Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Transformação Automática de Modelos UML para Modelos Markovianos Parametrizados

Paula Aguiar de Vasconcelos Gueiros Bernardes

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

Orientador
Prof. Dr. Vander Ramos Alves

Brasília
2015

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

Coordenador: Prof. Dr. Homero Luiz Piccolo

Banca examinadora composta por:

Prof. Dr. Vander Ramos Alves (Orientador) — CIC/UnB
Prof.ª Dr.ª Genaína Nunes Rodrigues — CIC/UnB
Prof. Dr. Pierre-Yves Schobbens — Universidade de Namur, Bélgica

CIP — Catalogação Internacional na Publicação

Bernardes, Paula Aguiar de Vasconcelos Gueiros.

Transformação Automática de Modelos UML para Modelos Markovianos Parametrizados / Paula Aguiar de Vasconcelos Gueiros Bernardes.
Brasília : UnB, 2015.
107 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2015.

1. Linha de produto de software, 2. Modelo de Markov paramétrico,
3. Verificação de Modelo

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

Transformação Automática de Modelos UML para Modelos Markovianos Parametrizados

Paula Aguiar de Vasconcelos Gueiros Bernardes

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

Prof. Dr. Vander Ramos Alves (Orientador)
CIC/UnB

Prof.ª Dr.ª Genaína Nunes Rodrigues
CIC/UnB

Prof. Dr. Pierre-Yves Schobbens
Universidade de Namur, Bélgica

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

Brasília, 1 de junho de 2015

Dedicatória

Dedico aos meus pais.

Agradecimentos

Gostaria de agradecer a todos que, em maior ou menor grau, contribuíram para a minha formação e para a realização deste trabalho: familiares, amigos, professores, e colegas de graduação.

Agradeço a Deus, razão da minha vida, sem O qual nada disso seria possível. Em especial, agradeço aos meu pais, que me priorizaram acima de tudo e apoiaram todas as minhas decisões. Às minhas irmãs, que se mostraram minhas melhores amigas nos tempos difíceis. Ao meu namorado, Ricardo, pela melhor companhia, pela boa temperança, pelo amor e pela constante disposição em ajudar.

Agradeço ao meu orientador, Vander, pela dedicação e confiança no meu trabalho. Também agradeço à professora Genaína, ao André e ao Thiago, pelo apoio e por toda a paciência.

Agradeço ao Pedro por todos os anos de amizade e muita sinceridade; à Gabi, pela presença e pelo apoio nas horas mais difíceis; à Babi, pelo carinho contínuo; e à Fê, pela diversão garantida.

Resumo

Na construção de produtos de software, é essencial a utilização de técnicas de verificação capazes de medir atributos de qualidade, para garantir que determinados padrões sejam alcançados. Dentre essas técnicas, se destaca o *model checking*, que recebe como entrada um modelo do sistema sob avaliação e a propriedade a ser avaliada; e verifica, de forma automática, se essa propriedade é satisfeita pelo modelo. No entanto, de maneira geral, a criação dos modelos utilizados nessa verificação ainda é feita manualmente e exige uma análise cuidadosa dos modelos comportamentais do sistema. Além disso, à medida que os produtos de software crescem, ela se torna mais trabalhosa e propensa a erros. Tendo em vista esse contexto, o presente trabalho visa solucionar o problema descrito através da automatização da transformação de modelos UML para modelos Markovianos parametrizados passíveis de análise. Para validar esse trabalho, serão considerados os casos da *Beverage Machine* e da BSN-SPL.

Palavras-chave: Linha de produto de software, Modelo de Markov paramétrico, Verificação de Modelo

Abstract

When building software products, the use of verification techniques capable of measuring quality attributes is essential, in order to ensure that certain standards be met. Among these techniques, we emphasize model checking, which takes as input a model of the system under analysis and the property to be evaluated; and verifies, automatically, if this property is satisfied by the model. However, in general, models used in this type of verification are still built manually, which requires careful consideration of the system's behavioral models. Moreover, as software products grow, this task requires more effort and becomes more error prone. Given this context, this work aims to solve the described issue by automating the transformation of UML models to parameterized Markov models that can be analyzed. To validate this work, the Beverage Machine and BSN-SPL cases will be considered.

Keywords: Software product line, Parameterized Markov model, Model checking

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Na construção de produtos de software, ou conjuntos inter-relacionados desses produtos, é frequente o surgimento da necessidade de avaliar esses produtos com relação a atributos de qualidade, tais como funcionalidade, confiabilidade, usabilidade, eficiência, manutenabilidade, portabilidade, entre outros, que podem ser vistos como requisitos não-funcionais dos produtos [12]. A estratégia que se encarrega de realizar essas avaliações é chamada verificação de produtos. À medida que esses produtos crescem, no entanto, essa tarefa pode se tornar muito custosa do ponto de vista de tempo e pessoal especializado.

Com a necessidade de atender rapidamente a diferentes demandas dentro de um mesmo domínio, a Engenharia de Linhas de Produtos de Software [1] se destaca como técnica promissora, visto que ela permite a criação de vários produtos para atender a diversas necessidades de um domínio através da reutilização de artefatos da Linha de Produtos de Software (SPLs). A verificação de produtos é ainda mais importante em abordagens que criam diversos produtos que necessitam ser verificados, tal como ocorre em SPLs.

Diversas técnicas utilizadas atualmente buscam garantir o atendimento a níveis mínimos de qualidade em softwares existentes. Dentre elas se destaca o *model checking* [2]. Trata-se de um procedimento que recebe como entrada um modelo comportamental do sistema sob avaliação e a propriedade a ser avaliada; e verifica, de forma automática, se essa propriedade é satisfeita pelo modelo. Essa automatização traz benefícios imensuráveis, visto que elimina a necessidade da verificação manual de sistemas, o que pode ser inviável, considerando, além do tamanho dos produtos, a necessidade de trabalhar com SPLs.

Dentre os atributos de qualidade de um produto de software, a confiabilidade pode ser calculada como a probabilidade de se alcançar estados desejáveis em modelos probabilísticos, tais como Cadeias de Markov de Tempo Discreto (DTMCs) [9]. A utilização de cadeias desse tipo na verificação de produtos constitui a base utilizada ao longo de todo este trabalho.

1.1 Problema

De um modo geral, a criação dos modelos utilizados no *model checking*, no entanto, ainda é feita manualmente. Essa tarefa exige uma análise cuidadosa dos modelos comportamentais do sistema, como diagramas de atividades e diagramas de sequência; e, à

medida em que os produtos de software crescem, ela se torna mais trabalhosa e propensa a erros.

Existe a necessidade, portanto, da criação de um sistema que busque solucionar o problema descrito. Para isso, é necessária a definição clara de regras de transformação de modelos comportamentais em modelos formais passíveis de análise, de forma a possibilitar a automatização do processo de modelagem.

1.2 Solução

A proposta de solução constitui uma abordagem de transformação de modelos comportamentais UML em modelos Markovianos parametrizados, chamados FDTMCs, que são modelos DTMC com variabilidade embutida. Os modelos comportamentais utilizados devem ser diagramas de atividades e diagramas de sequência. Além disso, a transformação de diagramas UML para modelos formais deve ser escalável, i.e., deve permitir que as partes de um sistema sejam modeladas isoladamente, analisadas e compostas posteriormente para representar o comportamento do software como um todo.

Essa proposta considera o refinamento de atividades do diagrama de atividades em diagramas de sequência, como uma maneira de dividir o problema de modelagem em unidades menores. Unidades comportamentais definidas em diagramas de sequência também podem ser modeladas isoladamente, analisadas e posteriormente compostas, para permitir o aumento da escalabilidade da proposta.

1.3 Avaliação

O método de avaliação envolve a realização de experimentos controlados em laboratório utilizando software com variabilidade. Os dois casos retratados serão a BSN-SPL e a *Beverage Machine*. O primeiro retrata um sistema de sensores para o corpo humano, que capta sinais do corpo do paciente e detecta variações inesperadas, podendo se reconfigurar para atender à nova necessidade e tratar situações de risco. O segundo retrata o comportamento de um sistema que recebe um pedido do usuário e prepara e entrega a bebida requerida por ele.

Espera-se que a ferramenta proposta produza os mesmos resultados das tarefas anteriormente feitas de forma manual. Isso significa que, a partir dos diagramas comportamentais de cada sistema, os modelos Markovianos parametrizados produzidos devem ser os mesmos. Isso vale tanto para os modelos isolados quanto para o modelo geral construído para o sistema completo.

1.4 Estrutura do Trabalho

Os procedimentos deste trabalho são definidos da seguinte forma:

- Inicialmente, regras de transformação devem ser cuidadosamente estabelecidas para cada elemento que possa ser utilizado nos diagramas comportamentais. Essas regras devem mapear um elemento comportamental em uma estrutura em FDTMC. Isso é demonstrado no capítulo 3.

- Em seguida, são feitos o design e a implementação da ferramenta, demonstrados no capítulo 4. A ferramenta consiste em dois passos: *parsing* e transformação.
  - O primeiro passo consiste em percorrer um arquivo de entrada que descreve textualmente todos os diagramas comportamentais, seus elementos e o relacionamento entre eles. Essas informações são armazenadas em estruturas de dados para posteriormente serem manipuladas. Cada elemento de um diagrama deve ser armazenado em uma ordem definida, de forma a garantir que as transformações não produzam resultados inesperados.
  - O segundo passo consiste em percorrer as estruturas armazenadas e aplicar as regras de transformação para cada elemento, produzindo um conjunto de modelos Markovianos inter-relacionados.

- Por fim, no capítulo 5, dois estudos de caso são considerados de forma a analisar os resultados obtidos: o caso da BSN e o caso da *Beverage Machine*. Questões e métricas são estabelecidas para possibilitar a análise de resultados.

Capítulo 2

Fundamentação Teórica

Para entender por completo este trabalho, é importante definir alguns conceitos que serão largamente utilizados nele, a maioria dos quais estão envolvidos no contexto da área de Engenharia de Software, em especial no que diz respeito a Reuso de Software e a Atributos de Qualidade.

2.1 Linhas de Produtos de Software

Em Engenharia de Software, o desenvolvimento de uma Linha de Produtos de Software (SPL) refere-se ao mecanismo de construção de grupos de aplicações que têm um conjunto comum de recursos entre eles, propósitos semelhantes, e que são construídos com um núcleo comum. Estes grupos interrelacionados de programas são classificados como uma Linha de Produtos de Software e esses sistemas, por vezes, precisam ser medidos em termos de variabilidade, ou seja, a capacidade do sistema de suportar mudanças na configuração para uso em um contexto particular.

Em uma SPL, uma *feature* pode ser vista como uma característica ou uma unidade de diferença de um produto de software. Portanto, diferentes configurações de *features* escolhidas a partir de um *feature model* previamente definido difere um produto do outro. Há cinco aspectos que devem ser assegurados em uma SPL a fim de obter produtos qualificados: disponibilidade, confiabilidade, segurança, integridade e manutenabilidade [17]. Garantir esses cinco aspectos no início do estágio de desenvolvimento de software pode ajudar a evitar problemas e danos posteriores. Uma solução encontrada para esse desafio era usar a verificação de modelos de forma a estimar as propriedades não-funcionais do sistema usando a documentação do software. No entanto, ao construir modelos de produtos de SPLs, é impraticável a construção de um modelo separado para cada configuração possível. Portanto, surge a ideia de criar um modelo geral que, conforme a escolha de parâmetros, tenha a capacidade de instanciar todos os produtos de uma SPL. Com o *model checking* paramétrico, usando *features* opcionais, é possível representar qualquer outro tipo de variabilidade.

Os elementos que definem uma SPL são descritos a seguir.

2.1.1 Feature Model

Na definição de SPLs, as possíveis configurações distintas são extraídas de um modelo essencialmente presente na documentação da SPL, o *feature model*. Esse modelo define e representa *features* variantes e comuns entre os produtos, bem como suas dependências, de forma hierárquica. Além de um diagrama, ele pode conter uma lista de *cross-tree constraints*, representando restrições que não puderam ser exibidas no diagrama. Cada programa em uma SPL é identificado por uma combinação válida e única de *features*.

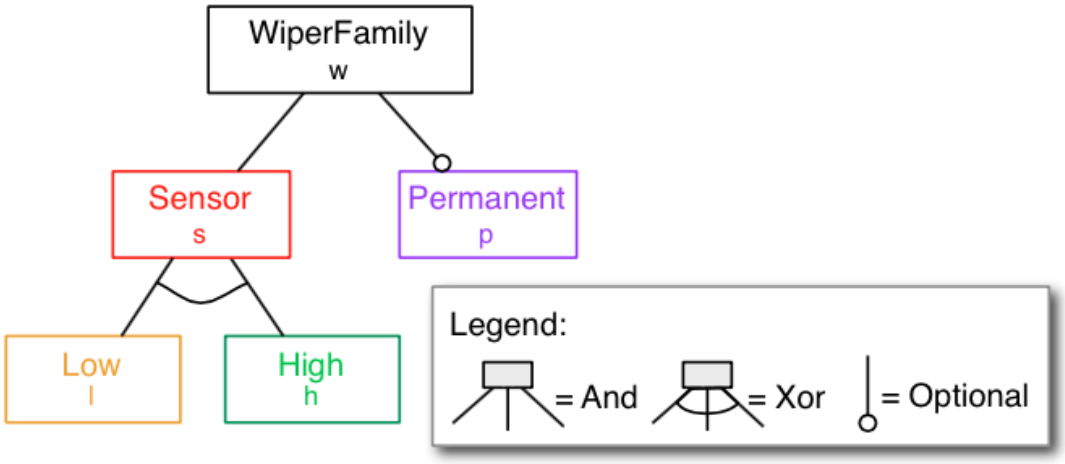

Figura 2.1: *Feature Model* para o *Windscreen Wiper Controller* [6]

A figura 2.1 representa o *feature model* do *Windscreen Wiper Controller*, do trabalho [6]. As seguintes regras estabelecem a relação existente entre as *features* e são utilizadas para determinar se um produto de software é bem formado ou não. Essas regras são chamadas regras de boa formação.

- Relações representadas por uma linha contínua com um círculo preenchido no final, ou simplesmente sem círculo, indicam a presença de uma *feature* obrigatória, i.e., sempre que o pai estiver presente, o filho também deve estar. Nesse caso, sempre que *WiperFamily* estiver presente, *Sensor* também deve estar. Portanto, a configuração mínima de cada produto dessa SPL é *{WiperFamily, Sensor}*;
- Relações representadas por uma linha contínua com um círculo não preenchido no final indicam a presença de uma *feature* opcional, i.e., sempre que o pai estiver presente, essa feature pode ou não estar presente. No nosso exemplo, sempre que *WiperFamily* estiver presente, *Permanent* pode ou não estar presente;
- Relações representadas por linhas com um ângulo não preenchido entre elas indicam a presença de *features* alternativas, ou XOR. Isso significa que, sempre que o pai estiver presente, necessariamente uma e somente uma dentre elas deve ser selecionada. Para o caso retratado, sempre que *Sensor* estiver presente, somente uma e exatamente uma dentre *Low* e *High* deve estar presente.
- Relações representadas por linhas com um ângulo preenchido entre elas indicam a presença de *features* OR. Isso significa que, sempre que o pai estiver presente, pelo

menos uma das *features* deve ser selecionada, mas não apresenta restrição quanto ao número máximo delas. No nosso exemplo, não existem relações OR.

- Relações representadas por linhas sem ângulo anotado indicam uma relação AND entre as features. Isso significa que, sempre que o pai estiver presente, todas as *features* devem ser selecionadas. No nosso exemplo, não existem relações AND.

2.1.2 Asset Base

O *Asset Base* de uma SPL constitui o conjunto de artefatos reutilizáveis que são usados na produção de um ou mais produtos de uma SPL. Isso inclui componentes de arquitetura, modelos de domínio, especificações de requisitos, documentos, casos de teste, classes, descrições de processo, bem como qualquer outro elemento útil na produção de software. Esses elementos são combinados de forma a produzir diferentes produtos.

Cada *asset* deve ter um processo associado a ele, que especifica como ele será utilizado no desenvolvimento de um produto. Esse processo também pode especificar o apoio a ferramentas automatizadas para realizar essas etapas. Restrições do produto, bem como a estratégia de produção, influenciam a especificação do processo. Os processos seguem uma abordagem de implementação chamada método de produção. Os processos anexados se tornam um plano de produção da SPL. A figura 2.2 ilustra esse processo.

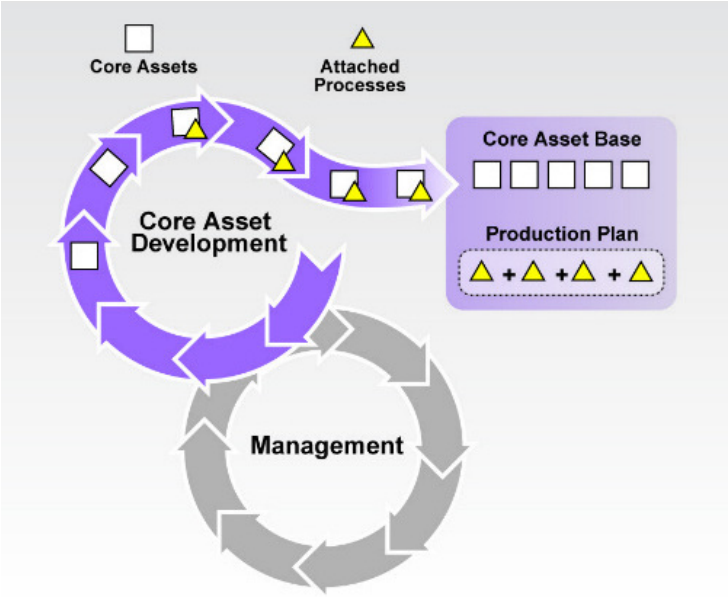

Figura 2.2: Processo de Desenvolvimento do *Core Asset Base* [1]

2.1.3 Configuration Knowledge

O *Configuration Knowledge* de uma SPL é o elemento responsável por associar, a cada *feature* do *feature model*, a combinação de *assets* que deve ser utilizada com ela. Dessa forma, extraindo do *feature model* uma configuração parcial válida, é possível identificar,

através do *Configuration Knowledge*, o conjunto de *assets* que deve ser combinado com essa configuração, de forma a obter um produto válido da SPL.

2.1.4 Relação entre os elementos de uma SPL

A construção de uma SPL em relação a um produto isolado não envolvido no contexto de uma SPL é mais custosa em estágios iniciais. No entanto, do ponto de vista de reuso de software, a utilização de SPLs se torna mais eficiente. Isso se dá pelo fato da possibilidade de reutilização de artefatos no desenvolvimento de novos produtos. O *feature model*, o *Asset Base* e o *Configuration Knowledge* se tornam úteis nesse processo.

2.2 UML Diagrams

The Unified Modeling Language was established in 1994 by Grady Booch, Ivar Jacobson, and James Rumbaugh. It is general-purpose visual language used to specify, visualize, construct, and document the artifacts of a software system [5]. It is used in modeling systems, and aims to establish design standards and modeling rules. In 1997, it was adopted as standard by the Object Management Group (OMG) and has been managed by it ever since. In 2005 it was approved as a standard by the International Organization for Standardization (ISO) as well.

UML diagrams offer various types of visions for the system. In the context of this work, however, we are interested in diagrams that represent the system behavior and state changes. Therefore, we work with activity and sequence diagrams. In this context, the general behavior of the system is described by a single activity diagram, wherein each activity, in turn, is described by one or more sequence diagrams.

2.2.1 Activity Diagrams

An activity diagram models organizational processes, displaying flows of activities or actions. It uses an activity graph, similar to a state machine, which represents the computational activities used in performing a calculation. This diagram consists of states, called activities, that represent operations or steps in an execution workflow. These activities may either be sequential or concurrent. This type of diagram, therefore, supports choice selection, iteration, and concurrency.

A set of shapes connected by arrows are used in the construction of diagrams of this kind. An initial node marks the start of a flow, a final nodes marks the end of a flow, and transitions, represented by the arrows, mark change of state. Some of the elements that make up an activity diagram are described below, and can be seen in figure 2.3 [5]:

- Transition:
  A transition describes the flow from a single element (usually an activity) to another. They do not need to be triggered by an explicit event. The completion of the source activity triggers its outgoing transitions. These elements are represented by arrows.
- Initial Node:
  An initial node represents the start of an execution workflow. An incoming transition

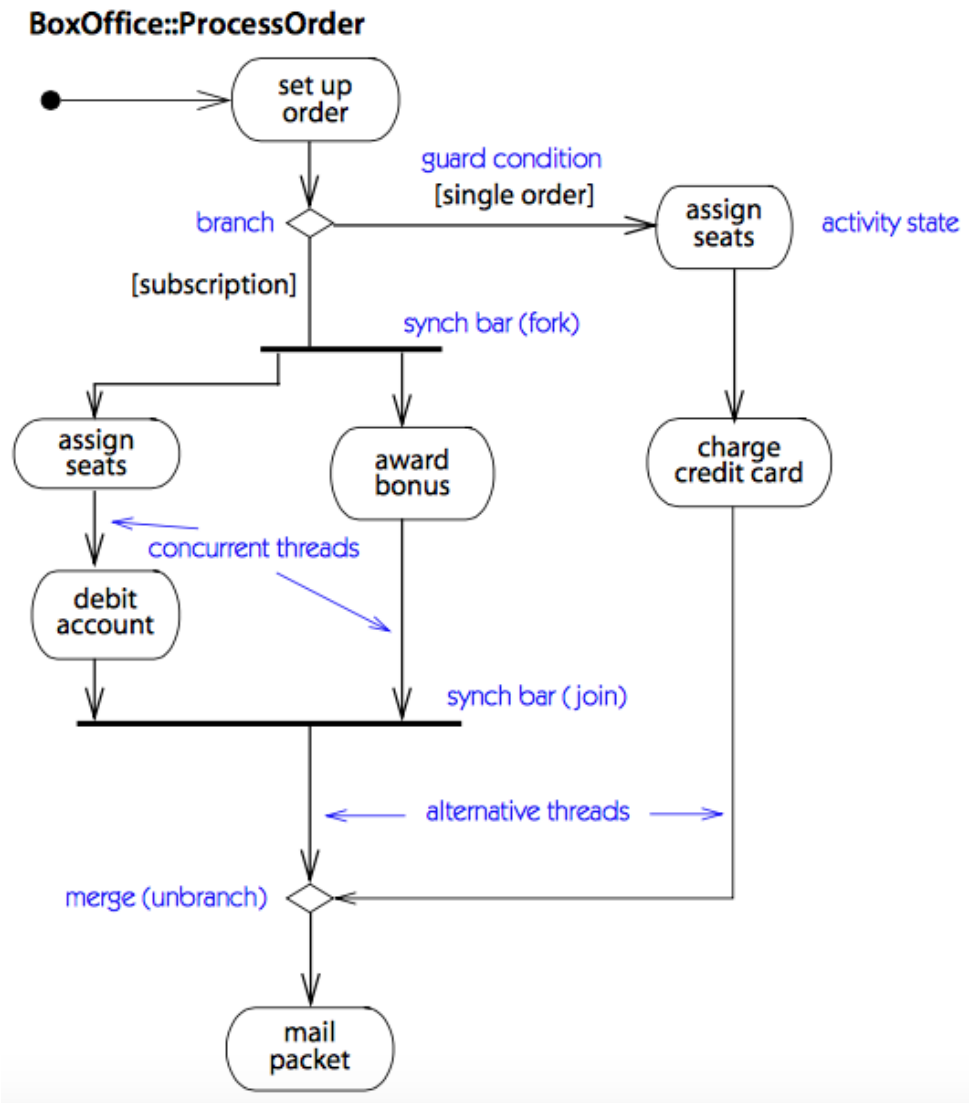

Figura 2.3: Activity Diagram Elements [5]

to a submachine causes this submachine's initial state to activate. An activity diagram may have more than one initial state, in which case multiple flows will start, one in each of these nodes, when the activity is invoked. Initial nodes are represented by a solid circle.

- Activity Node:
  An activity node represents the step in a workflow execution: the performance of an operation or a procedure. It can be terminated by an event that forces an output transition to be taken from that state. Therefore, the node does not need to terminate on its own. Activity nodes are represented by a box with rounded ends, which contains a description of the activity.

- Decision Node:
  A decision node is used for changing flow of control depending on conditionals in

activities. This type of node accepts one incoming transition and selects a single one of the outgoing transitions, depending on its guard condition. Decision nodes are represented by a diamond with multiple outgoing labeled arrows.

- Merge Node:
  A merge node is used to bring together various incoming transitions and produce a single outgoing transition. All incoming and outgoing flows of a merge node must be either object flows or control flows. Merge nodes are represented by a diamond with multiple incoming arrows and a single outgoing arrow.

- Final Node:
  A final node stops all flows in an activity and indicates activity completion. Final nodes are represented by a solid circle surrounded by a plain circle, notated as a bull's eye, or a target.

2.2.2 Sequence Diagrams

A sequence diagram represents an interaction, which is depicted as a two-dimensional chart, in which the vertical axis represents time flow, while the horizontal axis represents the involved participants as classifier roles, which may also be seen as objects.

The set of elements that make up a sequence diagram is described below and can be seen in figure 2.4 [5]:

- Lifeline
  Each role is represented by a vertical bar, called a lifeline. For the time an object exists, its role is shown as a dashed line. While the object is active, its role is shown as a double line. An object is active when a procedure is active on this object. This includes the time it waits for nested procedures to execute.

  Interactions are shown as messages from one lifeline to another. Messages sent in an earlier point in time are displayed above those sent in a later point in time.

- Synchronous Message
  A synchronous message is a message that forces the sending object to pause and wait for a response in order to continue processing. Synchronous messages are represented by an arrow with a solid line and a solid arrow head.

- Reply Message
  A reply message is sent in response to a previous synchronous message to the originator of this message. Reply messages are represented by an arrow with a dashed line.

- Asynchronous Message
  An asynchronous message is a message that does not force the sending object to pause and wait for a response in order to continue processing. Asynchronous messages are represented by an arrow with a solid line and an open arrow head.

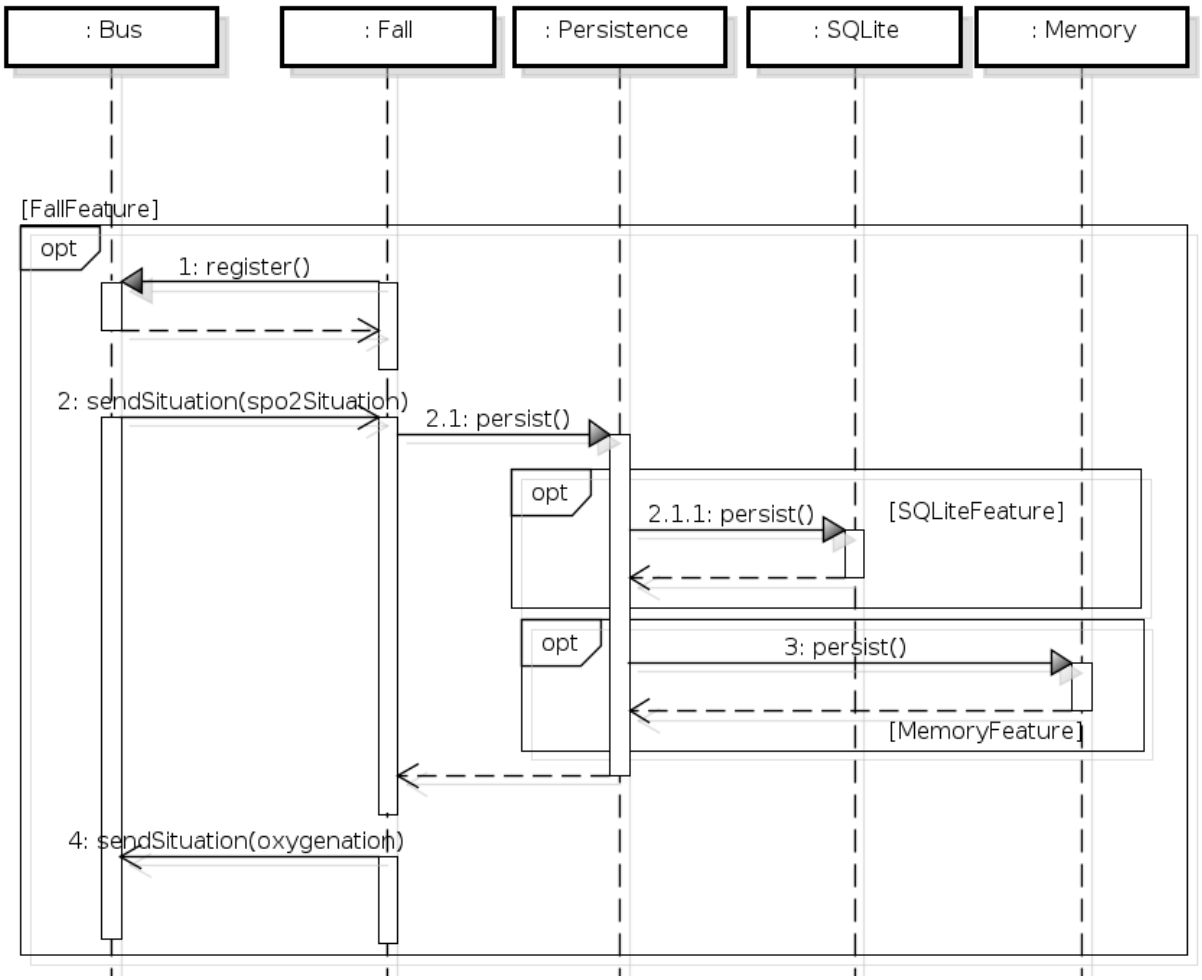

Figura 2.4: Sequence Diagram Elements [5]

- Combined Fragment
  A combined fragment is a structure used to create logical groupings and represent them in a compact manner. Its represented by interaction operands, called condition guards, and an interaction operator, which will specify the type of logic or conditional statement that describes the behavior of the combined fragment. For the purposes of this work, we'll consider combined fragments of type *optional*. Optional fragments describe behavior that may or may not occur.

2.3 Model Checking

Sistemas de Informação e Comunicação têm desempenhado tarefas cada vez mais essenciais nas atividades do dia-a-dia. Por isso, altos níveis de qualidade têm sido constantemente exigidos pela sociedade. Pequenos erros em aplicações podem ter consequências dramáticas, e, portanto, os padrões de confiabilidade demandados têm sido cada vez maiores, com o intuito de poupar perdas, seja do ponto de vista econômico, como do ponto de vista de segurança. Em 1995, um *bug* no processador Intel Pentium II, descoberto por

Thomas R. Nicely [16] causou um dano de centenas de milhões de dólares para a empresa multinacional. No ano seguinte, o foguete Ariane-5 caiu segundos após seu lançamento devido a uma conversão errônea de um número em ponto flutuante [15].

A maioria dos sistemas com os quais trabalhamos estão envolvidos em contextos maiores, o que os torna ainda mais vulneráveis a erros. Portanto, a verificação de sistema se faz necessária, tanto para componentes de hardware quando de software.

Para realizar uma verificação, primeiramente, é necessário descrever o comportamento esperado do sistema, i.e., definir uma especificação formal dele. Ela deve conter todas as funcionalidades do sistema, além de descrever quais são os erros esperados e como eles devem ser tratados. Portanto, a verificação será sempre relativa à especificação de um sistema [2].

De maneira geral, verificações de sistema podem ter duas naturezas distintas: de hardware ou de software.

Dentre as técnicas de verificação de Software, estão o *Peer reviewing* e o *Software testing* [2]. O primeiro envolve a participação de usuários especializados, os *peers*. Trata-se de uma análise estática do código puro, prévia à compilação; portanto, o código não é executado. Ao fim dela, cada *peer* fornece um feedback da análise, com uma declaração dos erros detectados. Em média, o *Peer reviewing* encontra em torno de 60% dos defeitos de software [3]. No entanto, por se tratar de uma análise estática, é compreensível que diversos erros não sejam detectados, tais como erros de segmentação, que são mais facilmente detectados em tempo de execução. O *Software testing*, por outro lado, faz a análise dinâmica do software, ou seja, efetivamente executa o software. Nessa verificação, uma série de entradas de teste é gerada. Esses testes são fornecidos ao programa, que percorre caminhos de execução coerentes com o conteúdo do teste. O problema desse tipo de verificação é que só é possível determinar a correção do software para os exemplos envolvidos em teste, e não de maneira global.

O *model checking* é uma técnica de verificação promissora que, dado um modelo de estados finitos de um sistema e uma certa propriedade, verifica de forma automática se essa propriedade é satisfeita pelo modelo. A análise do modelo é feita através da busca exaustiva pelo espaço de estados dele. Assim como em qualquer verificação, é necessário que haja, inicialmente, uma especificação formal do sistema em questão. A partir dessa especificação, portanto, o modelo do sistema é construído e as propriedades às quais o sistema deverá satisfazer são determinadas. Uma execução do *model checker* para determinada propriedade retorna um resultado positivo ou negativo, caso a propriedade seja satisfeita pelo modelo ou não, respectivamente. No segundo caso, é fornecida, também, uma indicação de como chegar no resultado inesperado. Essa estratégia é capaz de descobrir falhas antes mesmo da construção do sistema, pela análise do seu modelo.

O processo de verificação do model checking é retratado na figura 2.5.

Uma verificação só será tão eficiente quanto o seu modelo. Portanto, muito esforço deve ser atribuído à construção desse, de forma a assegurar que ele englobe todos os estados possíveis do sistema. Dependendo do tamanho do sistema a ser avaliado e do instrumento de avaliação, pode haver uma explosão do espaço de estados, situação na qual o tamanho do modelo supera o espaço disponível para armazená-lo.

Quando o verificador retorna uma situação de erro, esse deve ser investigado. De maneira geral, o erro pode ter sido causado por uma das seguintes falhas:

1. Uma falha na especificação do sistema;

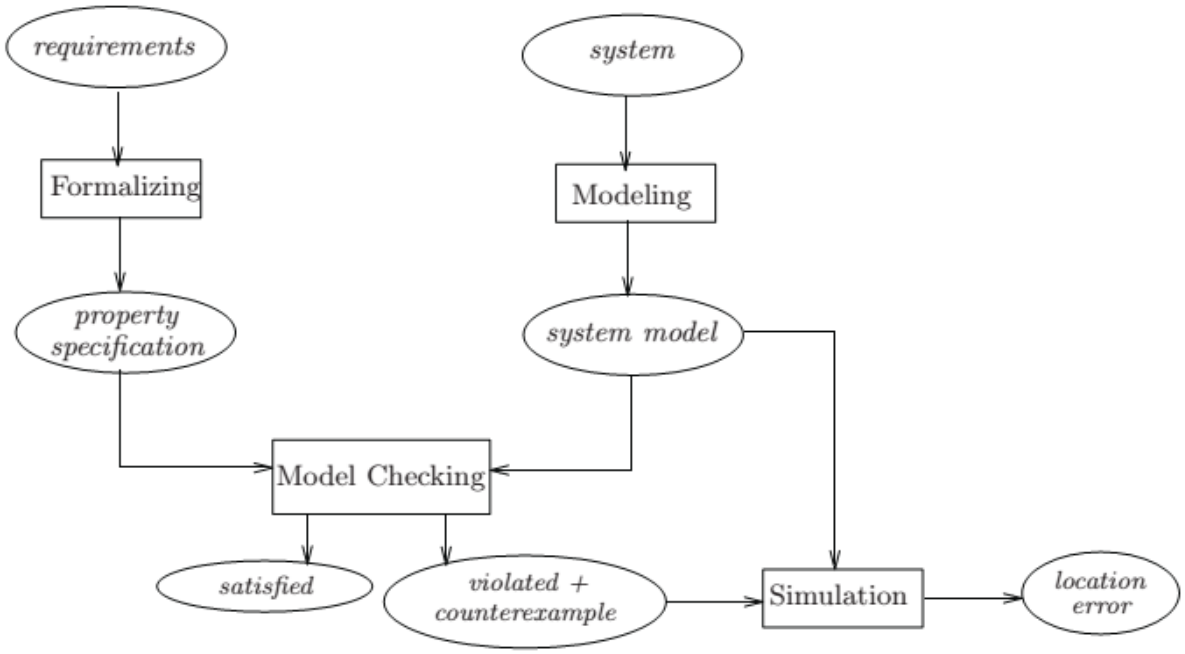

Figura 2.5: Processo de Verificação do Model Checking [2]

2. Uma falha no modelo;
3. Uma falha na definição da propriedade.

Na primeira situação, nota-se que o problema foi gerado por uma inconsistência entre a especificação e o que se desejava descrever. Nesse caso, dizemos que houve um erro de validação, i.e., a especificação formal do sistema não refletiu sua concepção informal. Na ocorrência de erros desse tipo, a especificação do sistema deve ser corrigida, o modelo deve ser modificado, e todas as propriedades anteriormente verificadas devem ser analisadas novamente.

Na segunda situação, temos a ocorrência de um outro tipo de erro de validação. Uma falha no modelo indica que, embora a especificação do sistema estivesse correta, a modelagem não conseguiu construir algo que refletisse, de fato, esse sistema. Nesse caso, um novo modelo deve ser construído e todas as propriedades anteriormente verificadas devem ser analisadas novamente.

Por fim, pode ser encontrada uma falha na definição da propriedade. Se a especificação está correta, assim como o modelo, a situação de erro na verificação indica que o sistema não satisfaz a propriedade definida. Portanto, essa propriedade não reflete a exigência do sistema e deve ser modificada. Como não houve modificação do modelo, as propriedades anteriores não devem ser modificadas, nem reverificadas. Essa situação constitui um erro de verificação.

Os modelos utilizados nesse processo são representados por autômatos de estados finitos, que consistem em um conjunto finito de estados e transições. Cada estado contém os valores das variáveis do sistema em determinado momento de sua execução; enquanto transições descrevem a evolução de um estado para outro. Mais especificamente, constumam ser utilizadas Cadeias de Markov, que podem ser sintetizadas como máquinas de

estados finitos em que a probabilidade de se chegar a um certo estado a partir de outro depende apenas do estado atual, e não de estados anteriores. Cadeias de Markov podem ser classificadas como de tempo contínuo (CTMC) ou de tempo discreto (DTMC). Essa última será largamente utilizada nesse trabalho. Nesse tipo de cadeia, as transições são tomadas com determinada probabilidade, e cada transição representa a decorrência de uma unidade de tempo. No caso de cadeias de tempo contínuo, transições são tomadas com determinada taxa. EM DTMCs, portanto, as probabilidades de todas as transições que partem de um estado devem ter soma igual a 1, como exemplificado a seguir.

2.3.1 Modelos Probabilísticos

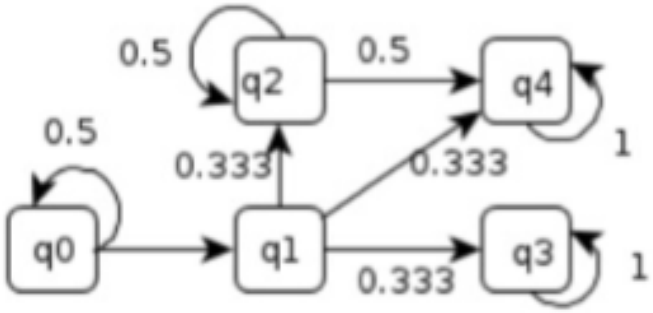

Figura 2.6: Exemplo de Cadeia de Markov [17]

O *model checking* probabilístico é usado em cenários que podem conter ocorrências de imprevistos. Ele utiliza um modelo previamente estabelecido e desenvolvido para verificar propriedades de dependabilidade, tais como confiabilidade, disponibilidade e integridade. Esse modelo contém transições anotadas com valores de probabilidade. O *model checking* "puro" utiliza um DTMC, tal como exemplificado na figura 2.6. A probabilidade de se chegar em q3 a partir de q1 é de 0.333. A partir de q4, a probabilidade de se chegar em q3, no entanto, é zero, pois, como se pode observar, não há caminho algum que tenha início em q3 e chegue em q4. A probabilidade de se chegar em q2 a partir de q0, por fim, é de 0.5 * 0.333 = 0.1665, pois o único caminho válido é $q0 \rightarrow q1 \rightarrow q2$.

O *Probabilistic Symbolic Model Checker* (PRISM) é uma ferramenta de verificação de modelos capaz de analisar automaticamente propriedades quantitativas em modelos probabilísticos, tais como Cadeias de Markov de Tempo Contínuo, DTMCs, Autômatos Probabilísticos, Autômatos Probabilísticos Temporários, entre outros [14]. Os modelos são descritos usando a linguagem específica do PRISM, que constitui uma linguagem baseada em estados, de simples utilização. Um exemplo de trecho simples de um arquivo em PRISM contém um tipo de cadeia, declaração de constantes e um ou mais módulos que contenham a descrição de um DTMC. O código abaixo descreve o DTMC da figura 2.6 em linguagem PRISM.

```
dtmc

module example

    q : [0..4] init 0;

    [] q=0 -> 0.5 : (q'=0) + 0.5 : (q'=2);
    [] q=1 -> 0.333 : (q'=2) + 0.333 : (q'=3) + 0.333 : (q'=4);
    [] q=2 -> 0.5 : (q'=2) + 0.5 : (q'=4);
    [] q=3 -> 1.0 : (q'=3);
    [optionalInterfaceName] q=4 -> 1.0 : (q'=4);

endmodule
```

O início de cada módulo contém a declaração de variáveis de estado. No nosso exemplo, nossa variável de estado q assume valores de 0 a 4. Em seguida, para cada estado, deve ser definido seu conjunto de transições, i.e., para cada transição, seu estado de destino e valor de probabilidade. No primeiro caso, temos $q = 0$ com probabilidade igual a 0.5 de tomar a primeira transição, mantendo o valor de $q = 0$; e (+) probabilidade igual a 0.5 de tomar a segunda transição, com q assumindo um novo valor q' igual a 2.

Opcionalmente, os estados podem ser associados a um rótulo ou *label* no PRISM. Isso pode ser proveitoso na utilização de linguagens de especificação de propriedade, muitas das quais são suportadas pela ferramenta, como é o caso da linguagem PCTL. Para encontrar a probabilidade de sucesso do sistema descrito, basta associar o rótulo "sucesso" a algum estado e calcular:

```
P = ? [(true) U ("success")]
```

Como se pode observar, a declaração do conjunto de transições do estado $q = 4$ foi associada ao rótulo de ação *optionalInterfaceName*. Esses rótulos podem ser usados para sincronizar transições em módulos diferentes. Se um módulo chega em uma transição com determinado rótulo que também existe em outro módulo, ele fica bloqueado aguardando a ocorrência da transição de mesmo rótulo no outro módulo, até que ambas sejam tomadas concomitantemente.

2.3.2 Modelos Paramétricos

Em SPLs, temos que lidar com conjuntos de produtos que têm propriedades semelhantes e distintas. Em modelos probabilísticos, os valores de probabilidade são apenas símbolos anteriormente à avaliação. É natural, portanto, estender a abordagem para o caso em que as probabilidades são dadas como parâmetros formais, o que torna possível a consideração de modelos paramétricos, permitindo que alguns valores de probabilidades não sejam especificados [7].

Cadeias de Markov são utilizadas para representar dois tipos de modelo: o modelo não parametrizado e o modelo parametrizado, que constitui em um modelo geral capaz de gerar cada modelo de configuração distinta, dependendo dos parâmetros selecionados. O primeiro é representado por um DTMC, enquanto o segundo utiliza um FDTMC (*Feature Discrete-Time Markov Chain*) para representá-lo, onde cada *feature* constitui um parâmetro no modelo.

Se inicialmente dispusermos dos modelos individuais representados na figura 2.7, é possível construir um modelo geral, parametrizado, como apresentado na figura 2.8, que

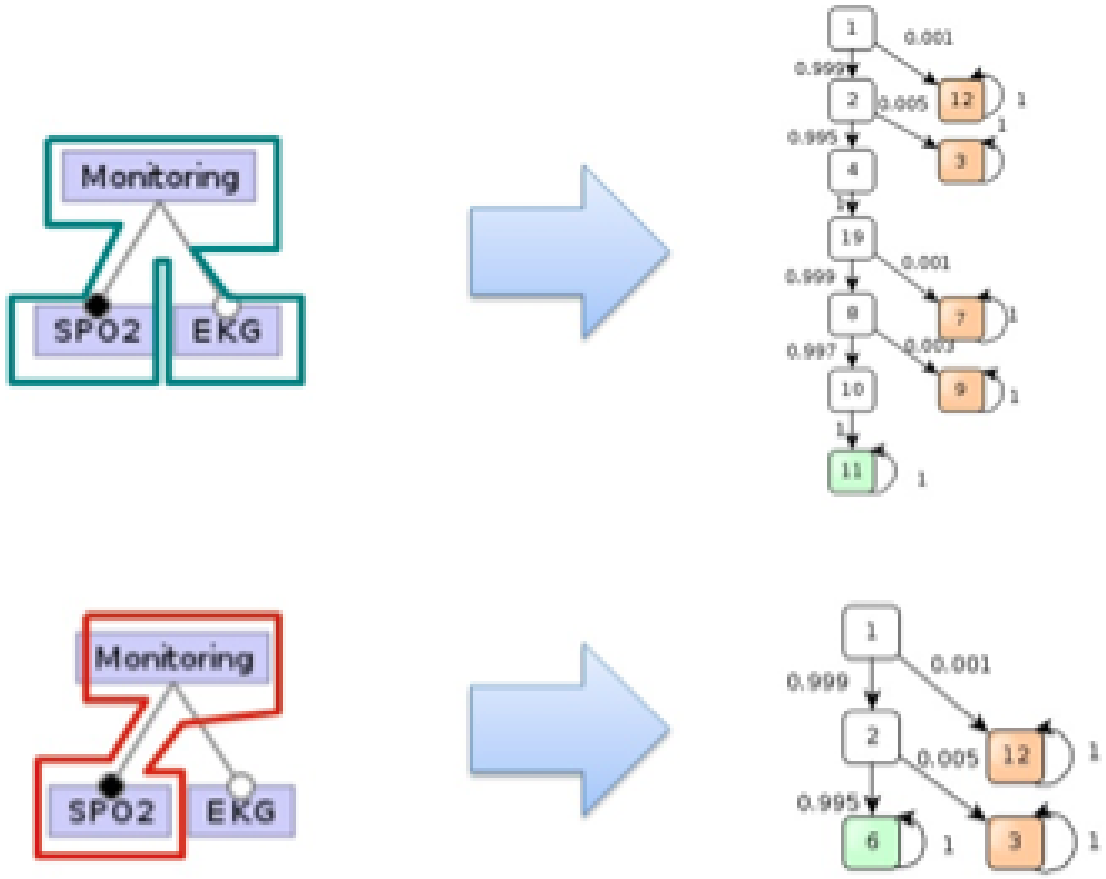

Figura 2.7: Exemplo de DTMC para o modelo não parametrizado [17]

abranja os modelos individuais; i.e., seja capaz de gerar cada modelo não parametrizado de configuração distinta, ilustrado na figura anterior. Nesse modelo geral, as transições podem depender de um certo parâmetro correspondente a uma *feature* do sistema, que podem ser avaliados em 0 ou 1. Quando ambas as *features* SPO2 e EKG estão presentes, fEKG e fSPO2 terão valores iguais a 1, caso em que a cadeia de Markov resultante para esta configuração excluirá o estado 6 e incluirá os estados 7, 8, 9, 10, 11 e 19. Por outro lado, quando EKG está ausente, fEKG corresponderá a 0, caso em que a cadeia de Markov resultante para esta configuração excluirá os estados 7, 8, 9, 10, 11 e 19 e incluirá o estado 6. A figura 2.7 ilustra esse comportamento. O valor da propriedade verificada será variável de acordo com a configuração selecionada. Ambas as figuras fazem referência ao *Body Sensors Network* (BSN) do trabalho [17].

Para realizar verificações de modelos parametrizados, o *Model Checker* PARAM surgiu como uma extensão do PRISM. Ele utiliza a mesma linguagem do PRISM e, além da verificação de modelos probabilísticos, permite o uso de parâmetros em rótulos de transições e no cálculo de fórmulas. No cálculo para avaliação de propriedades, ele retorna uma fórmula parametrizada. Nessa ferramenta, constantes ainda são suportadas, bem como o uso de expressões PCTL. O PARAM utiliza o mesmo processo do PRISM para sintetizar cadeias de Markov. Ele utiliza técnicas eficientes para manipular e representar polinô-

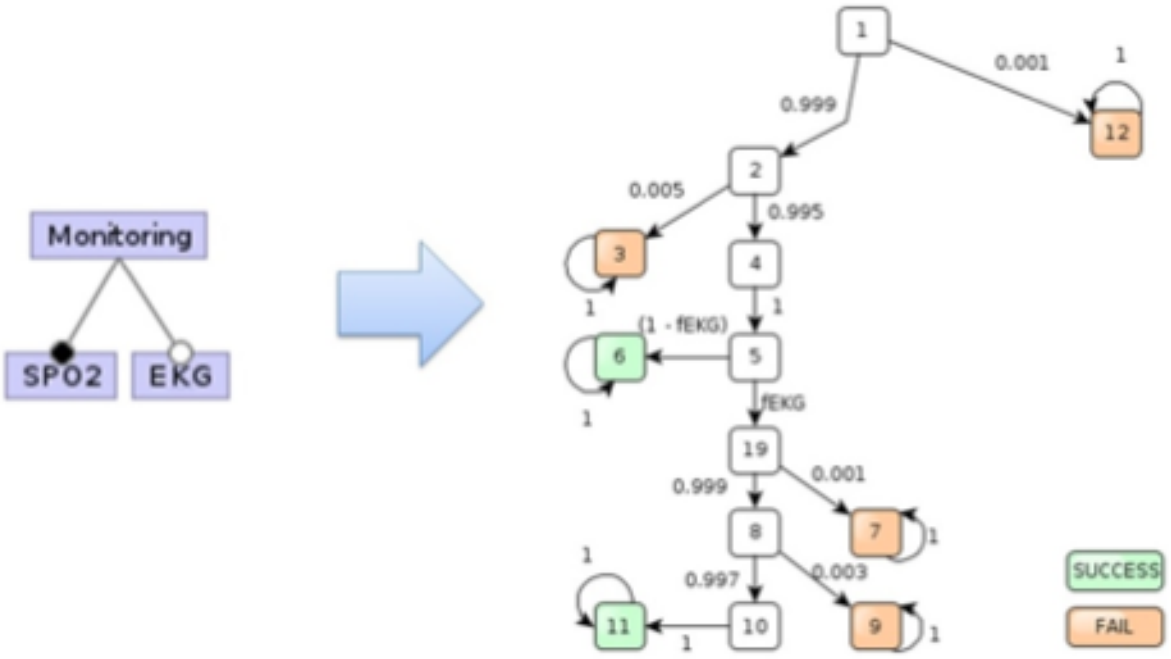

Figura 2.8: Exemplo de FDTMC para o modelo parametrizado [17]

mios, combinadas com técnicas dedicadas de *state-lumping*, baseadas em bissimulação. A análise básica do PARAM é o cálculo das probabilidades de alcançabilidade, mas ele também pode lidar com várias extensões dessa análise [13].

Essa ferramenta é especialmente útil quando existem características variáveis no modelo do sistema. Nesse caso, não é necessário modificá-lo para cada verificação de configuração distinta. O modelo é avaliado uma única vez e o valor de alcançabilidade resultante depende da avaliação das variáveis da fórmula resultante. Nesse trabalho, a principal propriedade verificada é a confiabilidade dos sistemas. Nas figuras 2.7 e 2.8, o PARAM poderia ser usado na verificação tanto dos modelos de cima quanto dos de baixo, pois suporta variabilidade, i.e., modelos com transições de probabilidade variável. O valor de confiabilidade de um produto, portanto, depende do conjunto de *features* selecionadas. O PRISM, no entanto, só poderia ser utilizado com modelos da primeira figura. Neste trabalho, consideraremos a utilização da ferramenta PARAM, visto que estaremos trabalhando com variabilidade constantemente.

Capítulo 3

Modeling

As previously seen, the PRISM and PARAM Model Checkers use the concept of Markov Chains to determine a system's reliability. Using the PRISM-specific language, the user builds a model for the system under analysis, which describes its behavior. This model is usually derived from UML behavioral diagrams, such as sequence and activity diagrams. Therefore, UML diagrams must become DTMCs or FDMTCs, which can later be written in PRISM language for model checking. An issue we face today is that this task is usually performed manually, which can be very expensive considering time spent and the need of skilled personnel. In fact, as systems grow, this task may become unfeasible. We'll exemplify with diagrams from the Body Sensors Network SPL, a system that will be introduced subsequently. This system's behavior is depicted by a single activity diagram, in which each action is further described by a sequence diagram.

Each UML element must be transformed to DTMCs and FDTMCs, which later must be written in PRISM language for model checking with PARAM, since we'll definitely be dealing with variability in these cases. In order to accomplish this, we must initially determine how each element in a behavioral diagram is mapped to a structure in a Markov Chain. We have been working with activity and sequence diagrams, each of which contain unique elements that must be taken into consideration. Therefore, transformation rules are defined so that they can be automated. The rules for sequence diagrams were previously defined in [9], and will hereafter be expressed as templates, in a more explicit manner. In addition, new rules are created for working with activity diagrams.

This work uses the activity-based approach. Therefore, the activities in the activity diagram and messages in the sequence diagram will be transformed sequentially into modules. Hereafter, we describe each pattern that will be used in the UML diagram and what they will become as a structure in Markov Chains.

Take the activity diagram of figure 2.3. The first activity, which we'll refer to simply by *Capture*, is described by a sequence diagram with six lifelines, four combined fragments, and 13 messages, in which each combined fragment represents the occurrence of a feature, i.e., a variability point. This activity alone becomes a single module in PARAM and each node in the sequence diagram becomes a state in the FDMTC. A node may be any source or target position of a message; or the beginning or end position of a combined fragment. In addition, each element type in a diagram becomes a determined structure in the FDTMC. This activity by itself is translated to a module of over 40 – manually written – lines of code in PARAM.

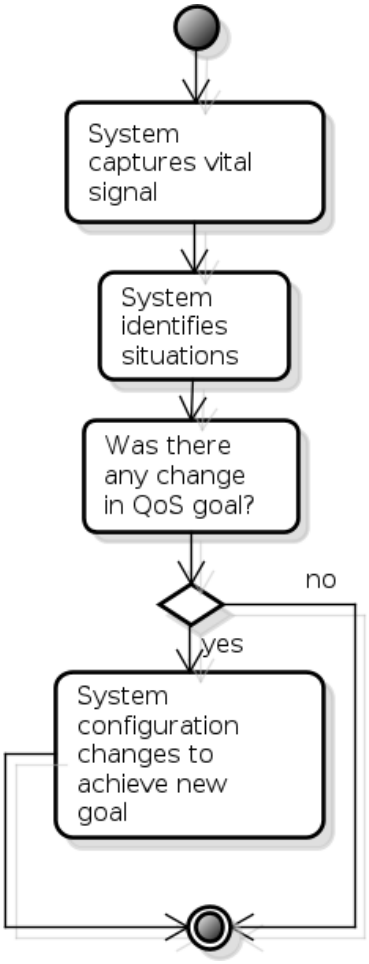

Figura 3.1: Activity Diagram for the Body Sensors Network [17]

Take the second action, which we'll simplify as *Situation*. This action is described by a considerably larger sequence diagram, with nine lifelines, 15 combined fragments and 50 messages. This activity also becomes a single module in PARAM. However, this module consists of 160 – manually written – lines of code in PARAM, a module four times larger than the previous one.

Notice that a relatively small system, whose behavior can be described by four activities and four sequence diagrams, becomes a PRISM file of over 200 lines of code. In addition, all diagrams must be carefully analyzed before being written into PRISM; i.e., it's not a straightforward process. Given that, imagine how costly the modeling of huge enterprise systems would be. On that account, the automation of this task has been proposed in the interest of solving this problem.

3.1 Process

The modeling process involves the accomplishment of three main activities, which are described in activity diagram of figure 3.2.

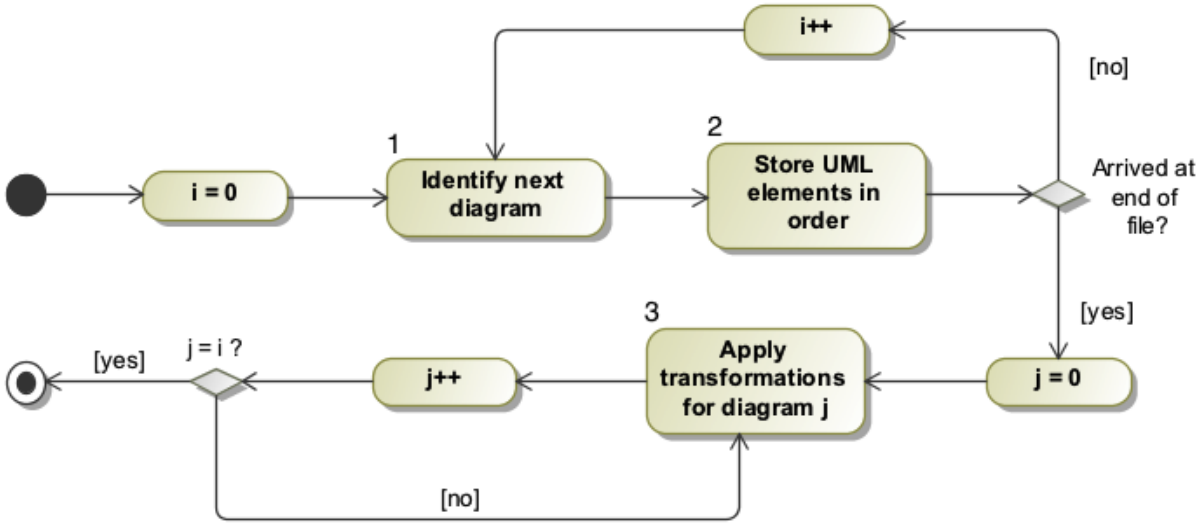

Figura 3.2: Activity Diagram for the Modeling Process

1. Identify all behavioral diagrams of the SPL.
   The identification of the diagrams comes from the parsing of an XML file exported from a UML design project, which contains all of the involved diagrams, including annotations concerning activity refinement in sequence diagrams.

2. For each behavioral diagram, identify the order in which the elements are arranged, and how they are interconnected.
   After the diagrams are detected, the elements must be identified, with respect to the order in which they are read. For sequence diagrams, this is more easily done, since the y axis represents time evolution, and, therefore, the order of occurrence of each element. For activity diagrams, however, the initial node must be detected, and the outgoing transitions covered. For each reached activity, its outgoing transitions must be covered again and so on, until all elements have been treated. In activity diagrams, nodes are connected by transitions, while in sequence diagrams, messages connect lifelines and combined fragments group messages with unique behavior.

3. Apply transformations for each identified element according to the established rules.
   Once the transformation rules have been defined for every UML element for activity and sequence diagrams, these rules must be applied in sequence for each occurring element in the diagram, which forms a single DTMC or FDTMC for every encountered diagram. These rules must be carefully defined, to prevent error in modeling.

The transformation rules that map each behavioral element to an FDTMC structure are described in detail below.

3.2 Templates

3.2.1 Activity Diagram Mapping

For the purpose of this work, the basic elements with which we'll work in an activity diagram are declared below. The left side of the figures show the UML activity diagram element, while the right side of the figures show what they will become in the FDTMC.

1. Initial Node
   An initial node becomes the start state in the FDTMC, as shown in figure 3.3.

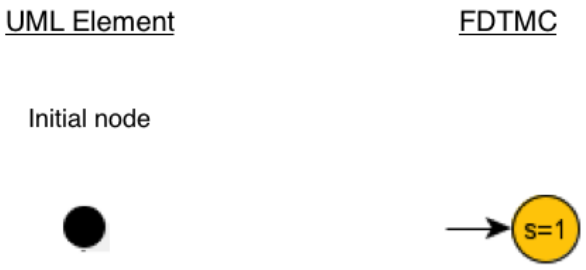

Figura 3.3: Activity Diagram Initial Node to FDTMC

2. Transition
   A transition from one activity to another becomes an FTDMC structure comprised of three states and two edges, as shown in figure 3.4. The first two states are regular, while the third one is an error state. The first edge in annotated with probability equal to *rAct1*, meaning "reliability of action 1" (source action). It flows from the first to the second state. The second edge in annotated with probability equal to $1 - rAct1$, which is the complement of *rAct1*. It flows from the first state to the error state. Note that the sum of these two probability values equals 1, maintaining the basic property of DTMCs. Therefore, the first state will have no other outgoing edges.

3. Decision Node
   A decision node becomes an FTDMC structure comprised of three regular states and two edges, as shown in figure 3.5. The first edge is annotated with probability equal to *[g1]*, the same parameter annotated in the first outgoing transition of the decision node. This edge flows from the first to the second state. The second edge in annotated with probability equal to *[g2]*, the same parameter annotated in the second outgoing transition of the decision node. It flows from the first state to the third state.

4. Merge Node
   A merge node becomes an FDTMC structure comprised of two regular states and a single edge, as shown in figure 3.6. A number of previous edges reach the first created state. An edge is created from the first to the second state with probability equal to 1.

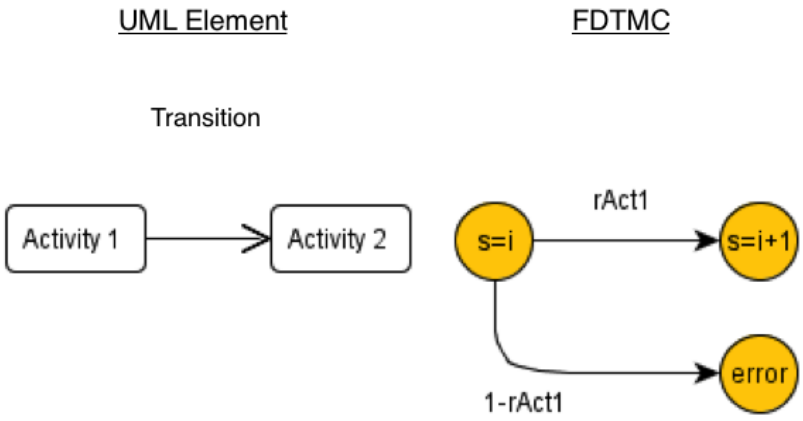

Figura 3.4: Activity Diagram Transition to FDTMC

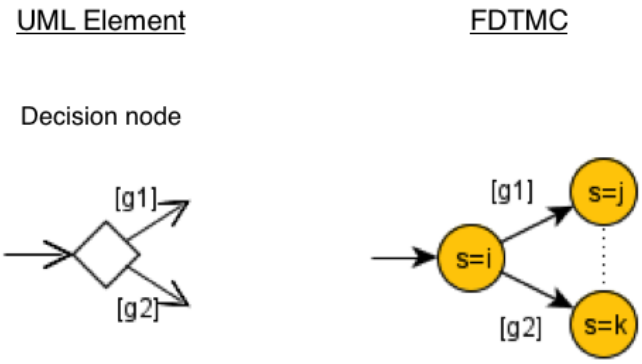

Figura 3.5: Activity Diagram Decision Node to FDTMC

5. Final Node
   A final node becomes a single state with a single edge (loop) in the FDTMC, as show in figure 3.7. This edge has probability equal do 1.

3.2.2 Sequence Diagram Mapping

Similarly, the basic elements with which we'll work in a sequence diagram are declared below. The left side of the figures show the UML sequence diagram element, while the right side of the figures show what they will become in the FDTMC.

1. Lifeline
   Each new interaction with a lifeline becomes a new state in the FDTMC, as show in figure 3.8. Therefore, disregarding the possibility of errors, if a message is sent from lifeline A to lifeline B, this will become an FDTMC comprised of two states: one

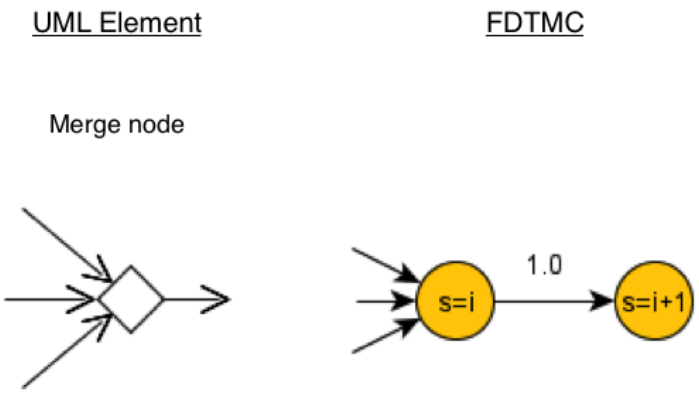

Figura 3.6: Activity Diagram Merge Node to FDTMC

Figura 3.7: Activity Diagram Final Node to FDTMC

for lifeline A, one for lifeline B; and a single edge. Since, in practice, errors must be considered, the following rules show how messages are actually translated.

2. Synchronous Message
   A synchronous message from a lifeline A to a lifeline B becomes an FDTMC comprised of three states and two edges, as shown in figure 3.9. The first two states are regular and consecutive, while the third one corresponds to the error state. **Note that the error state will be the same for all messages in the same sequence diagram.** The first edge is annotated with the message name followed by a probability value equal to rB, meaning "reliability of component B" (target lifeline). This edge flows from the first to the second state. The second edge is annotated with the message name followed by a probability value equal to $1 - rB$, which is the complement of rB. This edge flows from the first state to the error state. Note that the sum of the two probability values equals 1, maintaining the basic property of DTMCs. Therefore, the first state will have no other outgoing edges.

3. Asynchronous Message
   The precise definition of an asynchronous message is given in subsection

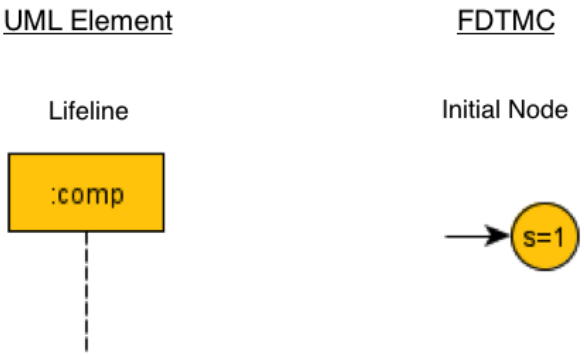

Figura 3.8: Sequence Diagram Lifeline to FDTMC

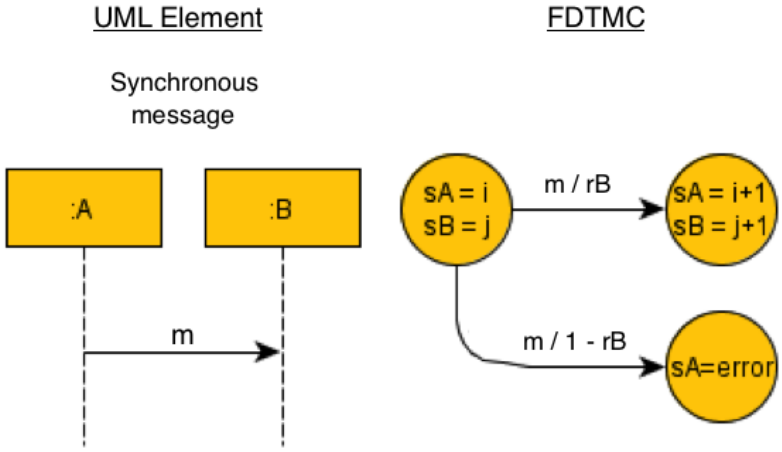

Figura 3.9: Sequence Diagram Synchronous Message to FDTMC

2.2.2. However, for the purpose of this work, asynchronous messages were used to represent actions that do not synchronize between components in the PRISM environment. An asynchronous message from a lifeline A to a lifeline B becomes an FDTMC comprised of three states and two edges, as shown in figure 3.10. The first two states are regular and consecutive, while the third one corresponds to the error state. The first edge is annotated with a probability value equal to rB. This edge flows from the first to the second state. **Note, however, that for asynchronous messages in a sequence diagram, the corresponding elements in the FDTMC are not annotated with the message name.** The second edge is annotated with a probability value equal to $1 - rB$, which is the complement of rB. This edge flows from the first state to the error state.

4. Reply Message
   A reply message from a lifeline B to a lifeline A becomes an FDTMC comprised of

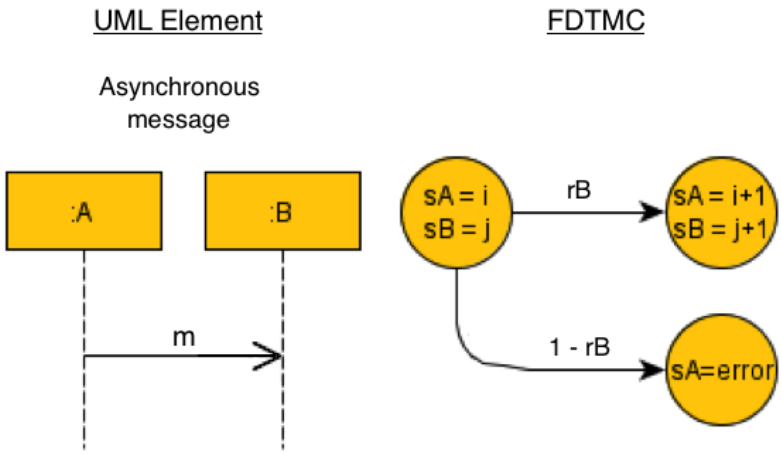

Figura 3.10: Sequence Diagram Asynchronous Message to FDTMC

three states and two edges, as shown in figure 3.11. The first two states are regular and consecutive, while the third one corresponds to the error state. The first edge is annotated with probability equal to rA, and it flows from the first to the second state. The second edge is annotated with probability equal to $1 - rA$, and it flows from the first state to the error state.

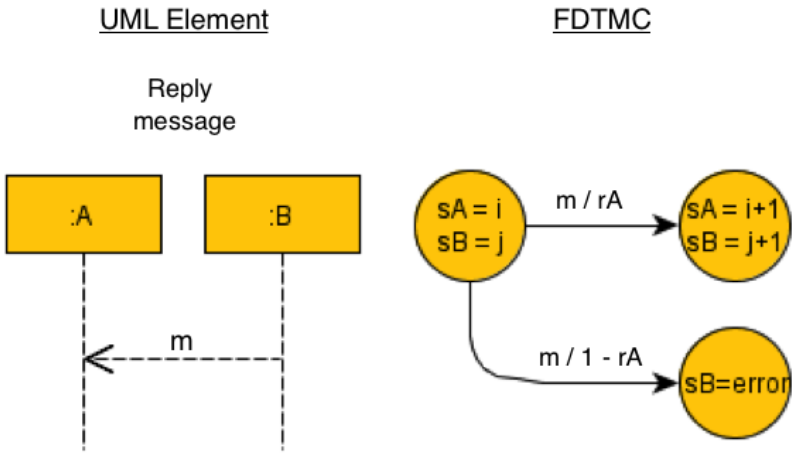

Figura 3.11: Sequence Diagram Reply Message to FDTMC

5. Combined Fragment: Optional
   A combined fragment of type optional indicates the presence of an optional feature. This is what adds variability to our models. With the occurrence of these types of fragments, the reliability of the system will depend on the combination of present features. When a feature is present, the optional fragment corresponding to its beha-

vior is taken into account in the computation of the system's reliability. Otherwise, this fragment is discarded. The occurrence of a combined fragment generates an FDTMC comprised of three regular states and two edges, as shown in figure 3.12. The first edge is annotated with probability equal to *feature*, which corresponds to the presence of the feature described in this fragment. It flows from the first to the second state. The second edge is annotated with probability equal to $1 - feature$, which corresponds to the absence of the feature. It flows from the first state to the third state. The content of the fragment is later transformed as a separate diagram, and becomes a single DTMC or FDTMC, using the same rules.

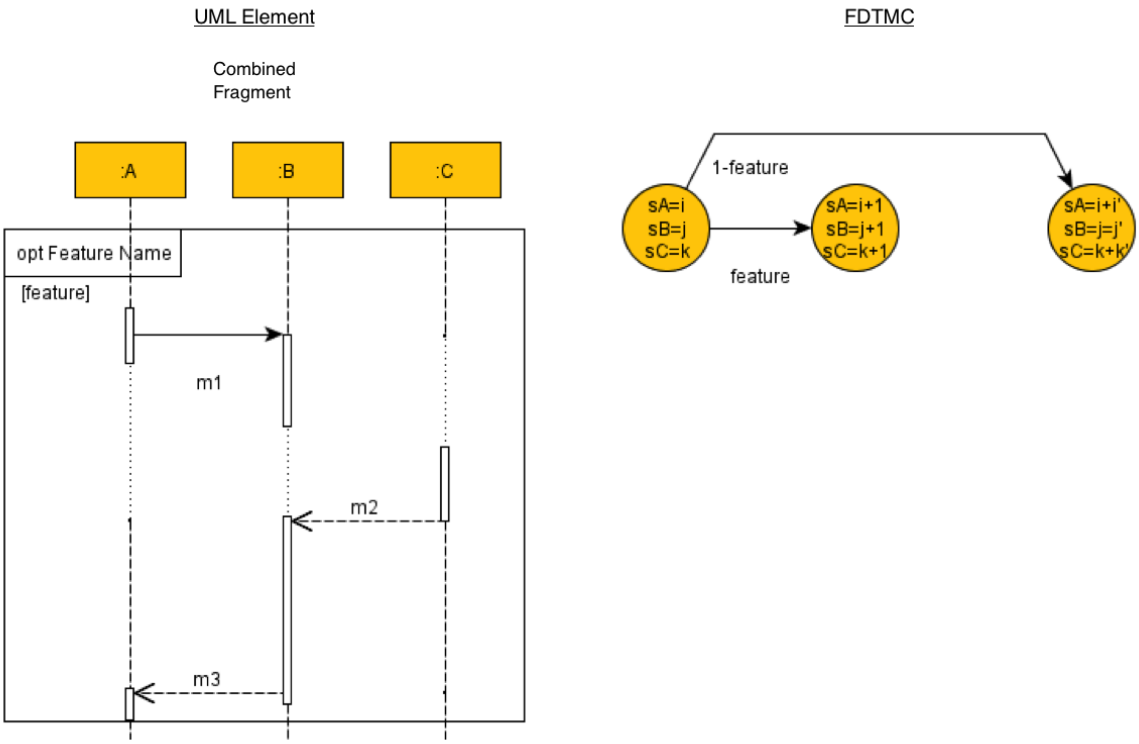

Figura 3.12: Sequence Diagram Optional Combined Fragment to FDTMC

Capítulo 4

Design and Implementation

The modeling algorithm consists of two main tasks: Parsing and Transformation. The first one involves traversing an XML file in search of the UML elements that make up the diagrams and the creation of UML objects in memory, while the second one involves putting into practice the transformation rules expressed in Chapter 3 for each of the identified elements, in order to build a set of FDTMCs. These tasks will be seen in detail further ahead in this chapter.

The algorithm was implemented in a MAC OS X Yosemite Version 10.10.2 environment, using Java ™ language [11] with compilance from execution environment JSE 1.7, in Eclipse IDE for Java Developers, Version Kepler Service Release 2. The *org.w3c.dom* API was used for XML parsing, and a number of structures were created for working with UML elements and FDTMCs. The U-MarmMo Tool [10] was used as a basis and part of the code was reused for performing sequence diagram transformations.

The application's architectural style was carefully chosen, in order to improve partitioning and allow future reuse. Architectural styles and patterns shape an application and determine the vocabulary of components and connectors that can be used with that style [18]. For the modeling tool, a combination of two architectural styles were used in the development of the algorithm: the component based and the object oriented styles. Hereafter, this is described more in detail.

4.1 Architecture

The component-based architectural style provides a high level of abstraction. It relies on the decomposition of the design into separate components with functional or logical relations, that expose well-defined communication interfaces containing methods and properties [18]. This style was chosen when taking into account that these components should be reusable, extensible, encapsulated and independent. Therefore, for our tool, classes that provide similar services and hold similar responsibilities were grouped into a single package, as shown in figure 4.1

The Modeling package is responsible for the overall operation and control of the modeling tool. It contains specific classes for initialization and relies on the remaining classes for parsing and transformation. The Parsing package contains specific classes for reading input data and building UML objects in memory. Control then returns to the Modeling package along with these objects. The Transformation package is responsible for

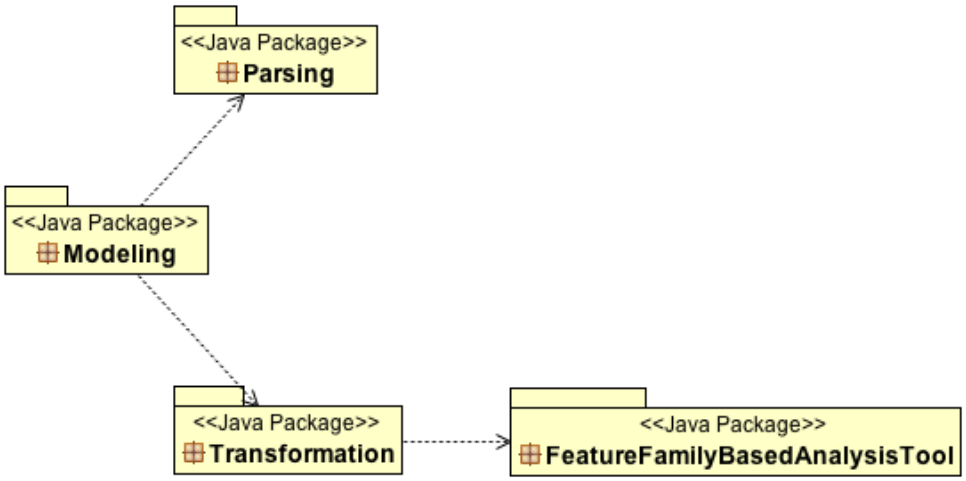

Figura 4.1: Component Based Architectural Style

transforming UML elements into Markov models. It relies on the FeatureFamilyBasedAnalysisTool package for the use of FDTMC objects.

The object-oriented architectural style encompasses the division of responsibilities for an application into individual, reusable, self-sufficient objects containing their own data and behavior. These objects cooperate with each other and are discrete, independent, and loosely coupled [18]. Communication is done through the use of interfaces or methods, by accessing object properties or by exchanging messages. This style was chosen when taking into account benefits such as reusability and cohesion.

Figure 4.2 combines both styles. For each component, represented by the packages, a series of objects were defined. Each object contains defined roles and responsibilities and, by the association of various objects, the modeling task is performed.

Some of the main elements in the modeling algorithm are described as follows.

- Class **DiagramAPI** is responsible for creating the parsing objects. For each diagram found in the input file, this class creates either an **ADReader** or an **SDReader** object, and calls the class methods responsible for parsing the correspondent diagram.
- Class **ADReader** is responsible for parsing an activity diagram from the input file. It contains a name, an identifier, and references a list of activities and a list of transitions. The last two are **ArrayList**s of type **Activity** and **Edge**, respectively. Since activity diagrams don't define a unique order for each element, parsing will consist of three main steps, in order:
  - Retrieve the activities;
    All activities must be stored in memory. At this point, the edges are not known. Therefore, incoming and outgoing edges cannot be linked to each activity yet.

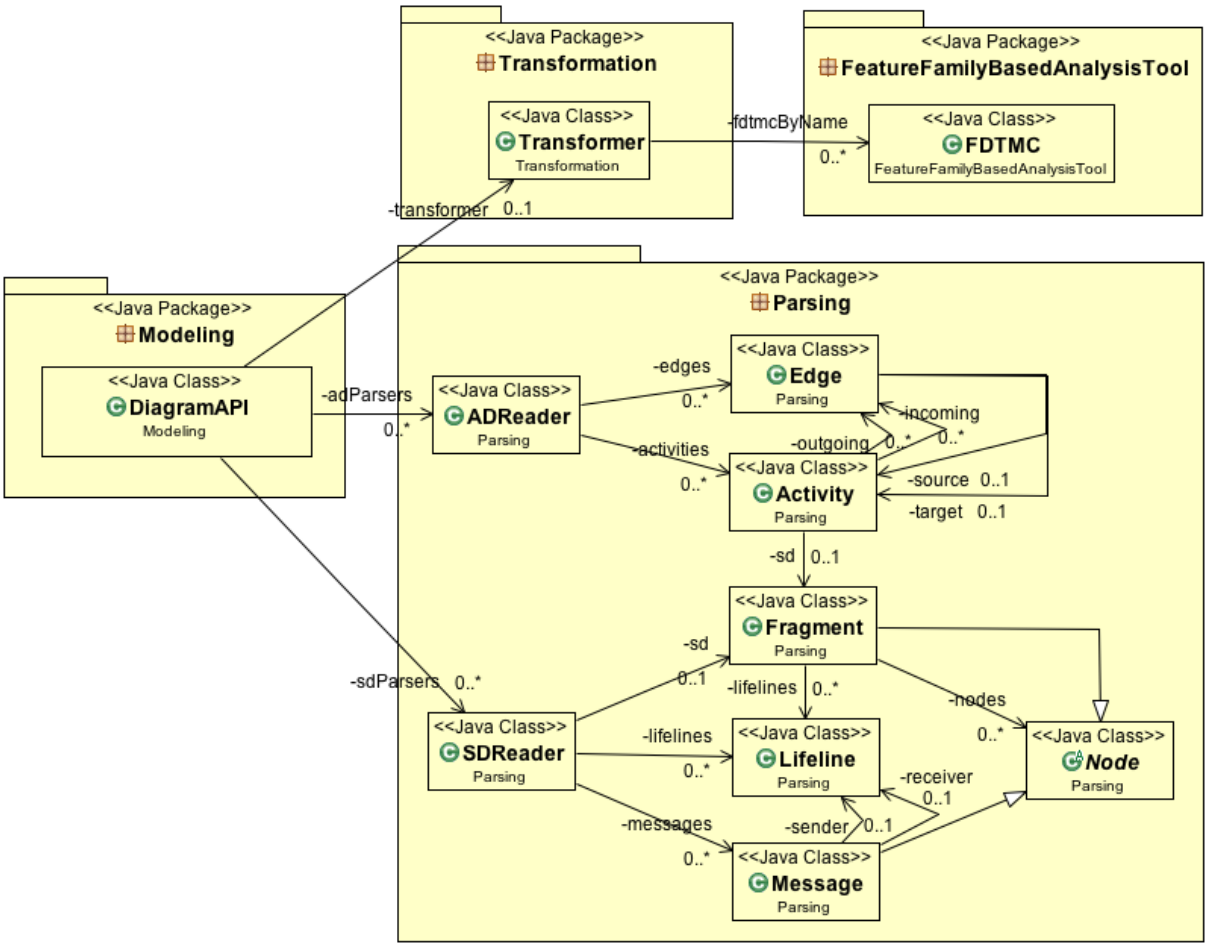

Figura 4.2: Component Based and Object Oriented Architectural Styles

  - Retrieve the edges;
    All edges must be stored in memory. Since the activities are already known, while the edges are being retrieved, the source and target activities of the edges can also be stored.
  - Finally, link the incoming and outgoing edges to each activity.
    This can be done at this point because the edges are known and may be detected through an identification number.

  This class also rearranges the order of the actions so that they can be read more intuitively. This means that the first node will be the initial node, followed by the first activity it leads to, and so on, for each flow in the activity diagram.

- Class **Activity** defines an object for the actions of an activity diagram. Each activity contains a name and an activity type. It references the sequence diagram that refines it, a list of edges that arrive at it and a list of edges that leave it. The activity types may be *InitialNode*, *ActivityFinalNode*, *CallBehaviorAction*, *DecisionNode*, *MergeNode*, *ForkNode* or *JoinNode*.

- Class **Edge** defines an object for the transitions that connect the activities of an activity diagram. Each edge contains a name, an identifier, an edge type and, possibly, a guard value. It also references the activity from which it leaves and the activity at which it arrives. The edge types may be *ControlFlow* or *ObjectFlow*.

- Class **SDReader** is responsible for parsing a sequence diagram from the input file. It contains an identifier, and references a list of lifelines and a list of messages. The last two are **ArrayLists** of type **Lifeline** and **Message**, respectively. Sequence diagrams may have combined fragments, that group messages with unique behavior. A combined fragment may be seen as a nested sequence diagram. Furthermore, a sequence diagram may be seen as a combined fragment, as well, since they have very similar behavior. Therefore, for each occurrence of either a sequence diagram or a combined fragment, an object of type **Fragment** is created. The **SDReader** references a single **Fragment**, corresponding to the actual sequence diagram being parsed. Combined fragments will be referenced by the **Fragment** within which they are nested. Sequence diagram parsing will consist of three main steps, in order:

  - Retrieve lifelines;
    All lifelines must be stored in memory. Since these object are actors which take turn in executing tasks, the order in which they are retrieved does not matter.
  - Retrieve messages;
    This includes all messages that will occur at any time in the sequence diagram, inside or outside combined fragments. At this point, the order in which elements occur in the diagram is still not known. Therefore, the elements are stored out of order. This decision was made in accordance with the input file's distribution of data.
  - Trace diagram.
    This step consists on referencing the previously stored objects in the order that they occur. The set of ordered elements will consist of messages and combined

fragments. When a combined fragment is found, it is referenced by its parent **Fragment**, but messages within that fragment will only be referenced by this fragment, and not by its parent. This task is accomplished through the use of a recursive method, which is called for each **Fragment** in the diagram.

- Class **Lifeline** defines an object for the actors involved in the sequence diagram. Messages will be exchanged between lifelines. Each lifeline contains a name and an identifier.
- Class **Node** defines an object for each element that appears in a defined order in the sequence diagram. Therefore, classes **Message** and **Fragment** will extend this class in accordance with additional elements that each one must define. Since we are working with the MARTE (Modeling and Analysis of Real-Time and Embedded Systems) profile, nodes may be annotated with important tags that describe further behavior or characteristics. Therefore, each node contains an identifier, and may contain values for probability, energy consumption and execution time, according to the annotated tags.
- Class **Message** extends class **Node** to define an object for the messages that are exchanged between lifelines. Each message contains a name and a message type. It references the lifeline corresponding to the actor who sends the message, and the lifeline corresponding to the actor who receives it. The message types may be *Synchronous*, *Asynchronous* or *Reply*.
- Class **Fragment** extends class **Node** to define an object with behavior similar to a combined fragment, i.e., for a sequence diagram or a combined fragment, since these two will be treated as the same kind of object. Each fragment contains a name, a fragment type, and may contain an operand name, which will occur for combined fragments, but not for sequence diagrams. It also references a list of lifelines and nodes. The lifelines correspond to the involved lifelines for the particular fragment, and the list of nodes will contain the set of messages and nested fragments in the order that they appear. This order will be necessary when applying the transformation rules.
- Class **Transformer** is responsible for applying transformation rules for each parsed element and creating a set of FDTMCs, which are mapped by name. Class **FDTMC** is called in order to create FDTMC structures: models, states and transitions.
- Class **FDTMC** defines data structures for FDTMCs, and is constantly called in the transformation process. For every new diagram and combined fragment, a new FDTMC is created. This class allows the creation of models, states annotated with labels and transitions annotated with probability and action values.

4.2 Parsing

Although a standard has been defined by OMG for representing UML diagrams through XML files, currently there are plenty of modeling tools, each of which extends this standard in accordance with their purposes. Therefore, we had to limit our work to the

use of a unique tool to generate these models. The tool that best met our needs in regard to modeling as well as in the production of an effective output was the MagicDraw tool, since it allows the extension of the XML representation in order to store information needed in our evaluation approach, such as reliability values. We used No Magic, Inc's MagicDraw tool, version 18.1 Enterprise Edition with MARTE profile 1.0.16.8. The tool exports diagrams to a UML 2.5 XML File, which is used as input for the Modeling algorithm.

MagicDraw supports the use of lifelines, synchronous messages, asynchronous messages, reply messages, and combined fragments of type optional, all of which are used to model our system. More specifically, the MARTE profile extension used with MagicDraw adds capabilities to UML for model-driven development of Real Time and Embedded Systems, providing support for specification, design, verification and validation. This profile allows us to specify essential constraints, named Tags, for each element in the UML model. The probability constraint was used for identifying values in sequence diagram messages, indicating the probability of the corresponding transition in a DTMC. On the other hand, the time and energy constraints are useful when working with CTMCs, for indicating elapsed time and energy consumption, respectively, in the occurrence of a sequence diagram message. Therefore, it is safe to say that the use of the MARTE profile is indispensable when working with the Modeling algorithm.

The MagicDraw tool, however, is a paid tool, and may not be very accessible. It does offer a trial version. However, the number of elements that can be used at a time is very limited, and the trial version only lasts for a limited number of days.

The MagicDraw output XML file consists of an EXtensible Markup Language file, which contains a written representation of all the involved behavioral diagrams. This includes all object metadata and interrelationships. Most of the file, however, contains application metadata that is never used.

In the activity diagram, elements are retrieved in the order in which they were modeled. This happens because, for activity diagrams, no unique order is specified for nodes, differently from what occurs with sequence diagrams. Therefore, these elements are rearranged in order to maintain an intuitive view. The most important details, however, are that the initial node be treated before all others, that an execution flow be treated in order, and that each transition be treated a single time when applying the transformation rules. Otherwise, it doesn't matter the order in which the elements are stored and flows are treated, as long as all properties are stored.

The parsing of an activity diagram comprises the use of important elements, which are described as follows. Figure 4.3 depicts the overall XML representation of an activity diagram.

- An activity diagram is indicated by the *packagedElement* tag along with the attribute *xmi:type* with value *uml:Activity*. The algorithm only works with a single activity diagram, which is fortunate, since the detection of an activity diagram in the XML file requires parsing and checking over one thousand tags with the same name. Using a large number of activity diagrams, therefore, has the potential of decreasing drastically the system's performance.

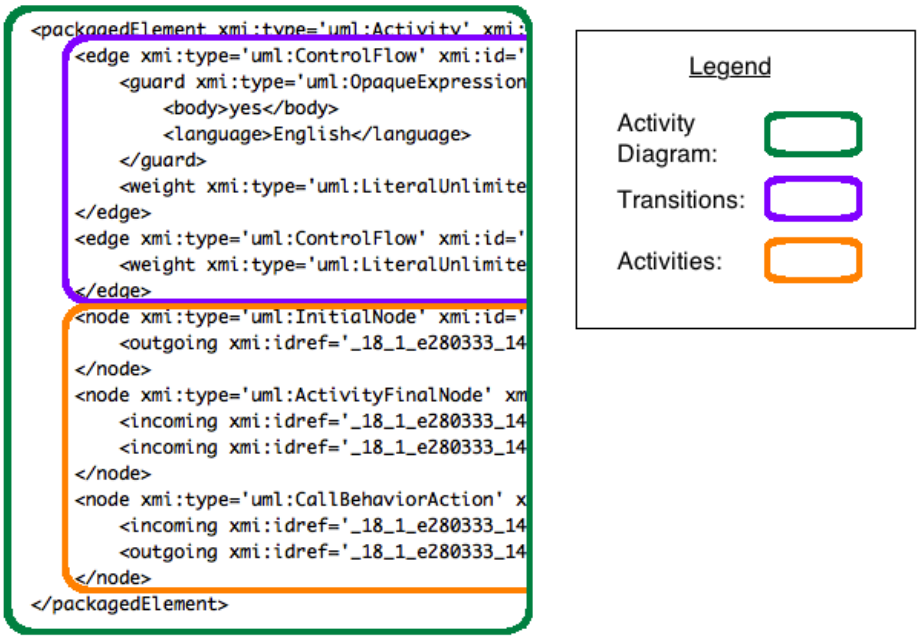

Figura 4.3: XML Representation of an Activity Diagram

- Child elements of *packagedElement* identified by the *edge* tag indicate a transition. All metadata for a transition is present as an attribute or as a child of this tag, such as name, identification, type, and guard value.
- Child elements of *packagedElement* identified by the *node* tag indicate any type of activity diagram node. All metadata for a node is present as an attribute or as a child of this tag, including the identification number of all incoming and outgoing edges. Furthermore, the *behavior* attribute of a node identifies the sequence diagram that refines the node's behavior.

The parsing of a sequence diagram is slightly different from that of the activity diagram. This happens due to the input file's distribution of data. What happens is that the identification of lifelines and messages, including some of their metadata, are presented on a section different from the one that lists the occurrence of each element in the order that they appear in the diagram. Therefore, a pre-analysis takes place, in order to identify these objects before they can be referenced, where the parser searches for the *lifeline* and *message* tags. During this step, some of the metadata of the objects are stored, but not their interrelationships.

The second step consists on parsing the section that presents these elements in order. This step is crucial, since the transformation rules will later be applied for these elements in this specific order. Other tags are used in this process and are described as follows. Figure 4.4 depicts part of the XML representation of a sequence diagram.

- The beginning of a sequence diagram is uniquely indicated by the *ownedBehavior* tag. The algorithm works with one sequence diagram for every activity in the activity diagram. The attributes of this tag indicate the diagram's type, identification and name.

Figura 4.4: Fragment of an XML Representation of a Sequence Diagram

- Child elements of *ownedBehavior* identified by the *ownedAttribute* tag indicate a lifeline. When this tag is encountered, its previously retrieved lifeline is referenced as a part of this diagram, and additional information concerning the lifeline is stored.
- Child elements of *ownedBehavior* identified by the *fragment* tag indicate not only the presence of a combined fragment, but also message occurrences. When this tag is encountered, either a message is referenced, or a new combined fragment is created. Note that the pre-analysis does not cover the presence of combined fragments. If a fragment is encountered, it must also go through the second step of parsing. In this case, elements are identified by different tags.
  - Child elements of a combined fragment identified by the *covered* tag indicate a lifeline that is covered by this fragment. When this tag is encountered, its previously retrieved lifeline is referenced as a part of this fragment.
  - Child elements of a combined fragment identified by the *operand* tag indicate the start of sequential behavior. Its child nodes will include message occurrences and other combined fragments identified by the *fragment* tag, which are equivalent to the *fragment* tags of a sequence diagram, applied to a combined fragment.
  - Child elements of a combined fragment identified by the *guard* tag indicate a guard condition for the combined fragment. All metadata for a guard condition is present as an attribute or as a child of this tag, such as type, identification, and value.

MARTE profile elements are identified outside these tags. For each message or fragment encountered, their identification number is used to search for MARTE profile elements, which are indicated by tags such as *GQAM:GaStep* and *GRM:ResourceUsage*.

4.3 Transformation

After parsing is done, the transformation rules defined in chapter 3 must be applied. For the activity diagram, this is done through the analysis of flows in the diagram. A recursive function is called a single time for each encountered transition, starting with the initial node. Each function call analyzes the involved elements, such as source node type, target node type and transition guard value; and applies the transformation rules accordingly. The probability of failure in activities is not considered in this diagram because, since each activity is further described by a sequence diagram, the failure case is considered for the sequence diagram, where all message errors are indicated by the same state. A single FDTMC is produced for the activity diagram.

For sequence diagrams, since a specific order of the elements is already defined, a second recursive function is called in this order, applying the transformation rules according to the encountered elements. This function is called once for each sequence diagram and for each unique combined fragment. Each call produces a new FDTMC.

The creation of a new FDTMC is done through the use of functions of class **FDTMC**. An interface with probability of success equal to f and probability of failure equal to $1-f$ is created using the following segment of code, represented by the structure in figure 4.5. Note that an edge may have an associated action, exemplified by *sendPackage*.

```
1  FDTMC fdtmc = new FDTMC();
2  State init, success, fail;
3
4  fdtmc.setVariableName("s");
5
6  init = fdtmc.createState("init");
7  success = fdtmc.createState("success");
8  fail = fdtmc.createState("fail");
9
10 fdtmc.createTransition(init, success, "sendMsg", "f");
11 fdtmc.createTransition(init, fail, "sendMsg", "1-f");
```

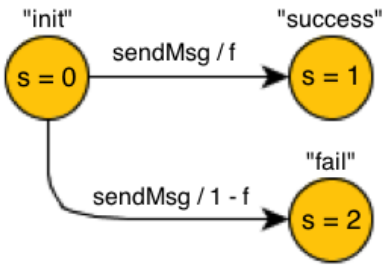

Figura 4.5: Resulting FDTMC from Code

Capítulo 5

Case Study

This work considers two case studies for analysis: the Beverage Machine and the Body Sensors Network (BSN), which will be seen in context as follows.

5.1 Context

5.1.1 Beverage Machine SPL

The Beverage Machine SPL consists on a simpler system than the BSN and is used to show the proposed work's feasibility. It was designed with the purpose of verifying non-functional properties of different configurations in an SPL, such as reliability and energy consumption, using probabilistic model checking tools and techniques. It aims to analyze the effects of applying parametric model checking instead of classic model checking [9]. The SPL produces various types of vending machines and its feature model is depicted in figure 5.1.

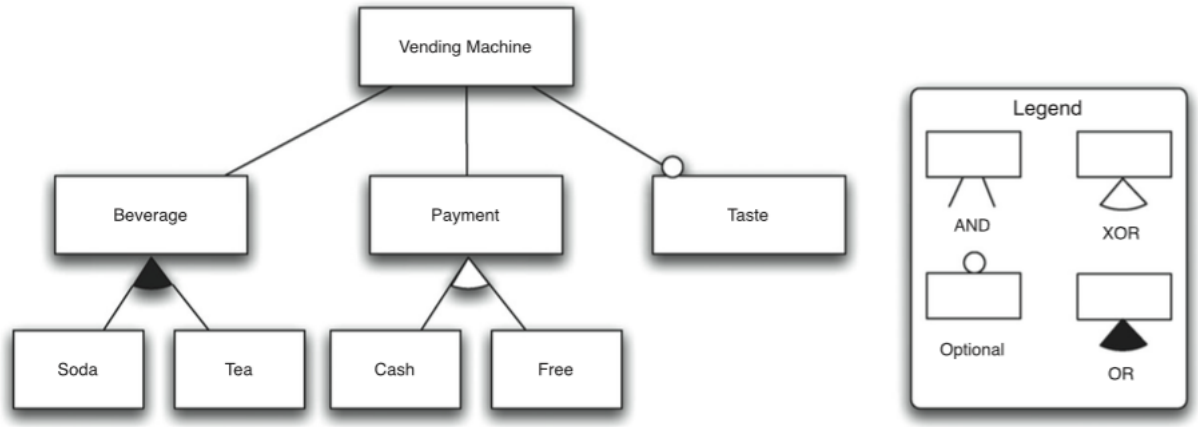

Figura 5.1: Beverage Machine Feature Model [9]

The system allows a user to pay for a beverage using coins, from which they may or may not receive change. Next, the user requests a type of beverage, which is prepared by the system and then served to the user. As can be seen from the feature model, a machine

may offer soda, tea or both; will either accept cash or offer a free beverage; and may or may not support adding taste to a beverage (for example, lemon to tea). Therefore, a beverage machine may support *tea*, *taste* and *free*, while another supports *soda* and *cash* only.

5.1.2 Body Sensors Network SPL

The Body Sensors Network is a more complex SPL, and consists of a system in which interconnected sensors communicate through a network and collect information that can then be analyzed in order to identify an individual's health situation. The wireless sensors capture vital signs of the human body and send data to a centralized system. The system analyzes this data in order to identify if the patient is in critical health condition. It can be useful for patients with chronic diseases, seniors, people going through intense medical treatment, or simply for anyone concerned about monitoring their health state [8].

The BSN is able to manipulate some of the data and even identify erroneous sensor capturing, if necessary. However, the system, by itself, cannot be responsible for evaluating all the information. The manipulation of data by specialized staff may and should be required. The feature model for this system is depicted in figure 5.2. Note that this model contains three semantically defined groups of features [19]:

- Sensor Features: represent sensors that may be present in a configuration. Each sensor is responsible for capturing particular types of data:
  - Pulse Oximeter (SPO2): measures the oxygen saturation in the blood;
  - Electrocardiogram (ECG): measures electrical potentials on the body surface, as well as electrical currents associated with heart muscle activity;
  - Temperature Sensor (TEMP): measures body temperature;
  - Accelerometer (ACC): measures the proper body tri-axial acceleration.
- Information Features: represent data captured by sensors;
- Storage Features: represent storage means for data read from sensors.

The activity diagram in figure 3.1 describes the BSN system process. Note that this behavior is repeated while the system is being executed. Initially, the system captures the vital signs using the body sensor. These sensors send data to a new sensor, the Control Sensor, which eliminates redundancies, optimizes the data, and sends it to the main system for analysis. Subsequently, the system analyzes the data in order to identify if the patient's health condition has been changed. This is important given the fact that body characteristics are not absolute. For instance, a blood pressure of 90/60 mmHg may be considered normal for an individual whose pressure is often around that; however, for chronically hypertensive patients, it could mean the person is going into shock. After identifying the patient's condition, the system must decide if there has been any change in the Quality of Service Goal. This involves the use of an adaptation manager, which evaluates whether the current configuration of the system is able to provide the services

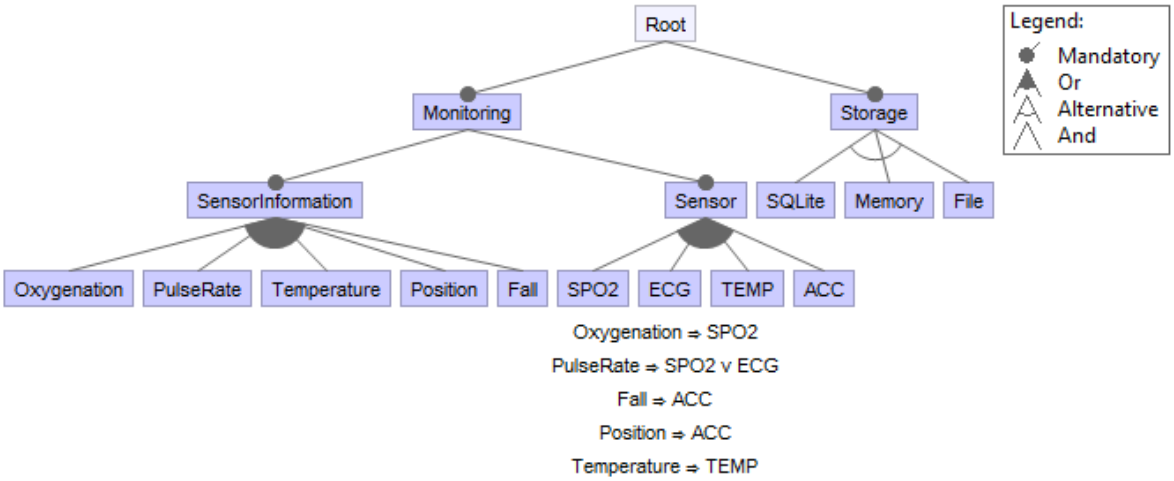

Figura 5.2: Body Sensors Network Feature Model [17]

needed for the current health status of the patient. If not, a reconfiguration takes place, in order to achieve the new goal. Otherwise, no reconfiguration is needed.

The BSN-SPL differs from the Beverage Machine SPL in the following aspects:

- It contains a more complex feature model, in which cross-tree constrains are used to represent restrictions that could not have been depicted in the tree structure;
- It allows feature redundancies, which means that the same feature may occur in different locations in the modeling process, but will only be modeled once. For other occurrences of the same feature, its model will be reused, thus eliminating the need to model the feature again.

5.2 Goals, Questions, Metrics

The Goal Question Metric approach consists of an approach to software metrics which defines a measurement model on three levels: conceptual, operational and quantitative [4]. A *goal* (conceptual level) is defined for an object, concerning quality, point of view and a particular environment. Then, a set of *questions* should be defined (operational level) in order to characterize the way a goal is going to be performed. These questions characterize the object of measure with respect to a quality issue. Finally, metrics (quantitative level) are associated with every question so that it can be measured in a quantitative way.

This model can be determined in the form of a table, as shown in table 5.1, which depicts the GQM determination for this work.

5.3 Design and Instrumentation

Specific methods were created with the purpose of measuring each described issue in terms of the defined metrics. The first method is used to measure *Q1* in terms of *M1*. It consists of associating, for each created model, an *integer* number indicating how many

Goal	Purpose	Reduce
	Issue	the cost of
	Object (process)	creating parametric Markov models for SPLs from behavioral diagrams
	Viewpoint	from the Software Engineer's point of view
Question	Q1	How much effort can be saved in modeling when reusing a previously built model that occurs in different locations in the modeling process?
Metric	M1	Number of times model is reused
Metric	M2	Size of model in number of states
Metric	M3	Size of model in number of transitions

Tabela 5.1: Determination of Goal Question Metric

times the model occurs in the modeling process. Therefore, the method is called for each sequence diagram as well as for every feature occurrence. Note that, when verifying that an occurring feature has been previously modeled, this method will be called, but the transformation process will be aborted for this feature, since its model already exists. At the end of the process, each constructed model is evaluated regarding this value. This method is depicted below.

```
1  public void countCallsModel (Fragment fragment) {
2
3     if (fragment.getOperandName() != null) { // consists of a fragment
4        if (fdtmcByName.get(fragment.getOperandName()) != null) { // model exists
5           nCallsByName.put(fragment.getOperandName(),
                nCallsByName.get(fragment.getOperandName()) + 1); // increment counter
6           return;
7        }
8        // model does not exist
9        nCallsByName.put(fragment.getOperandName(), 1); // set counter to 1
10    } else { // consists of a sequence diagram
11       if (fdtmcByName.get(fragment.getName()) != null) {
12          nCallsByName.put(fragment.getName(), nCallsByName.get(fragment.getName()) +
                1);
13          return;
14       }
15       nCallsByName.put(fragment.getName(), 1);
16    }
17 }
```

A separate transformation process occurs for every activity diagram, sequence diagram or combined fragment. For the last two, at the beginning of this process, method *countCallsModel* is called and given a fragment structure as input. This method checks for the fragment name and uses the **HashMap** *fdtmcByName* to retrieve its corresponding FDTMC, which will only exist if the current structure corresponds to a feature that has previously been modeled. At this point, a second **HashMap** is used, either to increment or to create a counter for the model, corresponding to the number of times it occurs in

the modeling process.

The second method is used to measure *Q1* in terms of *M2* and *M3*. Therefore, it consists of measuring the size of the model, in number of states and in number of transitions. This information is combined with the result from *Q1* using *M1* to evaluate the saved effort. This method is depicted below.

```
1  public void measureSizeModel (FDTMC fdtmc) {
2     Integer nStates, nTrans = 0;
3
4     nStates = fdtmc.getStates().size(); // captures number of states
5     Set<State> states = fdtmc.getTransitions().keySet(); // captures key set of map
6     Iterator <State> itStates = states.iterator();
7     while (itStates.hasNext()) { // iterates states
8        State temp = itStates.next();
9        nTrans += fdtmc.getTransitions().get(temp).size(); // increments transitions
10    }
11    /* store nStates, nTrans */
12 }
```

At the end of the modeling process, method *measureSizeModel* is called for each model occurring more than once in the process, and is given an FDTMC structure as input. The method captures the number of states in the model, which is retrieved by a simple method, and, for every state, increments to the total number of transitions the number of outgoing transitions of that state.

5.4 Results and Discussion

With respect to the Beverage Machine case, our work provides benefits such as being able to represent the overall behavior of the system through an activity diagram, in which each activity may be further refined in a sequence diagram. This is beneficial since it allows us to decompose the transformation problem into several smaller problems. In addition, system features may be represented by fragments, which allows a feature behavior to be defined independently form other features. Finally, it allows the reuse of Markov models for features with behavior that appears more than once in a system model. Since our question and metrics were defined on top of this last issue, they do not apply for the Beverage Machine, in which recurrent features do not occur.

Regarding the BSN-SPL, it consists of an SPL containing twelve features that are modeled in sequence diagrams. These features may be seen in figure 5.2, and correspond to the leaves of the feature model. Of these twelve features, we observed that three of them occurred more than once in the modeling process: *SQLite*, *Memory* and *File*. Table 5.2 shows the results of *Q1* in terms of *M1*, *M2* and *M3*.

Coincidentally, for a single feature, its model is reused four times, contains four states and four transitions. In addition, since all three features relate to storage options, they have very similar behavior. Therefore, it is not shocking that the results for each one of them are equivalent.

Take feature *SQLite*, for example. In the traditional approach, which doesn't allow the reuse of previously built models for recurrent features, *SQLite* is modeled four times more than in our approach, which allows feature redundancies. Each of these four additional

Feature	Q1 Results	
SQLite	M1	4 times
	M2	4 states
	M3	4 transitions
Memory	M1	4 times
	M2	4 states
	M3	4 transitions
File	M1	4 times
	M2	4 states
	M3	4 transitions

Tabela 5.2: Results of *Q1* in terms of *M1*, *M2* and *M3*

models, in turn, use an extra four states and four transitions. Therefore, considering *SQLite* alone, our approach uses 16 states and 16 transitions less than the traditional approach. Applying the same logic to all the recurrent features, we observe that our approach uses 48 states and 48 transitions less than the traditional approach. We may, therefore, conclude for *Q1* that our saved effort is of 96 elements consisting of states and transitions, for the BSN-SPL.

A combination of metrics *M1* and *M2* gives us the total effort saved, regarding number of states, in modeling a feature. Feature *SQLite*, for example, occurs five times in the process. Using the traditional approach, for each occurrence of the feature, four new states are created, resulting in a total of 20 states. By allowing feature redundancies, a single model is created, using four states. Therefore, a total of 16 states are saved for *SQLite*, using the new approach. This can be easily seen in figure 5.3, which compares the total effort, in number of states, for each recurring feature, considering the traditional and new approaches.

In the same manner, a combination of metrics *M1* and *M3* gives us the total effort saved, regarding number of transitions, in modeling a feature. Coincidentally, the number of states and transitions in these models are equivalent. Therefore, we observe the same results in figure 5.4. Using the traditional approach, for each occurrence of feature *SQLite*, for example, four new transitions are created, resulting in a total of 20 transitions. By allowing feature redundancies, a single model is created, using four transitions. Therefore, a total of 16 transitions are saved for *SQLite*, using the new approach.

Therefore, in the traditional approach, 120 elements consisting of states and transitions are created for the three recurring features, while, in the new approach, only 24 elements are created for the same features. Therefore, we may further conclude for *Q1* that our saved effort was equal to 80% for the features that occurred more than once. For features that appeared a single time, no effort was saved.

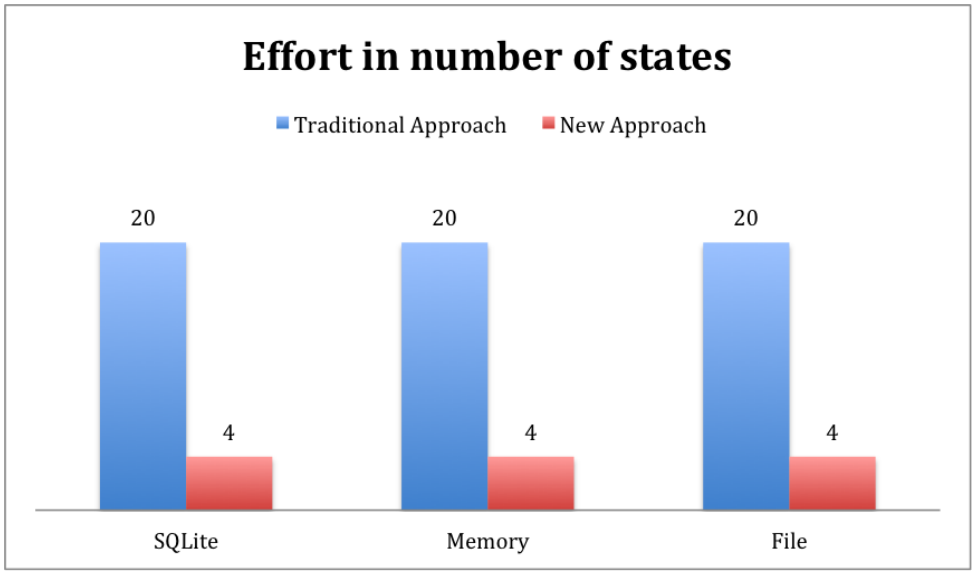

Figura 5.3: Comparison of the approaches regarding effort in number of states

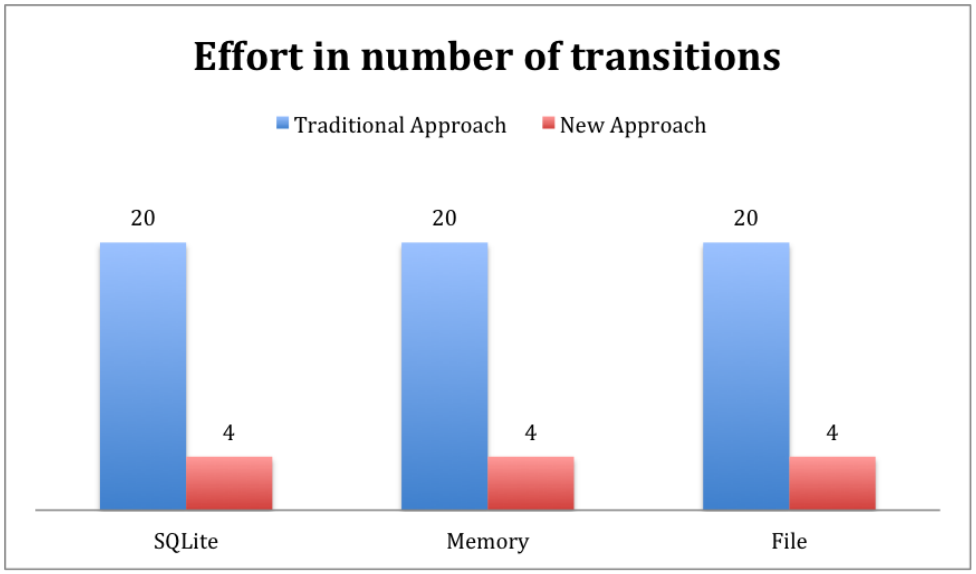

Figura 5.4: Comparison of the approaches regarding effort in number of transitions

Capítulo 6

Conclusion

As seen previously, this work consists on taking a set of UML behavioral models describing a system operation, and automatically transforming them to parameterized Markov models, for quality analysis. This is done by the Modeling algorithm, which requires a set of UML diagrams to be provided in XML format for initial parsing. The MagicDraw MARTE profile is used to annotate components with reliability values, essential for the transformation process.

The modeling algorithm consists on two main steps: Parsing and Transformation. Parsing consists of traversing an XML file and creating UML objects in memory, while Transformation consists of applying transformation rules for each object and creating a set of corresponding FDTMCs.

The model of system behavior is activity-based, since it considers refining activities of an activity diagram in sequence diagrams. It is also feature-based, since we consider that combined fragments in a sequence diagram represent system features, which make up our reference unit.

Previous work in the related field was considered when developing the modeling algorithm, and will be described as follows.

6.1 Related Work

Model-based verification of quantitative non-functional properties for software product lines [9] was introduced in 2013, and consists of creating a parameterized sequence diagram from a set of sequence diagrams representing the complete behavior of the SPL. This work uses an auxiliary structure that divides the parameterized sequence diagram into several other sequence diagrams. In addition, alternative behavior is modeled using a single parameter. Therefore, a sequence diagram fragment may represent several alternative features. Further, optional features are also represented by fragments. However, this work does not consider recurrent feature, i.e., features whose behavior appear more than once in the modeling process. This means that, for every occurrence of a feature, a new transformation to Markov model is performed.

Our work, when compared to the previous, presents certain differences. Initially, it considers activity refinement, i.e., further describing an activity of an activity diagram in a sequence diagram. This allows us to break down the problem of transforming models into smaller problems, since it consists of transforming activity diagrams into Markov models

that represent the way the activities are executed by the SPL. In addition, our model of system behavior is feature-based: for every system feature, there exists a specific fragment, which describes its behavior. This allows us to define a feature behavior independently of other features' behavior, which was not possible in the previous work. Further, our work allows the reuse of Markov models. Once a feature's model has been created, it can be reutilized in all parts of the UML model in which it appears.

6.2 Future Work

For future work, the following topics have been considered.

- Firstly, the modeling algorithm may be extended to work with component-based modeling, as well as the current activity and feature-based modeling.
- Secondly, it may be extended to perform Continuous Time Model Checking (CTMC) for evaluation of other SPL properties, such as performance and power consumption.

Referências

[1] *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. vii, 1, 6

[2] C. Baier and J. P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. vii, 1, 11, 12

[3] R. Baker and I. Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *Software Engineering, IEEE Transactions on*, 39(6):787–805, June 2013. 11

[4] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994. 37

[5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005. vii, 7, 8, 9, 10

[6] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 321–330, New York, NY, USA, 2011. ACM. vii, 5

[7] Conrado Daws. Symbolic and parametric model checking of discrete-time markov chains. In *Proceedings of the First International Conference on Theoretical Aspects of Computing*, ICTAC'04, pages 280–294, Berlin, Heidelberg, 2005. Springer-Verlag. 14

[8] P. Fernandes. Linha de produtos de software dinâmica direcionada por qualidade: o caso de redes de monitoração do corpo humano. 2012. 36

[9] Carlo Ghezzi and Amir Molzam Sharifloo. Model-based verification of quantitative non-functional properties for software product lines. *Information and Software Technology*, 55(3):508–524, 2013. vii, 1, 17, 35, 42

[10] Carlo Ghezzi and Amir Molzam Sharifloo. Tool: U-marmo, June 2015. https://sites.google.com/site/amirsharifloo/u-marmo. 26

[11] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996. 26

[12] Jennifer Greene and Andrew Stellman. *Applied Software Project Management*. O'Reilly, first edition, 2005. 1

[13] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang. PARAM: A model checker for parametric Markov models. In *Proc. 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 660–664. Springer, 2010. 16

[14] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011. 13

[15] J. Lions. Flight 501 failure. Technical report, July 1996. University of Minnesota. Accessed: 2014-09-29. 11

[16] T. R. Nicely. The pentium division flaw. *Virginia Scientists Newsletter*, volume 1:page 3, April 1995. 11

[17] V. Nunes, P. Fernandes, V. Alves, and G. N. Rodrigues. Variability management of reliability models in software product lines: An expressiveness and scalability analysis. In *SBCARS'12*, pages 51–60, 2012. vii, 4, 13, 15, 16, 18, 37

[18] Microsoft Patterns. *Microsoft Application Architecture Guide*. Microsoft Press, 2nd edition, 2009. 26, 27

[19] Leonardo Monteiro Pessoa. Flexibilidade em linhas de produtos dinâmicas cientes de qualidade : uma abordagem baseada em linguagens específicas de domínio, October 2014. 36