



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Interoperabilidade entre a biblioteca UnB-DALi e ferramentas UML

Pedro Paulo Struck Lima
Jefferson Leandro da Silva

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

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

Brasília
2017

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

Coordenador: Prof. Dr. Rodrigo Bonifácio de Almeida

Banca examinadora composta por:

Prof.^a Dr.^a Genáina Nunes Rodrigues (Orientadora) — CIC/UnB
Prof. Dr. Fernando Antônio de A. C. de Albuquerque — CIC/UnB
Prof. Dr. André Luiz Peron Martins Lanna — CIC/UnB

CIP — Catalogação Internacional na Publicação

Lima, Pedro Paulo Struck.

Interoperabilidade entre a biblioteca UnB-DALi e ferramentas UML /
Pedro Paulo Struck Lima, Jefferson Leandro da Silva. Brasília : UnB,
2017.

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

Monografia (Graduação) — Universidade de Brasília, Brasília, 2017.

1. dependabilidade, 2. transformação de modelos, 3. verificação de
modelos, 4. UML, 5. XMI, 6. UnB-DALi

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**

Interoperabilidade entre a biblioteca UnB-DALi e ferramentas UML

Pedro Paulo Struck Lima
Jefferson Leandro da Silva

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

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

Prof. Dr. Fernando Antônio de A. C. de Albuquerque Prof. Dr. André Luiz Peron Martins Lanna
CIC/UnB CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 06 de dezembro de 2017

Dedicatória

Pedro: Dedico este trabalho aos meus pais, Raifran e Rose, à minha namorada, Kássia, e à minha orientadora, Genáina. A todos vocês, meu muito obrigado.

Jefferson: Dedico este trabalho como uma pequena contribuição ao progresso da humanidade.

Agradecimentos

Pedro: Agradeço aos meus pais pelo incentivo e amor. À Kássia, companheira que me dá amor, carinho, ouvidos e conselhos. A Cal Newport e Timothy A. Pynchyl, por me ajudarem a procrastinar menos depois que procrastinei lendo seus livros. Por último à Genaína, orientadora compreensiva que me ajudou muito nas nossas conversas que fugiam do escopo deste trabalho. Muito obrigado a todos vocês.

Jefferson: Agradeço a minha família, por todo o apoio. Aos meus amigos, por todas as boas lembranças. Ao universo, por me dar essa chance de viver minha atual experiência. E à Genaína, por toda a paciência, compaixão e orientação.

Resumo

Uma possível forma de se aferir a dependabilidade de um sistema computacional é por meio de verificação formal de modelos. Esta técnica de verificação de modelos, implementada por ferramentas automáticas como o PRISM (Probabilistic Symbolic Model Checker), requer um modelo devidamente anotado e passível de análise, ou seja, em uma linguagem formal específica. Porém, considerável parte dos sistemas computacionais são modelados em UML, uma linguagem expressa, geralmente, por meio de desenhos, não passíveis de verificação formal. Sendo assim, é necessário transformar um modelo UML para um modelo passível de análise. Oliveira propôs e implementou uma biblioteca Java, a *UnB-DALi*, que fornece uma API capaz de conduzir essa transformação de forma automática com a ajuda do motor de grafos AGG (Attributed Graph Grammar tool). No entanto, a entrada de dados da *UnB-DALi* requer uma descrição textual do modelo em código Java, o que não é geralmente feito quando se modela em UML. Porém, várias ferramentas de modelagem UML são capazes de gerar um arquivo textual, em formato XMI, contendo todas as informações dos elementos do diagrama UML. O problema é que este arquivo é proprietário, único de cada ferramenta, e sem a preocupação de aderir aos padrões estabelecidos para arquivos XMI conforme a OMG. Dessa forma, a interoperabilidade dessas ferramentas UML com a biblioteca *UnB-DALi* pode ser seriamente comprometida. Esta monografia se propõe a fazer a interoperabilidade padronizada entre os arquivos em formatos XMI de diversas ferramentas e a biblioteca *UnB-DALi*. Para tanto, utilizamos recursos da ferramenta SDMetrics, a qual contém um analisador sintático para XMI. Por fim, avaliamos nossa proposta por meio de testes funcionais definidos em classes de equivalência entre os modelos UML nas ferramentas Papyrus e Astah e seus resultados gerados em PRISM por meio das chamadas à *UnB-DALi*.

Palavras-chave: dependabilidade, transformação de modelos, verificação de modelos, UML, XMI, *UnB-DALi*

Abstract

A possible way to assess the dependability of a computational system is by formal system verification. This technique of model verification, implemented by automatic tools like PRISM (Probabilistic Symbolic Model Checker), requires a properly annotated model and subject to analysis, that is, in a specific formal language. However, a considerable amount of computational systems are modeled using UML, a language commonly expressed with graphic representations, not subject to formal verification. Thus, it is necessary to transform an UML model to a model that is subject to analysis. Oliveira proposed and implemented a Java library, the UnB-DALi, that provides an API capable of performing this transformation automatically, using the graph engine of AGG (Attributed Graph Grammar tool). However, the data entry is reliant on a textual description of the model in Java language, which is not generally common when your modeling in UML. But many UML modeling tools are capable of exporting a textual file in XMI format that contains all the elements information of the UML diagram. The problem is that this file is proprietary, unique of each tool, and does not conform to the standards established for XMI files according to the OMG. Thus, the interoperability between these tools and UnB-DALi could be severely compromised. This monograph proposes to enable the interoperability between these files in different XMI formats and UnB-DALi library. To do so, we evaluate our solution with well defined functional tests in equivalence class partitioning between the UML models by the tools Papyrus and Astah, and their generated results in PRISM through calls to UnB-DALi.

Keywords: dependability, model transformation, model checking, UML, XMI, UnB-DALi

Sumário

1	Introdução	1
1.1	Objetivo Geral	3
1.2	Objetivos Específicos	3
1.3	Estrutura do Documento	3
2	Fundamentação Teórica	4
2.1	Dependabilidade	4
2.2	Verificação de modelos	5
2.2.1	Especificação de Modelos	6
2.2.2	Especificação das Propriedades para Análise	7
2.2.3	Ferramenta de Verificação de Modelos PRISM	8
2.3	Análise de Dependabilidade	8
2.3.1	Metodologia do processo de Análise de dependabilidade	8
2.4	UnB-DALi	10
2.5	UML	11
2.5.1	Diagramas de Atividade	12
2.5.2	Diagramas de Sequência	13
2.6	XMI	14
2.6.1	Formas de acesso a dados do XMI	16
2.6.2	Trabalho relacionado	17
2.7	SDMetrics	17
2.7.1	Metamodelo SDMetrics	19
2.7.2	Arquivo de transformação SDMetrics	20
2.8	Considerações finais	22
3	XMI-PRISM-Converter	24
3.1	Requisitos	25
3.2	Arquitetura e Implementação	25
3.2.1	Arquitetura	26
3.2.2	Implementação	27
3.3	Considerações finais	32
4	Validação	34
4.1	Estratégia de Teste	34
4.2	Testes e Resultados Obtidos	34
4.2.1	AD \rightarrow PRISM	35
4.2.2	SD \rightarrow PRISM	40

4.3	Considerações sobre os testes	42
5	Conclusão e Trabalhos Futuros	43
	Referências	45
I	Estrutura do XMI das ferramentas de modelagem utilizadas	47
I.1	Astah Professional	47
I.2	Papyrus	49
II	Arquivos de configuração SDMetrics	51
II.1	Metamodelo SDMetrics	51
II.2	XMI Transformations	52
II.2.1	Astah Professional	53
II.2.2	Papyrus	55
II.2.3	<i>Template para XMI Transformation File</i>	56

Lista de Figuras

1.1	O foguete Ariane-5 explode momentos depois do lançamento [12]	2
2.1	Dependabilidade: seus atributos, ameaças e meios	6
2.2	Visão esquemática da abordagem <i>Model Checking</i> [3]	7
2.3	Processo de Análise de Dependabilidade [14]	9
2.4	Visão geral do funcionamento da UnB-DALi [2]	11
2.5	Taxonomia dos diagramas de estrutura e comportamento (com adaptações) [10].	12
2.6	Exemplo de um Diagrama de Atividades contendo todos os elementos reconhecidos pela UnB-DALi.	14
2.7	Exemplo de um Diagrama de Sequência contendo todos os elementos reconhecidos pela UnB-DALi.	15
3.1	Resumo da macro-etapa automatizada de conversão de modelos UML para modelos PRISM	24
3.2	Visão geral da arquitetura do projeto.	27
3.3	Diagrama de Classes do XMI-PRISM-Converter.	28
3.4	Diagrama de Sequência com as principais chamadas de métodos do programa.	29
3.5	Execução do XMI-PRISM-Converter em um terminal do Windows.	33

Lista de Tabelas

4.1	Resultados dos testes de conversão de ADs para PRISM	35
4.2	Resultados dos testes de conversão de SDs para PRISM	40

Capítulo 1

Introdução

O avanço da computação e sua utilidade para organizar informações fez com sua aplicação se tornasse bastante ampla. A computação pode estar presente tanto em uma simples calculadora como em um sistema computacional complexo utilizado na área da saúde. Neste último exemplo, a garantia do funcionamento correto esperado é essencial, dado que uma falha ocorrida neste contexto pode causar uma fatalidade para o usuário. Em outros sistemas, o mau funcionamento do sistema pode causar a perda de recursos importantes.

Os sistemas computacionais são projetados por meio de *software*, que contém o conjunto de instruções, em alguma linguagem de programação, que o computador deve executar para que se chegue a um resultado esperado. Apesar dos esforços dos envolvidos na concepção e implementação, erros ainda podem ocorrer, devido à ação ser de natureza humana. Vários acidentes críticos de *software* já ocorreram nos últimos tempos, como no caso do foguete Ariane-5 [3], da Agência Espacial Europeia, que entrou em auto-destruição após 36 segundos da decolagem e, na ocasião, causou um prejuízo de sete bilhões e meio de dólares [6]. Após uma investigação, foi constatado que ocorreu um *overflow* devido à conversão de um número em ponto flutuante de 64 bits em um valor inteiro de 16 bits.

Outro acidente decorrido de falha de *software* foi causado pela máquina Therac-25 [9], que era usada para terapia radioativa. A Therac-25 tinha dois tipos de funcionamento para terapias diferentes: uma utilizava um raio de elétrons de baixa intensidade e um modo de raios-X megavolt, que requeria uma barreira como escudo, filtros e uma câmara de íons para manter os raios no alvo. O problema foi o acondicionamento do *software* do modelo antigo no modelo novo, sem que fosse adequadamente testado. O resultado foi a exposição de raios-X sem a devida proteção e filtragem a pacientes que necessitavam do tratamento com raios de elétrons e, em decorrência disto, alguns pacientes morreram devido à exposição à radiação.

Tendo em vista a importância do bom funcionamento de certos sistemas computacionais, principalmente os sistemas críticos, aqueles nos quais uma eventual falha pode configurar em vítimas fatais, os estudiosos da área de engenharia de *software* estudam formas de garantir o bom funcionamento do *software*, verificando se o comportamento deste está de acordo com o que foi proposto no início. Existem vários modos de se verificar esta conformidade e nesta monografia o método que será focado é o *Model Checking*, onde um modelo do sistema em questão tem todos os cenários possíveis do sistema analisados de maneira sistemática. [3]



Figura 1.1: O foguete Ariane-5 explode momentos depois do lançamento [12]

Esta característica de confiabilidade do sistema é avaliada sob a ótica do conceito de Dependabilidade, que, na definição original, é explicada por Avizienis et al. no artigo *Basic Concepts and Taxonomy of Dependable and Secure Computing* [1] como a habilidade de entregar um serviço no qual se possa justificadamente confiar. De outra forma, a Dependabilidade verifica os requisitos mínimos a serem satisfeitos de um sistema computacional para que o usuário possa, comprovadamente, confiar no sistema. Rodrigues et al. [14] propuseram uma metodologia para análise de Dependabilidade usando um sistema de Ambiente de Vida Assistida como exemplo. Nessa metodologia, a primeira etapa é a especificação do sistema utilizando a linguagem UML (*Unified Modelling Language*), mais especificamente os diagramas de atividade (*Activity Diagrams* ou "AD") e os diagramas de sequência (*Sequence Diagrams* ou "SD"), que são diagramas UML de uma visão alto-nível dos componentes do sistema derivados da especificação arquitetural.

Em uma parte da metodologia, há a necessidade de conversão do modelo UML para o modelo PRISM, um programa que viabiliza a análise de Dependabilidade do sistema. A ferramenta PRISM tem linguagem própria e requer que o interessado na conversão a realize manualmente, deixando o processo mais lento e também aumentando a chance de erros ocorrerem. Tendo em vista essa etapa específica de transformação de modelos UML para PRISM, Oliveira [2] propôs, implementou e validou, a UnB-DALi (*UnB Dependability Analysis Library*), uma biblioteca Java capaz de conduzir a transformação do modelo UML em questão para um modelo PRISM, passível de *model checking*, utilizando a sintaxe abstrata inerente ao modelo: o grafo. É importante lembrar que Oliveira [2] projetou a biblioteca de modo que esta independa de detalhes da sintaxe de descrição de modelos UML, o XMI. Essa opção do autor tornou a biblioteca mais independente e reusável, mas tornou difícil expressar diagramas UML, mais especificamente ADs e SDs, tendo em vista que os diagramas devem ter os elementos instanciados um-a-um em linguagem Java.

O presente trabalho tem a intenção de possibilitar a interoperabilidade entre diagramas de comportamento UML e a biblioteca de Oliveira [2], no sentido de prover como

possibilidade de entrada de dados um arquivo no formato XMI descrevendo um modelo UML feito utilizando-se as ferramentas Astah e Papyrus para a modelagem dos diagramas. E também, objetiva-se mostrar que a solução proposta é flexível o suficiente para acomodar várias outras ferramentas. A partir do arquivo XMI será possível extrair informações dos elementos do modelo UML por meio de transformação de metamodelos facilitados por meio da ferramenta SDMetrics.

1.1 Objetivo Geral

Propor, implementar e validar um programa em linguagem Java que torna possível a interoperabilidade de diversas ferramentas de modelagem UML e a biblioteca UnB-DALi por meio de arquivos XMI e do analisador sintático da ferramenta SDMetrics.

1.2 Objetivos Específicos

- Aprofundar conhecimentos sobre a UML, o XMI e a linguagem Java;
- Estudar trabalhos relacionados a Transformações de modelos utilizando XMI;
- Implementar um programa Java que realiza a conversão de arquivos XMI contendo diagramas UML para arquivos na linguagem PRISM, por meio das biblioteca SDMetrics e UnB-DALi;
- Validar parcialmente¹ a ferramenta.

1.3 Estrutura do Documento

Os demais capítulos deste documento foram estruturados de maneira a explicar os conceitos básicos utilizados para se conseguir uma solução de transformação automática de modelos integrada à UnB-DALi, que é o principal objetivo deste trabalho.

O Capítulo 2 fornece a fundamentação teórica necessária para entender o processo da solução.

O Capítulo 3 explica a solução proposta e desenvolvida, detalhando seu funcionamento.

O Capítulo 4 exhibe os resultados de vários testes em tabelas e contém algumas considerações.

O Capítulo 5 é uma conclusão sobre o desenvolvimento deste projeto e o resultado obtido, bem como uma proposta de trabalhos futuros que podem ser desenvolvidos a partir daqui.

O Anexo 1, ao final da monografia, é um estudo a parte detalhando a estrutura do XMI das ferramentas de modelagem utilizadas.

¹Foram realizados testes funcionais com base no critério de classes de equivalência

Capítulo 2

Fundamentação Teórica

Em sistemas computacionais os requisitos são descrições do que o sistema deve fazer, como os serviços que ele deve prover e as restrições à sua operação [16]. Os requisitos são classificados em dois grupos: Requisitos funcionais e requisitos não-funcionais.

- Requisitos funcionais: São asserções dos serviços que o sistema deve fornecer, como o sistema deve reagir a entradas específicas e como o sistema deve se comportar em situações específicas. Em alguns casos, os requisitos funcionais podem também explicitamente afirmar o que o sistema não deve fazer.
- Requisitos não-funcionais: São as restrições nos serviços ou funções oferecidas pelo sistema. São características como restrições temporais, restrições no processo de desenvolvimento e restrições impostas por padrões. Requisitos não funcionais geralmente se aplicam ao sistema como todo, em vez de serviços ou características individuais.

A Dependabilidade trabalha com requisitos não funcionais. Esses requisitos são geralmente quantificáveis e podem ser testados de alguma forma. Uma afirmação como “A funcionalidade X do Sistema Y deve produzir uma resposta correta 99,99% das vezes” é um exemplo de requisito não-funcional.

2.1 Dependabilidade

Segundo Avizienis et al., “Dependabilidade é a habilidade de entregar um serviço no qual se possa, justificadamente, confiar”. [1] Uma outra definição dos mesmos autores diz que “A Dependabilidade de um sistema é a habilidade de se evitar falhas de serviço que são mais frequentes e severas do que o aceitável”.

Atualmente há vários meios para se aferir a Dependabilidade de sistemas computacionais, tanto quantitativamente quanto qualitativamente. Para fazer essas aferições, os métodos buscam testar os atributos de Dependabilidade dos sistemas que estão sob teste. Os seguintes atributos fazem parte da definição de Dependabilidade:

- Disponibilidade: prontidão na entrega correta dos serviços.
- Confiabilidade: continuidade de serviço correto.

- Segurança Operacional: ausência de consequências catastróficas aos usuários e ao meio.
- Integridade: ausência de alterações impróprias no sistema.
- Manutenibilidade: habilidade de submeter-se a modificações e reparos.

Um serviço correto é entregue quando este executa a função proposta do sistema. Porém, nem sempre um serviço correto é entregue, o que pode comprometer a confiabilidade do sistema. As causas dessa execução errada são chamadas de ameaças. A seguir, estão as definições das ameaças à Dependabilidade:

- Defeito: Um evento que ocorre quando o serviço entregue é diferente do serviço correto. O defeito de um serviço ocorre porque este não está acordo com a especificação funcional ou porque a atual especificação não descreve adequadamente a função esperada do sistema.
- Erro: Um desvio de serviço correto em pelo menos um ou mais estados externos.
- Falha: A julgada ou hipotetizada causa de um erro.

Nos últimos 60 anos, meios têm sido buscados para se satisfazer os atributos da Dependabilidade. Esses meios podem ser agrupados em 4 grandes grupos:

- Prevenção de falhas: Meio para se prevenir a ocorrência ou introdução de falhas.
- Tolerância a falhas: Meio para evitar falhas de serviço na presença de falhas.
- Remoção de falhas: Meio para reduzir a quantidade e a severidade das falhas.
- Previsão de falhas: Meio para estimar a quantidade atual, a incidência no futuro e as prováveis consequências das falhas.

A Figura 2.1 a seguir resume todos os elementos relacionados à Dependabilidade.

Dada esta introdução à Dependabilidade, destacamos que a abordagem utilizada neste trabalho atua como um meio de previsão de falhas, dado que a metodologia de análise de Dependabilidade é aplicada nos estágios iniciais do desenvolvimento do *software*.

2.2 Verificação de modelos

Uma das formas de se verificar o funcionamento correto de um sistema de *software* é por meio de técnicas de verificação modelo-dirigidas, as quais são baseadas em modelos que descrevem os comportamentos do sistema de maneira precisa, matemática e não-ambígua. Uma destas técnicas é conhecida como verificação de modelos (ou *Model Checking*, em inglês).

O *Model Checking* é uma abordagem que lida com o seguinte problema: dado o modelo do sistema, exaustiva e automaticamente, verifique se o modelo corresponde à especificação [20]. Uma possível técnica para se aplicar a abordagem é com o uso de ferramentas de verificação automatizadas (*model checkers*, em inglês). Neste caso, a ferramenta de análise

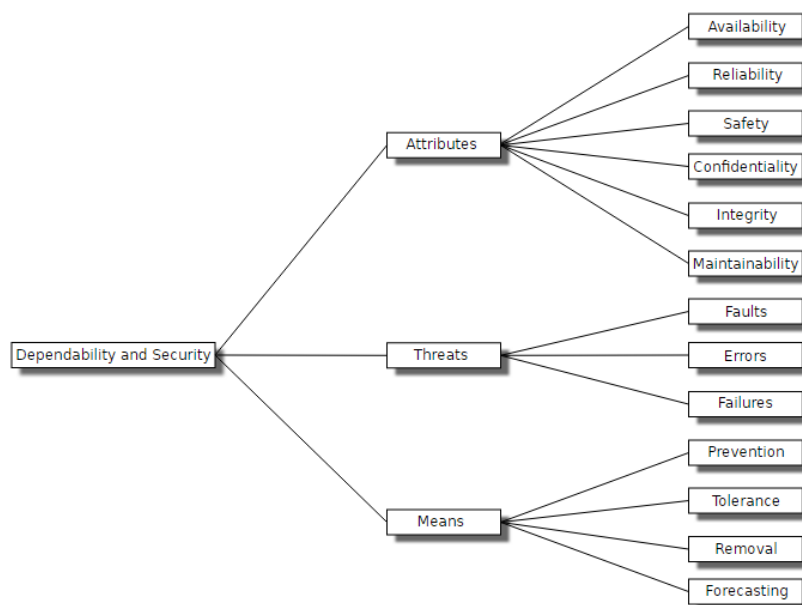


Figura 2.1: Dependabilidade: seus atributos, ameaças e meios

checa todos os cenários possíveis do sistema de maneira sistemática. Dessa forma, é possível verificar se o modelo do sistema satisfaz alguma propriedade específica que se quer analisar. Estas propriedades devem ser expressas em linguagem matemática não-ambígua.

Na simulação, se um estado que viola a condição da propriedade é encontrado, a ferramenta de *Model Checking* fornece um contra-exemplo que indica como o modelo foi capaz de atingir o estado indesejado. O contra-exemplo descreve um fluxo de execução que vai do estado inicial de execução até o estado que viola a propriedade sendo verificada. Com este caminho em mãos, o usuário analisador pode refazer o caminho e anotar informações relevantes para depuração e adaptação do modelo ou da propriedade posteriormente. A Figura 2.2 mostra um esquema geral da abordagem *Model Checking*:

Na prática, as propriedades medidas por ferramentas de *Model Checking* podem ser difíceis de se garantir. Uma pergunta como “O sistema nunca falha” é de natureza qualitativa e difícil de ser comprovada. Já uma propriedade menos abrangente como “O sistema funciona 98% das vezes.” tem natureza quantitativa e pode ser mais facilmente comprovada. Quando fazemos análises com afirmações menos absolutas, como a última, podemos aplicar a técnica de Verificação Probabilística de Modelos (*Probabilistic Model Checking*, em inglês). Esta abordagem nos permite fazer uma análise mais realista, incluindo elementos de natureza estocástica, ou seja, elementos que nem sempre são esperados que ocorram durante a execução mas podem ocorrer.

2.2.1 Especificação de Modelos

Para modelar tais ocorrências, é necessário ampliar as características do sistema de transição (o modelo) e incluir informações acerca da probabilidade de transições entre os estados. Neste caso, usaremos as chamadas Cadeias de Markov. Nas Cadeias de Markov o estado seguinte a ser transitado na execução é escolhido de acordo com uma distribuição probabilística que só depende do estado atual. Sendo assim, o desenrolar da execução

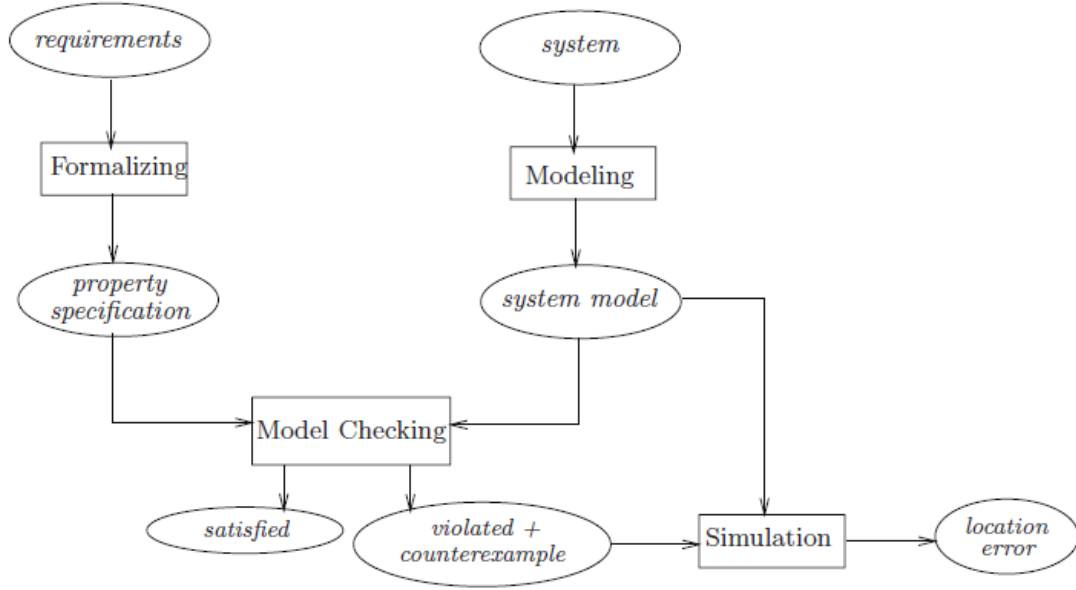


Figura 2.2: Visão esquemática da abordagem *Model Checking* [3]

do sistema não depende do caminho até então trilhado pelas transições anteriores. Essa propriedade das Cadeias de Markov é chamada de propriedade sem memória (*memoryless property*.) A seguir, a definição formal de Cadeias de Markov [3]:

Definição 1 (Cadeia de Markov de Tempo Discreto) Uma cadeia de Markov de Tempo Discreto (*DTMC*) é uma M-tupla $= (S, P, \iota_{init}, AP, L)$ onde:

- S é um conjunto de estados contável e não-vazio,
- $P : S \times S \rightarrow [0,1]$ é a função de transição probabilística tal que para todos os estados s :

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

- $\iota_{init} : S \rightarrow [0,1]$ é a distribuição inicial, tal que $\sum_{s \in S} \iota_{init}(s) = 1$, e
- AP é um conjunto proposições atômicas e $L : S \rightarrow 2^{AP}$ é uma função rotuladora.

M é chamado finito se S e AP são finitos. Para M finito, o tamanho de M , denotado $size(M)$, é o numero de estados mais o número de pares (s, s') pertencentes $S \times S$ com $P(s, s') > 0$.

2.2.2 Especificação das Propriedades para Análise

A visão esquemática da abordagem de verificação de modelos prevê que os requisitos do sistema sejam expressos de alguma forma. A forma utilizada mais comum para DTMCs é a lógica temporal PCTL (*Probabilistic Computational Tree Logic*).

Definição 2 (Gramática de PCTL) A gramática de PCTL é definida da seguinte forma:

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

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

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

Fórmulas PCTL são interpretadas sobre os estados e os caminhos de uma cadeia de Markov. Para a fórmula de estados, a relação de satisfatibilidade (\models) é uma relação entre os estados da cadeia de Markov e as fórmulas de estados. Em relação a um estado s , dizemos que $s \models \Phi$ se e somente se Φ é válido em s .

2.2.3 Ferramenta de Verificação de Modelos PRISM

A ferramenta PRISM fornece as capacidades esperadas de um verificador de modelos, de acordo com a figura anterior. Ela suporta sistemas modelados como DTMCs, que são o foco nesta monografia, e fornece algumas interfaces gráficas para que o usuário tenha controle da análise. As três principais interfaces gráficas para o acompanhamento da análise são: O editor de texto (para a representação do modelo em análise em linguagem PRISM), o simulador (com um gerador de caminhos com o objetivo de depuração do modelo) e o ambiente de especificação de propriedades em análise (para a entrada das propriedades, em PCTL, que serão testadas pela execução sistemática do modelo).

PRISM, inicialmente desenvolvido na Universidade de Birmingham e agora sendo mantido pela Universidade de Oxford, é multiplataforma, código-livre, licenciado sobre GPL e um ambiente de *Probabilistic Model Checking* em constante atualização com suporte para os mais diversos tipos de modelos e lógicas temporais. Maiores detalhes podem ser obtidos via endereço eletrônico da aplicação [17].

2.3 Análise de Dependabilidade

A análise de dependabilidade a ser feita varia de acordo com a representação do sistema a ser usada (o modelo ou o próprio sistema em si), a fase de desenvolvimento na qual o sistema se encontra (estágios iniciais, meio ou fim) e também a adequação da representação do modelo com o comportamento que é esperado de fato. A análise de dependabilidade em questão está sob a classificação de *Markov Analysis*, mais especificamente a análise de modelo probabilístico, onde o modelo do sistema é representado por um DTMC e as propriedades são formalizadas por meio de PCTL. Esta análise é baseada nos estágios iniciais do desenvolvimento do sistema em análise.

2.3.1 Metodologia do processo de Análise de dependabilidade

De acordo com a metodologia proposta por Rodrigues et al. [14], o processo de análise de dependabilidade consiste em oito passos divididos em duas macro etapas:

1. Especificação e conversão do sistema em UML para PRISM - Consiste nos passos 1 a 3.

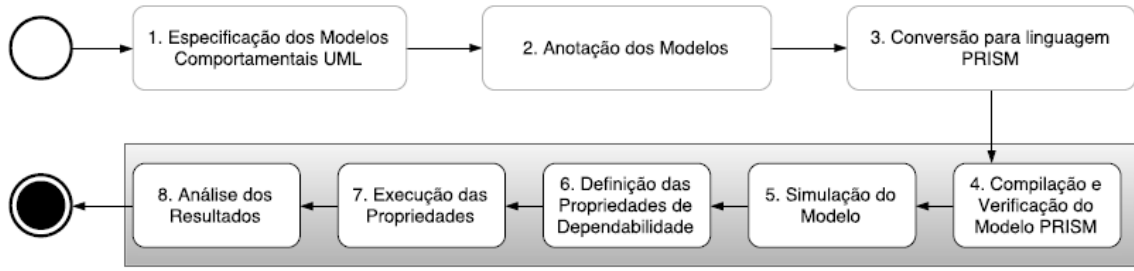


Figura 2.3: Processo de Análise de Dependabilidade [14]

2. Análise de dependabilidade do modelo PRISM - Consiste nos passos 4 a 8.

Focaremos nos passos 1 a 3, pertencentes à fase de especificação e conversão do modelo, tendo em vista que este trabalho se preocupa com a conversão automática dos modelos em si, e não sua análise de dependabilidade propriamente dita. Todavia, será realizada uma análise básica do modelo PRISM gerado, bem como sua compilação, o que consiste no passo 4. Tal análise terá o objetivo de validar parcialmente as transformações realizadas pela ferramenta construída nesta monografia.

Especificação dos Modelos Comportamentais UML

Como a análise será realizada no começo do desenvolvimento, utilizam-se diagramas UML para representar uma visão alto nível do comportamento dos componentes provenientes da especificação. Esses comportamentos compõem os cenários de uso e são representados por Diagramas de Atividade (*Activity Diagrams* ou ADs) e Diagramas de Sequência (*Sequence Diagrams* ou SDs). Os ADs são compostos por nós de ação e nós de decisão. Nós de ação representam uma execução de um cenário, o qual é representado por um SD [14].

Esses dois tipos de diagramas da UML e suas representações em XML, assunto de grande importância para o objetivo desta monografia, serão melhores explorados no capítulo seguinte.

Anotação dos Modelos

Após a modelagem do sistema é necessário enriquecê-lo com informações probabilísticas sobre seus componentes (no caso de SDs, a confiabilidade de seus componentes) e suas transições (no caso de ADs). A probabilidade de uma transição PT_{ij} é a probabilidade do controle de execução ser transferido diretamente do cenário S_i para o cenário S_j . A soma de todas as probabilidades de transição de um cenário S_i para um cenário S_j é igual a um.

Nos SDs, as seguintes três premissas são levadas em consideração:

1. Uma mensagem de um componente C para um componente C' representa uma invocação de C de um serviço oferecido por C'. A confiabilidade com a qual este serviço é realizado é a confiabilidade de C', $R_{C'}$.
2. A transferência de controle entre os componentes seguem uma propriedade de Cadeias de Markov de Tempo Discreto, o que significa que uma transição de um estado

de execução para outro é dependente somente do estado de origem e suas transições possíveis, e não das transições de estado passadas.

3. Falhas são independentes entre as transições

Conversão de Modelos UML para PRISM

Este passo diz respeito ao mapeamento dos modelos UML anotados para modelos em linguagem PRISM. Como dito anteriormente, os nós de ação dos ADs representam cenários de execução, cada um representado como um SD. Inicialmente, cada nó executável se torna um módulo em PRISM.

O processo de conversão consiste em primeiro construir um modelo de máquina de estados que representa cada nó do AD. Além disso, para cada execução de serviço de um componente no SD que modela o nó executável, há um estado correspondente na máquina de estados onde as mensagens trocadas pelos componentes no SD são representadas como transições rotuladas entre os estados. Logo, para cada estado no modelo de máquina de estados há um componente C executando um serviço. Por isso, cada estado está associado à confiabilidade de algum componente, R_C , ou seja, a probabilidade de uma execução de serviço bem sucedida, em contrapartida com uma execução com falha, de probabilidade $(1-R_C)$, representando uma transição para o estado de erro E .

2.4 UnB-DALi

Na conclusão do artigo, Rodrigues et al. [14] mencionam que a metodologia ainda precisa de refinamentos como estudos de caso em diferentes domínios e uma forma de automatizar a conversão de modelos UML em modelos PRISM. A etapa de conversão de modelos, além de ser trabalhosa, é muito propensa a erros humanos, podendo fazer com que os resultados obtidos na análise levem a conclusões equivocadas acerca do sistema.

Com o intuito de resolver o problema da automatização das transformações, Oliveira [2] propôs, implementou e validou parcialmente uma biblioteca Java, a *Unb-Dependability Analysis Library*, ou *UnB-DALi*, que é reusável, interoperável e matematicamente rigorosa, o que resolvia o problema da conversão automática, eliminando, em pelo menos um passo do processo, a possibilidade de erro humano.

Na Figura 2.4 abaixo, é apresentada uma visão geral do funcionamento da biblioteca.

A parte inicial da UnB-DALi, “Instância de Modelo Fonte”, utiliza métodos para criação de objetos e inclusão destes elementos do modelo como objetos passíveis de transformação para DTMC. Alguns exemplos de métodos, em um Diagrama de Atividades, por exemplo, são *addExecutableNode(ExecutableNode node)*, que recebe um Nó Executável e o adiciona ao Diagrama de Atividades anteriormente instanciado e *addControlFlow(ControlFlow edge)*, que recebe uma aresta ou transição e a adiciona ao Diagrama de Atividades.

No entanto, Oliveira [2] tomou uma decisão de projeto que tornou a entrada dos diagramas para a conversão ainda muito verbosa e exigia que o usuário codificasse os componentes do diagrama UML a ser transformado com comandos Java, um a um, o que ainda seria um pouco trabalhoso, mesmo que o usuário não precisasse mais escrever seus diagramas na linguagem PRISM. Para esta decisão, ele argumentou que o espaço

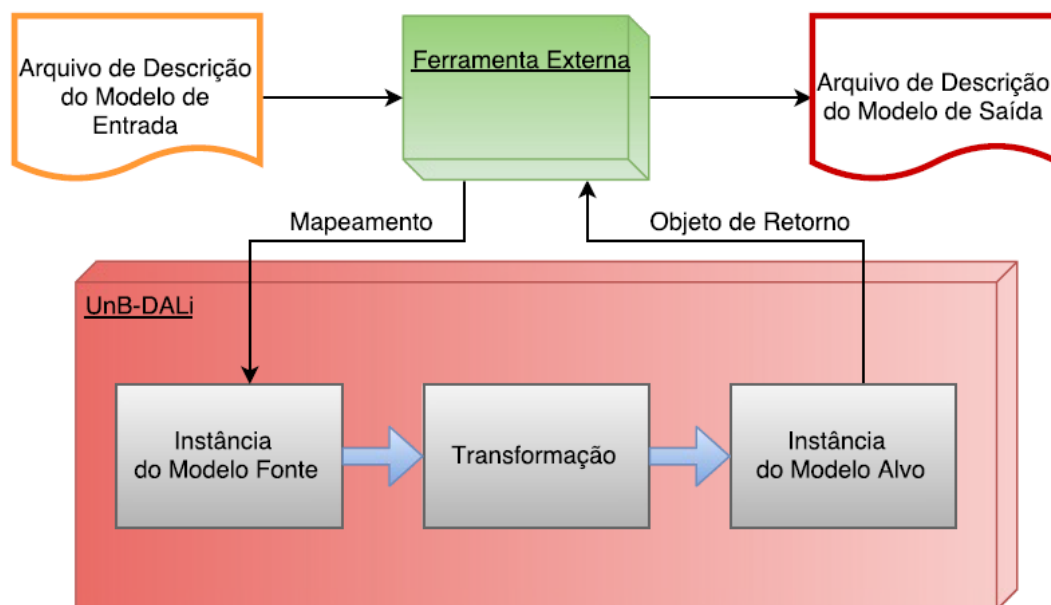


Figura 2.4: Visão geral do funcionamento da UnB-DALi [2]

tecnológico geralmente utilizado para persistir as informações de digramas UML, o XMI, é um padrão utilizado por diferentes ferramentas de modelagem que ainda causa muitas divergências e, portanto, não há garantias de que um modelo textual gerado por uma ferramenta seja igual ao mesmo modelo gerado por outra ferramenta. Dessa forma, a *UnB-DALi* trabalha de forma desacoplada de arquivos de persistência de modelos, como o XMI.

Com este fato, coube a trabalhos futuros averiguar a possibilidade de se trabalhar com arquivos de persistência no formato XMI, tendo em vista que a modelagem de sistemas através de diagramas UML é muito mais fácil de ser realizada com o auxílio de ferramentas gráficas. Várias são as ferramentas de modelagem UML no mercado e entre elas podem haver (e geralmente há) divergências entre os padrões de arquivos XMI. Esta monografia vem com o objetivo de propôr, implementar e validar parcialmente uma ferramenta Java, que usa a biblioteca criada por Oliveira (entre outras que serão explicadas mais adiante) para fazer esta ponte entre os arquivos de persistência em formato XMI gerado por ferramentas gráficas de modelagem UML do mercado. Logo, na etapa de “mapeamento” para a “instância do modelo fonte” é onde se encontra o foco deste trabalho.

Inicialmente a ferramenta dará suporte à conversão de diagramas gerados pelas ferramentas Astah e Papyrus. No entanto, a ferramenta foi construída de tal forma que a adição de suporte a novas ferramentas será facilmente realizada.

2.5 UML

Um sistema de *software*, preferivelmente antes de sua implementação, é pensado em um nível de abstração mais alto, de forma a se planejar como as diferentes partes do sistema se comunicarão e atuarão para resolver os problemas especificados nos requisitos. Uma forma natural de se planejar uma solução que vem à cabeça é “desenhar o que se quer fazer”. No entanto, deve-se haver certo cuidado com a forma deste desenho.

Devemos nos lembrar que os desenvolvedores de *software* não trabalham isolados, mas sim com uma série de pessoas interessadas no projeto (*stakeholders*), e nem todas essas pessoas possuem um *background* em sistemas de informação [16]. Logo, é importante se encontrar uma linguagem que satisfaça tanto a necessidade de compreensão por parte dos *stakeholders* que não possuem o *background*, tanto quanto a capacidade de se expressar o sistema em diferentes níveis de abstração para os desenvolvedores.

A UML (do inglês *Unified Modeling Language*) é uma linguagem padrão adotada pela indústria para representar modelos de estruturas de projetos de *software*. Foi estabelecida em 1994 por Grady Booch, Ivar Jacobson e James Rumbaugh. É uma linguagem de propósito geral utilizada por Arquitetos de *Software*, Engenheiros de *Software* e desenvolvedores para especificar, visualizar, construir e documentar artefatos de *software* [10]. O principal objetivo do desenvolvimento desta linguagem é estabelecer um padrão para modelagem de sistemas pela indústria em geral. Desde 1997, a linguagem UML é adotada pelo *Object Management Group* (OMG). Em 2005 ela também foi adotada como padrão pela *International Organization for Standardization* (ISO) [10].

Na Figura 2.5 abaixo observa-se que existem vários tipos de diagramas na UML, porém este trabalho terá um total enfoque nos Diagramas de Atividade e os Diagrama de Sequência (destacados em vermelho), uma vez que a biblioteca UnB-DALi, no momento de escrita deste trabalho, oferece suporte apenas para estes dois tipos de diagramas.

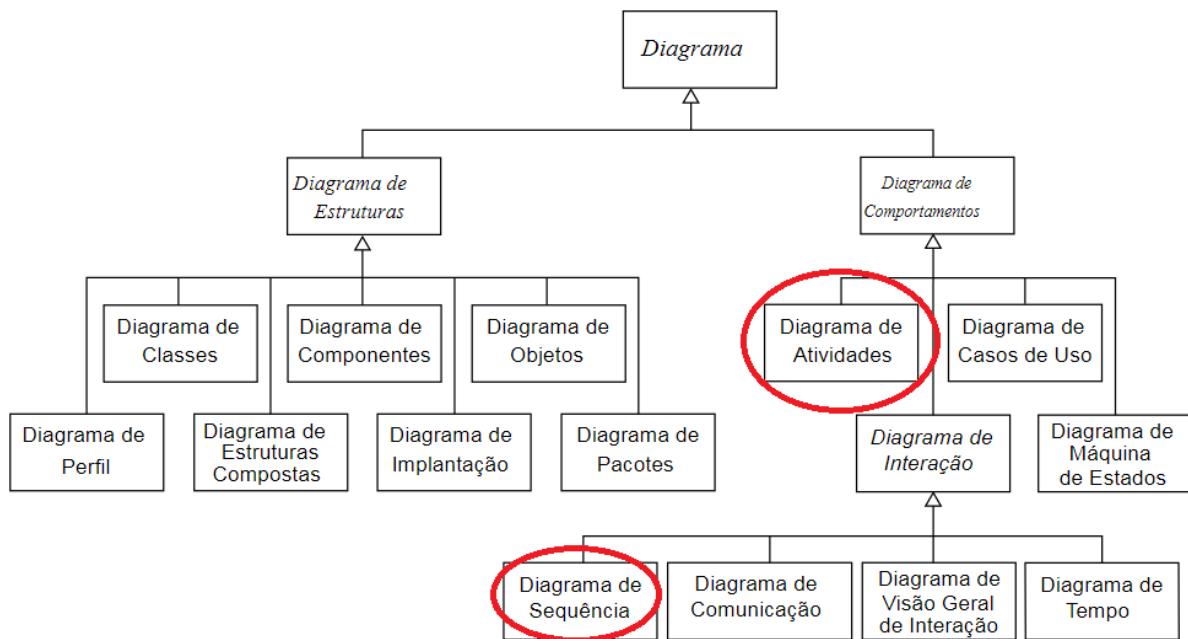


Figura 2.5: Taxonomia dos diagramas de estrutura e comportamento (com adaptações) [10].

2.5.1 Diagramas de Atividade

Os Diagramas de Atividade representam o fluxo de controle entre atividades de computação de uma certa tarefa ou um fluxo de trabalho [4]. Esta representação consiste em um grafo de Atividades, de forma similar a uma Máquina de Estados Finitos, onde

cada estado seria equivalente a uma atividade intrínseca ao modelo de computação representado. Estas Atividades podem ser sequenciais ou paralelas, e também suportam estruturas de Decisões, Iterações e Concorrência.

Alguns dos elementos que compõem um Diagrama de Atividades estão descritos a seguir.

- **Nó Executável:** Representa uma ação a ser realizada por um ou mais componentes do sistema.
- **Nó Inicial:** Representa o início da execução de um fluxo de atividades. Um Diagrama de Atividade pode conter um ou mais Nós Iniciais. É representado por um círculo preenchido com uma cor escura (preto, azul, etc).
- **Nó Final:** Representa o fim de um fluxo de atividades, indicando sua completude. Um Diagrama de Atividade pode conter um ou mais nós finais. É representado por um círculo de cor escura, envolto por um círculo de espessura menor.
- **Nó de Decisão:** É utilizado para a mudança de um caminho no fluxo de atividades, dependente do resultado da condição representada pelo Nó. Este nó aceita apenas uma transição de chegada, podendo escolher uma dentre as transições de saída, dependendo da condição do Nó. Nó de decisão são representados por um losango com várias transições de saída.
- **Nó de Merge:** É utilizado para receber várias transições de entrada e resultar em apenas uma transição de saída.
- **Transição:** Descreve um fluxo (em inglês, *control flow*) entre dois únicos elementos. Eles podem ou não ser ativados por algum evento específico. O final da atividade corrente representa uma transição para a próxima. Estes elementos podem ter algum valor associado como, por exemplo, a probabilidade do evento seguinte ocorrer (que é utilizado neste projeto).

A Figura 2.6 abaixo mostra Diagrama de Atividades que contém todos os elementos descritos acima.

2.5.2 Diagramas de Sequência

Os Diagramas de Sequência exibem um conjunto de mensagens (síncronas ou assíncronas) que são enviadas entre os componentes de um dado sistema em uma sequência temporal [4]. É um gráfico bidimensional com um eixo vertical que representa os componentes participantes da tarefa (ou objetos), e um eixo horizontal que representa o fluxo temporal.

O conjunto de elementos que compõem um Diagrama de Sequência são descritos abaixo:

- **Lifeline:** Cada objeto participante é descrito por uma barra vertical chamada de *Lifeline*. Interações entre componentes são descritas por mensagens representadas por linhas horizontais.

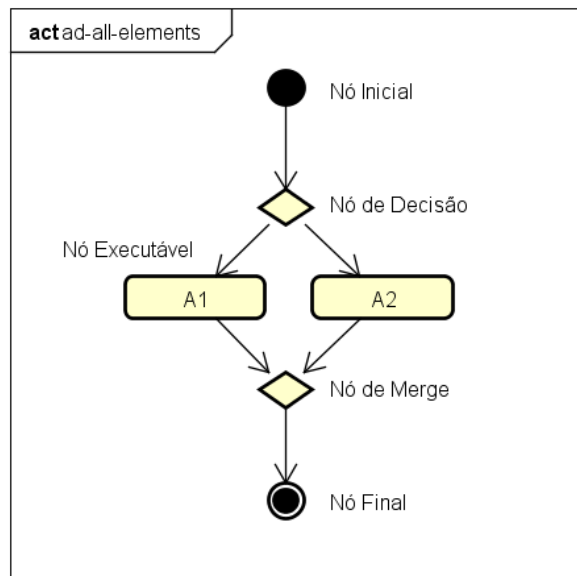


Figura 2.6: Exemplo de um Diagrama de Atividades contendo todos os elementos reconhecidos pela UnB-DALi.

- Mensagens Síncronas: São mensagens que forçam o objeto remetente a pausar e esperar pela resposta do objeto destinatário para continuar a computação. São representadas por linhas cheias com uma seta cheia no final.
- Mensagem de Resposta: É uma mensagem síncrona. É representada por linhas pontilhadas.
- Mensagem Assíncrona: É uma mensagem que não tem seu processamento pausado ao enviar mensagem ao destinatário. É representada por linhas cheias com uma seta não cheia ao final.
- Fragmento combinado: É uma estrutura utilizada para representar agrupamentos lógicos compactos. São representados por interações, chamadas de condições de guarda e um operador que especifica o tipo de lógica condicional que descreve o comportamento do fragmento. Fragmentos podem representar comportamentos que podem ocorrer ou não.

A Figura 2.7 a seguir, representa um diagrama de sequência básico, que contém os itens mais relevantes para este trabalho.

No momento da escrita desta monografia, a UnB-DALi não faz nenhuma distinção entre mensagens de resposta e mensagens assíncronas. Além disso, não há suporte a mensagens síncronas e fragmentos.

2.6 XMI

A linguagem XML, abreviação de *Extensible Markup Language*, ou “Linguagem de Marcação Extensível”, é um padrão que especifica como se devem guardar objetos de dados

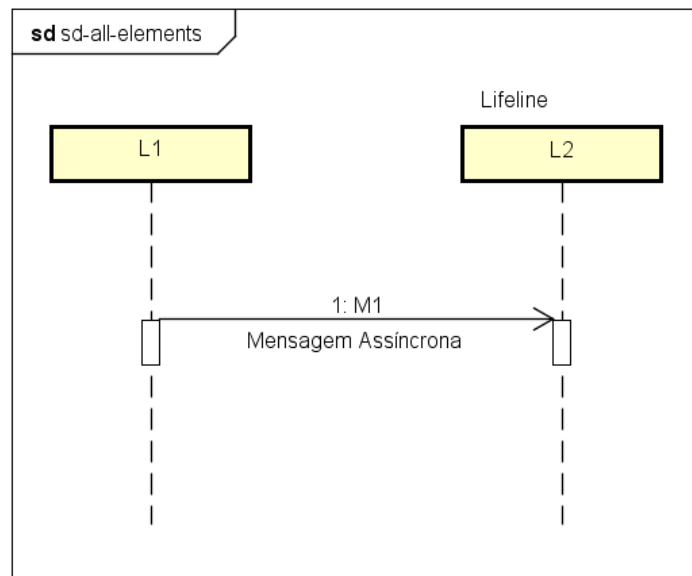


Figura 2.7: Exemplo de um Diagrama de Sequência contendo todos os elementos reconhecidos pela UnB-DALi.

chamados documentos XML. Documentos XML são compostos por unidades de armazenamento de dados. Um elemento consiste em uma *tag* de abertura, seu conteúdo, e uma *tag* de fechamento.

O XML-based Metadata Interchange (XMI) é um formato para o intercâmbio de dados definidos pelo padrão MOF (*Meta Object Facility*). A especificação oficial da UML define o meta-modelo UML como um meta-modelo MOF. Por este motivo, o padrão XMI é utilizado como modelo de intercâmbio de dados para a UML [8]. Este padrão possibilita o intercâmbio de modelos UML entre ferramentas de diferentes grupos de desenvolvimento e diferentes repositórios. Embora seu uso seja principalmente para representação de modelos UML, o formato também pode ser utilizado para serialização de objetos. Segue abaixo o conteúdo de um arquivo em formato XMI, seguindo o modelo estabelecido pelo OMG, gerada pela ferramenta Papyrus.

Código 2.1: Exemplo de arquivo em formato XMI

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <uml:Model xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI
  /20131001" xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML" xmi:id="
  _Zuy0AJyjEeaAh8z00Uz7wA" name="RootElement">
3 <packagedElement xmi:type="uml:Activity" xmi:id="_Z0FkwJyjEeaAh8z00Uz7wA
  " name="Activity1" node="_g5wm8JyjEeaAh8z00Uz7wA
  _i1mokJyjEeaAh8z00Uz7wA">
4 <edge xmi:type="uml:ControlFlow" xmi:id="_mUB3QJyjEeaAh8z00Uz7wA" target
  ="_i1mokJyjEeaAh8z00Uz7wA" source="_g5wm8JyjEeaAh8z00Uz7wA">
5 <weight xmi:type="uml:LiteralReal" xmi:id="_A9gQQJykEeaAh8z00Uz7wA" name
  ="PTS" value="0.9"/>
6 </edge>
7 <node xmi:type="uml:InitialNode" xmi:id="_g5wm8JyjEeaAh8z00Uz7wA" name="
  Inicio" outgoing="_mUB3QJyjEeaAh8z00Uz7wA"/>
  
```

```

8 <node xmi:type="uml:ActivityFinalNode" xmi:id="_i1mokJyjEeaAh8z00Uz7wA"
   name="Fim" incoming="_mUB3QJyjEeaAh8z00Uz7wA"/>
9 </packagedElement>
10 </uml:Model>

```

Pode-se observar que no modelo acima estão representados dois elementos do tipo *node*. Estes nós estão ligados por uma aresta, que é representada por um elemento *edge*, e esta possui um elemento filho, *weight*, que neste contexto corresponde a probabilidade de transição entre os estados que estão conectados por esta aresta.

2.6.1 Formas de acesso a dados do XMI

Podemos querer extrair informações acerca do conteúdo de um documento XMI. Uma ideia inicial seria varrer o documento utilizando-se de alguma forma de verificação de palavras, como as *tags* que o compõem. Para tal, criaria-se um programa chamado analisador sintático, em uma linguagem de programação qualquer. Essa abordagem é válida, mas seria muito custoso começar do zero.

Uma outra alternativa seria usar uma *Application Programming Interface*, ou API, já pronta. APIs para processamento de arquivos XML e XMI são desenvolvidas por programadores e pesquisadores que lidam constantemente com estes tipos de arquivos. No decorrer desta seção, apresentaremos algumas dessas APIs e apresentaremos o analisador sintático (*parser*) escolhido para utilização nesta monografia: o *parser* da ferramenta SDMetrics.

A *Extensible Stylesheet Language Transformations*, ou XSLT, é uma linguagem de transformação de arquivos XML para outros arquivos XML, de acordo com a transformação detalhada pelo usuário. Segundo a W3 Schools, “Com XSLT você pode adicionar ou remover elementos e atributos para ou do arquivo de saída. Você também pode reorganizar e ordenar elementos, executar tests e decidir sobre quais elementos exibir ou esconder, entre outras coisas. Uma forma comum de descrever o processo de transformação é dizer que XSLT transforma uma árvore-fonte em XML em uma árvore-resultado em XML” [18]. Caso fosse utilizada, a linguagem XSLT poderia nos ajudar limpar as informações de metadados não utilizadas dos arquivos XMI e dar como saída um arquivo mais fácil de se ler e se processar, mas como se quer um *parser* capaz de lidar com diferentes ferramentas, esta linguagem não foi escolhida para a solução.

De acordo com a especificação da W3C [19], o *Document Object Model* (DOM) é uma API para documentos HTML válidos e documentos XML bem formados. Ela define a estrutura lógica dos documentos e as maneiras pelas quais o documento é acessado e manipulado. O DOM funciona montando uma árvore de nós onde cada nó é um objeto representando um dos elementos do documento XML [5]. As vantagens do DOM é que sua árvore pode ser atravessada e modificada livremente pelo desenvolvedor. Sua desvantagem principal é que, como a árvore da estrutura do documento deve ser montada por completo e em memória, isso acarreta em um custo computacional elevado, deixando difícil a leitura de arquivos XML muito grandes.

Tendo em vista a lentidão (para arquivos grandes) e compatibilidade com diferentes arquivos XMI de diferentes ferramentas de modelagem do mercado, escolheu-se o *parser* da SDMetrics (que será explicado mais adiante). Este *parser*, utiliza a *Simple API for XML* (SAX), que, diferentemente da API DOM, lê o arquivo como uma corrente contínua

de informações, não necessitando montar a árvore de objetos antes de percorrê-la. É uma abordagem orientada a eventos, com vários gatilhos (*triggers*) preparados para analisar elementos específicos, de acordo com a vontade do desenvolvedor.

Assim, as duas APIs fazem a mesma coisa: Extrair informações de arquivos XML, e, geralmente, seus prós e contras são opostos. Por um lado, DOM fornece uma estrutura completa para o desenvolvedor atravessar e modificar à vontade mas a um custo computacional muito grande, dependendo do tamanho do arquivo. Por outro lado, SAX não fornece estrutura alguma e eventos passados que não foram disparados pelos gatilhos, não serão analisados novamente mas essa propriedade torna a análise do documento muito mais rápida, mesmo que o documento seja enorme.

2.6.2 Trabalho relacionado

Na monografia de conclusão de curso de Bernardes [11], intitulada *Transformação Automática de Modelos UML para Modelos Markovianos Parametrizados*, a autora criou uma ferramenta para automatizar o processo de transformação de modelos de *software*, construídos em uma ferramenta de modelagem UML, para modelos passíveis de *model checking* sob a ótica de Linhas de Produtos de *Software*, similarmente ao que está sendo feito neste trabalho. No entanto, existem algumas diferenças na abordagem do tratamento do arquivo XMI.

No trabalho de Bernardes, o *parsing* XMI foi focado nos arquivos exportados por uma ferramenta de modelagem específica, o *Magic Draw*, o que torna a ferramenta não-interoperável. Tal decisão foi tomada utilizando-se do argumento que existem muitos modelos diferentes de arquivos XMI e seria inviável realizar o mesmo trabalho para cada ferramenta. Para contrastar, a atual ferramenta desenvolvida nesta monografia se propõe a fornecer a capacidade de fácil adaptação a diferentes ferramentas de modelagem no mercado. Tal facilidade de adaptação se deve à utilização do *parser* da API *SDMetrics*, que será explicada a seguir.

Outra diferença foi a utilização do *Document Object Model*, ou DOM, uma API para extração de informações de arquivos em formato XML baseada em árvores, explicada anteriormente. Nesta monografia, utilizou-se a *Simple API for XML*, ou SAX, um algoritmo de *XML parsing* orientado a eventos, devidamente implementado na API *SDMetrics*.

Vale ressaltar que algumas propriedades da ferramenta de Bernardes são interessantes e podem servir como um modelo para os trabalhos futuros desta monografia, como a utilização de um *MARTE profile* para anotação de propriedades mais padronizada e também a autoria própria do *XMI parser*, o que dá mais liberdade de modificação para o autor.

2.7 SDMetrics

Segundo o próprio sítio *online* [15], a *SDMetrics* (*Software Design Metrics Tool for the UML*) é uma ferramenta para medição de qualidade de *designs* orientados a objetos para a UML. É uma ferramenta que analisa a estrutura de modelos UML e fornece informações, entre outros fins, para:

- Estabelecer *benchmarks* para identificar problemas potenciais de *design*;

- Prever qualidades relevantes do sistema, como a susceptibilidade a falhas ou manutenibilidade, para determinar onde focar os testes;
- Aumentar a qualidade sistema e a garantia de qualidade do mesmo, evitando custos desnecessários ao desenvolvimento já em fases iniciais do projeto.

A SDMetrics funciona com qualquer ferramenta de modelagem UML que forneça suporte ao formato XMI, pois é a partir destes arquivos que a ferramenta extrai informações sobre o modelo que se quer medir. A ferramenta conduz esta transformação através do seu *parser* XMI que, felizmente, está disponível para uso por terceiros, sendo de código-aberto (*open source*).

A implementação das funcionalidades principais da SDMetrics, além de ser código-aberto, está em Java, o que é oportuno já que a biblioteca UnB-DALi também está codificada nesta linguagem. Sendo assim, focaremos agora no funcionamento deste *parser* XMI da SDMetrics.

No manual da SDMetrics, Wüst [7], assim como Oliveira [2], ressaltou que há uma grande variedade de tipos de XMI diferentes. Esta variedade deve-se ao fato dos vendedores de ferramentas que fornecem suporte ao XMI nem sempre aderirem ao padrão especificado pelo OMG, o que torna o objetivo inicial de interoperabilidade entre as ferramentas de ser alcançado.

Wüst criou o *parser* XMI da SDMetrics com isso em mente. Para conseguir a flexibilidade de atender aos diferentes tipos de XMI existentes, foram pensados dois arquivos diferentes (também em formato XMI).

O primeiro arquivo, chamado de *metamodelo*, guarda informações referentes aos elementos dos modelos. Ele não precisa necessariamente de descrições de todos os elementos representados em UML, podendo ser resumido apenas aos elementos de enfoque do usuário. Isso é uma característica interessante para o presente trabalho, já que, na data da escrita, a UnB-DALi não oferece suporte a todos os tipos de diagrama UML e nem a todos os elementos.

O segundo arquivo, chamado de *XMI Transformation File*, estabelece o mapeamento entre os elementos presentes no metamodelo e as *tags* do arquivo XMI exportado por uma ferramenta de modelagem. Dessa forma, fica viável fazer a conexão do elemento com a *tag* específica no arquivo e, partir daí, extrair as informações que se deseja.

Os dois arquivos, tanto o metamodelo quando o *XMI Transformation*, devem ser elaborados pelo interessado em acessar os dados do XMI. No caso desta monografia, o metamodelo foi escrito pelos autores somente com base nos elementos reconhecidos pela UnB-DALi citados anteriormente, ignorando outros elementos de diagramas de atividades e diagramas de sequência da UML 2.5. Já o *XMI Transformation* precisou ser escrito duas vezes, uma versão para cada ferramenta UML, no caso Astah e Papyrus. Caso o usuário queira utilizar o programa aqui desenvolvido com outra ferramenta UML, o mesmo deve construir o *XMI Transformation* específico para aquela ferramenta. Exemplos dos XMI Transformation escritos e um *template* que facilita a escrita de um *XMI Transformation* estão ao final desta monografia, na seção de Anexos.

A partir da lógica de utilizar dois arquivos para extração de informações do XMI, podemos ver que facilmente pode-se fornecer suporte às diferentes ferramentas de modelagem do mercado. Com um arquivo, o *metamodelo*, especifica-se os elementos UML que se tem interesse. Com o outro arquivo, o de mapeamento (*XMI Transformation*)

faz-se a ligação dos elementos com as *tags* específicas de um tipo específico de arquivo gerado por uma ferramenta específica. Desta forma, precisa-se apenas criar diferentes *XMI Transformation Files* para cada ferramenta específica.

Abaixo estão explicações mais detalhadas sobre a estrutura destes dois arquivos. Para uma versão completa dos arquivos utilizados pela ferramenta aqui criada, veja o Anexo II, o qual contém o metamodelo e os arquivos “*XMI Transformation*” das duas ferramentas, Astah e Papyrus.

2.7.1 Metamodelo SDMetrics

O arquivo de metamodelo da SDMetrics define quais tipos de elementos UML serão reconhecidos pelo *parser*. Abaixo está um exemplo de metamodelo retirado do manual da SDMetrics [7]:

Código 2.2: Exemplo de estrutura do Metamodelo SDMetrics

```
1 <sdmetricsmetamodel version="2.0" >
2   <modelelement name="element1">
3     <attribute name="attr1" type="data" multiplicity="one" />
4     <attribute name="attr2" type="ref" multiplicity="many" />
5     ..
6   </modelelement>
7   <modelelement name="element2" parent="element1">
8     <attribute name="attr3" type="extref" multiplicity="one" />
9     ...
10  </modelelement>
11  ...
12 </sdmetricsmetamodel>
```

A sua estrutura é composta por uma lista de definições de elementos entre as *tags* `<sdmetricsmetamodel>`.

A definição de um elemento fica entre as *tags* `<modelelement>`, sendo o atributo *name* obrigatório e *parent* opcional (a herança entre *model elements* será explicada mais adiante).

Em cada definição de um elemento, pode haver um conjunto de atributos, que representam dados ou referências cruzadas a outros elementos no modelo. Cada atributo é definido dentro de uma *tag* `<attribute>`. Esta *tag* possui três atributos:

- *name* (Obrigatório)
- *type* (Opcional. O valor padrão é “data”)
- *multiplicity* (Opcional. O valor padrão é “one”)

Abaixo a definição de um elemento como exemplo chamado *operation*:

Código 2.3: Exemplo da definição de um elemento no metamodelo

```
1 <modelelement name="operation">
2   <attribute name="id" />
3   <attribute name="name" />
4   <attribute name="context" type="ref" />
5   <attribute name="visibility" />
6 </modelelement>
```

Os atributos especificados não possuem significado por si sós. Ainda é necessário de um arquivo para realizar o mapeamento das informações, que será a tarefa do *XMI Transformation*, explicado mais adiante.

Para facilitar a especificação de metamodelos, há também a característica de herança. Um elemento pode herdar atributos de um elemento definido anteriormente. Por padrão, todos os elementos do metamodelo herdam atributos de um elemento especial chamado *sdmetricsbase*. Por ter que ter sido definido anteriormente, este elemento é o primeiro a ser definidos em qualquer metamodelo SDMetrics.

A definição formal do arquivo de metamodelo e seus elementos internos (*sdmetricsmetamodel*, *modelelement* e *attribute*) está mostrada a seguir em formato DTD (*Document Type Definition*):

Código 2.4: Definição formal do arquivo de metamodelo

```

1 <!ELEMENT sdmetricsmetamodel ( modelelement+ ) >
2 <!--ATTLIST sdmetricsmetamodel
3     version CDATA #REQUIRED-->
4
5 <!--ELEMENT modelelement (#PCDATA|attribute)*-->
6 <!--ATTLIST modelelement
7     name CDATA #REQUIRED
8     parent CDATA "sdmetricsbase"-->
9
10 <!--ELEMENT attribute (#PCDATA)-->
11 <!--ATTLIST attribute
12     name CDATA #REQUIRED
13     type (ref|data) "data"
14     multiplicity (one|many) "one"-->

```

2.7.2 Arquivo de transformação SDMetrics

O arquivo de transformação, chamado de *XMI Transformation*, realiza o mapeamento das informações no arquivo XMI com os elementos do metamodelo definidos no arquivo explicado anteriormente. Uma representação do formato de um *XMI Transformation File* é:

Código 2.5: Exemplo de estrutura do *XMI Transformation File*

```

1 <xmitransformations version="2.0" >
2     <xmitransformation ...xmitransformation attributes... />
3     <trigger ...trigger attributes... />
4     <trigger ...trigger attributes... />
5     ...
6 </xmitransformation>
7 <xmitransformation ...xmitransformation attributes... />
8     <trigger ...trigger attributes... />
9     ...
10 </xmitransformation>
11 ...
12 </xmitransformations>

```

Esta estrutura é similar à vista anteriormente no metamodelo, mas os nomes das *tags* e suas funções são diferentes. Uma lista de transformações, caracterizadas pelas *tag*

<xmitransformation> estão contidas na *tag* mais externa <xmitransformations>. Cada uma das transformações possui gatilhos (*triggers*) dentro de seu corpo, e cada gatilho é expressado pela *tag* <trigger>.

Para mostrar como uma *xmitransformation* funciona, exibimos um exemplo de arquivo XMI hipotético gerado por alguma ferramenta de modelagem:

```

1 <umlDiagram xmi:type="uml:Activity" xmi:id="0001" name="AD-
  todos elementos" node="0002 0003 0004">
2   <node xmi:type="uml:InitialNode" xmi:id="0002" name="InitialNode1"
    outgoing="0003"/>
3   <node xmi:type="uml:OpaqueAction" xmi:id="0003" name="A1" incoming="
    0002" outgoing="0004"/>
4   <node xmi:type="uml:ActivityFinalNode" xmi:id="0004" name="FinalNode
    " incoming="0004"/>
5   <edge xmi:type="uml:ControlFlow" xmi:id="_9001" target="0003" source
    ="0002">
6     <weight xmi:type="uml:LiteralReal" xmi:id="_y5KR" value="1.0"/>
7   </edge>
8   <edge xmi:type="uml:ControlFlow" xmi:id="_9002" target="0004" source
    ="0003">
9     <weight xmi:type="uml:LiteralReal" xmi:id="_9JTA" value="1.0"/>
10   </edge>
11 </umlDiagram>

```

Neste simples Diagrama de Atividades, podemos perceber três nós e duas arestas (*control flows*). Do diagrama, que se encontra na *tag* “<umlDiagram>”, temos como atributos de interesse o tipo, pelo atributo “xmi:type”, o número de identificação do elemento, ou ID, em “xmi:id” e o nome do diagrama no atributo “name”. Dos nós, localizados nas *tags* “<node>”, queremos somente o ID (para referência nas arestas) e o nome. Das arestas, encontradas nas *tags* “<edge>”, queremos as informações de fonte (*source*, destino (*target*) e probabilidade da transição (*probability*). Tais informações se encontram nos atributos “source”, “target” e “value” (da *tag* mais interna “<weight>”), além do identificador. Note que o identificador é um atributo essencial de um elemento, pois ele o identifica unicamente no modelo e evita que haja repetições indevidas no momento da criação do diagrama em memória.

Suponhamos, para fins de um exemplo mais simples, que queremos apenas extrair informações associadas às arestas (*control flows*) do diagrama. Assume-se que o elemento no metamodelo que representa as arestas se chama “controlflow”. As informações são extraídas de acordo com a *XMI Transformation* abaixo:

Código 2.6: XMI Transformation que captura as informações das arestas em um Diagrama de Atividades

```

1 <xmitransformation modelelement="controlflow" xmi:pattern="edge" recurse=
  "true">
2   <trigger name="id" type="attrval" attr="xmi:id" />
3   <trigger name="source" type="attrval" attr="source"/>
4   <trigger name="target" type="attrval" attr="target"/>
5   <trigger name="probability" type="cattrval" src="weight" attr="value
    "/>
6 </xmitransformation>

```


Primeiramente, na tag `<xmitransformation>` temos o atributo “modelelement”, que se refere ao nome de um elemento no metamodelo. O atributo “xmipattern” refere-se ao nome da tag *tag* de referência no arquivo XMI por onde a busca pelas informações se iniciará, similarmente a um nó raiz em uma estrutura computacional do tipo árvore. O atributo “recurse” está com o valor *true* para indicar que a *tag* atual pode ser usada como contexto para buscas mais internas.

Cada um dos *triggers* tem, no mínimo, dois atributos:

- *name*: O nome do atributo no Metamodelo SDMetrics anteriormente definido ao qual este *trigger* pertence.
- *type*: Indica como o *trigger* recupera a informação do arquivo XMI.

No primeiro *trigger*, por exemplo, o atributo “name” está com o valor “id”, indicando que o atributo “id” do elemento “controlflow” terá o seu conteúdo preenchido por este *trigger*. O atributo “type” com o valor “attrval” indica que este dado será recuperado de um atributo da *tag* raiz, no caso “edge”. O atributo “attr” mostra para o *trigger* o nome do atributo no arquivo XMI onde ele deve procurar a informação, no caso o atributo “xmi:id”. O *triggers* para dos atributos “source” e “target” funcionam exatamente como o primeiro, somente mudando, é claro, o nome do atributo de interesse na *tag* raiz (que, coincidentemente, também se chamam “source” e “target”, mas nem sempre o nome do atributo no XMI corresponde ao nome do atributo no metamodelo). Por último, o *trigger* que recupera a informação de probabilidade da transição funciona de uma maneira um pouco diferente, pois esta informação não se encontra na *tag* raiz “edge” e sim na *tag* mais interna (filha) “weight”. Para acessar o valor da probabilidade, o tipo da busca por atributo será “cattrval”, significando *child attribute value*, ou valor do atributo filho. Este tipo de busca requer que a *tag* filha seja explicitada, pois há a possibilidade de haver diferentes *tags* filhas com nomes distintos. O nome da *tag* filha é colocado no atributo “src”, que aqui é “weight”. O último atributo, “attr”, é similar aos dos *triggers* anteriores, mas este atributo, que aqui tem o valor “value”, não pertence à *tag* raiz “edge” e sim à *tag* filha “weight”.

Há várias outras formas de busca de informações que podem ser especificadas no atributo “type”, mas por ora, focaremos apenas nos valores “attrval” e “cattrval”. Quando valores diferentes surgirem no capítulo posterior, onde as *XMI Transformations* das ferramentas Astah e Papyrus serão mais detalhadas, eles serão explicados corretamente.

2.8 Considerações finais

Neste capítulo explicamos o termo Dependabilidade, propriedade de interesse de sistemas computacionais. Vimos também como funciona a verificação de modelos computacionais, bem como as ferramentas utilizadas para dar auxílio a estas verificações. Apresentamos também o método para análise de dependabilidade proposto por Rodrigues et al., seus pontos que necessitam refinamento e o trabalho inicial de Oliveira para corroborar para o fortalecimento de um passo importante da metodologia. Vimos os Diagramas UML de interesse nesta monografia, que são os de Atividade e os de Sequência. Foi apresentado brevemente o formato XML e sua variação para intercâmbio de dados, o XMI. Foram apresentadas também formas diferentes de buscar informações a partir de

arquivos em formato XMI, e mostrada a forma escolhida, bem como as razões para sua escolha. Na última seção, a ferramenta SDMetrics foi focada e o funcionamento de seu *parser* XMI foi explicado com o auxílio de exemplos.

No capítulo seguinte, explicamos o funcionamento do *XMI-PRISM-Converter*, programa Java criado para habilitar a interoperabilidade entre a API SDMetrics e a biblioteca UnB-DALi, fornecendo um transformador de modelos automático a partir de arquivos XMI.

Capítulo 3

XMI-PRISM-Converter

O objetivo da ferramenta criada nesta monografia é converter automaticamente um modelo UML a partir de seu arquivo de persistência em formato XMI para um modelo DTMC na linguagem da ferramenta de verificação de modelos PRISM, ferramenta esta que verifica automaticamente os estados do modelo, sendo uma das metodologias de análise de dependabilidade.

Como visto no capítulo 2, a biblioteca UnB-DALi realiza a conversão do modelo a partir de um diagrama instanciado utilizando código Java para adicionar cada nó e transição a um diagrama (AD ou SD). Para automatizar o processo de instanciação dos diagramas, vimos no capítulo 2 que há formas de extrair informações de diagramas UML quando estes estão no formato XMI. Vimos que a ferramenta capaz de realizar esta tarefa é um *parser* XMI, e escolheu-se usar o *parser* XMI fornecido na API da ferramenta SDMetrics. Juntando-se essas duas partes (instanciação do modelo e conversão) temos um processo completo e automatizado que agilizará a macro-etapa de Modelagem UML no processo de análise de dependabilidade em foco nesta monografia.

A Figura 3.1 abaixo mostra um resumo da macro-etapa automatizada. Esta macro-etapa consiste na primeira parte do processo de Análise de Dependabilidade proposto por Rodrigues et al. [14] e é composta pelos passos de especificação do modelo em UML(1), anotação dos modelos(2) e conversão dos modelos UML para PRISM(3), como visto na figura 2.2.



Figura 3.1: Resumo da macro-etapa automatizada de conversão de modelos UML para modelos PRISM

3.1 Requisitos

Abaixo seguem alguns requisitos que nos deram uma direção para as decisões do projeto. Por exemplo, ao utilizar diferentes ferramentas de modelagem, vimos que o XMI gerado pode ter uma extensão de arquivo diferente de uma extensão de “.xmi”. Portanto, a ferramenta deve fornecer suporte aos diferentes tipos de extensões. Já em termos de usabilidade, vimos que seria interessante o usuário conseguir converter vários arquivos XMI em um mesmo diretório com apenas uma execução. Em termos de usabilidade, foi desejável que o programa emitisse mensagens relevantes ao término da conversão de cada arquivo, exibindo o sucesso ou fracasso, bem como as causas da falha na conversão. Dividimos os requisitos em funcionais e não-funcionais, a seguir:

Requisitos Funcionais:

1. Utilizar a linguagem de programação Java, já que este projeto é uma junção de outros dois projetos que utilizam essa linguagem (UnB-DALi e SDMetrics).
2. Receber como entrada arquivos em padrão XMI, quaisquer sejam suas extensões (.xml, .xmi, .uml, etc);
3. Criar um arquivo de saída no formato .pm (PRISM) no mesmo diretório do arquivo com o modelo após o término de transformações bem-sucedidas, para que o usuário não necessite copiar o texto gerado no *output*;
4. Fornecer uma interface amigável que exiba mensagens adequadas após a conclusão da conversão para cada arquivo, exibindo sucesso ou falha na conversão. No caso de falha, exibir mensagens de erro para que o usuário possa corrigir seus diagramas.

Requisitos Não-funcionais:

1. Facilidade ao se expandir o suporte para ferramentas de modelagem UML que ainda não tem um arquivo de transformações (*XMI Transformation*) específico para elas;
2. Facilidade de integração com as diversas ferramentas de modelagem UML existentes.

3.2 Arquitetura e Implementação

Para atender os requisitos citados anteriormente, adotamos algumas práticas na construção do programa:

1. A biblioteca foi feita em Java, tendo em vista a utilização dos outros dois projetos também codificados em Java que a suportam, atendendo o requisito funcional 1.
2. O projeto possui um conjunto de extensões de arquivos aceitas que pode ser facilmente alterado caso existam outros nomes de extensões de arquivos .XMI, atendendo ao requisito funcional 2.
3. O programa fornece um gerador de arquivo de *output* que já coloca o resultado em um arquivo com extensão .pm (PRISM *File*), pronto para ser aberto diretamente na ferramenta PRISM, atendendo ao requisito funcional 3.

4. O programa possui métodos específicos para exibir mensagens de erros em transformações mal-sucedidas e indica onde está o erro na construção do diagrama fornecido. Em caso de sucesso, uma mensagem adequada é exibida para o usuário, com o objetivo de mantê-lo explicitamente informado do resultado de suas transformações, atendendo ao requisito funcional número 4.
5. O programa utilizou simples métodos de entrada de argumentos e todos os métodos estão documentados em Javadoc e escritos seguindo boas práticas de programação. Cada classe tem sua função específica e uma alteração ou adição em alguma não influencia diretamente o funcionamento da outra. Além disso, o projeto do programa está disponível *online* no sítio *GitHub*, sendo de fácil acesso e facilitando uma possível manutenção no futuro. Desta forma, atendemos os requisitos não-funcionais 1 e 3.
6. Ao utilizarmos a biblioteca SDMetrics como auxílio para o *parsing* dos arquivos XMI, tornamos o projeto mais flexível, tendo em vista que este *parser* utiliza um arquivo facilmente modificável para se adaptar a diferentes padrões de XMI gerados por ferramentas de modelagem do mercado. Para auxiliar ainda mais, fornecemos um arquivo comentado que contém uma estrutura de uma *XMI Transformation*, o arquivo XMI responsável por mapear os elementos do modelo com as *tags* do arquivo XMI que contém o modelo. Desta forma, atendemos ao requisito não-funcional 2.

3.2.1 Arquitetura

O programa realiza uma ou mais conversões de modelo a partir dos modelos em arquivo XMI. Para que as conversões ocorram corretamente, deve-se observar os seguintes fatores

1. O modelo que se quer converter de UML-XMI para linguagem PRISM deve ter sido feito em uma ferramenta de modelagem atualmente suportada pelo programa. No momento da escrita da monografia, apenas as ferramentas *Astah Professional* e *Papyrus* têm suporte.
2. Os diagramas no modelo só podem conter elementos suportados pela biblioteca UnB-DALi. Atualmente, os elementos de um Diagrama de Atividade são os Nós inicial, final, executável, de decisão e de merge e as Arestas (transições). Os elementos de um Diagrama de Sequência são as Linhas de vida (*Lifelines*) e as mensagens assíncronas.
3. A anotação das probabilidades nos diagramas são feitas nas arestas, no caso de ADs e em componentes (*Lifelines*) no caso de SDs. Caso haja algum elemento destes sem probabilidade, haverá uma mensagem de erro acusando a ausência de alguma probabilidade. Atualmente, as probabilidades são anotadas em atributos, como o *tagged value*, que comportam números reais.
4. Atualmente, assume-se que o usuário utiliza o programa como um JAR executável em terminal e que o programa esteja em uma pasta com um ou mais arquivos XMI, cada arquivo contendo apenas um diagrama UML e todos os arquivos no mesmo diretório tendo sido exportados por uma mesma ferramenta. Dessa forma, a execução do arquivo será acompanhada pelo nome da ferramenta de modelagem

que gerou os arquivos. Mais informações dos comandos de execução, se encontram no arquivo README do projeto.

A Figura 3.2 abaixo mostra uma visão geral funcionamento do conversor no formato de um diagrama de componentes.

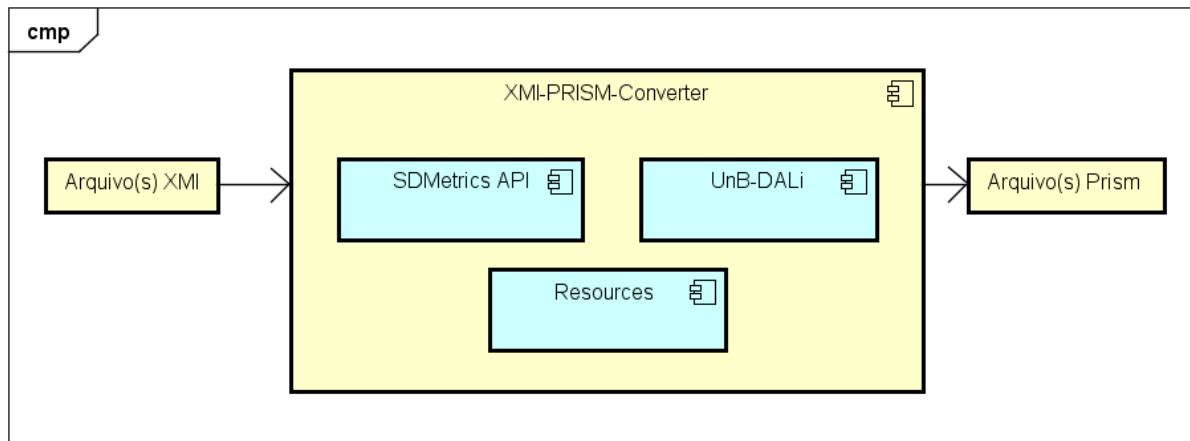


Figura 3.2: Visão geral da arquitetura do projeto.

Note que a aplicação tem um *input* e um *output* bem definidos. O número de arquivos PRISM gerados no *output* deve corresponder ao número de arquivos XMI que foram bem-sucedidos na conversão. Em outras palavras, a menos que ocorram falhas no processo de conversão, a quantidade de arquivos de entrada deve ser igual a de saída. O XMI-PRISM-Converter se utiliza de duas APIs externas, SDMetrics e UnB-DALi, e também um componente genérico chamado de *resources* (recursos), que guarda os arquivos necessários para que a criação dos modelos ocorra. Entre estes arquivos estão o metamodelo e o *XMI Transformation File* da SDMetrics.

3.2.2 Implementação

Como visto no diagrama de componentes anterior, o XMI-PRISM-Converter é um agregador de bibliotecas com uma lógica de programação envolvida para a avaliação do tipo de elemento e sua instanciação. Ele é composto por sete classes Java (fora as classes pertencentes às outras bibliotecas), que se relacionam entre si para realizar a conversão, cada classe com uma responsabilidade definida. A seguir, uma breve descrição de cada uma delas:

- *MainClass.java*: Classe de ponto de entrada do programa. Esta classe apenas faz as chamadas para a conversão de cada arquivo que o usuário optou por converter.
- *Converter.java*: Classe em alto nível que faz as chamadas do processo passo-a-passo de conversão. Os passos englobam construção do modelo SDMetrics, avaliação dos elementos, construção do modelo AGG e geração do modelo PRISM em um arquivo de saída.

- *ModelBuilder.java*: Classe que constrói o modelo SDMetrics para posterior consulta aos seus elementos. O modelo é construído com o arquivo XMI e os dois arquivos de configuração da SDMetrics, o *SDMetrics Metamodel* e o *XMI Transformation file*, previamente configurados.
- *DiagramBuilder.java*: Classe que constrói o diagrama UML a partir do modelo SDMetrics pronto. O diagrama é um modelo no formato AGG e é construído com uma lógica de programação simples que verifica o tipo do elemento, o instancia e adiciona-o ao diagrama utilizando o método adequado da biblioteca UnB-DALi.
- *FileUtil.java*: Classe de apoio que lida com operações referentes a endereços de arquivos no sistema operacional, endereço de pastas, lista de arquivos XMI e criação do arquivo de saída.
- *MessageUtil.java*: Classe de apoio que gera mensagens acerca do processamento da conversão, como indicação de sucesso ou falha.
- *TimeUtil.java*: Classe de apoio que mede o tempo da conversão do modelo AGG para o modelo PRISM, para fins de análise.

A Figura 3.3 abaixo mostra o Diagrama de Classes relacionando as classes do projeto:

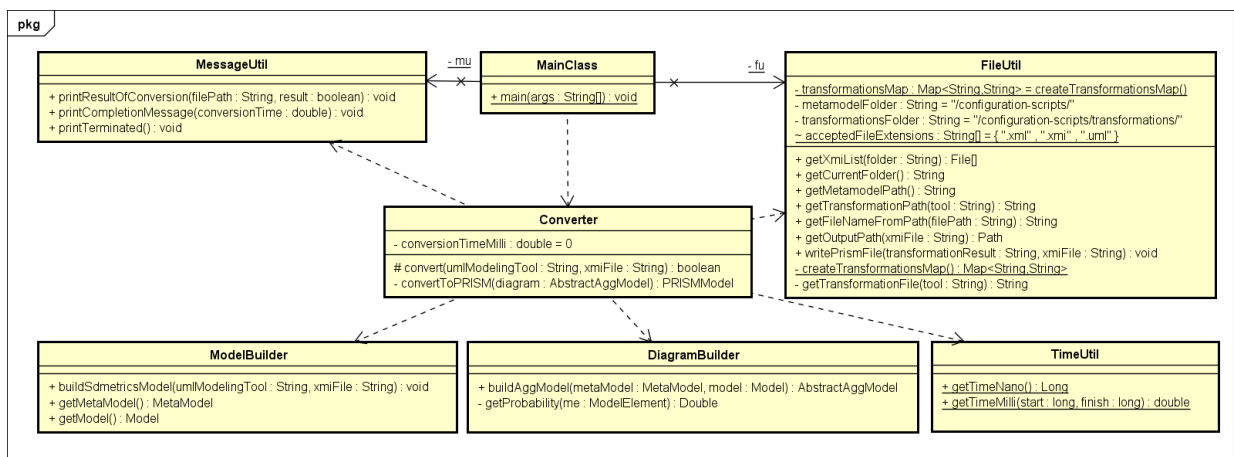


Figura 3.3: Diagrama de Classes do XMI-PRISM-Converter.

O programa começa a execução recebendo um ou mais arquivos XMI e em seguida constrói um modelo em memória, para cada arquivo, utilizando os arquivos de configuração da biblioteca SDMetrics. Após essa construção, o modelo em memória é consultado, elemento a elemento, em uma estrutura *switch*, onde cada tipo de elemento vai sendo instanciado em um modelo abstrato do AGG (tipo utilizado pela biblioteca UnB-DALi). Com o modelo abstrato do AGG construído, utiliza-se o método de conversão fornecido pela UnB-DALi e a conversão ocorre, retornando um modelo PRISM. A partir do modelo PRISM, é feito o arquivo de saída e o programa encerra a execução.

A Figura 3.4 abaixo mostra o Diagrama de Sequência das trocas de mensagens envolvidas na conversão:

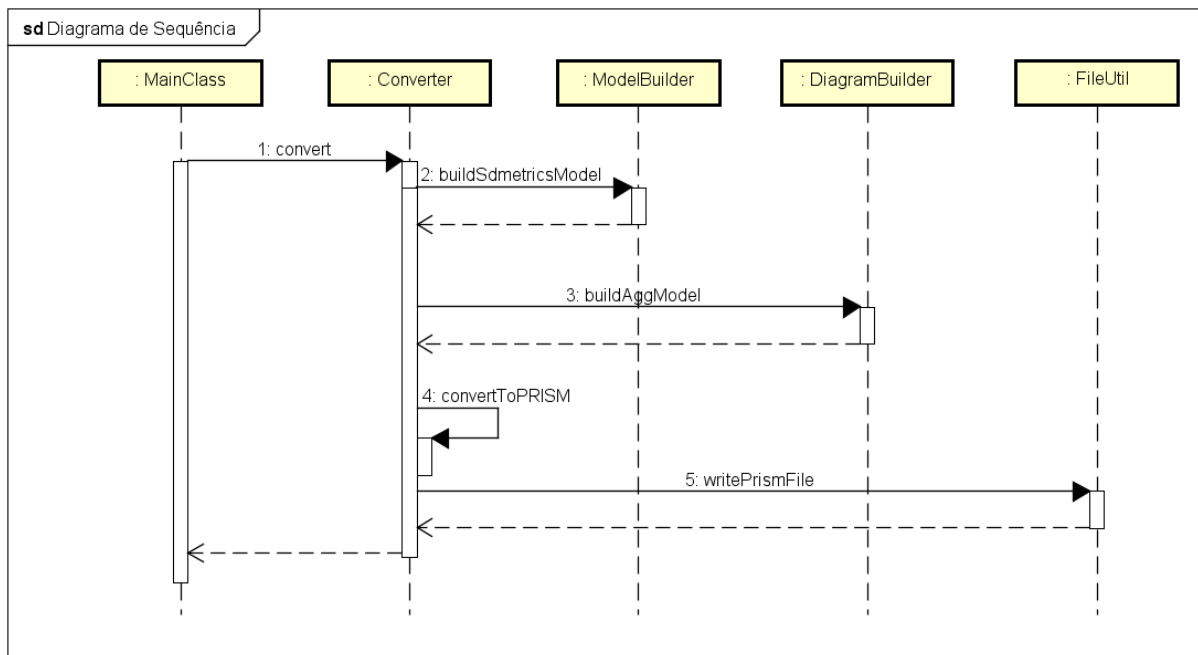


Figura 3.4: Diagrama de Sequência com as principais chamadas de métodos do programa.

Fluxo de execução do programa

Como mencionado anteriormente, o programa interpreta arquivos XMI com apenas um diagrama por arquivo, mas ele pode processar quantos arquivos o usuário quiser. Dependendo do comando de execução, o usuário pode escolher transformar alguns arquivos ou todos os arquivos no formato XMI presentes na pasta onde o programa está sendo executado. O código abaixo mostra o método *main*, que lida com o modo de execução escolhido pelo usuário:

Código 3.1: Trecho da classe MainClass

```

1 public static void main(String[] args) {
2     String modelingTool = args[0];
3     Converter converter = new Converter();
4
5     if (args.length == 1) {
6         System.out.println();
7         File[] listXMIFiles = fu.getXmiList(fu.getCurrentFolder());
8         for (File file : listXMIFiles) {
9             converter.convert(modelingTool, file.getName());
10        }
11    } else {
12        for (int i = 1; i < args.length; i++) {
13            String filename = args[i];
14            converter.convert(modelingTool, filename);
15        }
16    }
17    mu.printTerminated();
18 }

```

O método *convert*, da classe *Converter*, é o responsável por chamar os métodos de construção do modelo SDMetrics, de construção do modelo AGG e do método interno “convertToPRISM”, que invoca os métodos da UnB-DALi de transformação para o modelo PRISM.

Após ter o metamodelo e o modelo preparado (pela classe *ModelBuilder*), a classe *DiagramBuilder* utiliza estes objetos para obter as informações dos elementos do diagrama. Em uma estrutura *switch*, o método *buildAggModel()* vai analisando todos os tipos de elementos conhecidos do metamodelo e verificando se existem elementos no modelo que correspondem àquele tipo. Este é o método mais importante da aplicação pois ele é o responsável por realizar a ponte de ligação entre a automatização fornecida pelo *parser* SDMetrics e a instanciação de objetos adequados da UnB-DALi. Abaixo encontra-se um trecho de código deste método:

Código 3.2: Exemplo de um caso do tipo “node” na estrutura switch da classe Diagram-Builder

```
1 // *** ACTIVITY DIAGRAM NODE ***
2 case "node":
3     elements = model.getAcceptedElements(type);
4     for (ModelElement me : elements) {
5         String nodeId = me.getXMIID();
6
7         String nodeType = me.getPlainAttribute("type");
8         switch (nodeType) {
9             case "uml:OpaqueAction":
10                 ExecutableNode en = new ExecutableNode(nodeId,
11                     (ActivityDiagram) diagram);
12                 ((ActivityDiagram) diagram).addExecutableNode(en);
13                 break;
14
15             case "junction":
16             case "uml:DecisionNode":
17             case "uml:MergeNode":
18                 if (me.getSetAttribute("incomingEdges").size() == 1) {
19                     DecisionNode dn = new DecisionNode(nodeId,
20                         (ActivityDiagram) diagram);
21                     ((ActivityDiagram) diagram).addDecisionNode(dn);
22                 } else {
23                     MergeNode mn = new MergeNode(nodeId,
24                         (ActivityDiagram) diagram);
25                     ((ActivityDiagram) diagram).addMergeNode(mn);
26                 }
27                 break;
28
29             case "initial":
30             case "uml:InitialNode":
31                 InitialNode in = new InitialNode(nodeId,
32                     (ActivityDiagram) diagram);
33                 ((ActivityDiagram) diagram).addInitialNode(in);
34                 break;
35
36             case "uml:ActivityFinalNode":
37                 FinalNode fn = new FinalNode(nodeId,
38                     (ActivityDiagram) diagram);
39                 ((ActivityDiagram) diagram).addFinalNode(fn);
40                 break;
41
42             default:
43                 System.out.println("Node type not found: "
44                     + me.getPlainAttribute("type"));
45                 break;
46         }
47     }
48     break;
```

Neste exemplo, se houver algum elemento do tipo “node” (de um Diagrama de Atividades), é verificado qual o tipo do nó e então este nó é instanciado e adicionado ao Diagrama de Atividade em construção. Os nomes dos tipos de nós, por exemplo “uml:ActivityFinalNode”, são definidos previamente no arquivo *XMI Transformations*. O

método retorna a chamada com um diagrama do tipo *AbstractAggModel*, caso não tenha ocorrido falhas em sua construção. Caso contrário, é retornado o valor *null*.

A classe *Converter* continua a execução do programa a partir de uma verificação no diagrama retornado. Caso ele esteja correto, ele será convertido para PRISM com o método adequado e seu resultado será escrito em um arquivo de saída. Caso contrário, ele retornará um booleano, que é verificado em testes específicos na ferramenta (caso tenha dado algum erro durante a construção do diagrama, o usuário já teria sido avisado com mensagens pertinentes). O código abaixo mostra essa verificação:

Código 3.3: Verificação do diagrama construído e chamada do método conversor para o modelo PRISM

```
1 ...
2 boolean conversionResult = false;
3 if (aggModel != null) {
4     PRISMModel prismModel = convertToPRISM(aggModel);
5     if (prismModel != null) {
6         conversionResult = true;
7         FileUtil fu = new FileUtil();
8         fu.writePrismFile(prismModel.toString(), xmiFile);
9     }
10 }
11 MessageUtil mu = new MessageUtil();
12 mu.printResultOfConversion(xmiFile, conversionResult);
13 return conversionResult;
14 }
```

O programa é executado como um JAR executável, rodando em console. O comando para rodá-lo em uma pasta que contém arquivos XMI é “java -jar xpconverter.jar <ferramenta>”, onde <ferramenta> é a ferramenta que gerou os arquivos XMI ¹. Certifique-se de que os arquivos XMI presentes na pasta foram gerados com a ferramenta adequada pois o programa ainda não possui meios de verificar a ferramenta de modelagem que gerou o XMI.

Para cada arquivo executado é exibida uma mensagem com o resultado da conversão ao lado do nome do arquivo. O resultado de uma conversão pode ser “[FAIL]” (falha) ou “[SUCCESS]” (sucesso). A Figura 3.4 abaixo mostra o resultado de uma execução do programa em uma pasta com diversos arquivos XMI gerados pela ferramenta Astah Professional, todos convertidos com sucesso.

3.3 Considerações finais

Neste capítulo foi apresentado o programa Java XMI-PRISM-Converter, que visa automatizar ainda mais o processo de transformação de modelos UML para modelos PRISM. Falou-se de seus requisitos (funcionais e não-funcionais), sua arquitetura e sua implementação. A implementação focou-se no fluxo de controle do programa, já que sua função é ordenar chamadas a outros métodos de bibliotecas exteriores e aplicar um pouco de lógica de programação na construção do modelo a ser convertido.

¹No momento da escrita desta monografia, somente são válidos os valores “astah” e “papyrus” mas no futuro poderão ser acrescentados outros valores correspondentes a novas ferramentas de modelagem.

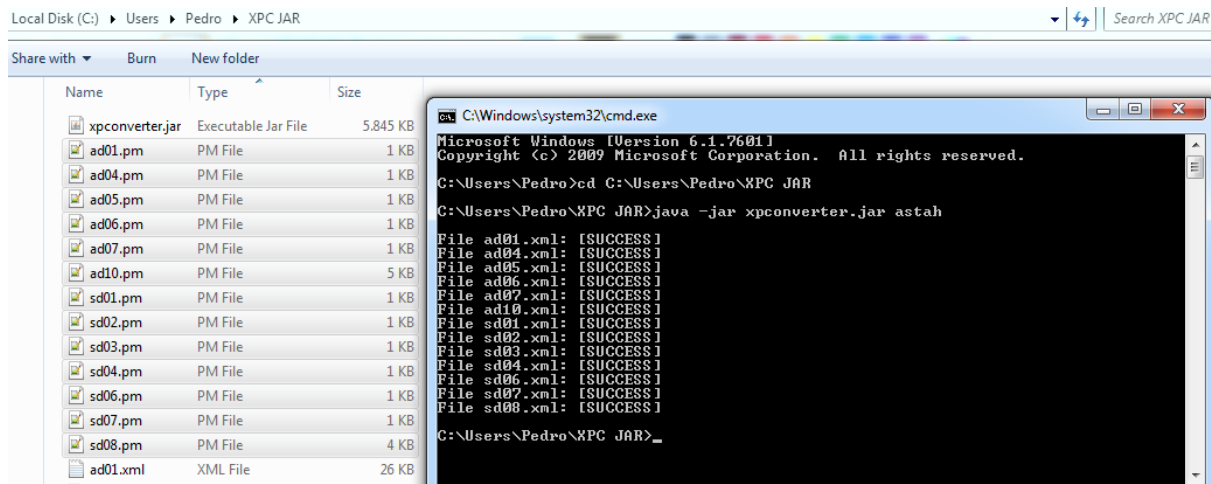


Figura 3.5: Execução do XMI-PRISM-Converter em um terminal do Windows.

No próximo capítulo, mostraremos os resultados de testes realizados com o programa e atestaremos a validade das transformações de diagramas UML gerados com as ferramentas de modelagem Astah Professional e Papyrus.

Capítulo 4

Validação

Este capítulo tem como objetivo descrever a abordagem utilizada para o processo de testes e validar parcialmente a solução proposta.

4.1 Estratégia de Teste

Adotou-se uma estratégia de testes de caixa-preta, que consiste na análise dinâmica da solução a partir do resultado obtido conforme entradas de diferentes tipos são utilizadas, comparando-se o resultado obtido com o resultado esperado após a execução.

Foram definidas diferentes classes de equivalência de entradas, baseadas em restrições atômicas da própria biblioteca UnB-DALi [2] e algumas outras construções de diagramas que foram julgadas importantes para este trabalho.

Vale ressaltar que testes de quaisquer tipos de sistemas de software têm o objetivo de provar sua correteza, não sendo possível garantir totalmente a ausência de defeitos [13].

Para uma maior precisão e automatização, foram utilizadas diversas classes de teste com o *framework JUnit*. O tempo de execução de um teste é o tempo que o método *toDTMC()* leva para converter um diagrama de Atividade ou Sequência (do tipo *AbstractAGGModel*) para a linguagem DTMC do PRISM. Cada teste foi executado 10 vezes consecutivas, sendo que apenas o tempo médio de execução e seu desvio-padrão são exibidos nas tabelas abaixo. Os testes foram conduzidos em um computador Dell, com memória RAM de 6 GB, processador 2.67 GHz Intel Core i5.

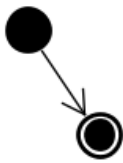




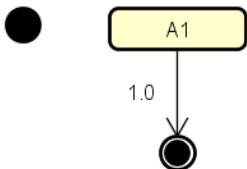




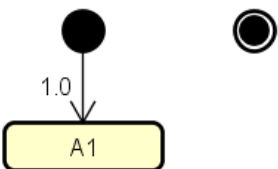




Todos os testes se encontram no repositório do projeto no *GitHub* (link no capítulo seguinte de conclusão), dentro do diretório “tests”. Todos os testes aqui realizados foram feitos a partir de diagramas modelados na ferramenta Astah. Testes iguais foram feitos para a ferramenta Papyrus e geraram resultados praticamente iguais, a não ser pelo tempo da conversão, que diferiu ligeiramente.



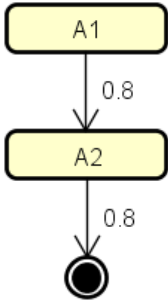

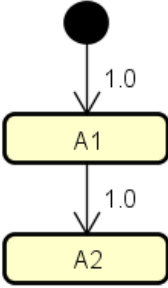

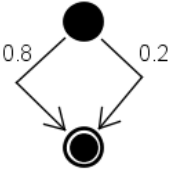

4.2 Testes e Resultados Obtidos

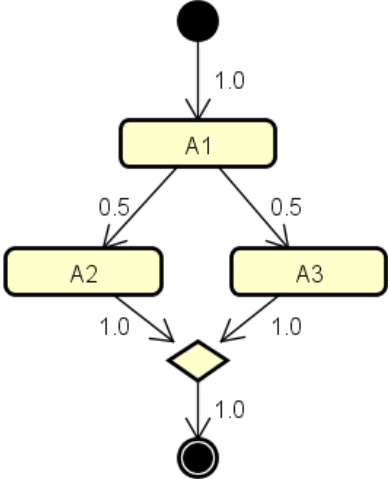

As tabelas a seguir, relacionam os testes realizados de acordo com as classes de equivalência escolhidas.

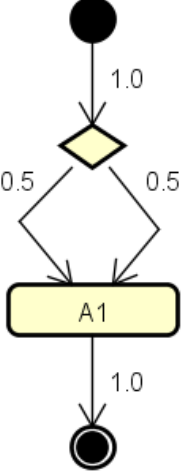

4.2.1 AD \rightarrow PRISM

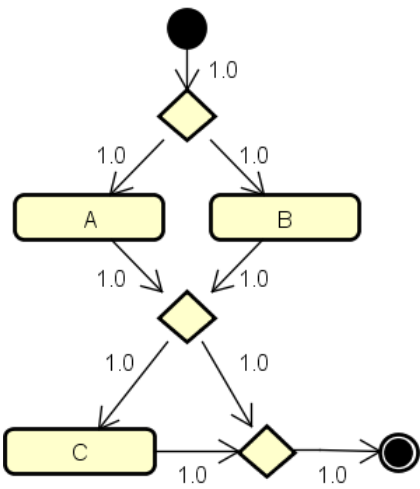

Tabela 4.1: Resultados dos testes de conversão de ADs para PRISM

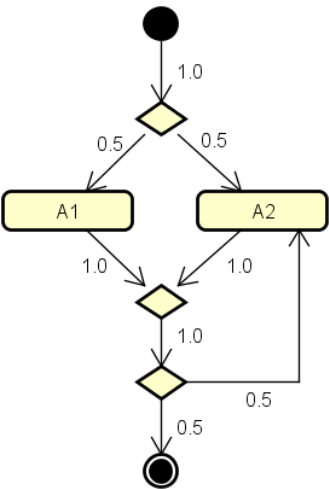

Teste 1: <i>Control flow</i> sem probabilidade associada			
Modelo de Entrada		Modelo de Saída - PRISM	
			
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
			-
Teste 2: <i>Control flow</i> obrigatório partindo de um Nó Inicial			
Modelo de Entrada		Modelo de Saída - PRISM	
			
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
			-
Teste 3: <i>Control flow</i> obrigatório chegando a um Nó Final			
Modelo de Entrada		Modelo de Saída - PRISM	
			
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
			-

Teste 4: <i>Control flow</i> obrigatório partindo de um Nó de Decisão			
Modelo de Entrada		Modelo de Saída - PRISM	
			
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✗	✗	✓	-
Teste 5: <i>Control flow</i> obrigatório chegando a um Nó Executável			
Modelo de Entrada		Modelo de Saída - PRISM	
			
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✗	✗	✓	-
Teste 6: <i>Control flow</i> obrigatório partindo de um Nó Executável			
Modelo de Entrada		Modelo de Saída - PRISM	
			
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✗	✗	✓	-
Teste 7: Múltiplos <i>Control flows</i> chegando a um Nó Final			
Modelo de Entrada		Modelo de Saída - PRISM	
			
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✗	✗	✓	-

Teste 8: Múltiplos <i>Control flows</i> partindo de um Nó Executável			
<p>Modelo de Entrada</p> 		<p>Modelo de Saída - PRISM</p> 	
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✗	✗	✓	-

Teste 9: Múltiplos <i>Control flows</i> chegando a um Nó Executável			
<p>Modelo de Entrada</p> 		<p>Modelo de Saída - PRISM</p> 	
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✗	✗	✓	-

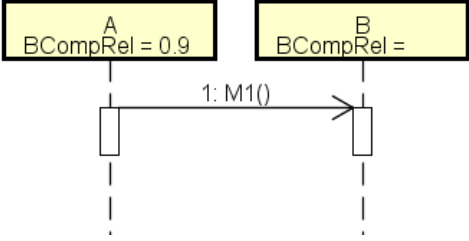

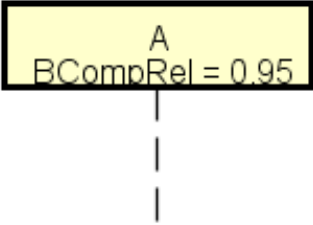
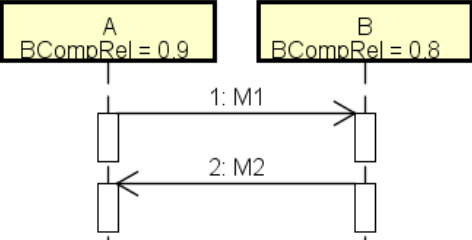
Teste 10: Nó misto de Decisão e Merge, não reconhecido pela UnB-DALi			
<p>Modelo de Entrada</p> 		<p>Modelo de Saída - PRISM</p> 	
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✗	✗	✓	-

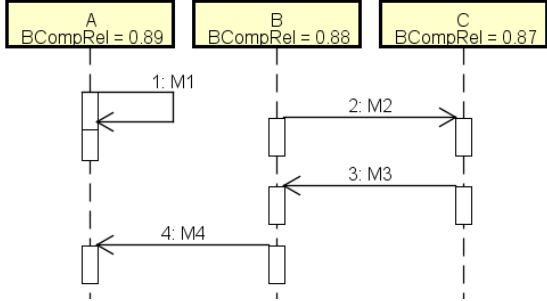
Teste 11: <i>Loop de Control flow</i>			
<p>Modelo de Entrada</p> 		<p>Modelo de Saída - PRISM</p> 	
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✗	✗	✓	-

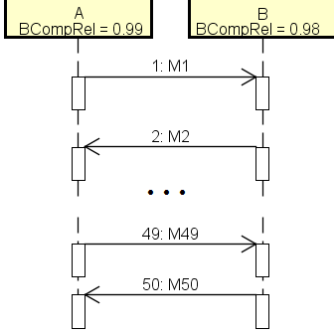
Teste 12: Diagrama simples com Nô Inicial para Nô Final			
Modelo de Entrada	Modelo de Saída - PRISM		
	<pre> 1 dtmc 2 3 module IN_To_FN 4 sA : [-1..2] init 0; 5 6 [A0_fail] sA=-1 -> 1.0:(sA'=-1); 7 [A0_final] sA=2 -> 1.0:(sA'=2); 8 [] sA=0 -> 1.0:(sA'=1); 9 [A0_initial] sA=1 -> 0.9:(sA'=2) + 0.1:(sA'=-1); 10 endmodule </pre>		
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✓	✓	✓	$\sim 137.5ms \pm 24.5ms$
Teste 13: Diagrama contendo todos os elementos reconhecíveis de um AD			
Modelo de Entrada	Modelo de Saída - PRISM		
	<pre> 1 dtmc 2 3 module All_Elements 4 sA : [-1..4] init 0; 5 6 [A0_initial] sA=1 -> 0.5:(sA'=3) + 0.5:(sA'=4); 7 [] sA=3 -> 1.0:(sA'=2); 8 [A0_final] sA=2 -> 1.0:(sA'=2); 9 [] sA=0 -> 1.0:(sA'=1); 10 [A0_fail] sA=-1 -> 1.0:(sA'=-1); 11 [] sA=4 -> 1.0:(sA'=2); 12 endmodule </pre>		
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✓	✓	✓	$\sim 294.7ms \pm 21.8ms$
Teste 14: Diagrama contendo todos os elementos reconhecíveis de um AD e com repetições			
Modelo de Entrada	Modelo de Saída - PRISM		
	<pre> 1 dtmc 2 3 module All_Elements_Repetition 4 sA : [-1..5] init 0; 5 6 [A0_fail] sA=-1 -> 1.0:(sA'=-1); 7 [A0_final] sA=2 -> 1.0:(sA'=2); 8 [] sA=0 -> 1.0:(sA'=1); 9 [] sA=3 -> 1.0:(sA'=2); 10 [] sA=5 -> 1.0:(sA'=2); 11 [A0_initial] sA=1 -> 0.5:(sA'=3) + 0.5:(sA'=4); 12 [] sA=4 -> 1.0:(sA'=5); 13 endmodule </pre>		
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✓	✓	✓	$\sim 555.7ms \pm 185.0ms$
Teste 15: AD com 100 Nós Executáveis			
Modelo de Entrada	Modelo de Saída - PRISM		
	<pre> 1 dtmc 2 3 module One_Hundred_EN 4 sA : [-1..102] init 0; 5 6 [] sA=61 -> 1.0:(sA'=62); 7 [A0_final] sA=2 -> 1.0:(sA'=2); 8 [] sA=41 -> 1.0:(sA'=42); 9 [] sA=80 -> 1.0:(sA'=81); 10 [] sA=33 -> 1.0:(sA'=34); 11 [] sA=24 -> 1.0:(sA'=25); </pre>		
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✓	✓	✓	$\sim 5072.6ms \pm 397.0ms$

4.2.2 SD \rightarrow PRISM

Tabela 4.2: Resultados dos testes de conversão de SDs para PRISM

Teste 1: <i>Lifeline</i> sem probabilidade associada			
<p>Modelo de Entrada</p> 		<p>Modelo de Saída - PRISM</p> 	
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✗	✗	✓	-
Teste 2: 1 <i>Lifeline</i> sem mensagens			
<p>Modelo de Entrada</p> 		<p>Modelo de Saída - PRISM</p> <pre> 1 dtmc 2 3 module One_Lifeline 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 </pre>	
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✓	✓	✓	$\sim 123.7ms \pm 10.6ms$
Teste 3: 2 <i>Lifelines</i> trocando mensagens			
<p>Modelo de Entrada</p> 		<p>Modelo de Saída - PRISM</p> <pre> 1 dtmc 2 3 module Two_Lifelines 4 sA : [-1..2] init 0; 5 sB : [-1..2] init 0; 6 7 [] sA=1 & sB=2 -> 1.0: (sA'=2); 8 [end_A] sA=2 -> 1.0: (sA'=2); 9 [fail_A] sA=-1 -> 1.0: (sA'=-1); 10 [init_A] sA=0 -> 0.8: (sA'=1) + 0.2: (sA'=-1); 11 [init_B] sB=0 & sA=1 -> 1.0: (sB'=1); 12 [] sB=1 -> 0.9: (sB'=2) + 0.1: (sB'=-1); 13 [end_B] sB=2 -> 1.0: (sB'=2); 14 [fail_B] sB=-1 -> 1.0: (sB'=-1); 15 endmodule </pre>	
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✓	✓	✓	$\sim 172.8ms \pm 19.9ms$

Teste 4: 3 Lifelines trocando mensagens			
Modelo de Entrada		Modelo de Saída - PRISM	
		<pre> 1 dtmc 2 3 module Three_Lifelines 4 sA : [-1..2] init 0; 5 sB : [-1..3] init 0; 6 sC : [-1..2] init 0; 7 8 [end_A] sA=2 -> 1.0: (sA'=2); 9 [] sA=1 & sB=3 -> 1.0: (sA'=2); 10 [fail_A] sA=-1 -> 1.0: (sA'=-1); 11 [init_A] sA=0 -> 0.89: (sA'=1) + 0.11: (sA'=-1); 12 [fail_B] sB=-1 -> 1.0: (sB'=-1); 13 [] sB=1 & sC=2 -> 1.0: (sB'=2); 14 [init_B] sB=0 -> 0.87: (sB'=1) + 0.13: (sB'=-1); 15 [end_B] sB=3 -> 1.0: (sB'=3); 16 [] sB=2 -> 0.89: (sB'=3) + 0.11: (sB'=-1); 17 [init_C] sC=0 & sB=1 -> 1.0: (sC'=1); 18 [fail_C] sC=-1 -> 1.0: (sC'=-1); 19 [] sC=1 -> 0.88: (sC'=2) + 0.12: (sC'=-1); 20 [end_C] sC=2 -> 1.0: (sC'=2); 21 endmodule </pre>	
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✓	✓	✓	~ 226.0ms ± 20.7ms

Teste 5: 2 Lifelines trocando 50 mensagens			
Modelo de Entrada		Modelo de Saída - PRISM	
		<pre> 1 dtmc 2 3 module Fifty_Messages 4 sA : [-1..50] init 0; 5 sB : [-1..50] init 0; 6 7 [] sA=24 -> 0.98: (sA'=25) + 0.02: (sA'=-1); 8 [] sA=31 & sB=32 -> 1.0: (sA'=32); 9 [] sA=11 & sB=12 -> 1.0: (sA'=12); 10 [] sA=23 & sB=24 -> 1.0: (sA'=24); 11 [] sA=21 & sB=22 -> 1.0: (sA'=22); </pre>	
Sucesso	PRISM Compilável	Conforme Esperado	Tempo
✓	✓	✓	~ 1431.7ms ± 117.5ms

4.3 Considerações sobre os testes

Uma análise do resultado mostra que a solução desenvolvida atende todas as classes de equivalência que foram testadas. Algumas regras estabelecidas na monografia de Oliveira [2] foram ignoradas nestes testes pelo fato das próprias ferramentas utilizadas para a elaboração dos modelos (Astah e Papyrus) já terem as mesmas restrições em suas próprias regras de definição. Por exemplo, sabe-se que a biblioteca UnB-DALi, no momento do desenvolvimento deste projeto, não oferece suporte a *loops* de *control flow* em Diagramas de Atividade, fato que pôde ser comprovado a partir de testes nas próprias ferramentas de modelagem.

Em relação ao tempo de execução, pode-se concluir que ele aumenta conforme complexidade do modelo. Tais resultados podem fornecer informações para otimização da solução desenvolvida em projetos futuros.

Portanto, pode-se concluir que o principal objetivo do projeto, que é demonstrar a capacidade de interoperabilidade entre ferramentas de modelagem UML de diferentes fabricantes e a UnB-DALi, foi atingido.

Capítulo 5

Conclusão e Trabalhos Futuros

Neste trabalho foi implementado e parcialmente validado um conversor automatizado de modelos UML em modelos PRISM, utilizando a biblioteca *UnB-DALi* e a API da *SDMetrics*.

A ideia deste trabalho partiu de um dos possíveis trabalhos futuros sugeridos por Oliveira no capítulo de conclusão de sua monografia, o qual sugeria um “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” [2]. Tal estudo era de interesse para o avanço da biblioteca *UnB-DALi*, pois a entrada de dados dependia da digitação, em código Java, dos diagramas UML a serem transformados, o que é diferente da maneira usual de se expressar diagramas UML, que geralmente são desenhados em alguma ferramenta de modelagem utilizando-se uma interface gráfica.

O argumento de Oliveira de que XMI não é interoperável, deve-se à não conformidade dos arquivos XMI gerados por várias ferramentas de modelagem ao padrão estabelecido pela OMG. A ideia do XMI era justamente o intercâmbio de metadados, o que tornaria possível um diagrama criado em uma ferramenta A poder ser lido e reconhecido em uma ferramenta B, mas tal fato não se confirma na prática. Logo, estaríamos trabalhando com diversos formatos diferentes de arquivos XMI.

Era necessário um *parser* para extrair as informações dos diagramas UML em formato XMI e, além disso, este *parser* precisava ser facilmente adaptável aos diferentes XMI gerados por diferentes ferramentas de modelagem. Caso contrário, o projeto ficaria engessado e somente seria utilizável por uma ferramenta de modelagem específica. O *parser* encontrado que tinha estas características foi o da API da *SDMetrics*, uma ferramenta de medição de qualidade de *designs* para a UML orientados à objeto. O XMI *parser* desta API trabalha com dois arquivos para extração das informações dos diagramas: um é o metamodelo, um arquivo que representa os elementos reconhecíveis pelo *parser* e o outro é o *XMI transformation*, um arquivo que mapeia as informações dos elementos e as *tags* e respectivos atributos de interesse. A partir daí, após uma verificação de tipo do elemento, chama-se os métodos de construção de um modelo abstrato fornecidos pela *UnB-DALi* de Oliveira.

A escolha da linguagem Java foi natural devido ao fato das duas bibliotecas utilizadas no trabalho estarem nessa linguagem.

O conversor funciona como um arquivo JAR executável, ou seja, o interessado na utilização deve construir o projeto seguindo as instruções no sítio onde se encontra a ferramenta. A execução é via console (por meio do comando `java -jar`), na pasta onde estão os arquivos XML. O resultado da conversão para cada arquivo gera um *output* de mesmo nome mas com a extensão “.pm”, no formato PRISM.

Alguns pontos da ferramenta precisam de melhoria:

- O tratamento de exceções lançadas pela *UnB-DALi* com mensagens mais explicativas
- Padronização no lançamento de exceções da ferramenta

A ferramenta atual conta com um arquivo-modelo para a customização do arquivo de mapeamento, o *XMI transformation*. Tal medida foi pensada para ajudar na evolução da ferramenta em relação ao suporte a novas ferramentas de modelagem que não foram compreendidas neste trabalho.

Como possíveis trabalhos futuros, temos:

- Integração da ferramenta às ferramentas de modelagem no mercado, por meio de *plugin*, a fim de viabilizar uma opção de exportação como “Exportar como PRISM” diretamente na ferramenta.
- Suporte a mais ferramentas de modelagem no mercado através da criação de novos mapeamentos específicos para estas ferramentas, por meio do arquivo-modelo do *XMI transformation*.
- Implementação de um programa simples, com campos a serem preenchidos pelo usuário, para gerar automaticamente o *XMI Transformation* de sua ferramenta UML específica.
- Acompanhamento da evolução lado a lado desta ferramenta junto com a *UnB-DALi*, tendo em vista que são trabalhos complementares. A medida que a *UnB-DALi* expandir suas capacidades, será possível também expandir a ferramenta que a utiliza.

O código-fonte do XMI-PRISM-Converter está disponível para *download* via repositório público no *GitHub* do autor Pedro Lima: <https://github.com/PedroPlima/XMI-PRISM-Converter>. O conversor está documentado conforme padrão Javadoc e instruções sobre a compilação do projeto são dadas no arquivo *README.md*.

Referências

- [1] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):1–23, 2004. 2, 4
- [2] A.E.C. Oliveira. Unb-dali: Biblioteca para transformação de modelos em análise de dependabilidade. UnB - Monografia de conclusão de curso, 2016. vii, 2, 10, 11, 18, 34, 42, 43
- [3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008. vii, 1, 7
- [4] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley Object Technology Series, 2005. 12, 13
- [5] J. Grose, G.C. Doney, and S.A. Brodsky. *Mastering XMI - Java Programming with XMI, XML and UML*. John Wiley and Son, 2002. 16
- [6] James Gleick. Sometimes a bug is more than a nuisance. <https://around.com/ariane.html>, acessado em 31/10/2017, 1996. 1
- [7] Jürgen Wüst. Sdmetrics user manual. <http://www.sdmetrics.com/Demo.html>, 28/11/2016, 2016. 18, 19, 51
- [8] J. Kovse and T. Harder. Generic xmi-based uml model transformations. 2002. 15
- [9] Nancy Leveson. Medical devices: The therac-25, 1995. 1
- [10] OMG. Omg unified modeling language. <http://www.omg.org/spec/UML/2.0/>, acessado em 08/06/2017, 2015. vii, 12
- [11] P.A. de G.V. Bernardes. Transformação automática de modelos uml para modelos markovianos parametrizados. UnB - Monografia de conclusão de curso, 2015. 17
- [12] Peter Bond. Cluster to rise from the ashes. <https://spaceflightrightnow.com/cluster2/000714feature/>, acessado em 31/10/2017, 2000. vii, 2
- [13] R. Pressman. *Engenharia de Software - 7a Edição*. AMGH, 2011. 34
- [14] G.N. Rodrigues, V. Alves, R. Franklin, and L. Laranjeira. Dependability analysis in the ambient assisted living domain: An exploratory case study. *Journal of Systems and Software*, 85(010028):112–131, 2011. vii, 2, 8, 9, 10, 24

- [15] SDMetrics. Sdmetrics - *The Software Design Metrics tool for the UML*. <http://www.sdmetrics.com/>, acessado em 17/11/2016, 2017. 17
- [16] I. Sommerville. *Software Engineering - 9th Edition*. Addison-Wesley, 501 Boylston Street, Suite 900, Boston, Massachusetts, USA, 2011. 4, 12
- [17] University of Oxford. Prism website. <http://www.prismmodelchecker.org/>, acessado em 28/06/2016, 2016. 8
- [18] W3 Schools. Xsl(t) *Languages*. https://www.w3schools.com/xml/xsl_languages.asp, acessado em 24/08/2017, 2017. 16
- [19] W3C. Document object model (dom) level 2 core specification. <https://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/>, acessado em 31/08/2017, 2000. 16
- [20] S. Yang and G. Li. Using model checking to verify the logic module of flight control software. *Proceedings of the 2nd International Conference On Systems Engineering and Modeling*, 1, 2013. 5

Anexo I

Estrutura do XMI das ferramentas de modelagem utilizadas

I.1 Astah Professional

O *Astah Professional*, antigamente conhecido como JUDE (*Java and UML Developers Environment*), é um *software* para modelagem UML e criação de mapas mentais. Há uma versão gratuita denominada *Astah Community* com menos funcionalidades. A versão *Professional* é paga, porém pode ser gratuita desde que o interessado demonstre que deseja utilizar a ferramenta para fins educativos e forneça um *email* acadêmico para adquirir uma licença.

O Astah foi escolhido por ser uma ferramenta popular, fácil de mexer e que os autores já tinham familiaridade, além da funcionalidade de exportação de diagramas UML em formato XMI.

Abaixo estão as estruturas de dois arquivos XMI gerados pelo Astah. Um deles contém um Diagrama de Atividades e o outro um Diagrama de Sequência, ambos contendo todos os elementos relevantes para esta monografia. Os nomes dos elementos, em preto, indicam em qual *tag* as informações pertinentes àquele elemento se encontram.

Código I.1: Estrutura dos elementos de um AD do XMI do Astah

```

1 <XMI>
2   <XMI.content>
3     <UML:Model>
4       <UML:Namespace.ownedElement>
5         <UML:ActivityGraph> Activity Diagram
6         <UML:StateMachine.top>
7           <UML:CompositeState>
8             <UML:CompositeState.subvertex>
9               <UML:ActionState>Executable Node</UML:ActionState>
10              <UML:Pseudostate>
11                Initial, Merge and Decision Node
12              </UML:Pseudostate>
13              <UML:FinalState>Final Node</UML:FinalState>
14            </UML:CompositeState.subvertex>
15          </UML:CompositeState>
16        </UML:StateMachine.top>
17        <UML:StateMachine.transitions>
18          <UML:Transition> Transition
19            <UML:ModelElement.taggedValue>
20              <UML:TaggedValue>
21                Transition probability
22              </UML:TaggedValue>
23            </UML:ModelElement.taggedValue>
24            <UML:Transition.source>
25              Transition source
26            </UML:Transition.source>
27            <UML:Transition.target>
28              Transition target
29            </UML:Transition.target>
30          </UML:Transition>
31        </UML:StateMachine.transitions>
32      </UML:ActivityGraph>
33    </UML:Namespace.ownedElement>
34  </UML:Model>
35 </XMI.content>
36 </XMI>

```

Código I.2: Estrutura dos elementos de um SD do XMI do Astah

```

1 <XMI>
2   <XMI.content>
3     <UML:Model>
4       <UML:Namespace.collaboration>
5         <UML:Collaboration>
6           <UML:Namespace.ownedElement>
7             <UML:ClassifierRole> Lifeline
8               <UML:ModelElement.taggedValue>
9                 <UML:TaggedValue>Lifeline probability</UML:TaggedValue>
10              </UML:ModelElement.taggedValue>
11            </UML:ClassifierRole>
12          </UML:Namespace.ownedElement>
13        </UML:Collaboration>
14      <UML:Collaboration.interaction>
15        <UML:Interaction>
16          <UML:Interaction.message>
17            <UML:Message> Message
18              <UML:Message.sender>
19                <UML:ClassifierRole>
20                  Message sender
21                </UML:ClassifierRole>
22              </UML:Message.sender>
23            <UML:Message.receiver>
24              <UML:ClassifierRole>
25                Message receiver
26              </UML:ClassifierRole>
27            </UML:Message.receiver>
28          <UML:Message.action>
29            <UML:Action>Async message attribute</UML:Action>
30          </UML:Message.action>
31        </UML:Message>
32      </UML:Interaction.message>
33    </UML:Interaction>
34  </UML:Collaboration.interaction>
35 </UML:Namespace.collaboration>
36 </UML:Model>
37 </XMI.content>
38 </XMI>

```

I.2 Papyrus

O *Papyrus* é uma ferramenta de código aberto baseada no ambiente de desenvolvimento integrado *Eclipse*, desenvolvida para oferecer suporte à criação e edição de diagramas UML 2.0. É uma ferramenta utilizada por vários grupos de pesquisa acadêmica e indústrias de software por todo o mundo.

Após uma pesquisa por várias ferramentas de *design* de diagramas UML que se baseavam no padrão XMI, a ferramenta Papyrus foi escolhida por ser fácil de usar e aderir ao padrão UML estabelecido pelo grupo *OMG*. O arquivo XMI que descreve o diagrama UML desenvolvido é significativamente mais simples do que os equivalentes gerados por outras ferramentas.

Seguem abaixo as estruturas de dois arquivos XMI gerados pela ferramenta Papyrus. Um fornece a estrutura de um Diagrama de Atividades e outro de um Diagrama de Sequência.

Código I.3: Estrutura dos elementos de um AD do XMI do Papyrus

```
1 <uml:Model>
2   <packagedElement xmi:type="uml:Activity"> Diagram (Attributes: type,
3     id, name, etc)
4     <edge> Transition (Has: id, source, target, etc)
5       <weight/> Transition probability
6     </edge>
7     <node/> Nodes. Differentiated by "xmi:type" attribute: Initial,
8       Final, Decision, Merge and Executable. Has: name, id, incoming and
        outgoing edges.
9   </packagedElement>
10 </uml:Model>
```

Código I.4: Estrutura dos elementos de um SD do XMI do Papyrus

```
1 <uml:Model>
2   <packagedElement> Diagram (Attributes: type, id, name, etc)
3     <lifeline>
4       <selector/> Lifeline probability
5     </lifeline>
6     <fragment/> Two \textit{fragment} tags are created for each message,
7       each containing sender and receiver of message, respectively. The
8       two tags reference the same \textit{message} tag by the "message"
9       attribute.
10    <message/> Has: name and ID, which is referenced by the \textit{
11      fragment} tag, in a cross reference manner. Also has the asynchronous
12      attribute.
13    </packagedElement>
14 </uml:Model>
```

Anexo II

Arquivos de configuração SDMetrics

Para deixar sua ferramenta mais flexível, Wüst [7] criou dois arquivos de configuração em formato XMI para reconhecer elementos de diagramas UML de diferentes ferramentas de modelagem do mercado. Os arquivos são o *SDMetrics Metamodel* e o *XMI Transformation File*. As estruturas destes dois arquivos são explicadas no Capítulo 2, na seção SDMetrics 2.7. A seguir são mostrados os arquivos de fato utilizados na ferramenta desenvolvida nesta monografia.

II.1 Metamodelo SDMetrics

Como explicado no Capítulo 2, o metamodelo é um arquivo XMI que guarda informações a respeito dos tipos de elementos que a SDMetrics consegue reconhecer. No caso do nosso metamodelo, somente é necessário oferecer suporte aos diagramas de Atividade e Sequência e a um sub-grupo de elementos reconhecíveis destes diagramas atualmente pela UnB-DALi.

Caso haja alguma evolução nos elementos reconhecidos pela UnB-DALi no futuro, este arquivo deve ser modificado de acordo.

Código II.1: Metamodelo SDMetrics para elementos da *UnB-DALi*

```
1 <sdmetricsmetamodel version="2.0">
2   <!-- Base Element -->
3   <modelelement name="sdmetricsbase">
4     <attribute name="context" type="ref" multiplicity="one">Owner of the
       element in the UML model.</attribute>
5     <attribute name="id" type="data" multiplicity="one">Unique
       identifier of the model element.</attribute>
6     <attribute name="name" type="data" multiplicity="one">Name of the
       element in UML model.</attribute>
7   </modelelement>
8
9   <!-- Diagrams -->
10  <modelelement name="diagram">
11    <attribute name="type" type="data" multiplicity="one">The type of
       the UML Diagram. AD or SD.</attribute>
12  </modelelement>
13
14  <!-- Activity Diagram - Nodes -->
```

```

15 <modelelement name="node">
16   <attribute name="type" type="data" multiplicity="one">The type of
    the node: executable, initial, final, merge or decision.</attribute>
17   <attribute name="incomingEdges" type="data" multiplicity="many">The
    incoming edges of a node.</attribute>
18 </modelelement>
19
20 <!-- Activity Diagram - Edges -->
21 <modelelement name="controlflow">
22   <attribute name="type" type="data" multiplicity="one">The type of
    the control flow.</attribute>
23   <attribute name="source" type="data" multiplicity="one">Source of
    the control flow (transition).</attribute>
24   <attribute name="target" type="data" multiplicity="one">Target of
    the control flow(transition).</attribute>
25   <attribute name="probability" type="data" multiplicity="one">
    Probability of the control flow (transition).</attribute>
26 </modelelement>
27
28 <!-- Sequence Diagram - Lifelines -->
29 <modelelement name="lifeline">
30   <attribute name="type" type="data" multiplicity="one">The type of the
    lifeline.</attribute>
31   <attribute name="BCompRel" type="data" multiplicity="one">The
    reliability of a lifeline (component).</attribute>
32 </modelelement>
33
34 <!-- Sequence Diagram - Messages -->
35 <modelelement name="message">
36   <attribute name="type" type="data" multiplicity="one">The message
    type (ONLY asynchronous for now).</attribute>
37   <attribute name="source" type="data" multiplicity="one">The sender
    of the message.</attribute>
38   <attribute name="target" type="data" multiplicity="one">The receiver
    of the message.</attribute>
39 </modelelement>
40 </sdmetricsmetamodel>

```

II.2 XMI Transformations

Como visto no Capítulo 2, o *XMI Transformation* é o arquivo XMI responsável por mapear os elementos definidos no metamodelo para as *tags* específicas dos XMI gerados por cada ferramenta de modelagem. Atualmente, só há suporte para duas ferramentas: Astah e Papyrus. No futuro, pode-se criar novos *XMI Transformation* para ferramentas de modelagem diversas. Para a criação deste arquivo de mapeamento, deve-se analisar o XMI gerado e anotar as *tags* que guardam informações relevantes acerca do diagrama reconhecidas pelo metamodelo da UnB-DALi. Note que à medida que o metamodelo for evoluindo, o mapeamento deve acompanhá-lo.

Na pasta do projeto do XMI-PRISM-Converter, há um arquivo de modelo para guiar mais facilmente a criação de um novo *XMI Transformation File* para uma nova ferramenta de modelagem.

Abaixo estão os arquivos *XMI Transformation* das ferramentas Astah Professional e Papyrus.

II.2.1 Astah Professional

Código II.2: XMI Transformation do Astah Professional

```
1 <xmitransformations version="2.0">
2   <!-- Base Element -->
3   <xmitransformation modelelement="sdmetricsbase" xmipattern="
4     sdmetricsbase" recurse="true">
5     <trigger name="id" type="attrval" attr="xmi.id" />
6     <trigger name="name" type="attrval" attr="name" />
7   </xmitransformation>
8
9   <!-- id -->
10  <xmitransformation modelelement="diagram" xmipattern="UML:Namespace.
11    ownedElement" recurse="true">
12    <trigger name="id" type="cattrval" src="UML:ActivityGraph" attr="xmi
13      .id" />
14  </xmitransformation>
15
16  <xmitransformation modelelement="diagram" xmipattern="UML:Namespace.
17    collaboration" recurse="true">
18    <trigger name="id" type="cattrval" src="UML:Collaboration" attr="xmi
19      .id" />
20  </xmitransformation>
21
22  <!-- name -->
23  <xmitransformation modelelement="diagram" xmipattern="UML:Model"
24    recurse="true">
25    <trigger name="name" type="attrval" attr="name" />
26  </xmitransformation>
27
28  <!-- type -->
29  <xmitransformation modelelement="diagram" xmipattern="UML:
30    ActivityGraph" recurse="true">
31    <trigger name="type" type="constant" attr="uml:Activity" />
32  </xmitransformation>
33
34  <xmitransformation modelelement="diagram" xmipattern="UML:
35    Collaboration" recurse="true">
36    <trigger name="type" type="constant" attr="uml:Interaction" />
37  </xmitransformation>
38
39  <!-- Executable Node -->
40  <xmitransformation modelelement="node" xmipattern="UML:ActionState"
41    recurse="true">
42    <trigger name="type" type="constant" attr="uml:OpaqueAction" />
43  </xmitransformation>
44
45  <!-- Initial Node, Decision Node and Merge Node -->
46  <xmitransformation modelelement="node" xmipattern="UML:Pseudostate"
47    recurse="true">
48    <trigger name="type" type="attrval" attr="kind" />
49    <trigger name="incomingEdges" type="gcattrval" src="UML:StateVertex.
50      incoming" attr="xmi.idref" />
51  </xmitransformation>
52</xmitransformations>
```



```

38 </xmitransformation>
39
40 <!-- Final Node -->
41 <xmitransformation modelelement="node" xmipattern="UML:FinalState"
42   recurse="true">
43   <trigger name="type" type="constant" attr="uml:ActivityFinalNode" />
44 </xmitransformation>
45
46 <!-- Control Flows (transitions) -->
47 <xmitransformation modelelement="controlflow" xmipattern="UML:
48   Transition" recurse="true">
49   <trigger name="type" type="constant" attr="controlflow"/>
50   <trigger name="source" type="gcattrval" src="UML:Transition.source"
51   attr="xmi.idref" />
52   <trigger name="target" type="gcattrval" src="UML:Transition.target"
53   attr="xmi.idref" />
54   <trigger name="probability" type="gcattrval" src="UML:ModelElement.
55   taggedValue" attr="value" />
56 </xmitransformation>
57
58 <!-- Lifelines -->
59 <xmitransformation modelelement="lifeline" xmipattern="UML:
60   ClassifierRole" recurse="true">
61   <trigger name="type" type="constant" attr="lifeline"/>
62   <trigger name="BCompRel" type="gcattrval" src="UML:ModelElement.
63   taggedValue" attr="value" />
64 </xmitransformation>
65 </xmitransformations>

```

II.2.2 Papyrus

Código II.3: XMI Transformation do Papyrus

```
1 <xmitransformations version="2.0">
2   <xmitransformation modelelement="sdmetricsbase" xmipattern="uml:Model"
3     recurse="true">
4     <trigger name="id" type="attrval" attr="xmi:id" />
5     <trigger name="name" type="attrval" attr="name" />
6   </xmitransformation>
7   <xmitransformation modelelement="diagram" xmipattern="packagedElement"
8     recurse="true">
9     <trigger name="type" type="attrval" attr="xmi:type"></trigger>
10  </xmitransformation>
11  <xmitransformation modelelement="node" xmipattern="node" recurse="true"
12    ">
13    <trigger name="type" type="attrval" attr="xmi:type" />
14    <trigger name="incomingEdges" type="attrval" attr="incoming" />
15  </xmitransformation>
16  <xmitransformation modelelement="controlflow" xmipattern="edge"
17    recurse="true">
18    <trigger name="type" type="constant" attr="controlflow"/>
19    <trigger name="source" type="attrval" attr="source" />
20    <trigger name="target" type="attrval" attr="target" />
21    <trigger name="probability" type="cattrval" src="weight" attr="value"
22      "/>
23  </xmitransformation>
24  <xmitransformation modelelement="lifeline" xmipattern="lifeline"
25    recurse="true">
26    <trigger name="type" type="constant" attr="lifeline"/>
27    <trigger name="BCompRel" type="cattrval" src="selector" attr="value"
28      />
29  </xmitransformation>
30  <xmitransformation modelelement="message" xmipattern="fragment"
31    recurse="true">
32    <trigger name="type" type="constant" attr="asynchronous" />
33    <trigger name="source" type="attrval" attr="covered" />
34    <trigger name="target" type="attrval" attr="covered" />
35  </xmitransformation>
36 </xmitransformations>
```

II.2.3 *Template para XMI Transformation File*

Código II.4: *Template para criação de novos XMI Transformation Files*

```
1 <?xml version="1.0"?>
2 <!DOCTYPE xmitransformations SYSTEM 'xmitrans.dtd'>
3
4 <!--
5 XMI transformations for the SDMetrics V2.3 default metamodel
6 for UML2.x and XMI 2.x source files.
7
8 Copyright (c) 2002-2013 Juergen Wuest
9
10 The MIT License
11
12 Permission is hereby granted, free of charge, to any person obtaining a
13   copy
14 of this SDMetrics project file (the "Project File"), to deal in the
15   Project File
16 without restriction, including without limitation the rights
17 to use, copy, modify, merge, publish, distribute, sublicense, and/or
18   sell
19 copies of the Project File, and to permit persons to whom the Project
20   File is
21 furnished to do so, subject to the following conditions:
22
23 The above copyright notice and this permission notice shall be included
24   in
25 all copies or substantial portions of the Project File.
26
27 THE PROJECT FILE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
28   EXPRESS OR
29 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
30 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
31   THE
32 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
33 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
34   FROM,
35 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
36   IN
37 THE PROJECT FILE.
38
39 Note: this license grant only applies to this Project File, and not to
40   any
41 other part of SDMetrics.
42   -->
43
44 <!-- Default Transformation - Adaptable to the XMI structure of an UML
45   modeling tool -->
46
47 <!-- Check the SDMetrics Manual inside the docs folder for specific
48   questions.
49   Or the website: http://www.sdmetrics.com/manual/MetaTrans.html -->
50
51 <!-- GENERAL TIPS AND EXPLANATION -->
52
53 <!-- "modelelement" is the name of the Metamodel element you are
54   referring to -->
```

```

40 <!-- "xmipattern" is where you put the tag name of where the model
    element information is inside the XMI file -->
41
42 <!-- The trigger attributes: -->
43 <!-- name: Name of the attribute in the Metamodel element -->
44 <!-- type: Depends on where the information is. It can be attrval,
    ctext, cattrval, gattrval, constant, etc. Check
45     SDMetrics manual section 7.2.2 for more information -->
46 <!-- attr: The name of the attribute you want, inside the XMI tag -->
47
48 <!-- id is generally "xmi.id" or "xmi:id". Change according to your
    tool. -->
49 <!-- id and name of the element may be in different tags. Create 2 <
    xmitransformation> elements in this case. -->
50 <!-- src="" attribute depends on the type of the trigger. Used in
    cattrval and gattrval (child and grandchild) -->
51 <xmitransformations version="2.0">
52
53 <!-- Base Element - Don't change anything here other than the "xmi.id"
    attribute (to xmi:id, for example). -->
54 <xmitransformation modelelement="sdmetricsbase" xmipattern="
    sdmetricsbase" recurse="true">
55     <trigger name="id" type="attrval" attr="xmi.id" />
56     <trigger name="name" type="attrval" attr="name" />
57 </xmitransformation>
58
59 <!-- * Diagram * -->
60 <!-- If in different tags, create an <xmitransformation> for each one.
    -->
61 <xmitransformation modelelement="diagram" xmipattern="
    TAG_OF_THE_DIAGRAM_INFORMATIONS" recurse="true">
62     <trigger name="id" type="" attr="xmi.id" />
63     <trigger name="name" type="" attr="name" />
64     <trigger name="type" type="" attr="Activity Diagram" />
65 </xmitransformation>
66
67 <!-- * AD Elements * -->
68 <!-- Activity Control Nodes -->
69
70 <!-- Executable Node/Opaque Action -->
71 <xmitransformation modelelement="node" xmipattern="
    TAG_OF_THE_NODE_INFORMATIONS" recurse="true">
72     <trigger name="type" type="constant" attr="executable" />
73 </xmitransformation>
74
75 <!-- Initial Node - Decision Node - Merge Node -->
76 <xmitransformation modelelement="node" xmipattern="
    TAG_OF_THESE_KINDS_OF_NODES" recurse="true">
77     <trigger name="type" type="" attr="ATTRIBUTE_OF_THE_TYPE" />
78     <trigger name="incomingEdges" type="" attr="
    ATTRIBUTE_OF_INCOMING_EDGES" />
79 </xmitransformation>
80
81 <!-- Final Node -->
82 <xmitransformation modelelement="node" xmipattern="TAG_OF_FINAL_NODE"
    recurse="true">

```

```

83     <trigger name="type" type="constant" attr="activityfinal" />
84 </xmitransformation>
85
86 <!-- Activity Diagram Control Flow -->
87 <xmitransformation modelelement="controlflow" xmipattern="
88     TAG_OF_CONTROL_FLOW" recurse="true">
89     <trigger name="type" type="constant" attr="controlflow"/>
90     <trigger name="source" type="" attr="" />
91     <trigger name="target" type="" attr="" />
92     <trigger name="probability" type="" attr="" />
93 </xmitransformation>
94
95 <!-- * SD Elements * -->
96 <!-- Sequence Diagram Lifeline/Component -->
97 <xmitransformation modelelement="lifeline" xmipattern="TAG_OF_LIFELINE
98     " recurse="true">
99     <trigger name="type" type="constant" attr="lifeline"/>
100     <trigger name="BCompRel" type="" attr="" />
101 </xmitransformation>
102
103 <!-- Sequence Diagram Asynchronous Message -->
104 <xmitransformation modelelement="asynchronousmessage" xmipattern="
105     TAG_OF_ASYNC_MESSAGE" recurse="true">
106     <trigger name="source" type="" src="" attr="" />
107     <trigger name="target" type="" src="" attr="" />
108 </xmitransformation>
109 </xmitransformations>

```