e usabilidade;
2. Apresente facilidade de integração com as diversas ferramentas de modelagem existentes; e
3. Seja de fácil uso.

5.2 Arquitetura e Implementação

Para aderir aos requisitos listados na Seção 5.1, temos que foram adotadas as seguintes estratégias:

1. Adotamos a linguagem Java para implementação da biblioteca dado que a maioria das ferramentas de modelagem também adotam-na como linguagem de programação. Como consequência, essa escolha facilita a integração entre essas ferramentas e o *UnB-DALi*, dado a existência de mecanismos de extensão como *plugins*, por exemplo. Ao mesmo tempo, Java também auxilia na manutenibilidade, reusabilidade e usabilidade dessa biblioteca, pois, por meio de Interfaces, Classes Abstratas, Heranças e Polimorfismo (estruturas e princípios básicos de orientação a objetos), é possível definir bem o escopo e o comportamento esperado dos componentes que fazem parte do UnB-DALi. Isso faz com que o próprio código da solução se torne documentação para sua eventual evolução, horizontal ou vertical. Dessa forma, os requisitos não-funcionais 1 e 2 são atendidos.
2. Por ser construída em cima do formalismo de sistemas de transformação de grafos, possuindo assim propriedades matemáticas interessantes, adotamos a ferramenta AGG (Capítulo 3) como motor de transformação de modelos. Dessa forma, atendemos facilmente os requisitos funcionais 1 e 2.
3. Metamodelagens minimalistas dos modelos de entrada e saída, contendo somente as estruturas e informações necessárias para uma análise de dependabilidade, foram criadas como classes Java na biblioteca. Dessa forma, temos que transformações de modelos são conduzidas sobre instâncias dessas classes e não diretamente sobre arquivos de descrição de modelos, situando a biblioteca em conformidade, portanto, com o requisito funcional 3. Essas metamodelagens também encapsulam detalhes da estrutura de grafos usada pelo motor de transformação subjacente, de forma que fique mais intuitivo a definição de certos componentes semânticos, facilitando assim o uso da ferramenta (requisito não funcional 3 acima).

5.2.1 Arquitetura

Como consequência das estratégias adotadas para implementação da biblioteca, temos que ferramentas externas que desejem fazer o uso do *UnB-DALi* para transformação de modelos precisam, inevitavelmente, implementar a lógica de construção das instâncias dos metamodelos presentes na biblioteca. Naturalmente, essa construção se dará a partir de um mapeamento das informações contidas em arquivos de descrição de modelos para respectivas instâncias das classes que definem seus metamodelos minimalistas, conforme Figura 5.1. Com isso, a base de código da biblioteca é simplificada, dado não ser mais preciso oferecer suporte às inúmeras diferentes formas de persistência de modelos, que variam de acordo com a ferramenta de modelagem adotada pelo arquiteto/analista do sistema.

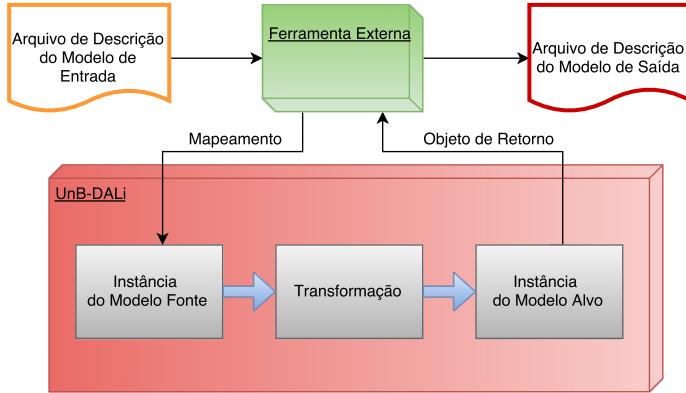


Figura 5.1: Visão geral do comportamento esperado da biblioteca *UnB-DALi*.

Como o motor para transformações de modelos escolhido foi o AGG, da biblioteca exigiu-se que:

1. todos os metamodelos minimalistas possuam internamente uma estrutura que defina esse metamodelo em termos de um grafo-tipo do AGG (classe *TypeGraph*);
2. todas as instâncias desses metamodelos possuam representações internas em termos de um grafo do AGG (classe *Graph*);
3. todos os elementos que definem um modelo possuam representações internas em termos das estruturas de nós ou arestas do AGG (classes *Node* e *Arc*), respectivamente.

Dessa forma, para orientar a implementação e evolução dos metamodelos da biblioteca e auxiliar no reuso de código, esses metamodelos diversos precisam seguir os contratos estabelecidos por três classes abstratas fundamentais (ver Figura 5.2): *AbstractAggModel*, que define a estrutura e comportamento de modelos passíveis de transformação via AGG a partir de representações internas de um Grafo-Tipo e um Grafo do AGG; *AbstractAggNode* que define a estrutura e comportamento esperado dos elementos de um metamodelo que forem identificados como uma estrutura de nó do AGG; *AbstractAggEdge* que define a estrutura e comportamento de elementos de um metamodelo que forem identificados como uma estrutura de aresta do AGG.

Uma estrutura especial para a definição de uma transformação de modelos via AGG também foi definida. A classe abstrata *LayeredAggTransformation* (Figura 5.2) define o comportamento e estrutura esperados de uma transformação de modelos que utiliza internamente o motor do AGG, distinguindo de forma mais clara o que se espera entre de um modelo de entrada e o que se espera de um modelo de saída em uma transformação.

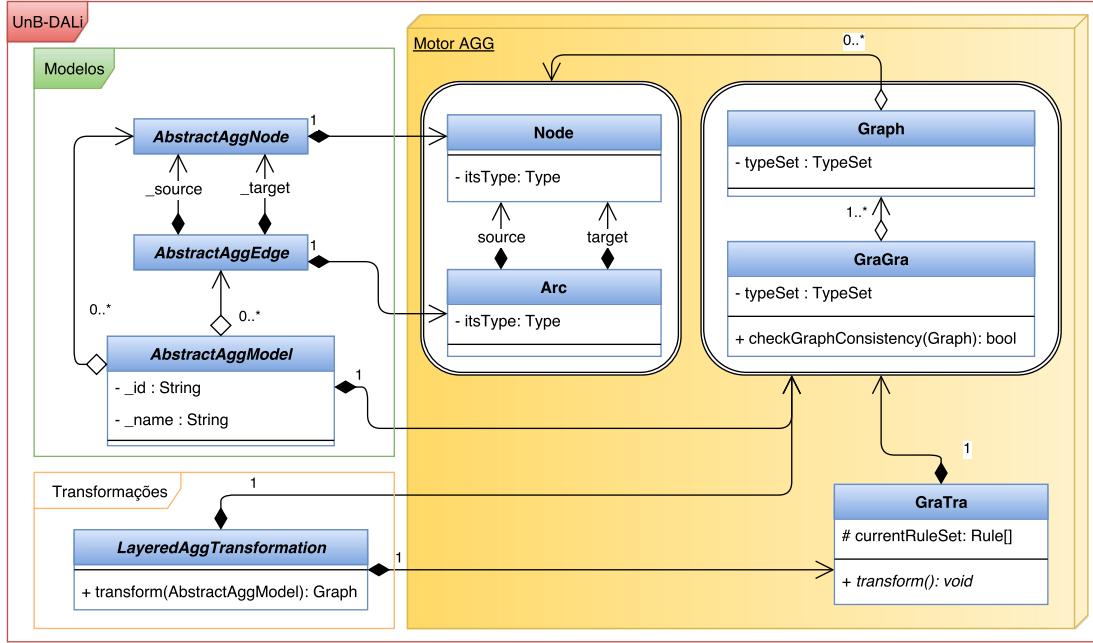


Figura 5.2: Relacionamento estrutural entre o *UnB-DALi* e o motor do AGG para transformação de grafos.

5.2.2 Implementação

Numa perspectiva mais geral da implementação do *UnB-DALi*, temos que a classe *AbstractAggModel* implementa a interface *IModel*, que define o método `+checkModel():void` para verificação da conformabilidade sintática de um modelo frente a seu metamodelo. Como a parte de verificação sintática pode ser conduzida por meio do AGG, a partir de uma verificação de consistência (*Consistency Checking* [14]), temos que *AbstractAggModel* implementa o método *checkModel()* de forma a executar essa rotina. Essa implementação também verifica se todos os atributos dos elementos do grafo subjacente a um modelo foram adequadamente atribuídos (por meio de chamada ao método `+isReadyForTransform():bool`, vide Código 5.1). Como a verificação semântica de um modelo varia de acordo com seu respectivo metamodelo, temos que *AbstractAggModel* permite que suas generalizações reimplementem o método *checkModel()*. Dessa forma, transformações que estendam *LayeredAggTransformation* podem fazer uso do método *checkModel()* para garantir que tanto o modelo de entrada quanto o modelo de saída sejam construídos de forma correta.

Código 5.1: Método *checkModel* implementado pela classe *AbstractAggModel*.

```

1 public abstract class AbstractAggModel implements IModel {
2     ...
3     @Override
4     public void checkModel() throws ModelSemanticsVerificationException {
5         if (!_gragra.checkGraphConsistency(_graph) || !_graph.isReadyForTransform())
6             throw new ModelSemanticsVerificationException("...");
7     }
8 }
```

Ainda nessa perspectiva, temos que a classe *LayeredAggTransformation* implementa a interface *ITransformation<AbstractAggModel, Graph>* que define o método *transform* para a condução da rotina de transformação de modelos. Com o intuito de reuso e clareza de leitura do código, temos que *LayeredAggTransformation* se preocupa em prover a lógica para condução da transformação do **grafo** subjacente a um objeto do tipo *AbstractAggModel*, delegando às suas generalizações a responsabilidade de construir o modelo de saída a partir do grafo resultante dessa transformação, conforme Código 5.2.

Código 5.2: Método *transform* implementado pela classe *LayeredAggTransformation*.

```

1 public abstract class LayeredAggTransformation
2     implements ITransformation<AbstractAggModel, Graph> {
3     ...
4     @Override
5     public Graph transform(AbstractAggModel source)
6         throws ModelSemanticsVerificationException {
7         source.checkModel();
8         Graph graph = source.getGraph().copy(_gragra.getTypeSet());
9         if (_gragra.resetGraph(graph)) {
10             _gratra.setHostGraph(graph);
11             _gratra.transform();
12         } else {
13             throw new ModelSemanticsVerificationException("...");
14         }
15         return _gratra.getHostGraph();
16     }
17     ...
18 }
```

AbstractAggModel e *LayeredAggTransformation* são construídos a partir de um arquivo de descrição de uma grafo-gramática do AGG, identificados com extensão *.gxl* (vide

Códigos 5.3 e 5.4). Espera-se que os arquivos *.gxl* fornecidos à construção de um *AbstractAggModel* possuam um grafo-tipo referente somente a seu metamodelo, onde estejam presentes também todas as regras de boa formação de um grafo tipado sobre esse grafo-tipo. Dos arquivos *.gxl* fornecidos à construção de extensões a *LayeredAggTransformation*, espera-se a presença de um grafo-tipo T onde os grafos-tipo dos modelos fonte e entrada, T_S e T_T , estejam contidos em T ; espera-se também a presença de todas as regras de transformação necessárias (assim como seus NACs) para a condução do morfismo $m : T_S \rightarrow T_T$. Esses arquivos são definidos a partir da interface gráfica do AGG (vide Capítulo 3) e mapeados em memória para um objeto do tipo *GraGra*.

Código 5.3: Construtor de um *AbstractAggModel*.

```

1  public abstract class AbstractAggModel implements IModel {
2
3      ...
4
5      public AbstractAggModel(String id, Graph graph, String GGXResource)
6          throws AggModelConstructionException {
7
8          _gragra = AggHelper.loadGraGra(GGXResource);
9          _gragra.destroyAllGraphs();
10         _graph = (graph!=null)?graph:new Graph(_gragra.getTypeSet());
11         _gragra.resetGraph(_graph.copy(_gragra.getTypeSet()));
12
13         if (!_gragra.checkGraphConsistency(_graph))
14             throw new AggModelConstructionException("... ");
15         if (id == null || id.isEmpty())
16             throw new AggModelConstructionException("... ");
17
18         _id = id;
19         _nodes = new HashMap<Node, AbstractAggNode>();
20         _edges = new HashMap<Arc, AbstractAggEdge>();
21         _nodesByString = new HashMap<String, AbstractAggNode>();
22         _edgesByString = new HashMap<String, AbstractAggEdge>();
23         setUp();
24         checkModel();
25     }
26
27     ...
28 }
```

Código 5.4: Construtor de um *LayeredAggTransformation*.

```

1  public abstract class LayeredAggTransformation
2      implements ITransformation<AbstractAggModel, Graph> {
3          ...
4
5          public AggLayeredTransformation(String fileName) {
6              _gragra = AggHelper.loadGraGra(fileName);
7              _gragra.destroyAllGraphs();
8              _gratra = new LayeredGraTralImpl();
9              _gratra.setCompletionStrategy(CompletionStrategySelector.getDefault());
10             _gratra.setGraGra(_gragra);
11         }
12     }

```

Observe que, da forma como planejada a biblioteca, uma rotina de transformação somente será executada caso o modelo de entrada esteja em conformidade com seu metamodelo. Caso o modelo de saída não possa ser criado a partir do grafo de saída da transformação, a exceção *ModelSemanticsVerificationException* é lançada e a transformação anulada. É recomendado, portanto, que os metamodelos do *UnB-DALi* possibilitem sua construção a partir de um grafo do AGG, pois não obrigatoriamente um modelo de entrada de uma transformação não possa ser o modelo de saída de outra transformação. Isso é facilitado pela obrigatoriedade de implementação do método *+setUp():void*, que será chamado sempre que um *AbstractAggModel* é construído.

5.3 Transformação de Modelos Comportamentais UML para DTMC

Temos que o modelo comportamental UML utilizado por Rodrigues *et al.* em [12] para análise de dependabilidade trata-se do *Interaction Overview Diagram*³ (ou IOD), um tipo de metamodelo definido pela especificação da UML 2.5 [30] que proporciona uma visão geral de como e quando cenários de comunicação entre os componentes de um sistema entram em execução. Isso é obtido a partir de uma modificação simples de um AD (Seção 5.3.1), onde nós de execução são substituídos por referências a SDs (Seção 5.3.1). Observa-se, portanto, que a construção de um IOD pode ser realizada a partir de uma composição inteligente entre um diagrama de atividades, denotando o controle de fluxo, e diagramas de sequência, denotando cenários de execução.

³Diagrama de Visão Geral de Interações, numa tradução livre

Para a automatização do processo de transformação de um IOD anotado ⁴ em um DTMC, adotou-se uma estratégia de *divisão-e-conquista*, em que inicialmente propomos as transformações de Diagramas de Atividades e Sequência para DTMCs ($AD \rightarrow DTMC$ e $SD \rightarrow DTMC$), via transformação de grafos conforme estabelecido na Seção 5.2.2. A composição dessas transformações de maneira compatível com o que se espera da transformação $IOD \rightarrow DTMC$, no entanto, ficou como trabalho futuro.

5.3.1 Metamodelos de Entrada e Saída

Explorando melhor as capacidades do *UnB-DALi* e do motor do AGG, temos que os metamodelos de entrada e saída de qualquer transformação devem ser sempre planejados independentemente de transformação, de forma a proporcionar assim um reuso sistemático dos mesmos. Essa recomendação foi seguida no planejamento das transformações $AD \rightarrow DTMC$ e $SD \rightarrow DTMC$, dado que o grafo-tipo de saída dos dois morfismos é exatamente o mesmo.

Diagrama de Atividades (AD)

Um AD proporciona às partes interessadas uma visão geral do fluxo de controle e de informações do sistema, possibilitando inclusive a modelagem de comportamentos concorrentes do mesmo. Conforme a sua especificação [30], todo AD é composto basicamente de dois tipos de elementos: nós de atividade e arestas de atividade. Neste trabalho, optou-se por restringir o metamodelo de um *AD* de forma que (1) não existam elementos de concorrência em sua composição e de forma que (2) sejam ignorados componentes que denotam fluxo de informação.

Com essas restrições, e conforme a especificação de um AD em [30], os elementos que compõem um AD no contexto do *UnB-DALi* se reduzem a basicamente:

- **Nó de Execução (ExecutableNode)**, que encapsula uma determinada ação a ser executada pelo sistema em determinado ponto da sequência de execução de ações;
- **Nó de Início (InitialNode)**, que representa o ponto de entrada para a execução das atividades do AD;
- **Nó de Fim (FinalNode)**, que representa o ponto final de execução das atividades e ações do AD;
- **Nó de Decisão (DecisionNode)**, que redireciona o fluxo de controle de acordo com a satisfatibilidade de determinadas condições;

⁴Anotado com informações de probabilidade de transição e confiabilidade dos componentes, vide Capítulo 2

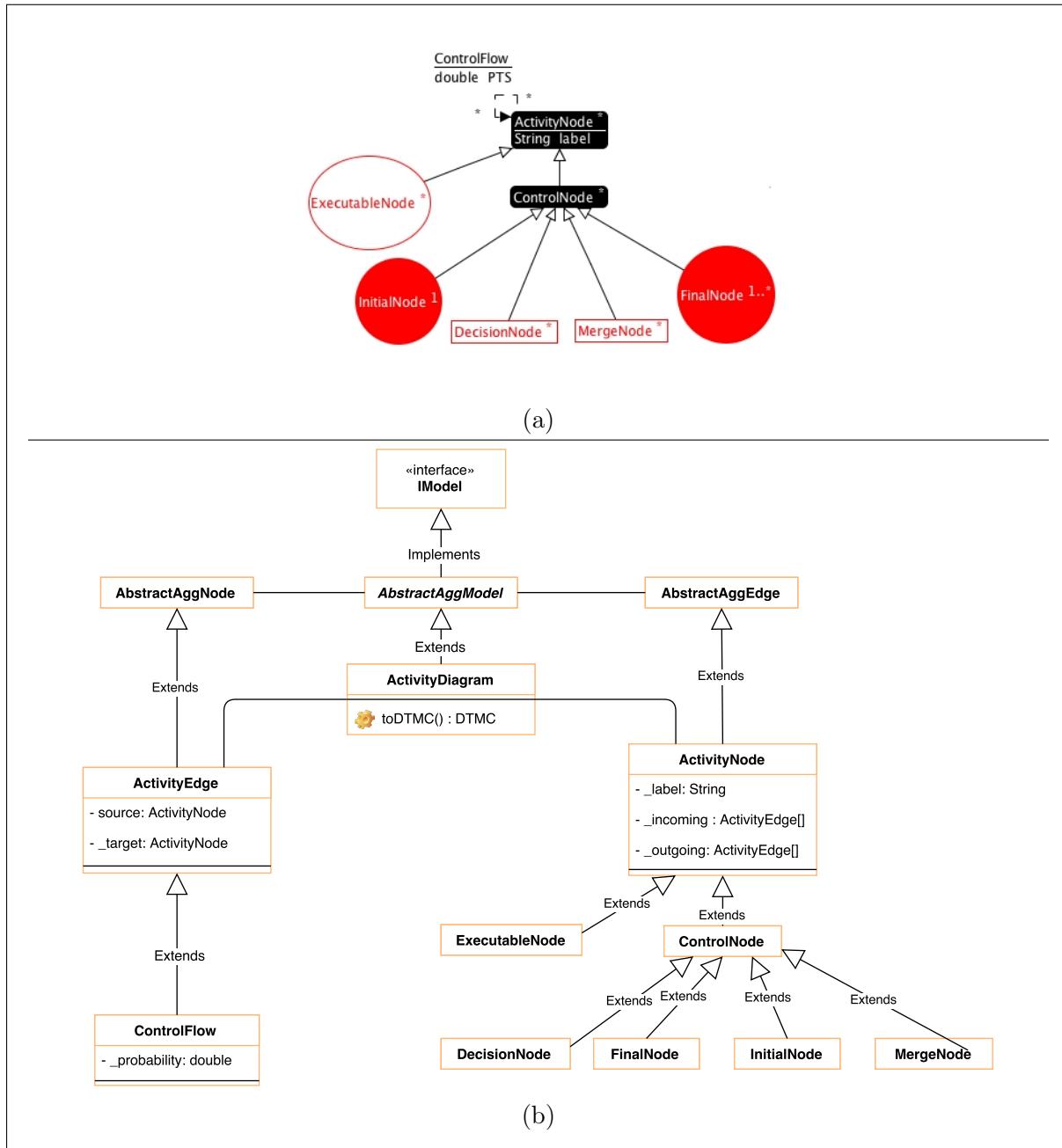


Figura 5.3: Diagrama de Atividades: (a): Grafo-Tipo do AGG. (b): Diagrama de Classes do UnB-DALi.

- **Nó de Merge (*Merge*)**, que indica a junção de fluxos condicionais diferentes;
- **Aresta de Fluxo de Controle (*ControlFlow*)**, que indica a transição de um nó de atividade para outro nó de atividade.

Dessa forma, foram modelados a representação em um grafo-tipo AGG de um AD (vide Figura 5.3a) e, concomitantemente, codificada essa modelagem em conformidade com as estruturas básicas do *UnB-DALi* (Figura 5.3b). Observe que a aresta *ControlFlow* possui a propriedade *PTS*, denotando a probabilidade de transição de um nó de atividade para outro, conforme estabelecido em [11] para uma análise de dependabilidade. Todos os nós de atividade (*ActivityNode*) possuem a propriedade *label*, denotando de forma apropriada o que aquele nó semanticamente representa na execução do sistema.

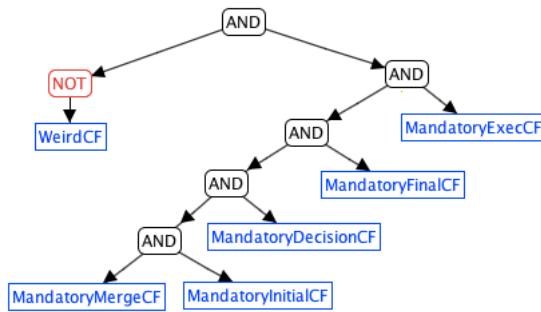


Figura 5.4: Condição de consistência que governa a boa formação de um grafo subjacente a um diagrama de atividades no AGG.

O método `+toDTMC():DTMC` da classe *ActivityDiagram* conduz a transformação de um diagrama de atividades para um DTMC, conforme Seção 5.3.2. Essa transformação somente ocorre caso a condição de consistência da Figura 5.4 seja satisfeita. Essa condição estabelece que nenhuma associação incomum via *ControlFlow* pode ocorrer entre nós de um diagrama de atividades (*WeirdCF*) e estabelece quais padrões de associação via *ControlFlow* são obrigatórios para os diferentes tipos de nós de um AD (*MandatoryExecCF*, *MandatoryFinalCF*, *MandatoryDecisionCF*, *MandatoryInitialCF* e *MandatoryMergeCF*). As restrições atômicas utilizadas para compor essa condição de consistência estão enumeraadas no Anexo A.1.

Diagrama de Sequência (SD)

Um SD faz parte do que se entende em UML por diagramas de interação, dado que indicam de que forma atores e componentes do sistema interagem entre si. Tal interação ocorre por meio de troca de mensagens, síncronas ou assíncronas, entre linhas de vida de dois ou mais objetos, trazendo assim uma noção de ordem bem determinada para esses

eventos [30]. Esses diagramas podem ser utilizados tanto pelo engenheiro do sistema quanto pelo usuário do mesmo (ou até mesmo pelo próprio sistema, na tentativa de mapear e posteriormente validar o comportamento dele mesmo); fato é que tais diagramas podem ser definidos em qualquer camada do ciclo de vida da aplicação, o que justifica

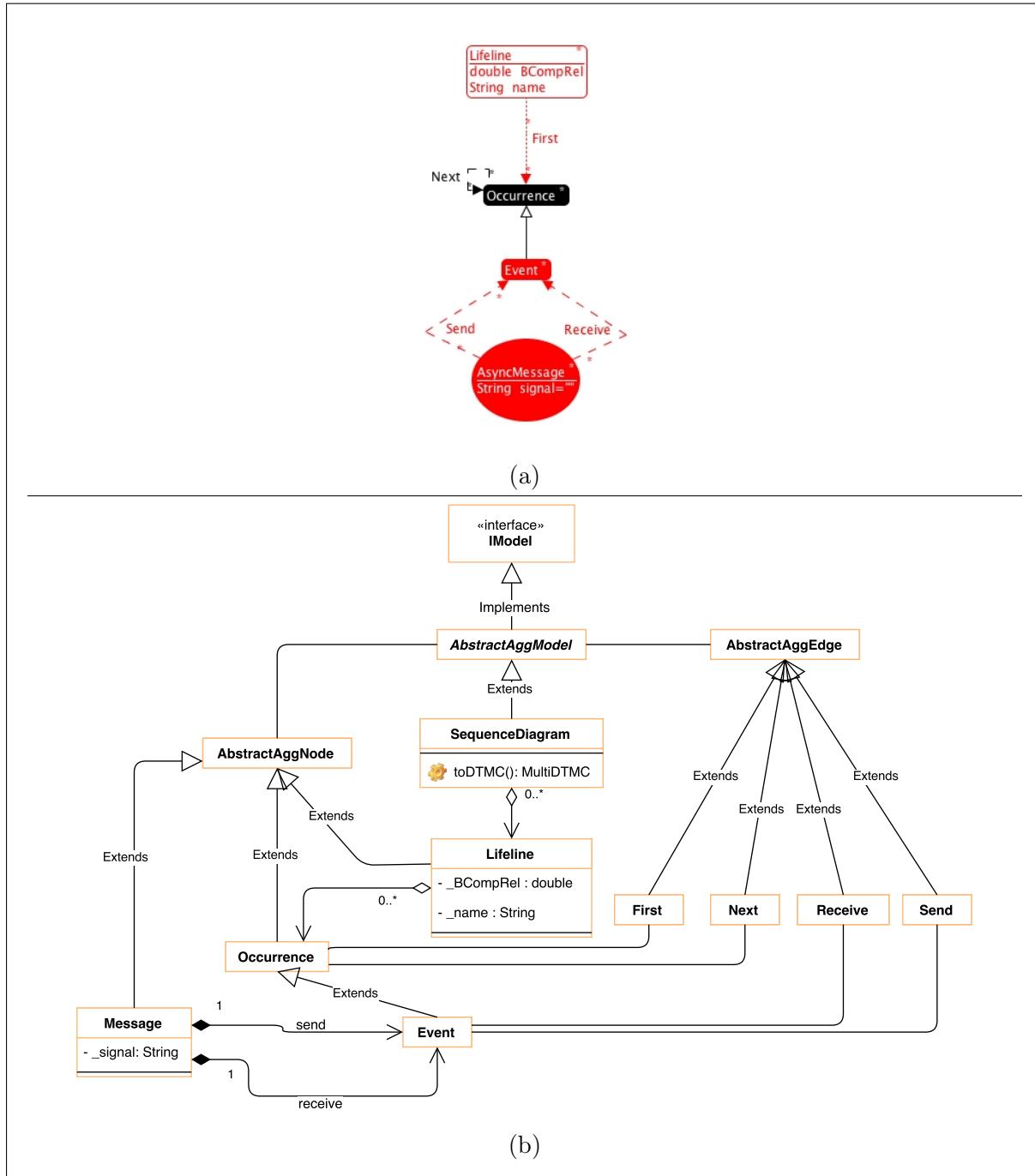


Figura 5.5: Diagrama de Sequência: (a): Grafo-Tipo do AGG. (b): Diagrama de Classes do UnB-DALi.

sua importância.

A especificação da UML para Diagramas de Sequência determina que troca de mensagens entre diferentes linhas de vida sejam estruturalmente definidas a partir da presença de eventos nas mesmas. Um evento (*Event*), assim como outros conceitos (fragmentos, etc), são tipos diferentes de ocorrências em uma linha de vida (*Lifeline*). Pela especificação da UML, cada linha de vida denota um sistema concorrente e cada nova ocorrência (*Occurrence*) indica uma nova transição de estados dentro desse sistema.

Para os objetivos deste trabalho, no entanto, determinados conceitos pertinentes a um SD não foram trabalhados, a citar:

- **Fragments:** conceito que denota sub-interações condicionadas dentro de um SD. O intuito é de explorar possíveis diferentes cenários de interação entre componentes de um sistema dado diferentes condições de execução do mesmo;
- **Mensagens Síncronas:** tipos de mensagens que quando disparadas por uma linha de tempo a fazem obrigatoriamente entrar em estado de espera por uma resposta.

Dessa forma, propomos a modelagem de um SD em termos do grafo-tipo AGG representado pela Figura 5.5a. Nesse grafo, estabeleceu-se que todo nó do tipo *Lifeline* possua uma única primeira ocorrência, denotada pela aresta do tipo *First*. Todas as demais ocorrências de uma linha de vida são encadeadasumas as outras a partir de uma aresta do tipo *Next*. Estabeleceu-se também que apenas mensagens assíncronas serão utilizadas como forma de interação entre duas linhas de vida. Cada mensagem está associada com dois diferentes nós de evento por meio de duas arestas de tipos distintos (*Send* e *Receive*), definindo assim a ocorrência de origem e a ocorrência de destino da mensagem; por conseguinte, definindo também as linhas de vida de origem e destino da mesma. Observe que *Lifeline* possui duas propriedades: *BCompRel*, o índice de confiabilidade do componente, de acordo com o estabelecido para análise de dependabilidade em [11]; e *name*, o nome do componente representado por nós desse tipo. *AsyncMessage* também possui uma propriedade, *signal*, denotando o conteúdo da mensagem trocada.

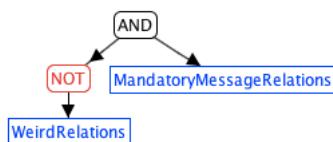


Figura 5.6: Condição de consistência que governa a boa formação de um grafo subjacente a um diagrama de sequências no AGG.

A organização da codificação do grafo-tipo AGG de um SD no contexto do *UnB-DALi* está ilustrado pela Figura 5.5b. O método *+toDTMC():MultiDTMC* de um *SequenceDiagram*

gram realiza a transformação de um SD para um DTMC adequado (conforme Seção 5.3.2). Essa transformação não ocorre, no entanto, caso a condição de consistência da Figura 5.6 não seja satisfeita. Essa condição estabelece que grafos subjacentes a um SD não podem possuir padrões incomuns de associação entre seus nós (*WeirdRelations*), como por exemplo um mesmo *Event* associado a duas diferentes *AsyncMessages*, e estabelece a obrigatoriedade da presença de uma associação *Send* e uma associação *Receive* para cada *AsyncMessage* (*MandatoryMessageRelations*). As restrições atômicas que fazem parte dessa condição de consistência e suas explicações estão enumeradas no Anexo A.2.

Cadeias de Markov de Tempo Discreto (DTMC)

DTMCs, da forma como definido no Capítulo 2, tratam-se de modelos capazes de representar comportamentos probabilísticos e estocásticos de um sistema qualquer. São adequados, dessa forma, para a condução de análises de diversos tipos, especialmente análise de dependabilidade. A Definição 2 de um DTMC pode facilmente ser representada a partir de um grafo-tipo AGG (conforme Figura 5.7a), onde:

- o conjunto de estados S é representado pelo conjunto de nós do tipo *AbstractState*;
- o estado inicial \bar{s} é representado por um nó do tipo *InitialState*;
- a matriz de probabilidades de transições $P : S \times S \rightarrow [0, 1]$ é representada pelo conjunto de arestas do tipo *Transition*, de forma que a ausência de uma transição entre dois estados significa uma entrada com valor 0 na matriz P ;
- a função rotuladora $L : S \rightarrow 2^{AP}$ é representada pelo atributo *label* de um *AbstractState*.

Pela Figura 5.7a percebe-se, no entanto, que o conceito de um DTMC foi expandido de forma a incluir (1) estados de erro (*ErrorState*) e (2) condições de guarda para transições (atributo *guard* de uma aresta do tipo *Transition*). Essas expansões são devidas, respectivamente, a dois fatores:

- DTMCs precisam garantir que as probabilidades das transições a partir de um determinado estado $s \in S$ somem 1. O conceito de estado de erro vem para garantir que um DTMC sempre será construído de forma a satisfazer essa propriedade, dado que caso as probabilidades de transição não somem 1, uma transição para um estado de erro sempre será criada com *PTS* possuindo o valor de probabilidade restante;
- como mencionado anteriormente, cada linha de vida de um SD define um sistema concorrente individual, onde a transição de estados em um desses sistemas depende

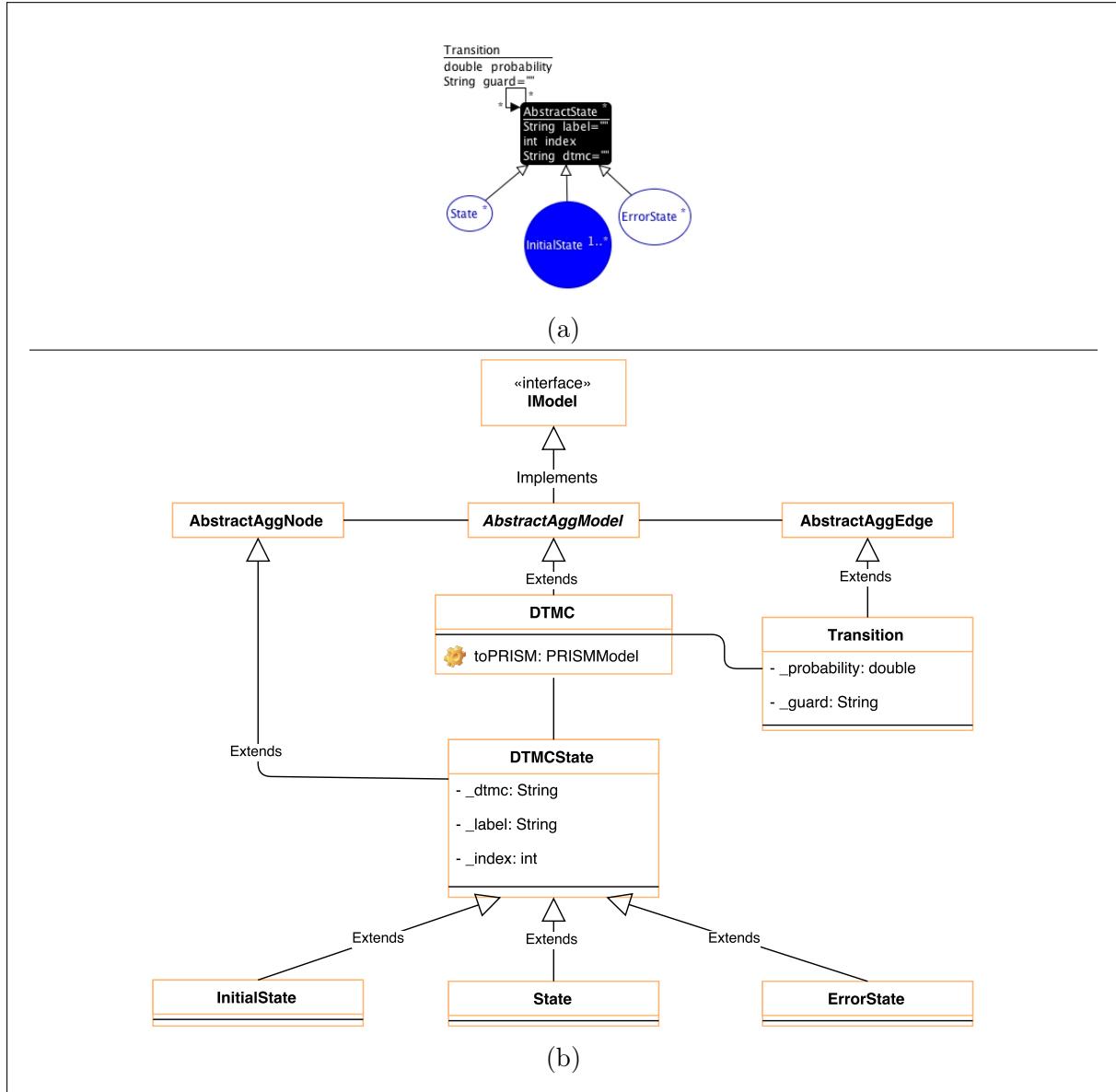


Figura 5.7: Cadeias de Markov de Tempo Discreto (DTMC): (a): Grafo-Tipo do AGG. (b): Diagrama de Classes do UnB-DALi.

de um ou mais estados de um ou mais sistemas. Logo, para a representação adequada de um SD em um DTMC surgiu a necessidade de modelar um DTMC que possua um ou mais estados iniciais, onde a existência de *guardas* (ou condições) sobre transições garantem que esses sistemas estejam sincronizados entre si.



Figura 5.8: Condição de consistência que governa a boa formação de um grafo subjacente a um DTMC no AGG.

Naturalmente, a codificação de um DTMC na biblioteca *UnB-DALi* (Figura 5.7b) seguiu os conceitos trabalhados acima. Destaca-se nesse diagrama de classes o método `+toPRISM():PRISMModel`, que transforma esse DTMC em um modelo PRISM conforme a sintaxe do PRISM discutido no Capítulo 2. Essa transformação somente ocorrerá caso a condição de consistência da Figura 5.8 seja satisfeita. A restrição atômica que faz parte dessa condição é elaborada com maiores detalhes no Anexo A.3.

5.3.2 Transformações

Relembrando o Capítulo 3, temos que uma transformação de modelos pode ser representada por um sistema de transformação de grafos $GTS = (T, R)$, T o grafo-tipo da transformação e R o conjunto de regras da mesma, onde os grafos-tipo T_S e T_T dos modelos-fonte e modelos-alvo, respectivamente, respeitam a relação $T_S \subseteq T \supseteq T_T$. Dessa forma, propomos nessa seção as transformações $AD \rightarrow DTMC$ e $SD \rightarrow DTMC$, onde para cada uma dessas foi implementado no AGG seu grafo-tipo (que contém os grafos-tipos dos modelos de entrada e saída) e seu conjunto de regras. Detalhes e explicações das mesmas estão detalhadas abaixo.

AD para DTMC

Pela Figura 5.9, vemos que além dos grafos-tipo dos modelos-fonte e modelos-alvo (AD e DTMC), o grafo-tipo $AD2DTMC$ contém dois outros nós e duas outras relações:

- nó **scount**: usado para contar o número de estados criados na transformação, facilitando posteriormente uma visualização mais adequada do DTMC resultante;
- nó **A2D** e suas relações $A2D \rightarrow ActivityNode$ e $A2D \rightarrow AbstractState$: denotando como os nós de um AD podem se relacionar com os nós de um DTMC. A existência desse nó trata-se de uma estratégia para transformações de grafos que limita quem

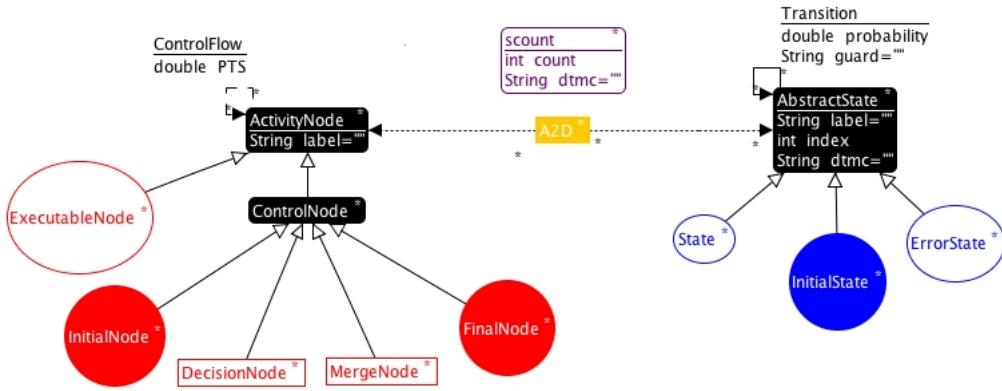


Figura 5.9: Grafo-tipo *AD2DTMC* do GTS de um diagrama de atividades para um DTMC.

do grafo-tipo fonte pode se transformar em quem do grafo-tipo alvo, além de proporcionar a existência de um padrão para identificar quando um elemento de um modelo fonte já foi transformado em um elemento de um modelo alvo.

A dinâmica implementada para a transformação de um AD para um DTMC é o de, primeiramente, resolver encadeamentos de nós do tipo *DecisionNode* e *MergeNode* para um cálculo mais correto das probabilidades de transição entre dois nós distintos dentro de um AD. Concluída essa etapa, a transformação prossegue para inicializar um contador de estados, o estado inicial e os estados finais. Tendo isso feito, as regras da forma como planejadas forçam a transformação a ser conduzida a partir do nó inicial da seguinte maneira: para cada conexão via *ControlFlow* entre um nó N_o de origem já mapeado em um estado S_o do DTMC e um nó N_d de destino ainda não mapeado para um estado do DTMC, é criado um novo estado S_d no DTMC associado a N_d , removido do grafo resultante o nó N_o e criado uma transição adequada de S_o para S_d . Após isso, dependendo do grafo de entrada, é esperado que determinados nós de um AD ainda permanecem no grafo; esses, que se encaixam em padrões pré-estabelecidos, são removidos do grafo restando assim apenas o DTMC resultante.

No *UnB-DALi*, essa transformação é representada pela classe *AD2DTMC*, que estende a classe *LayeredAggTransformation* e se constrói a partir do arquivo *AD2DTMC.gxl* presente na pasta *resources/transformations* da biblioteca. As regras de transformação que implementam a dinâmica acima estão listadas no Anexo B.1.

SD para DTMC

Temos que numa transformação de um SD para um DTMC, o mapeamento básico feito é que cada linha de vida possuirá seu próprio DTMC. Dessa forma, o grafo-tipo

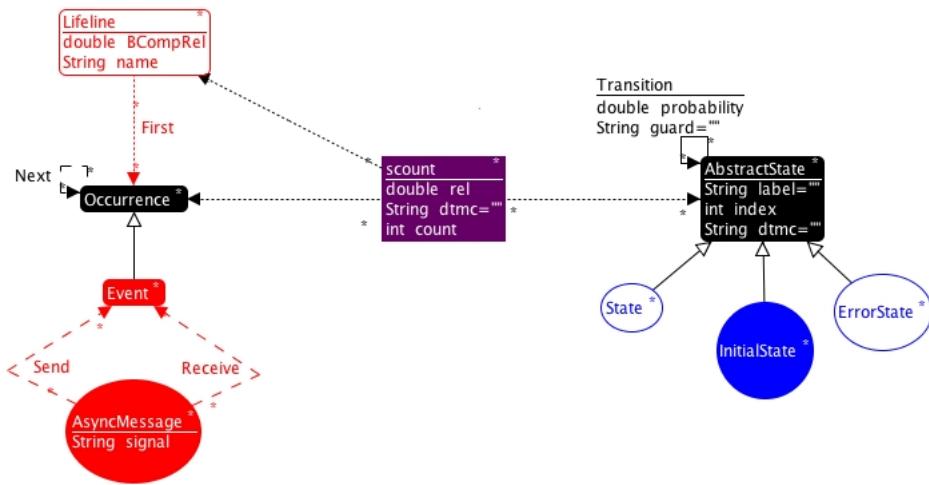


Figura 5.10: Grafo-tipo *SD2DTMC* do GTS de um diagrama de sequência para um DTMC

implementado para a transformação $SD \rightarrow DTMC$, ilustrado na Figura 5.10, possui, além dos grafos-tipo fonte e alvo, o nó *scount* que relaciona um SD a seu respectivo DTMC. Esse nó mantém um contador do número de estados criados de um DTMC (atributo *count*), mantém também o nome do DTMC de uma linha de vida (atributo *dtmc*), além de manter o índice de confiabilidade do respectivo componente do sistema (por meio do atributo *rel*).

A dinâmica básica da transformação de um SD para um DTMC é forçar uma inicialização da transformação a partir do mapeamento de uma linha de vida para um estado inicial de seu DTMC correspondente. Após isso, é preciso transformar os casos de troca de mensagens que envolvam eventos que são ocorrências primárias (*Lifeline* \xrightarrow{First} *Event*) dentro de sua respectiva linha de vida. Resolvido a etapa anterior, a tarefa seguinte é transformar as trocas de mensagens que envolvem apenas eventos secundários (*Event* \xrightarrow{Next} *Event*).

Essas duas etapas funcionam de forma que, para cada evento nos dois extremos de uma mensagem assíncrona, sejam (1) criados novos estados nos respectivos DTMCs, (2) removidos os nós anteriores a esses eventos (nós de tipo *Lifeline* ou *Event*), (3) criadas transições de forma adequada entre os estados associados aos nós antecessores e esses novos estados e, por último, (4) seja removido do grafo o nó referente a essa mensagem assíncrona. Dentro desse raciocínio, temos que para o evento associado com essa mensagem via *Receive*, a condição de guarda da transição adicionada é formatada de forma a conter o estado atual do evento associado via *Send*; para o evento associado com essa mensagem via *Send*, a probabilidade da transição adicionada é computada para ser igual ao índice de confiabilidade da linha de vida do evento associado via *Receive*. Essas medidas colocam essa transformação em conformidade com o método de Rodrigues *et al.*

discutido no Capítulo 2.

Concluídas essas etapas, é prevista ainda a presença de um nó do tipo *Event* para cada linha de vida no grafo. Esses são então removidos do grafo resultante de forma que um estado final é criado para cada um deles. Caso ainda persistam nós do tipo *Lifeline*, temos que essa linha de vida não possuiu nenhum tipo de interação nesse diagrama de sequência. Logo, um *DTMC* simples com apenas 2 estados (um inicial e outro de erro) são criados.

No *UnB-DALi*, essa transformação é representada pela classe *SD2DTMC*, que estende a classe *LayeredAggTransformation* e se constrói a partir do arquivo *SD2DTMC.gxl* presente na pasta *resources/transformations* da biblioteca. As regras de transformação que implementam a dinâmica acima estão listadas no Anexo B.2.

DTMC → PRISM

Após a transformação de um *AD* ou de um *SD* para um *DTMC*, supomos que é de interesse da ferramenta externa que utiliza o *UnB-DALi* a exportação desse *DTMC* para diversos formatos diferentes. No momento, porém, o único formato suportado para persistência de um *DTMC* é o definido pela ferramenta *PRISM* (Capítulo 2).

Para auxiliar nesse mapeamento, criou-se as classes *PRISMMModel* e *PRISMMModule* que, respectivamente, denotam um modelo *PRISM* e um módulo *PRISM*. Essas classes apenas definem a metaestrutura de um modelo *PRISM* a partir da definição de como os módulos estão organizados dentro de um modelo e como as variáveis e comandos estão organizados dentro de um módulo, delegando a lógica de construção de variáveis e comandos para as entidades que as utilizam.

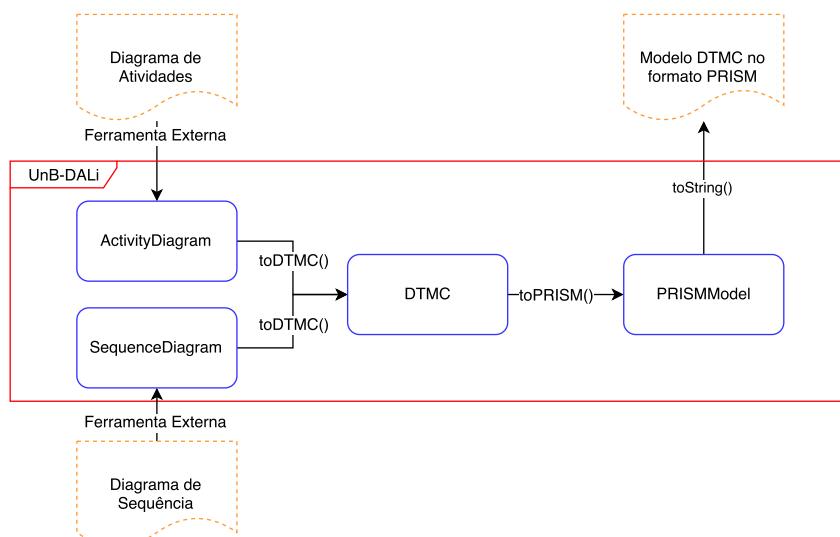


Figura 5.11: *Workflow* da transformação de um *AD* ou *SD* para um *DTMC* no formato *PRISM*.

Dessa forma, temos que o método *toPRISM()* implementa a lógica de representação de um DTMC em um objeto do tipo *PRISMMModel*. Basicamente, para cada DTMC é criado uma variável do tipo *array* de inteiros, tendo como identificador o nome do DTMC prefixado com "s" e as posições do *array* definidas de -1 a $|DTMC| - 2$, onde o estado erro é identificado por -1 e o estado inicial por 0 . Para cada estado do DTMC é criado um comando novo do PRISM, tendo como guarda a variável igualada a posição do próprio estado, isso concatenado com as guardas presentes em todas as transições saindo do estado em questão; o par $<\text{'probability}'>:<\text{'update}'>$ é criado a partir de cada transição saindo desse estado, onde 'update' é o nome da variável igualada ao estado de destino da transição e 'probability' é o valor de probabilidade associado a essa mesma transição.

Tem-se que a classe *PRISMMModel* sobrescreve o método *toString*, facilitando o acesso ao modelo PRISM resultante. Dessa forma, temos que uma transformação de um AD ou de um SD para um modelo DTMC descrito no formato PRISM segue a dinâmica da Figura 5.11.

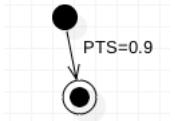
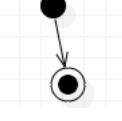
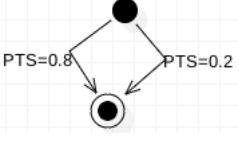
5.4 Validação

A validação da biblioteca implementada nesse trabalho foi conduzida de forma experimental. Um novo projeto Java [2] foi criado para a condução dos testes, que consumiu a última versão da biblioteca *UnB-DALi* encapsulada em um arquivo *.jar*. As seguintes métricas foram coletadas: (1) sucesso ou não da transformação no contexto da biblioteca; (2) média e desvio padrão do tempo de transformação (cada teste foi executado 10 vezes); (3) sucesso de compilação do modelo resultante na ferramenta PRISM; (4) correspondência entre o modelo PRISM resultante e o esperado.

Todos os testes foram conduzidos em um *MacBook Air* ano 2013, memória 4 GB 1600 MHz DDR3, processador 1.3 GHz Intel Core i5, com armazenamento de 120 GB em Unidade Flash.

5.4.1 $AD \rightarrow PRISM(DTMC)$

Tabela 5.1: Testes conduzidos sobre a transformação de um AD para um DTMC no formato PRISM.

Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 3 module Teste1 4 sTeste1 : [-1..2] init 0; 5 6 [initial] sTeste1=0 -> 1.0:(sTeste1'=1); 7 [final] sTeste1=2 -> 1.0:(sTeste1'=2); 8 [fail] sTeste1=-1 -> 1.0:(sTeste1'=-1); 9 [] sTeste1=1 -> 1.0:(sTeste1'=2); 10 endmodule </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
		$\sim 343.4ms \pm 181.9ms$	
Modelo de Entrada		Modelo Esperado	
			
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
			
Modelo de Entrada		Modelo Esperado	
			
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
			

Continua na próxima página

Tabela 5.1 – (Continuação)

Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 3 module Teste4 4 sTeste4 : [-1..4] init 0; 5 6 [] sTeste4=3 -> 1.0:(sTeste4'=2); 7 [fail] sTeste4=-1 -> 1.0:(sTeste4'=-1); 8 [final] sTeste4=2 -> 1.0:(sTeste4'=2); 9 [] sTeste4=1 -> 0.5:(sTeste4'=3) + 0.5:(sTeste4'=4); 10 [] sTeste4=4 -> 1.0:(sTeste4'=2); 11 [initial] sTeste4=0 -> 1.0:(sTeste4'=1); 12 endmodule </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
		~ 523.6ms ± 237.8ms	
Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 3 module Teste5 4 sTeste5 : [-1..4] init 0; 5 6 [] sTeste5=4 -> 1.0:(sTeste5'=2); 7 [initial] sTeste5=0 -> 0.5:(sTeste5'=3) + 0.5:(sTeste5'=4); 8 [fail] sTeste5=-1 -> 1.0:(sTeste5'=-1); 9 [] sTeste5=3 -> 1.0:(sTeste5'=2); 10 [final] sTeste5=2 -> 1.0:(sTeste5'=2); 11 endmodule </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
		~ 584.2ms ± 227.9ms	
Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 3 module Teste6 4 sTeste6 : [-1..5] init 0; 5 6 [] sTeste6=3 -> 1.0:(sTeste6'=2); 7 [] sTeste6=1 -> 0.5:(sTeste6'=3) + 0.5:(sTeste6'=4); 8 [] sTeste6=4 -> 1.0:(sTeste6'=5); 9 [] sTeste6=5 -> 1.0:(sTeste6'=2); 10 [fail] sTeste6=-1 -> 1.0:(sTeste6'=-1); 11 [final] sTeste6=2 -> 1.0:(sTeste6'=2); 12 [initial] sTeste6=0 -> 1.0:(sTeste6'=1); 13 endmodule </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
		~ 558.8ms ± 232.1ms	

Continua na próxima página

Tabela 5.1 – (Continuação)

Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 module Teste7 3 sTeste7 : [-1..5] init 0; 4 5 [final] sTeste7=2 -> 1.0:(sTeste7'=2); 6 [] sTeste7=3 -> 1.0:(sTeste7'=2); 7 [] sTeste7=5 -> 1.0:(sTeste7'=2); 8 [] sTeste7=4 -> 1.0:(sTeste7'=5); 9 [] sTeste7=1 -> 0.2:(sTeste7'=1) + 0.4:(sTeste7'=3) + 0.4:(sTeste7'=4); 10 [initial] sTeste7=0 -> 1.0:(sTeste7'=1); 11 [fail] sTeste7=-1 -> 1.0:(sTeste7'=-1); 12 13 endmodule </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
✓	✓	~ 564.8ms ± 241.9ms	✓
Modelo de Entrada		Modelo Esperado	
		X	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
✗	✓		
Modelo de Entrada		Modelo Esperado	
		X	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
✗	✓		
Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 module Teste10 3 sTeste10 : [-1..102] init 0; 4 5 [] sTeste10=20 -> 1.0:(sTeste10'=21); 6 [] sTeste10=39 -> 1.0:(sTeste10'=40); 7 [] sTeste10=59 -> 1.0:(sTeste10'=60); 8 [] sTeste10=48 -> 1.0:(sTeste10'=49); 9 [] sTeste10=53 -> 1.0:(sTeste10'=54); 10 [] sTeste10=43 -> 1.0:(sTeste10'=44); 11 ... 12 13 endmodule </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
✓	✓	~ 5464.8ms ± 893.1ms	✓

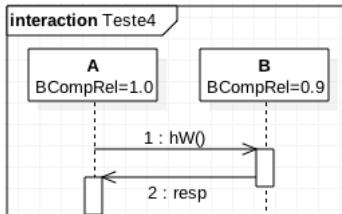
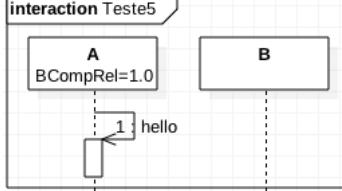
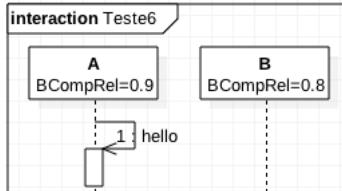
5.4.2 $SD \rightarrow PRISM(DTMC)$

Tabela 5.2: Testes conduzidos sobre a transformação de um SD para um DTMC no formato PRISM.

Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 3 module Teste1 4 sA : [-1..0] init 0; 5 6 [init_A] sA=0 -> 1.0:(sA'=0); 7 [fail_A] sA=-1 -> 1.0:(sA'=-1); 8 endmodule 9 </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
		$\sim 453.8ms \pm 213.7ms$	
Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 3 module Teste2 4 sA : [-1..0] init 0; 5 sB : [-1..0] init 0; 6 7 [init_A] sA=0 -> 1.0:(sA'=0); 8 [fail_A] sA=-1 -> 1.0:(sA'=-1); 9 [init_B] sB=0 -> 1.0:(sB'=0); 10 [fail_B] sB=-1 -> 1.0:(sB'=-1); 11 endmodule 12 </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
		$\sim 414.9ms \pm 187.0ms$	
Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 3 module Teste3 4 sA : [-1..1] init 0; 5 sB : [-1..1] init 0; 6 7 [fail_A] sA=-1 -> 1.0:(sA'=-1); 8 [init_A] sA=0 -> 0.9:(sA'=1) + 0.1:(sA'=-1); 9 [end_A] sA=1 -> 1.0:(sA'=1); 10 [init_B] sB=0 & sA=1 -> 1.0:(sB'=1); 11 [fail_B] sB=-1 -> 1.0:(sB'=-1); 12 [end_B] sB=1 -> 1.0:(sB'=1); 13 endmodule 14 </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
		$\sim 454.0ms \pm 182.8ms$	

Continua na próxima página

Tabela 5.2 – (Continuação)

Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 3 module Teste4 4 sA : [-1..2] init 0; 5 sB : [-1..2] init 0; 6 7 [init_A] sA=0 -> 0.9:(sA'=1) + 0.1:(sA'=-1); 8 [fail_A] sA=-1 -> 1.0:(sA'=-1); 9 [] sA=1 & sB=2 -> 1.0:(sA'=2); 10 [end_A] sA=2 -> 1.0:(sA'=2); 11 [fail_B] sB=-1 -> 1.0:(sB'=-1); 12 [] sB=1 -> 1.0:(sB'=2); 13 [end_B] sB=2 -> 1.0:(sB'=2); 14 [init_B] sB=0 & sA=1 -> 1.0:(sB'=1); 15 endmodule </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
		~ 536.1ms ± 232.4ms	
Modelo de Entrada		Modelo Esperado	
			
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
			
Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 3 module Test6 4 sA : [-1..1] init 0; 5 sB : [-1..0] init 0; 6 7 [fail_A] sA=-1 -> 1.0:(sA'=-1); 8 [init_A] sA=0 -> 0.9:(sA'=1) + 0.1:(sA'=-1); 9 [end_A] sA=1 -> 1.0:(sA'=1); 10 [fail_B] sB=-1 -> 1.0:(sB'=-1); 11 [init_B] sB=0 -> 1.0:(sB'=0); 12 endmodule </pre>	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
		~ 429.1ms ± 200.4ms	

Continua na próxima página

Tabela 5.2 – (Continuação)

Modelo de Entrada		Modelo Esperado	
Sucesso	Conforme Esperado	Tempo	PRISM Compilável
		$\sim 487.2ms \pm 218.0ms$	
Modelo de Entrada		Modelo Esperado	
		<pre> 1 dtmc 2 3 module Teste8 4 sA : [-1..50] init 0; 5 sB : [-1..50] init 0; 6 7 [] sA=33 & sB=34 -> 1.0:(sA'=34); 8 [] sA=17 & sB=18 -> 1.0:(sA'=18); 9 [] sA=31 & sB=32 -> 1.0:(sA'=32); 10 [] sA=9 & sB=10 -> 1.0:(sA'=10); 11 [] sA=8 -> 0.8:(sA'=9) + 0.2:(sA'=-1); 12 [] sA=29 & sB=30 -> 1.0:(sA'=30); 13 [] sA=40 -> 0.8:(sA'=41) + 0.2:(sA'=-1); 14 [] sA=48 -> 0.8:(sA'=49) + 0.2:(sA'=-1); 15 [] sA=16 -> 0.8:(sA'=17) + 0.2:(sA'=-1); 16 [] sA=2 -> 0.8:(sA'=3) + 0.2:(sA'=-1); 17 [] sA=6 -> 0.8:(sA'=7) + 0.2:(sA'=-1); 18 [] sA=45 & sB=46 -> 1.0:(sA'=46); 19 [] sA=15 & sB=16 -> 1.0:(sA'=16); 20 [] sA=27 -> 0.8:(sA'=23) + 0.2:(sA'=-1); </pre>	
		$\sim 1848.5ms \pm 687.4ms$	

5.5 Discussão

Um problema encontrado com a *engine* do AGG foi com o método *read_from_xml* da classe *XMLHelper*, rotina para deserialização de uma grafo-gramática do AGG. Esse método não funciona caso o arquivo que persiste a grafo-gramática esteja compactado dentro de um arquivo *.jar*, devido à utilização da classe *File* do Java para carregar o conteúdo do arquivo em memória. A maneira correta de implementar essa rotina, no entanto, seria utilizar *InputStream* para a mesma tarefa. Dado que planeja-se o uso do *UnB-DALi* justamente dessa forma (como um *.jar* contendo internamente as grafo-gramáticas como

resources), essa capacidade se tornou requisito obrigatório. Assim, para superar essa adversidade, a classe *CustomXMLHelper* que estende *XMLHelper* foi implementada de forma que essa deserialização possa ser executada a partir de objetos do tipo *InputStream*.

Outro problema enfrentado com o AGG foi com relação a transformação $SD \rightarrow DTMC$. Temos que o grafo resultante dessa transformação aplicada a um SD com n linhas de vida possui exatamente n DTMCs disconexos entre si. Como discutido no Capítulo 3, o motor de transformação de grafos do AGG não oferece suporte para transformações com múltiplos grafos como saída. Dessa forma, para reaproveitar a classe *DTMC* criada para a transformação $AD \rightarrow DTMC$, uma implementação de um algoritmo para extrair as componentes fracamente conexas de um grafo do AGG foi exigida. Essa implementação faz parte da classe *AggHelper* do *UnB-DALi* e é dada pelos métodos *getWeaklyConnectedSubGraphs* e *getWeaklyConnectedGraph*.

O AGG teve seu desenvolvimento iniciado há mais de 20 anos. Essa característica faz com que seu código possua padrões de projeto defasados e uso de métodos e classes depreciados. Dessa forma, observou-se que não há um uso adequado de mecanismos de exceção do Java. Com documentação pouco clara, não foi possível encontrar como tratar mensagens de erro e ajustar assim o *UnB-DALi* para entregar ao usuário da biblioteca mensagens mais significativas sobre onde se encontra o possível problema.

Pelos testes conduzidos na seção anterior, observou-se uma tendência para o desvio-padrão do tempo de execução corresponder próximo a 50% da média. Esse desvio discrepante possivelmente se deve pelo alto número de mensagens de *log* que o motor do AGG emite para *stdout*. Até o momento da escrita desta monografia não foi encontrado um mecanismo para que essas mensagens não sejam exibidas.

A validade sintática das transformações aqui conduzidas foram atestadas experimentalmente compilando-se os modelos de saída concretos na ferramenta de destino (no caso, PRISM). Isso, porém, não garante que todas as possíveis transformações estejam também sintaticamente corretas. Entretanto, como as transformações foram descritas utilizando AGG e, por conseguinte, descritas pelo formalismo de Transformação de Grafos, temos a possibilidade de garantir a correção sintática de todas as possíveis transformações por meio de uma *Análise de Pares Críticos*. Dado a complexidade do estudo e aplicação desse fundamento, decidimos que para não estender de forma demasiada este trabalho tal rotina seja conduzida em um trabalho futuro.

5.6 Considerações Finais

Neste capítulo foram discutidas e apresentadas as características estruturais e comportamentais da biblioteca *UnB-DALi*, onde testes da execução das rotinas de transformação

implementadas e comentários sobre a experiência com o AGG foram dados.

O próximo capítulo encerra essa monografia com observações gerais sobre o tema trabalhado e sobre a biblioteca implementada, além de dar considerações sobre possíveis trabalhos futuros.

Capítulo 6

Conclusão e Trabalhos Futuros

Nesse trabalho, uma biblioteca interoperável e reusável para a transformação de modelos numa análise de dependabilidade é proposta e implementada. O *UnB-DALi* faz um uso inteligente da técnica de transformação de grafos associada a ferramenta AGG, delegando a ferramentas externas o mapeamento dos arquivos de descrição de modelos (sintaxe concreta) aos metamodelos definidos pela biblioteca (sintaxe abstrata). Dessa forma, argumentamos que o *UnB-DALi* é independente de espaços tecnológicos para modelos de entrada, muitas vezes inconsistentes e redundantes em suas especificações, distinguindo assim o *UnB-DALi* de outras ferramentas que possuem o mesmo propósito, concomitantemente aumentando sua reusabilidade.

A decisão de implementação da biblioteca em Java se deu por dois motivos: primeiro porque o motor de transformação de grafos do AGG também está implementado nessa linguagem e segundo pois há uma expectativa que essa escolha impulsione uma interoperabilidade maior com ferramentas de modelagem presentes no mercado, dado que essas, em grande maioria, são também implementadas em Java.

Além de testes mais rigorosos que precisam ser feitos no futuro, temos que como consequência da escolha da técnica de transformação de grafos como motor para o *UnB-DALi*, o índice de confiabilidade associado ao *UnB-DALi* pode ser aumentado caso uma análise de pares críticos seja conduzida para cada um dos respectivos sistemas de transformação implementados ($AD \rightarrow DTMC$ e $SD \rightarrow DTMC$), verificando-se assim, formalmente, a correção sintática das mesmas.

Alguns aspectos da biblioteca precisam de melhorias, a citar:

- suporte a mensagens síncronas e fragmentos *alt*, *loop*, *opt* e *par* no metamodelo proposto para um SD;
- suporte a elementos de concorrência num AD;
- padronização de lançamentos de exceção por parte da biblioteca;

- melhor tratamento de erros lançados pelo AGG;
- distribuição da biblioteca via Maven, para melhor administração do uso da biblioteca por parte do desenvolvedor.

Como observado no Capítulo 5, foi-se hipotetizado que a transformação de um IOD para um DTMC na notação PRISM pode ser obtida a partir de uma composição inteligente das transformações de um SD e de um AD para um DTMC. Essa composição, os aspectos listados acima e uma análise de pares críticos sobre os sistemas de transformação propostos compõem trabalhos de melhoria e fechamento do que foi elaborado nessa monografia.

Possíveis trabalhos futuros são:

- Um gerador automático de classes Java que estendem as classes *AbstractAggModel*, *AbstractAggNode* e *AbstractAggEdge* a partir de um grafo-tipo do AGG. Um gerador desse tipo possibilitaria uma evolução horizontal da biblioteca de forma mais veloz;
- Replicação via *UnB-DALi* da transformação de um CRGM (Contextual-Runtime Goal Model) para um DTMC como proposto por Mendonça em [7];
- Condução de *benchmarks* entre a biblioteca desenvolvida e demais ferramentas com o mesmo propósito;
- Estudo aprofundado da interoperabilidade via XMI entre as ferramentas de modelagens disponíveis no mercado, dado que é argumentado no decorrer de toda a monografia que XMI não é interoperável.

O código-fonte da biblioteca *UnB-DALi* está disponível para *download* via repositório público no *GitHub* do grupo de pesquisas *LaDeCiC*: <https://github.com/lesunb/UnB-DALi>. A biblioteca está documentada conforme padrão Javadoc e instruções sobre os padrões adotados para versionamento, controle de dependências e compilação do projeto são dados no arquivo *README.md*. Por fim, a biblioteca está licenciada sobre a Licença MIT.

Referências

- [1] A. Avižienis, JC. Laprie, B. Randell e C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transaction on*, 1(1):11–33, 2004. [2](#), [5](#), [6](#)
- [2] A.E.C. de Oliveira. UnB-DALi: The UnB Dependability Analysis Library, url="<https://github.com/lesunb/UnB-DALi/>", 2016. [57](#)
- [3] C. Baier e JP. Katoen. *Principles of Model Checking*. The MIT Press, 2008. [x](#), [1](#), [2](#), [7](#), [8](#), [10](#), [11](#), [13](#)
- [4] C. Ermel, M. Rudolf e G. Taentzer. Handbook of graph grammars and computing by graph transformation. chapter The AGG Approach: Language and Environment, pages 551–603. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999. [31](#)
- [5] C. Ghezzi e A.M. Sharifloo. *Dependable Computer Systems*, chapter Quantitative Verification of Non-functional Requirements with Uncertainty, pages 47–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. [35](#), [38](#)
- [6] C.A.R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. [16](#)
- [7] D.F. Mendonça. *Dependability Verification for Contextual-Runtime Goal Modelling*. Universidade de Brasília, UnB, Brasília, DF, Brazil, 2015. [67](#)
- [8] E.M. Clarke, E.A. Emerson, e A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986. [10](#)
- [9] G. Taentzer. Agg: A graph transformation environment for modeling and validation of software. In JohnL Pfaltz, Manfred Nagl, and Boris Böhnen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer Berlin Heidelberg, 2004. [27](#), [31](#)
- [10] G.N. Rodrigues. LaDeCiC: <http://dgp.cnpq.br/dgp/espelhogrupo/098777969802265>, 2015. [Online; acessado em 2016-01-02]. [2](#)
- [11] G.N. Rodrigues, D.S Rosenblum e S. Uchitel. Sensitivity analysis for a scenario-based reliability prediction model. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005. [8](#), [48](#), [50](#)

- [12] G.N. Rodrigues, V. Alves, R. Franklin e L. Laranjeira. Dependability analysis in the ambient assisted living domain: An exploratory case study. In *Software Components, Architectures and Reuse (SBCARS), 2010 Fourth Brazilian Symposium on*, pages 150–159, 2010. [x](#), [2](#), [7](#), [14](#), [15](#), [17](#), [18](#), [20](#), [45](#)
- [13] Object Management Group. XML Metadata Interchange (XMI) Specification. Technical report, Object Management Group, April 2014. [35](#), [38](#)
- [14] H. Ehrig, K. Ehrig, U. Prange e G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. [42](#)
- [15] H. Ehrig, U. Prange e G. Taentzer. Fundamental theory for typed attributed graph transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 161–177. Springer Berlin Heidelberg, 2004. [21](#), [27](#), [29](#), [30](#)
- [16] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange e G. Taentzer. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.*, 376(3):139–163, May 2007. [28](#), [29](#)
- [17] J. Gelick. A Bug and a Crash: Sometimes a Bug Is More Than a Nuisance, Dezembro 1996. [Online; acessado em 2015-06-06]. [2](#)
- [18] K. Czarnecki e U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. [21](#)
- [19] K. Ehrig, E. Guerra, J. De Lara, L. Lengyel, U. Prange, G. Taentzer, D. Varro e et al. Model transformation by graph transformation: A comparative study. In *In MTIP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MODELS 2005)*, 2005. T. Rohit Gheyi, pages 71–80, 2006. [x](#), [29](#), [30](#), [31](#)
- [20] L. Hoffman. In search of dependable design. *Commun. ACM*, 51(7):14–16, July 2008. [7](#)
- [21] M. Biehl. Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology, July 2010. [x](#), [21](#), [22](#)
- [22] M. Elaasar e Y. Labiche. *Modelling Foundations and Applications: 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, chapter Model Interchange Testing: A Process and a Case Study, pages 49–61. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. [38](#)
- [23] M. Kwiatkowska, G. Norman e D. Parker. PRISM: Probabilistic symbolic model checker. In P. Kemper, editor, *Proc. Tools Session of Aachen 2001 International Multi-conference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 7–12, September 2001. [11](#)

- [24] M. Kwiatkowska, G. Norman e D. Parker. Advances and challenges of probabilistic model checking. In *Proc. 48th Annual Allerton Conference on Communication, Control and Computing*, pages 1691–1698. IEEE Press, 2010. 9
- [25] M. Kwiatkowska, G. Norman e D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan e S. Qadeer, editor, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011. 8, 9
- [26] Modeliosoft. Modelio: <https://www.modelio.org/>, 2015. [Online; acessado em 2015-06-08]. 27
- [27] M.R. Lyu. Software reliability engineering: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 153–170, Washington, DC, USA, 2007. IEEE Computer Society. 7
- [28] NuSMV. Nusmv website: <http://nusmv.fbk.eu/>, 2016. [Online; acessado em 2016-01-06]. 8
- [29] OMG. *Meta Object Facility (MOF) Core Specification, Version 2.0*. Object Management Group, 2006. 22, 23, 38
- [30] OMG. *OMG Unified Modeling Language*. Object Management Group, 2015. 26, 27, 35, 45, 46, 49
- [31] OMG. Website: <http://www.omg.org/>, 2015. [Online; acessado em 2015-06-05]. 38
- [32] R. Heckel, J.M. Küster e G. Taentzer. Confluence of typed attributed graph transformation systems. In *Proceedings of the First International Conference on Graph Transformation*, ICGT '02, pages 161–176, London, UK, UK, 2002. Springer-Verlag. 30
- [33] R.S. Scowen. Extended bnf - a generic base standard. In *Proceedings of the 1993 Software Engineering Standards Symposium*, SESS '93, 1993. 22
- [34] S. Bernardi, J. Merseguer e D.C. Petriu. Dependability modeling and analysis of software systems specified with UML. *ACM Comput. Surv.*, 45(1):2:1–2:48, December 2012. 5
- [35] S. Bernardi, J. Merseguer e D.C. Petriu. *Model-Driven Dependability Assessment of Software Systems*. Springer Publishing Company, Incorporated, 2013. xii, 13, 38
- [36] S. Garfinkel. History's Worst Software Bugs: <http://archive.wired.com/software/coolapps/news/2005/11/69355?currentPage=all>, November 2005. [Online; acessado em 2015-06-05]. 2
- [37] SPIN. Spin website: <http://spinroot.com/>, 2016. [Online; acessado em 2016-01-06]. 8
- [38] T. Mens e P. Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, March 2006. xii, 20, 22, 28, 33

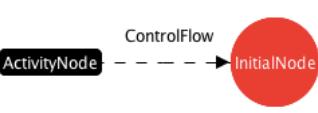
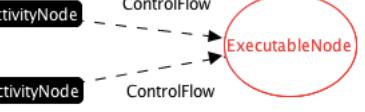
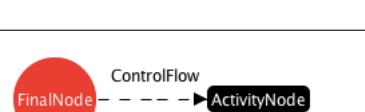
- [39] TU Berlin. Agg website: <http://user.cs.tu-berlin.de/~gragra/agg/index.html>, 2016. [Online; acessado em 2016-01-12]. 3, 33, 36
- [40] University of Oxford. Prism website: <http://www.prismmodelchecker.org/>, 2015. [Online; acessado em 2015-03-14]. 8, 13
- [41] W3C. Extensible Markup Language (XML) 1.0, 2013. [Online; acessado em 2016-01-28]. 38

Anexo A

Restrições Atômicas

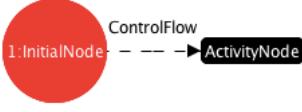
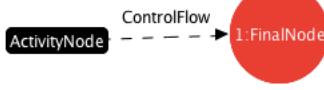
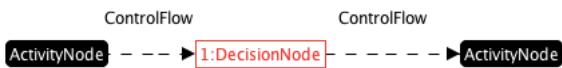
A.1 Diagrama de Atividades (AD)

Tabela A.1: Restrições atômicas aplicadas a grafos-instância do grafo-tipo de um AD.

Premissa	Conclusões	
		<i>WeirdCF:</i>
	 	Denota padrões incomuns de associações entre os nós de um grafo tipado sobre AD. Note que a premissa é vazia, logo todo o grafo está sujeito ao reconhecimento desses padrões. Dessa forma, temos que são incomuns: <i>InitialNode</i> como destino de um <i>ControlFlow</i> ; múltiplas associações via <i>ControlFlow</i> tendo um nó do tipo <i>DecisionNode</i> como destino; múltiplos <i>ControlFlows</i> com destino a um nó do tipo <i>ExecutableNode</i> ; múltiplos <i>ControlFlows</i> com destino a um nó do tipo <i>FinalNode</i> ; múltiplos <i>ControlFlows</i> com um nó do tipo <i>ExecutableNode</i> como origem; múltiplos <i>ControlFlows</i> com um nó do tipo <i>MergeNode</i> como origem; <i>FinalNode</i> como origem de um <i>ControlFlow</i> ; e loops de <i>ControlFlow</i> .
	 	
		

Continua na próxima página

Tabela A.1 – (Continuação)

<p>Premissa</p> 	<p>Conclusões</p> 	<p><i>MandatoryInitialCF:</i> Denota padrões obrigatórios de associações via <i>ControlFlow</i> entre um nó do tipo <i>Initial</i> e nós do tipo <i>ActivityNode</i>. Dessa forma, temos que um nó do tipo <i>InitialNode</i> não pode ocorrer solto.</p>
<p>Premissa</p> 	<p>Conclusões</p> 	<p><i>MandatoryFinalCF:</i> Denota padrões obrigatórios de associações via <i>ControlFlow</i> entre nós do tipo <i>ActivityNode</i> e um nó do tipo <i>FinalNode</i>. Dessa forma, temos que um nó do tipo <i>FinalNode</i> não pode ocorrer solto.</p>
<p>Premissa</p> 	<p>Conclusões</p> 	<p><i>MandatoryExecCF:</i> Denota padrões obrigatórios de associações via <i>ControlFlow</i> envolvendo nós do tipo <i>ExecutableNode</i>. Dessa forma, temos que um nó do tipo <i>ExecutableNode</i> necessita ter pelo menos um <i>ControlFlow</i> de entrada e um <i>ControlFlow</i> de saída.</p>
<p>Premissa</p> 	<p>Conclusões</p> 	<p><i>MandatoryDecisionCF:</i> Denota padrões obrigatórios de associações via <i>ControlFlow</i> envolvendo nós do tipo <i>DecisionNode</i>. Dessa forma, temos que um nó do tipo <i>DecisionNode</i> necessita ter pelo menos um <i>ControlFlow</i> de entrada e um <i>ControlFlow</i> de saída.</p>

Continua na próxima página

Tabela A.1 – (Continuação)

Premissa	Conclusões	
		<p><i>MandatoryMergeCF:</i> Denota padrões obrigatórios de associações via <i>ControlFlow</i> envolvendo nós do tipo <i>MergeNode</i>. Dessa forma, temos que um nó do tipo <i>MergeNode</i> necessita ter pelo menos um <i>ControlFlow</i> de entrada e um <i>ControlFlow</i> de saída.</p>

A.2 Diagrama de Sequência (SD)

Tabela A.2: Restrições atômicas aplicadas a grafos-instância do grafo-tipo de um SD.

Premissa	Conclusões	
		<p><i>WeirdRelations:</i> Identifica padrões incomuns de associações entre nós de um grafo tipado sobre AD. Dessa forma, temos que é incomum: <i>lifelines</i> com duas ou mais associações via <i>First</i>, seja com nós do tipo <i>Occurrence</i> diferentes ou iguais; nós do tipo <i>Occurrence</i> com duas ou mais associações via <i>Next</i> com outros nós do mesmo tipo; nós do tipo <i>AsyncMessage</i> com duas associações do tipo <i>Send</i> ou <i>First</i> com nós do tipo <i>Event</i>.</p>

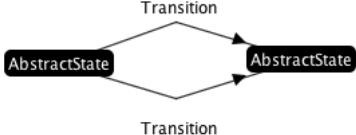
Continua na próxima página

Tabela A.2 – (Continuação)

Premissa	
	<i>MandatoryMessageRelations:</i> Identifica padrões obrigatórios de associações com nós do tipo <i>AsyncMessage</i> . Dessa forma, temos que todo nó <i>AsyncMessage</i> precisa obrigatoriamente estar associado via <i>Send</i> e <i>Receive</i> com nós do tipo <i>Event</i> .
	

A.3 Cadeias de Markov de Tempo Discreto (DTMC)

Tabela A.3: Restrições atômicas aplicadas a grafos-instância do grafo-tipo de um DTMC.

Premissa	
	<i>WeirdTransitions:</i> Identifica padrões incomuns de associações entre nós do tipo <i>AbstractState</i> via <i>Transition</i> . Dessa forma, temos que é incomum: múltiplas transições entre dois estados e múltiplas transições entre o mesmo estado.
	

Anexo B

Regras de Transformação

B.1 $AD \rightarrow DTMC$

Tabela B.1: Regras para transformação $AD \rightarrow DTMC$, organizadas por camadas. Camada 0: conecta diretamente todos os possíveis nós alcançáveis dos nós de decisão iniciais. Camadas 1 – 3 inicializa o contador, o nó inicial e os nós finais. Camada 5 resolve os casos intermediários. Camadas 6 e 7 limpam o grafo resultante.

LHS	RHS
<pre> graph TD 3[3:ActivityNode] -- "ControlFlow PTS=X" --> 2[2:MergeNode] 2 -- "S:ControlFlow PTS=Y" --> 1[1:ControlNode] </pre>	<pre> graph TD 3[3:ActivityNode] -- "ControlFlow PTS=X*Y" --> 2[2:MergeNode] 2 -- "S:ControlFlow PTS=Y" --> 1[1:ControlNode] </pre>
NACs	
3:DecisionNode	3:MergeNode
1.FinalNode	

[0] *ControlNodeMergeChain*:

Como nós do tipo *MergeNode* não possuem estado correspondente no DTMC resultante, é preciso resolver primordialmente os fluxos que passam por um *MergeNode*, de forma que todos os possíveis fluxos a partir do *ActivityNode* de origem estejam diretamente relacionados a este último.

Continua na próxima página

Tabela B.1 – (Continuação)

LHS	RHS	
		<p>[0] <i>ControlNodeDecisionChain</i>: Como nós do tipo <i>DecisionNode</i> não possuem estado correspondente no DTMC resultante, é preciso associar diretamente todos os possíveis fluxos via este <i>DecisionNode</i> com o nó de origem dos mesmos, ajustando adequadamente as probabilidade de transição.</p>
NACs		
		<p>[0] <i>ClearUselessDecision</i>: Responsável por remover os nós de decisão cujos fluxos já foram todos resolvidos.</p>
NACs		
		<p>[0] <i>ClearUselessMerge</i>. Responsável por remover os nós de merge cujos fluxos já foram todos resolvidos.</p>
NACs		
		<p>[1] <i>Count</i>: Responsável por inicializar o nó contador de estados. Por padrão, o nome dado ao DTMC resultante é "A0".</p>
NACs		

Continua na próxima página

Tabela B.1 – (Continuação)

LHS	RHS	
 <code>1:InitialNode</code> <code>2:scount count=X dtmc=D</code>	 <code>1:InitialNode</code> ← A2D → <code>initialState</code> <code>initialState</code> <code>label=initial</code> <code>index=X-1</code> <code>dtmc=D</code> <code>Transition probability=1.0 guard=""</code> <code>Transition probability=1.0 guard=""</code> <code>ErrorState</code> <code>label=D+"_fail"</code> <code>index=-1</code> <code>dtmc=D</code>	<p>[2] <i>Initial</i>:</p> <p>Responsável por inicializar um nó do tipo <i>InitialNode</i> e seu respectivo DTMC. Como <i>InitialNode</i> trata-se de uma execução, o mesmo traduz-se a uma transição entre dois estados, de <i>InitialState</i> para <i>State</i>, com probabilidade 1.0. O estado de erro global do DTMC resultante é criado, uma transição do estado correspondente ao <i>InitialNode</i> para o estado de erro é criada e o contador é adequadamente iterado. Como o estado de erro é estado final, temos que uma transição para ele mesmo com probabilidade 1.0 é criada.</p>
NACs		
	 <code>1:InitialNode</code> ← A2D	
LHS	RHS	
 <code>1:FinalNode</code> <code>2:scount count=X dtmc=D</code>	 <code>1:FinalNode</code> ← A2D → <code>State</code> <code>State</code> <code>label=final</code> <code>index=X-1</code> <code>dtmc=D</code> <code>Transition probability=1.0 guard=""</code>	<p>[3] <i>Final</i>:</p> <p>Responsável por inicializar um nó do tipo <i>FinalNode</i>, de forma que seu estado final correspondente, <i>FinalNode</i>, seja criado e o contador adequadamente iterado.</p>
NACs		
	 <code>1:FinalNode</code> ← A2D	

Continua na próxima página

Tabela B.1 – (Continuação)

LHS	RHS
<p>ActivityNode ControlFlow PTS=X</p> <p>1:ActivityNode</p> <p>6: A2D → 3:State</p> <p>10: 3:State → 5:ErrorState 13:Transition probability=Z</p> <p>8: 1:ActivityNode ← 2: A2D → 4:State</p> <p>9: 4:State → 3:State</p>	<p>1:ActivityNode</p> <p>8: 1:ActivityNode ← 2: A2D → 4:State</p> <p>9: 4:State → 3:State Transition probability=X guard=""</p> <p>13: 3:State → 5:ErrorState Transition probability=Z-X</p>
NACs	
<p>6: A2D → 3:State</p> <p>10: 3:State → 5:ErrorState Transition</p> <p>9: 2: A2D → 4:State</p>	
NACs	
LHS	RHS
<p>1:ActivityNode</p> <p>2:DecisionNode</p> <p>3:ActivityNode</p> <p>8:ControlFlow PTS=X</p> <p>10: 1:ActivityNode ← 4: A2D → 5:State</p> <p>11: 5:State → 6:ErrorState 12:Transition probability=Z</p> <p>7:scount count=T dtmc=D</p>	<p>1:ActivityNode</p> <p>2:DecisionNode</p> <p>3:ActivityNode</p> <p>8:ControlFlow PTS=X</p> <p>10: 1:ActivityNode ← 4: A2D → 5:State</p> <p>11: 5:State → 6:ErrorState Transition probability=X*Y guard=""</p> <p>7:scount count=T+1</p> <p>12: 5:State → 3:ActivityNode label="" index=T-1 dtmc=D Transition probability=1.0 guard=""</p>
NACs	
	<p>3:ActivityNode ← A2D</p>
LHS	RHS
<p>1:ActivityNode</p> <p>2:DecisionNode</p> <p>3:ActivityNode</p> <p>9:ControlFlow PTS=X</p> <p>11: 1:ActivityNode ← 4: A2D → 5:State</p> <p>12: 5:State → 6:ErrorState 13:Transition probability=Z</p> <p>7:A2D → 3:ActivityNode</p> <p>14: 3:ActivityNode ← 7: A2D → 8:State</p> <p>15: 8:State → 3:ActivityNode</p>	<p>1:ActivityNode</p> <p>2:DecisionNode</p> <p>3:ActivityNode</p> <p>9:ControlFlow PTS=X</p> <p>11: 1:ActivityNode ← 4: A2D → 5:State</p> <p>12: 5:State → 6:ErrorState Transition probability=X*Y guard=""</p> <p>7:A2D → 3:ActivityNode</p> <p>14: 3:ActivityNode ← 7: A2D → 8:State</p> <p>15: 8:State → 3:ActivityNode 13:Transition probability=Z-X*Y</p>

Continua na próxima página

Tabela B.1 – (Continuação)

LHS	RHS	
<p>LHS diagram illustrating the transformation rule [5]. It shows a state transition from 5 to 6 with probability Z. A merge node 2 leads to two activity nodes, 3 and 4. Activity node 3 has control flow PTS=X. Activity node 4 has control flow PTS=Y and leads to state 8 via transition 14.</p>	<p>RHS diagram for rule [5]. A merge node 2 leads to an activity node 3. Activity node 3 has control flow PTS=Y. Activity node 3 leads to state 8 via transition 14. State 8 leads to an error state 6 via transition 15 with probability $Z \cdot Y$.</p>	<p>[5] <i>AN2Merge2AppAN</i>: Identifica o já aplicado <i>3:ActivityNode</i> como possível fluxo a partir de um determinado <i>ActivityNode</i> via um <i>MergeNode</i>. Replica-se, portanto, esse fluxo como uma <i>Transition</i> no DTMC resultante, removendo o <i>ActivityNode</i> de origem do grafo e ajustando adequadamente as probabilidades de transição no DTMC. Não ocorre quando a transição no DTMC já existe.</p>
NACs		
<p>NACs diagram for rule [5]. A state transition from 5 to 8 via 14 is shown.</p>		
NACs		
<p>LHS diagram for rule [5]. An executable node 2 leads to state 4 via A2D. State 4 leads to an error state 5 via transition 10 with probability Z.</p>	<p>RHS diagram for rule [5]. An executable node 2 leads to state 4 via A2D. State 4 leads to an error state 5 via transition 10 with probability $Z \cdot X$. State 4 also leads to state 5 via transition 10 with probability $X \cdot Y$.</p>	<p>[5] <i>AN2Exec</i>: Identifica um encadeamento de nós de execução. Elimina o nó de origem, itera o contador e replica o encadeamento no DTMC resultante. Não ocorre caso <i>2ExecutableNode</i> já tenha sido aplicado.</p>
NACs		
<p>NACs diagram for rule [5]. An executable node 2 leads to A2D.</p>		
NACs		
<p>LHS diagram for rule [5]. It shows a state transition from 5 to 6 via 12 with probability Z. A merge node 3 leads to two activity nodes, 2 and 4. Activity node 2 has control flow PTS=X. Activity node 4 has control flow PTS=Y and leads to state 7 via transition 13.</p>	<p>RHS diagram for rule [5]. A merge node 3 leads to an activity node 2. Activity node 2 has control flow PTS=Y. Activity node 2 leads to state 7 via transition 13. State 7 leads to an error state 6 via transition 12 with probability $Z \cdot X \cdot Y$.</p>	<p>[5] <i>AN2Merge2AN</i>: Identifica <i>3:ActivityNode</i> é alcançável a partir de um determinado <i>ActivityNode</i> via um <i>MergeNode</i>. Cria-se um novo estado representando <i>2:ActivityNode</i>, replica-se esse fluxo como uma <i>Transition</i> no DTMC resultante, removendo o <i>ActivityNode</i> de origem do grafo, ajustando adequadamente as probabilidades de transição no DTMC e iterando o contador. Não ocorre quando o estado a ser criado já existe.</p>
NACs		
<p>NACs diagram for rule [5]. An activity node 2 leads to A2D.</p>		

Continua na próxima página

Tabela B.1 – (Continuação)

LHS	RHS	
		<p>[6] <i>ClearFinalAndCount</i>: Remove o mapeamento de um estado final a seu respectivo estado do DTMC.</p>
		<p>[6] <i>UntieDecisionNodes</i>: Remove do grafo nós do tipo <i>Decision-Node</i>. Não ocorre caso ainda existam fluxos saindo de um nó de decisão.</p>
NACs		
		<p>[6] <i>UntieMergeNodes</i>: Remove do grafo nós do tipo <i>Merge-Node</i>. Não ocorre caso ainda existam fluxos chegando a um nó de merge.</p>
NACs		
		<p>[7] <i>ClearActivityNodes</i>: Remove qualquer <i>ActivityNode</i> do grafo que já não possua mais associações de <i>ControlFlow</i> com nenhum outro elemento.</p>
NACs		

B.2 $SD \rightarrow DTMC$

Tabela B.2: Regras da transformação $SD \rightarrow DTMC$, organizadas por camada. Camada de nível 0 inicializa as *lifelines* do SD em transformação. Camada de nível 1 transforma os padrões associados com arestas do tipo *First*. Camada de nível 2 lida com padrões de trocas de mensagens envolvendo apenas arestas do tipo *Next*. E por fim, a camada de nível 5 limpa o grafo resultante.

LHS	RHS	
<div style="border: 1px solid red; padding: 5px; display: inline-block;"> 1:Lifeline BCompRel=R name=N </div>		<p>[0] <i>InitLifelines</i>: Inicializa as linhas de vida existentes no modelo de entrada, de forma que o contador de estados do DTMC resultante seja inicializado (<i>scount</i>), e de forma que um estado inicial e um estado de erro sejam adequadamente criados. O nome do DTMC correspondente é inicializado como o nome da <i>lifeline</i> em questão.</p>
<div style="border: 1px solid red; padding: 5px; display: inline-block;"> 1:Lifeline BCompRel=R name=N </div>	<div style="text-align: center;">NACs</div>	
	<div style="border: 1px solid red; padding: 5px; display: inline-block;"> 1:Lifeline BCompRel=R name=N </div>	
LHS	RHS	
		<p>[1] <i>FirstSend2FirstReceive</i>: Lida com o padrão de troca de mensagens em que as primeiras ocorrências nas <i>lifelines</i> de origem e destino fazem parte da mensagem em troca. Assim, removem-se os nós das <i>lifelines</i>, criam-se os respectivos estados e transições, ajustam-se as probabilidades e guardas das transições associadas e iteram-se os respectivos contadores de estados. Perceba que os contadores passam a ser associados aos respectivos eventos da mensagem.</p>

Continua na próxima página

Tabela B.2 – (Continuação)

LHS	RHS	
		<p>[1] <i>FirstSend2Receive</i>:</p> <p>Lida com o padrão de troca de mensagens em que o evento de origem é a primeira ocorrência da sua linha de vida e o evento de destino um evento que não é o primeiro na <i>lifeline</i> de destino. As transformações aplicadas são análogas as da regra <i>FirstSend2FirstReceive</i>.</p>
		<p>[1] <i>Send2FirstReceive</i>:</p> <p>Lida com o padrão de troca de mensagens em que o evento de destino é a primeira ocorrência da sua linha de vida e o evento de origem um evento que não é o primeiro na <i>lifeline</i> de origem. As transformações aplicadas são análogas as da regra <i>FirstSend2FirstReceive</i>.</p>
		<p>[1] <i>FirstSelfSend</i>:</p> <p>Trata uma troca de mensagens onde os eventos de origem e destino fazem parte da mesma linha de vida e onde o evento de origem é a primeira ocorrência na <i>lifeline</i> em questão.</p>

Continua na próxima página

Tabela B.2 – (Continuação)

LHS	RHS	
		<p>[2] <i>NextMessage</i>:</p> <p>Faz a transformação do SD para um DTMC quando uma troca de mensagens assíncronas é identificada entre dois eventos ordinários^a de duas <i>lifelines</i> distintas. O comportamento é análogo ao definido pela regra <i>FirstSend2FirstReceive</i>.</p> <p>^aeventos que não são a primeira ocorrência em sua respectiva linha de vida</p>
		<p>[2] <i>NextSelfSend</i>:</p> <p>Transforma uma troca de mensagens onde os eventos de origem e destino fazem parte da mesma linha de vida, eventos esses ocorridos em posições que não a primeira com respeito a sua <i>lifeline</i>.</p>
		<p>[5] <i>CleanOccurrences</i>:</p> <p>Dado a dinâmica que governa a estrutura das regras até então, os eventos soltos, ou seja, sem mensagens associadas, são removidos do grafo e convertidos em estados finais do DTMC correspondente.</p>
		<p>[5] <i>CleanLifelines</i>:</p> <p>As <i>lifelines</i> que não possuem ocorrência são tratadas nessa regra. Nesse caso, o DTMC resultante será apenas o estado inicial transitando com probabilidade 0.0 para o estado de erro do respectivo DTMC.</p>