PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UTILIZAÇÃO DE REDES DE AUTÔMATOS ESTOCÁSTICOS NO PROCESSO UNIFICADO, VISANDO A GERAÇÃO DE CASOS DE TESTE DE *SOFTWARE*

ANDRÉ DE ALMEIDA BARROS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica de Rio Grande do Sul.

Orientador: Paulo Henrique Lemelle Fernandes

Porto Alegre, Brasil
2006

B277u Barros, André de Almeida
Utilização de redes autômatos estocásticos no processo unificado, visando a geração de casos de teste de software / André de Almeida Barros. – Porto Alegre, 2006.
123 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Paulo Henrique Lemelle Fernandes.

1. Informática. 2. Engenharia de Software. 3. Redes de Autômatos Estocásticos. 4. Software - Avaliação. I. Fernandes, Paulo Henrique Lemelle. II. Título.

CDD 005.1

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Utilização de Redes de Autômatos Estocásticos no Processo Unificado, Visando a Geração de Casos de Teste de Software***", apresentada por André de Almeida Barros, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Ciência Computacional, aprovada em 20/03/2007 pela Comissão Examinadora:

Prof. Dr. Paulo Henrique Lemelle Fernandes – Orientador — PPGCC/PUCRS

Prof. Dr. Toacy Cavalcante de Oliveira – PPGCC/PUCRS

Prof. Dr. Michael da Costa Móra – FACIN/PUCRS

Homologada em 20/03/2012, conforme Ata No. 006/12. pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

“Os grandes navegadores devem sua reputação
aos temporais e tempestades”
- Epicuro -

AGRADECIMENTOS

Gostaria de agradecer a todas a pessoas que, de uma forma ou de outra, me ajudaram nessa importante etapa da minha vida.

Ao meu orientador, professor Paulo Fernandes, pela orientação, pelos ensinamentos compartilhados, pela ajuda e compreensão dispensadas, principalmente nos momentos mais críticos desta minha empreitada.

Agradeço aqui a minha família que, mesmo de longe, sempre estiveram presentes com seu incentivo e apoio, sempre acreditando e torcendo pelo meu êxito. Essa conquista é nossa!

Aos meus amigos, com os quais compartilhei grande parte das alegrias e preocupações, sempre me incentivando e apoiando. Em especial, gostaria de agradecer ao Guilherme, Fábio, Virgínia, Joseane, Leonel, Ijuy, Odorico e Rafael, que ajudaram a formar o consulado da ec6 em Porto alegre, fazendo do PPGCC uma espécie de extensão da FURG. Agradeço ainda ao Diego, Sérgio e Gustavo, pelo apoio e pelos momentos de descontração durante esse período.

Ao integrantes do GSP, Guilherme, Eduardo e Vinícius, por compartilharem bons momentos (outros nem tanto) durante o nosso projeto, e por terem abraçado a causa dessa dissertação, com trabalho, esforço e muita Coca-Cola.

Por fim, gostaria de fazer um agradecimento especial à Mari. Pela inspiração, incentivo, apoio, ajuda, paciência e compreensão, presenciando tudo de perto, desde o ingresso no mestrado, até a entrega dessa dissertação. Somente alguém que já passou por isso e entende a importância desse momento, pode saber o quanto valioso é ter uma pessoa assim ao seu lado. Você foi essencial nessa conquista!

UTILIZAÇÃO DE REDES DE AUTÔMATOS ESTOCÁSTICOS NO PROCESSO UNIFICADO, VISANDO A GERAÇÃO DE CASOS DE TESTE DE SOFTWARE

RESUMO

Esse trabalho apresenta método para a construção de SAN, a partir de informações extraídas de diagramas UML concebidos sob a abordagem do Processo Unificado. Nele foi formalizado um *framework* para a transcrição de diagramas de estado UML, utilizados para a descrição comportamental de um sistema, para uma estrutura equivalente em SAN. Essa SAN é utilizada como um modelo de uso do sistema, de onde é possível a extração de casos de teste de *software*, conforme verificado em estudo anteriores. Foi proposta a geração dos modelos sob duas óticas: a primeira focada nas funcionalidades disponibilizadas aos usuários do sistema, e a segunda analisando o sistema como um todo. Para essa última, foi especificado um método de simplificação da SAN, viabilizando assim a sua análise na ferramenta PEPS2003. Baseado no *framework* descrito, foi implementado um protótipo para a construção automática de SAN, a partir de arquivos gerados pelo Rational Rose, arquivos esses contendo informações sobre os diagramas UML utilizados na descrição do sistema. O trabalho também descreve um estudo de caso, onde são aplicadas as técnicas descritas.

Palavras-chave: Redes de Autômatos Estocásticos; Soluções Numéricas; Simulação Exata.

USING STOCHASTIC AUTOMATA NETWORK ON UNIFIED PROCESS, AIMING SOFTWARE TEST CASES GENERATION

ABSTRACT

This research proposes a method to build the SAN, using information extracted from UML diagrams defined according to Unified Process methodology. A framework was formalized to translate the UML state diagrams into SAN structures. The UML state diagrams are basically used to describe the system features. The SAN provided reflect a usage model of the system, what can be used to generate *software* test cases. This research proposal is to generate the models considering two approaches: the first just focus on functionalities available for the system user, and the second one consider the whole system. This last approach also specifies a simplified function to reduce the SAN status to be visible in the PEP2003 Tool. Based on the framework proposed, a prototype was developed to generate the SAN automatically, based on UML diagrams provided by Rational Rose Tool. Finally, this search describes a case of study, where this framework is described in a real example.

Keywords: Stochastic Automata Networks; Numerical Solutions, Exact Simulation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIAÇÕES

UML	Unified Modeling Language
MC	Markov Chain
SAN	Stochastic Automata Network
UP	Unified Process
SAN	Stochastic Automata Network
CPTS	Centro de Pesquisa em Teste de *Software*
STAGE	State Based Test Generator

SUMÁRIO

1. INTRODUÇÃO

Nesse capítulo introdutório é apresentado, em linhas gerais, um resumo sobre o contexto onde se insere o produto dessa pesquisa, o objetivo almejado com esse trabalho e a estrutura utilizada na sua construção.

1.1. Contexto Geral

Com o aumento do tamanho e da complexidade dos sistemas de *software*, técnicas de modelagem relacionadas à abstração e a decomposição do sistema têm se mostrado muito importantes [BPL06]. Essas técnicas possibilitam uma visão antecipada do *software* a ser desenvolvido, permitindo assim uma análise e avaliação prévias, antes mesmo desse modelo ser implementado [JBR99]. Devido a isso, o emprego de modelos é sugerido em metodologias que rejam o processo de desenvolvimento de um sistema de *software*, visando a organização das atividades para obtenção de alta qualidade e baixo custo. Nesse contexto, a UML (Unified Modeling Language) é a linguagem-padrão utilizada para a elaboração da estrutura de projetos de *software*, sendo adequada para a modelagem de sistemas [BRJ00].

Dentre as metodologias empregadas no desenvolvimento de sistemas de *software*, destaca-se o *Processo Unificado*. Esse processo utiliza iterações para evitar o impacto de mudanças no projeto, auxiliando assim o gerenciamento de mudanças, e que concentrem esforços nos pontos críticos do sistema, o mais cedo possível [JBR99].

Uma das etapas que compõem o Processo Unificado é a fase de testes. Segundo [RPG03], a definição dos casos de teste demanda uma importante etapa de todo o processo de desenvolvimento do *software*. O aumento da eficiência na obtenção desses casos de teste e o alcance da cobertura dos mesmos possibilitam o aumento na qualidade e produtividade, reduzindo assim o custo e o tempo de desenvolvimento. Sendo assim, a utilização de técnicas que, agregadas ao processo de desenvolvimento, auxiliem na geração de casos de teste de forma automática, são de grande valia e têm sido alvo de várias pesquisas, conforme será visto no decorrer desse trabalho.

1.2. Estudos anteriores

Dentre as técnicas pesquisadas, algumas utilizam métodos estatísticos para a análise e como base para a construção de casos de teste. Nesse contexto, Walton e Poore (em [WPT95, WP00]) e Whittaker (em [WP93, WT94, Whi97]) descrevem em seus estudos a utilização de Cadeias de Markov (MC) na geração de casos de teste de *software*. Limitações referentes ao espaço total de estados gerados, podendo não oferecer uma precisão necessária para a descrição de sistemas complexos, torna desvantajosa a utilização de Cadeias de Markov.

As Redes de Autômatos Estocásticos (SAN) introduzem um formalismo baseado em MC com grande poder de resolução, além de apresentar mecanismos de suporte à modelagem de sincronismo e paralelismo. A grande diferença e principal vantagem de SAN em relação a MC é a modularização,

propiciando uma forma eficiente de armazenamento dos dados, mantendo as características de modelagem das MC, com a vantagem de amenizar problemas como o da *explosão do espaço de estados*.

Nesse contexto, pesquisas foram efetuadas com o intuito de integrar a análise viabilizada com SAN sobre as técnicas de modelagem de *software* e geração de casos de teste. Em seu estudo, Farina [FFO02, Far02] relatou a viabilidade da utilização de SAN na representação de modelos de uso, tendo como alvo a geração de testes de *software*. Em [BFF+04, Ber05], Bertolini destacou as vantagens da utilização de casos de teste gerados a partir de modelos SAN, principalmente no que se refere ao ganho obtido na análise de cobertura dos testes gerados. Ainda, Neuwald [Neu05] concentrou esforços na consolidação de um método para construção de SAN a partir de modelos UML.

1.3. Proposta

O presente trabalho tem por objetivo propor a integração do formalismo SAN ao Processo Unificado, visando assim o auxílio da geração automática de casos de teste de *software*, tomando como base os diagramas e outros artefatos disponibilizados durante a aplicação do referido processo.

Dentre os objetivos alcançados com a presente pesquisa, podemos destacar:

- a consolidação de um *framework* para a transcrição de diagramas de estados UML para uma estrutura equivalente em SAN, diagramas esses construídos através da metodologia do Processo Unificado;

- a construção de um protótipo para a elaboração automática de SAN, a partir de informações extraídas dos modelos gerados no Processo Unificado. Essas informações foram extraídas de arquivos XML gerados a partir da modelagem do sistema, modelagem essa efetuada no Rational Rose.

- a formalização das possíveis análises disponibilizadas através da aplicação da técnica descrita, bem como um método de simplificação de SAN que agiliza à análise do modelo, mantendo as informações necessárias para construção de casos de teste.

- uma demonstração do método proposto através da elaboração de um estudo de caso, com o qual procurou-se evidenciar os detalhes de especificação necessários, a tradução dos modelos especificados e uma eventual simplificação dos modelos SAN.

Durante o trabalho serão salientadas as limitações não amparadas na presente pesquisa, tais como: os níveis dos testes gerados pelo método apresentado, sugerindo a possibilidade de agregação de outros diagramas UML para a construção de casos de testes; as limitações impostas pelo Rational Rose ao não disponibilizar alguns elementos dos diagramas de estados; a necessidade de inclusão de mecanismos para o tratamento de eventuais condições de guarda existentes em transições dos diagramas de estados; e a necessidade da interação manual com o *framework* de geração de casos de teste. Algumas possíveis soluções para essas são propostas como trabalhos futuros.

1.4. Estrutura do Trabalho

Esse trabalho é dividido em sete capítulos. O Capítulo 2 retoma o referencial teórico com os conceitos necessários para a compreensão do restante do trabalho. No Capítulo 3 é apresentado o estado da arte sobre teste de *software*, formas de geração e métodos empregados na sua construção.

O método proposto para a construção de SAN, a partir de informações disponibilizadas pela metodologia do Processo Unificado, é descrito no Capítulo 4. O Capítulo 5 descreve os detalhes do protótipo proposto para esse trabalho, utilizado para construir SAN, seguindo o método proposto. Um estudo de caso é apresentado no capítulo 6 como um exemplo de aplicação do método proposto, sendo ainda discutidos os resultados obtidos. Por fim, as conclusões feitas nesses estudo, bem como algumas sugestões para incrementar o trabalho feito, são expostas no capítulo 7.

2. REVISÃO TEÓRICA

Esse capítulo retoma o referencial teórico necessário para a compreensão do restante do trabalho, bem como as definições necessárias para a compor a base para as definições que são propostas nesse trabalho.

2.1. Processo Unificado de Desenvolvimento de *Software*

Segundo [JBR99], o Processo Unificado de Desenvolvimento de *Software* (UP) é o conjunto de atividades necessárias para a elaboração de uma sistema de *software* a partir de requisitos do usuário (Figura 2.1). O sistema é construído a partir de componentes de *software* interconectados via interfaces muito bem definidas. O processo unificado utiliza a UML no preparo dos artefatos do sistema. Os aspectos que distinguem o processo unificado são capturados em três conceitos chave:

- direcionado a casos de uso;
- centrado na arquitetura;
- iterativo e incremental.

Cada um desses princípios será abordado em detalhes no decorrer dessa seção.

2.1.1. Processo Direcionado a Casos de Uso

Um sistema de *software* é concebido com o intuito de atender ao usuário, sendo esse uma pessoa ou alguma entidade (outro sistema, por exemplo) que interage com o sistema em questão. Um caso de uso [JBR99] é uma descrição de um comportamento do sistema que pode retornar um determinado valor ao usuário. Casos de uso capturam requisitos funcionais, e a junção desses resulta no modelo de casos de uso, que descreve a funcionalidade completa do sistema.

Os casos de uso direcionam o processo de desenvolvimento, já que baseados no modelo de casos de uso, os analistas criam uma série de modelos para o projeto e a implementação do sistema. Os responsáveis pelos testes realizam seu trabalho com o propósito de garantir que os componentes do modelo de implementação cumpram corretamente os objetivos estabelecidos nos casos de uso. Desta forma, os casos de uso não somente iniciam o processo de desenvolvimento, mas também mantém a integridade com as demais fases do processo. Os casos de uso são especificados, projetados e servem de base para a construção dos casos de teste.

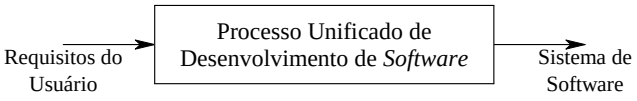


Figura 2.1 - Processo Unificado de Desenvolvimento de *Software*.

2.1.2. Processo Centrado na Arquitetura

O conceito de arquitetura de *software* incorpora os aspectos estáticos e dinâmicos mais importantes do sistema. A arquitetura é influenciada por muitos fatores, tais como a plataforma de *software* sobre a qual o sistema vai rodar (sistema operacional, sistema gerenciador de banco de dados, protocolos para comunicação em rede, etc.), blocos de construção reutilizáveis (por exemplo, um *framework* para construção de interface gráfica com o usuário), considerações sobre distribuição e requisitos não funcionais (performance, confiabilidade, etc.). Ela representa uma visão do projeto como um todo, na qual as características mais importantes são colocadas em destaque.

Define-se como *produto* os artefatos de desenvolvimento, tais como modelos, códigos, documentação e planos de trabalho [BRJ00]. Um produto possui uma função e uma forma, e nenhum desses elementos sozinho é suficiente. Nesse caso, a função corresponde aos casos de uso e a forma à arquitetura. Os casos de uso devem, quando construídos, adequar-se à arquitetura. Por outro lado, a arquitetura deve fornecer espaço para a construção de todos os casos de uso necessários.

2.1.3. Processo Iterativo e Incremental

O desenvolvimento de um *software* é uma tarefa que pode se estender por meses. É conveniente dividir o trabalho em pedaços menores ou miniprojetos. Cada miniprojeto é uma iteração (passos em um fluxo de trabalho) que resulta em um incremento (relacionado ao crescimento do *software*). Cada iteração envolve um ciclo completo de desenvolvimento, que correspondem a uma versão executável de um produto e resultam em um incremento no *software*.

Um incremento não é necessariamente a adição do código executável correspondente aos casos de uso que pertencem à iteração em andamento. Especialmente nas primeiras fases do ciclo de desenvolvimento, os desenvolvedores podem substituir um projeto superficial por um mais detalhado ou sofisticado. Em fases avançadas os incrementos são tipicamente aditivos.

Em cada iteração, os desenvolvedores identificam e especificam os casos de uso relevantes, criam um projeto utilizando a arquitetura escolhida como guia, implementam o projeto em componentes e verificam se esses componentes satisfazem os casos de uso. Se uma iteração atinge seus objetivos, o desenvolvimento prossegue com a próxima iteração, caso contrário, os desenvolvedores devem rever suas decisões e tentar uma nova abordagem.

Foi definido em apud [SK99] que um modelo iterativo de desenvolvimento de *software* pode ser representado por uma escada do tipo espiral, onde cada ciclo corresponde a uma iteração e cada degrau pode ser interpretado como um incremento. Essa representação é apresentada na Figura 2.2 (adaptada de [RPG03]).

Os três conceitos apresentados são igualmente importantes, e a remoção de um deles poderia reduzir o valor do processo unificado.

2.1.4. Ciclo de Vida do Processo Unificado

O processo unificado consiste da repetição de uma série de ciclos durante a vida de um sistema, como mostrado na Figura 2.3. Cada ciclo é concluído com uma versão do produto pronta

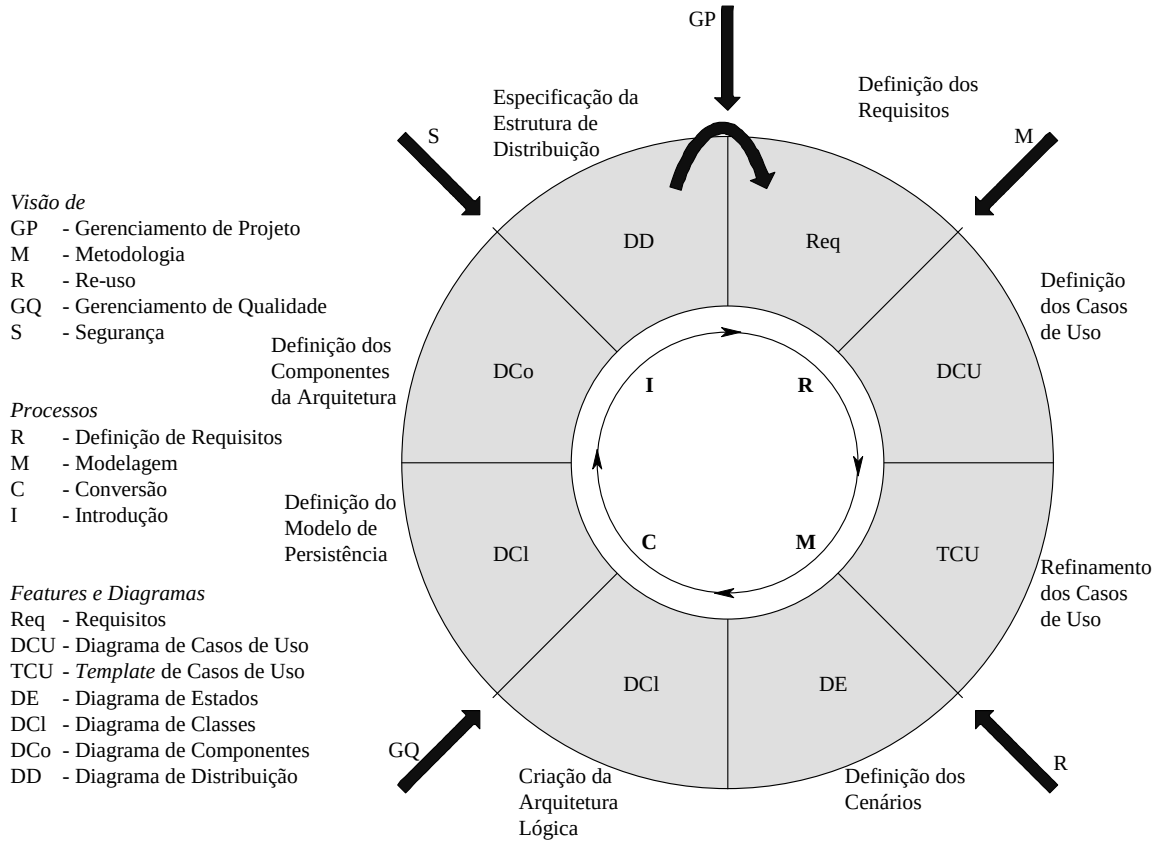


Figura 2.2 - Processo de Desenvolvimento de *Software* Baseado no Processo Unificado.

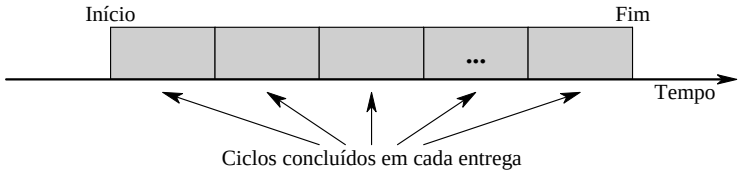


Figura 2.3 - Ciclos que compõem o tempo de vida do sistema.

para distribuição. Essa versão é um conjunto relativamente completo e consistente de artefatos, possivelmente incluindo manuais e um módulo executável do sistema, que podem ser distribuídos para usuários internos ou externos.

Cada ciclo consiste de quatro fases: concepção, elaboração, construção e transição. Cada fase é também subdividida em iterações, como discutido anteriormente. A Figura 2.4 mostra um ciclo, descrevendo ainda suas fases e iterações.

O Produto

Cada ciclo resulta na entrega de uma nova versão do sistema, e cada entrega é um produto pronto para a utilização. O produto final inclui artefatos de interesse do usuário (tais como manuais e código fonte incorporado em componentes que podem ser compilados e executados), artefatos que interessem a pessoas que irão trabalhar no produto (requisitos, casos de uso, especificações não funcionais e casos de teste), incluindo também modelos da arquitetura. Dessa forma, os elementos mencionados permitem especificar, projetar, implementar, testar e utilizar o sistema.

Mesmo que componentes executáveis sejam os artefatos mais importantes do ponto de vista dos

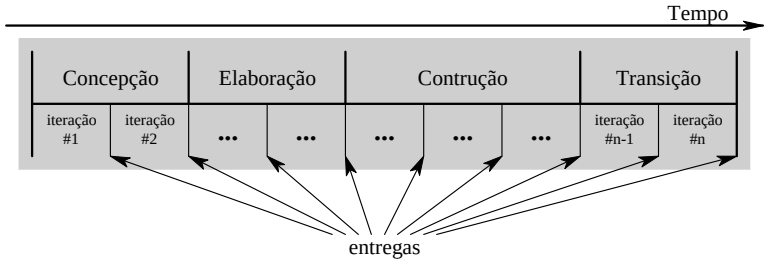


Figura 2.4 - Fases e iterações de um ciclo.

usuários, sozinhos eles não são suficientes. Isso é devido à evolução de sistemas operacionais, sistemas de bancos de dados e máquinas. Ainda, os próprios requisitos podem ser alterados à medida que compreendermos melhor a missão do sistema. Para executar um novo ciclo eficientemente, os desenvolvedores precisam de todas as representações do produto:

- Um modelo de casos de uso, contendo todos dos casos de uso e seus relacionamentos com os usuários.
- Um modelo de análise, que tem dois propósitos: refinar os casos de uso em mais detalhes e fazer uma alocação inicial do comportamento do sistema a um conjunto de objetos.
- Um modelo de projeto que define a estrutura estática do sistema em termos de subsistemas, classes e interfaces, e a realização dos casos de uso como sendo colaborações entre subsistemas, classes e interfaces.
- Um modelo de implantação que define os nós físicos dos computadores e o mapeamento entre os componentes e esses nós.
- Um modelo de implementação, incluindo componentes (código fonte) e o mapeamento entre classes e componentes.
- Um modelo de teste, o qual descreve os casos de teste que serão usados para verificar os casos de uso.
- Uma representação da arquitetura.

O sistema pode ter também um modelo de domínio ou um modelo de negócios que descreva o contexto no qual ele está inserido. Os modelos do processo unificado e suas relações com o modelo de casos de uso, estão representados na Figura 2.5.

Todos esses modelos estão relacionados e juntos representam o sistema como um todo. Elementos em um modelo têm uma ligação com elementos de outro modelo. Por exemplo, um caso de uso pode estar relacionado à sua realização no modelo de projeto e a um caso de teste no modelo de teste. Essas ligações entre os elementos definem um *trace*, o que facilita a compreensão e a modificação.

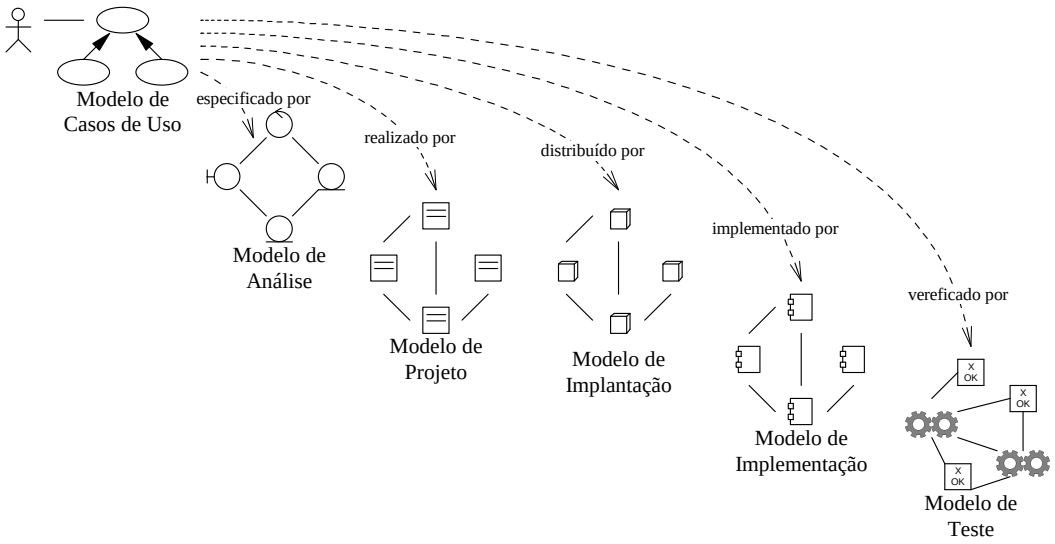


Figura 2.5 - Modelos do Processo Unificado.

Fases em um Ciclo

Um ciclo está dividido em quatro fases, cada qual podendo ser subdividida em iterações e conseqüentes incrementos, como mostrado na Figura 2.6. O final de uma fase é marcado por um ponto de verificação (*milestone*), onde cada um define a disponibilidade de um conjunto de artefatos que formalizam o sistema, tais como modelos e outros documentos.

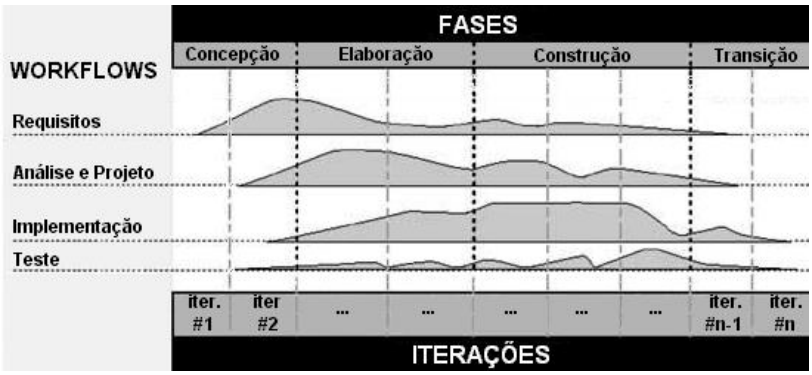


Figura 2.6 - *Workflows* que ocorrem nas quatro fases.

Os pontos de verificação servem a diversos propósitos, dentre eles: como base para tomada de decisão, monitoração do progresso do trabalho, a observação do tempo e esforço gastos em cada fase.

A Figura 2.6 lista na coluna à esquerda os *workflows* que devem ser realizados em cada fase (requisitos, análise e projeto, implementação e testes). As curvas representam uma aproximação do esforço despendido com os *workflows* em cada fase. A seguir é feita uma breve descrição sobre cada uma das fases:

Concepção: durante essa fase são definidos os objetivos do sistema, e como eles contemplam as necessidades do usuário. É definida também uma possível arquitetura para esse sistema. Ainda é efetuado um planejamento e mensurado um custo para o desenvolvimento do produto. Um modelo de casos de uso simplificado, que contenha os casos de uso mais críticos, podem ajudar na definição dos objetivos do sistema. Nesta fase a arquitetura é

experimental, tipicamente apenas um esboço contendo os subsistemas fundamentais. Os riscos mais importantes são identificados e priorizados nessa fase.

Elaboração: aqui, a maioria dos casos de uso do produto são especificadas em detalhes e a arquitetura do sistema é projetada. A arquitetura é representada considerando todos os modelos do sistema. Durante esta fase, os casos de uso mais críticos são realizados. No final da fase de elaboração, o gerente do projeto têm condições de planejar as atividades e estimar os recursos necessários para a conclusão do projeto. Nesse ponto, a arquitetura e os planos devem se mostrar estáveis o suficiente e os riscos sob controle, visando o comprometimento com o desenvolvimento do trabalho como um todo.

Construção: durante essa fase o produto é efetivamente construído. Apesar da possível estabilidade da arquitetura do sistema, os desenvolvedores podem descobrir maneiras melhores de estruturá-lo, podendo ainda sugerir pequenas mudanças. Ao final desta fase, o produto deve conter todos os casos de uso previstos para essa versão. Entretanto, é possível que ele não esteja totalmente livre de defeitos, os quais podem ser detectados e corrigidos durante a fase de transição.

Transição: a fase de transição cobre o período onde uma versão produto é disponibilizado aos usuários. Essa versão é utilizada por um pequeno grupo de usuários experientes que relatam defeitos e deficiências. Os desenvolvedores efetuam as correções e incorporam algumas das melhorias sugeridas. Essa fase envolve atividades como treinamento de clientes, fornecimento de assistência on-line e correção de defeitos encontrados depois da distribuição. A equipe de manutenção freqüentemente divide os defeitos em duas categorias: defeitos operacionais que justificam correção imediata, e aqueles que podem ser corrigidos na próxima versão regular.

2.2. Teste de *Software*

Testes de *software* são amplamente utilizados e aceitos para a validação e aceitação de um sistema de *software*, e podem ser considerados como uma revisão completa da especificação, do projeto e da implementação desse sistema [Gro03]. Inicialmente, os casos de teste devem sempre estar centrados nos requisitos e na especificação e não baseado no código, o que significa que os testes devem sempre visar a conformidade do produto final de acordo com os documentos de requisitos ou especificação. Caso de teste é um conjunto de condições sobre as quais será determinado se um requisito de uma aplicação é parcialmente ou totalmente satisfeito.

Como veremos no próximo capítulo, diversos autores sustentam em suas pesquisas a utilização da UML para a elaboração de casos de teste, uma vez que essa linguagem é utilizada para especificar e modelar adequadamente um sistema computacional de uma forma gráfica e textual.

2.2.1. Estratégias de Teste

Existem várias maneiras de efetuar um teste de *software*. Há técnicas que foram bastante utilizadas em sistemas desenvolvidos em linguagens estruturadas que ainda hoje tem grande valia para os sistemas orientados à objeto. Apesar de os paradigmas de desenvolvimento serem completamente diferentes, o objetivo principal destas técnicas continua a ser o mesmo: encontrar falhas no *software*. Os métodos mais conhecidos são a *caixa-branca* e a *caixa-preta*. Esses métodos se complementam e devem ser aplicados em conjunto a fim de garantir um teste de boa qualidade. A seguir estão descritas essas duas estratégias [Mye04].

Caixa-Branca Nessa estratégia a estrutura interna do programa é examinada, produzindo testes a partir do estudo da lógica do programa. Isso é feito através da análise do código fonte e da elaboração de casos de teste que cubram todos os caminhos de fluxo possíveis do programa, permitindo assim que todo o *software* seja testado.

Caixa-Preta Essa categoria de teste de *software* encara o sistema a ser testado como uma função que mapeia um conjunto de valores de entrada em um conjunto de valores de saída, sem se preocupar com a forma como esse mapeamento foi implementado. Testes caixa-preta baseiam-se exclusivamente na especificação de requisitos para determinar que tipo de saídas são esperadas para um determinado conjunto de entradas.

2.2.2. Categoria de Testes

As diferentes categorias de teste de *software* possibilitam a detecção de falhas no sistema sob diferentes perspectivas. Em determinadas situações, devido à existência de poucos cenários ou à baixa criticidade do sistema, certas categorias de teste podem ser planejadas em conjunto. A seguir são descritas as categorias mais utilizadas:

Teste de Unidade: os testes de unidade (ou testes unitários) têm por objetivo testar pequenas partes ou unidades do sistema, tais como métodos ou pequenos trechos de código.

Teste de Componente: o universo alvo desse tipo de teste seria um pouco maior que o teste unitário, testando o componente como um todo, mas ainda não considerando a iteração com as outras partes do sistema.

Teste de Integração: são os testes que buscam falhas provenientes da integração dos componentes do sistema, geralmente relacionadas ao envio e recebimento de dados.

Teste de Sistema: nesse tipo de teste a busca de falhas é feita através da utilização do sistema, tal como seria feita pelo usuário final.

Teste de Aceitação: geralmente são realizados por um restrito grupo de usuários finais do sistema, através de simulação de operações de rotina do sistema, buscando verificar se o comportamento do mesmo está de acordo com a especificação.

2.2.3. Análise de cobertura

As técnicas de análise de cobertura de código está entre as primeiras técnicas utilizadas para sistematizar os testes de *software*. Esse tema foi tratado inicialmente por Miller e Maloney [MM63]. A partir de então, diferentes autores iniciaram suas pesquisas, onde surgiram variações para técnicas de análise de cobertura, algumas sendo mesclas de diferentes abordagens. As principais técnicas desenvolvidas foram as seguintes [Mye04]:

- *Statement coverage*: nessa verificação, é avaliado se cada sentença de execução é encontrada. A vantagem dessa abordagem é que esse pode ser aplicada diretamente ao código do objeto, não sendo necessário o processamento do código fonte. Por ser insensível a algumas estruturas de controle, como avaliar as condições de um *if* ou *while*, seu uso torna-se desvantajoso. Essa técnica também é conhecida *line coverage*, *segment coverage* [Nta88], ou *basic block coverage*.

- *Decision coverage*: nessa abordagem são verificadas se as expressões booleanas testadas nas estruturas de controle do tipo *if* e *while* são avaliadas como *true* ou *false*. Também conhecida como *branch coverage*, *basis path coverage* ou *decision-decision-path testing* apud [Rop94], essa técnica possui a vantagem de tratar as estruturas de controle não amparadas na *statement coverage*. Sua desvantagem está em ignorar trechos necessitem obrigatoriamente de avaliação. Por exemplo, se no teste do trecho `if( condição1 || condição2 )` a `condição1` for satisfeita, o teste com a `condição2` nunca será testada.

- (*Condition coverage*): essa técnica trata cada sub-expressão booleana separadamente, avaliando-as independentemente uma das outras. A sua desvantagem está em não poder assegurar que todos os caminhos possíveis, oriundos de sentenças condicionais. Para melhor ilustrar essa abordagem, é apresentada a Figura 2.7 (extraída e adaptada de [Mye04])

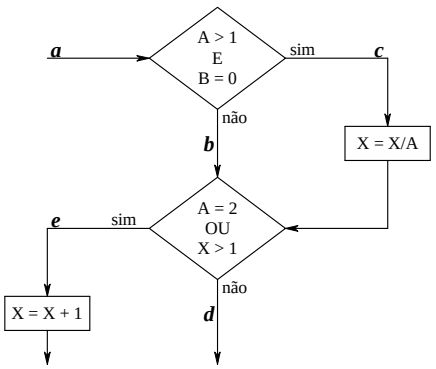


Figura 2.7 - Fluxograma de um program a ser testado.

Suponhamos que o seguinte trecho de código seja testado com os seguintes casos de teste, cobrindo assim os respectivos caminhos:

Case de Teste 1: A=2, B=0, X=4 → ***ace***

Case de Teste 2: A=1, B=1, X=1 → ***abd***

Apesar desses casos de teste cobrirem todos os caminhos, as seqüências ***acd*** e ***abe*** não são executadas. Logo, esses caminhos não são testados e eventuais falhas nesses trechos não são executadas.

- *Path coverage*: essa técnica avalia se cada um dos possíveis caminhos em um função pode ser percorrido. Um caminho é uma seqüência única de ramificações desde a entrada da função até a sua saída. Nessa abordagem, também conhecida como *predicate coverage* [Bei90], os caminhos **acd** e ***abe*** da Figura 2.7 são testados.

Existem outras técnicas, derivadas das descritas anteriormente. Elas serão aqui relacionadas em um caráter apenas informativo, não sendo detalhadas as suas características (maiores detalhes em [Cor06]): *function coverage*, *call coverage*, *linear code sequence and jump coverage*, *data flow coverage*, *object code branch coverage*, *loop coverage*, *race coverage*, *relational operator coverage*, *weak mutation coverage*, *table coverage*, etc.

2.3. Redes de Autômatos Estocásticos

Redes de Autômatos Estocásticos (SAN) é um formalismo, baseado em Cadeias de Markov, para modelagem de sistemas. Foi proposto na tese de doutorado de Brigitte Plateau, em 1984 [Pla84]. A idéia principal do formalismo SAN é modelar um sistema em vários subsistemas chamados de autômatos, que podem ou não interagir entre si. SAN é um formalismo para modelagem de sistemas complexos e com grande espaço de estados.

A grande diferença e principal vantagem de SAN em relação a MC é a modularização, pois propicia uma forma eficiente de armazenamento dos dados do sistema, mantendo as características de modelagem das MC, com a vantagem de amenizar problemas como o da *explosão do espaço de estados* [BFS04].

As próximas subseções apresentam alguns conceitos básicos necessários para a elucidação do formalismo SAN.

2.3.1. Autômatos Estocásticos

Um autômato è um modelo matemático de um sistema, com entradas e saídas discretas. O sistema pode estar em qualquer um dentre um número finito de estados ou configurações internas [AFJ+99]. Esses estados sintetizam as informações relativas às entradas anteriores e informam ainda os possíveis comportamentos do sistema diante das entradas seguintes. A denominação de *estocásticos* atribuída a esses autômatos é atribuído ao fato do tempo ser tratado como uma variável aleatória, e essa obedece uma distribuição exponencial na escala de tempo contínua, ou geométrica no caso de escala de tempo discreta [Pre06].

2.3.2. Estados Locais e Globais

Os estados de um modelo SAN podem ser observados através de dois aspectos: local (refere-se ao estado individual de cada autômato pertencente ao modelo) e global (considera a combinação dos estados locais de todos os autômatos do modelo). Intuitivamente, a mudança do estado local de qualquer autômato do modelo altera o estado global do sistema. A Figura 2.8 exemplifica uma SAN com seus estados locais, mostrando também os estados globais equivalentes.

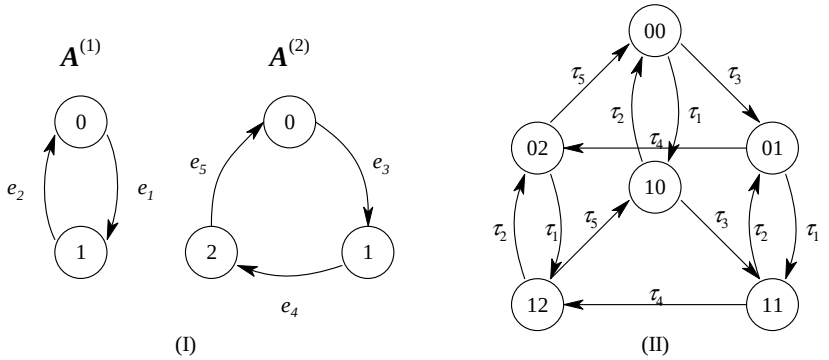


Figura 2.8 - Estados locais (I) e globais (II) de uma SAN.

A SAN representada na Figura 2.8 possui um autômato de 2 estados locais e outro de 3 estados locais. O número de estados globais equivalentes a este exemplo é o número de estados igual ao produtório do número de estados locais de todos os autômatos da SAN. Para o nosso exemplo, tem–se $2 \times 3 = 6$ estados globais (2 estados locais do autômato $\mathcal{A}^1$ e 3 estados locais do autômato $\mathcal{A}^2$).

O estado global 01 corresponde a situação em que o autômato $\mathcal{A}^1$ encontra-se no estado 0 e o autômato $\mathcal{A}^2$ no estado 1. Essa analogia é adotada para os demais estados globais equivalentes. Vale ressaltar que o autômato equivalente da Figura 2.8 tem fim meramente elucidatório, uma vez que a solução dessa SAN pode ser obtida independentemente para cada um dos seus subsistemas, pois só há transições locais, e nenhuma interação entre os dois autômatos.

2.3.3. Eventos Locais e Sincronizantes

Evento é a entidade do modelo responsável pela ocorrência de uma transição, mudando assim o estado global do modelo. Um ou mais eventos podem estar associados a uma transição e esta é disparada através da ocorrência de qualquer um dos eventos a ela associada. No formalismo SAN podem ser modelados dois tipos de eventos: *locais* (são aqueles que modificam o estado local de um único autômato sem interferir no estado local dos demais autômatos do modelo) e *sincronizantes* (eventos que disparam transições em mais de um autômato, modificando assim os estados locais dos autômatos envolvidos).

Para uma descrição completa, as transições sincronizadas necessitam de uma estrutura auxiliar para especificar os tipos de transições e eventos de um modelo SAN. Isso é feito através de uma *tabela de eventos*, onde cada linha da tabela contém a identificação de um evento, a sua taxa de ocorrência e o tipo desse evento [AFJ+99]. Essa tabela é consultada para distinguir os eventos locais dos sincronizantes, visto que a representação gráfica não apresenta tais informações. A Figura 2.9

apresenta um exemplo de SAN com eventos sincronizantes e o autômato global equivalente. Na Tabela 2.1 são especificadas as referidas taxas de ocorrência.

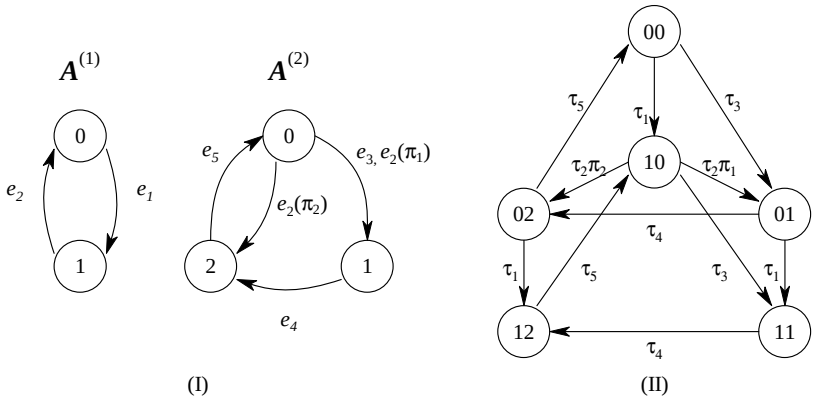


Figura 2.9 - SAN com eventos sincronizantes (I) e seu autômato global equivalente (II).

Tabela 2.1 - Taxa de ocorrência e tipo dos eventos do modelo SAN apresentado na Figura 2.9.

Evento	Taxa de Ocorrência	Tipo
e_1	τ_1	Local
e_2	τ_2	Sincronizante
e_3	τ_3	Local
e_4	τ_4	Local
e_5	τ_5	Local

2.3.4. Taxas e Probabilidades Funcionais

Como visto anteriormente, todo evento em um modelo SAN deve ter associado a si uma taxa de ocorrência e uma probabilidade de ocorrência[1], mas essa última é abstraída caso sua taxa seja igual a *1*. Tanto a taxa de ocorrência como a probabilidade de ocorrência podem ser definidas como valores constantes ou valores funcionais. Quando as taxas e probabilidades são definidas como valores funcionais, essas são ditas *taxas funcionais* e *probabilidades funcionais*. Nesses casos, os valores assumidos por estas taxas e probabilidades dependem dos estados locais dos demais autômatos do modelo.

As taxas e probabilidades funcionais proporcionam uma segunda possibilidade de interação entre autômatos nos modelos SAN (a outra possibilidade é a utilização de eventos sincronizantes[2], como visto anteriormente). As funções associadas com as taxas e probabilidades permitem atribuir a um mesmo evento diferentes valores conforme o estado global do sistema.

O autômato $\mathcal{A}^{(1)}$ da Figura 2.10 (I) apresenta uma transição local funcional (do estado 0 pro estado 1), que depende do estado interno do autômato $\mathcal{A}^{(2)}$. Essa função f[3] é dada por:

[1]Também conhecida como probabilidade de rotação

[2] A utilização de taxas e probabilidades funcionais não está limitada aos eventos locais e podem ser empregadas similarmente nos eventos sincronizantes.

[3]Notação utilizada pela ferramenta PEPS2003

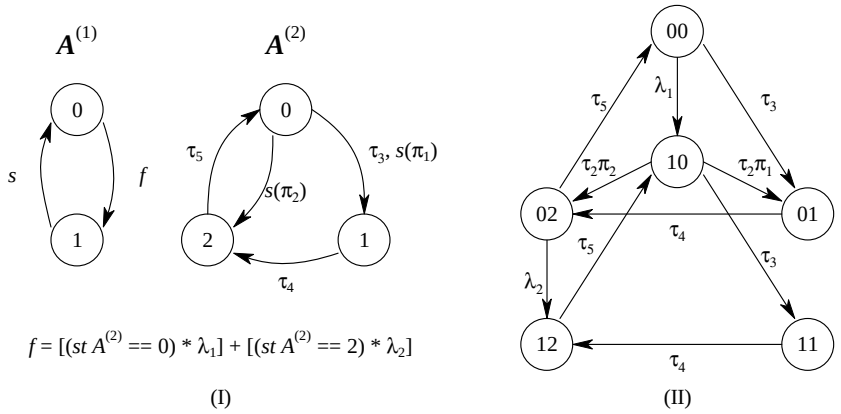


Figura 2.10 - SAN com transição funcional (I) e seu autômato global equivalente (II).

Tabela 2.2 - Taxa de ocorrência e tipo dos eventos do modelo SAN apresentado na Figura 2.11

Evento	Taxa de Ocorrência	Tipo
e_1	f	Local
e_2	μ	Sincronizante
e_3	σ	Local
e_4	δ	Local
e_5	τ	Local

$$f = \begin{cases} \lambda_1 & \text{se autômato } \mathcal{A}^{(2)} \text{ está no estado } 0; \\ 0 & \text{se autômato } \mathcal{A}^{(2)} \text{ está no estado } 1; \\ \lambda_2 & \text{se autômato } \mathcal{A}^{(2)} \text{ está no estado } 2; \end{cases}$$

Isto significa que a transição do estado 0 para o estado 1 ocorre com uma taxa de ocorrência λ_1, caso o autômato $\mathcal{A}^{(2)}$ esteja no estado 0, ocorre com uma taxa λ_2, caso o autômato $\mathcal{A}^{(2)}$ esteja no estado 2, e não ocorre caso o autômato $\mathcal{A}^{(2)}$ esteja no estado 1. A Figura 2.10 também apresenta mostra o autômato equivalente para essa SAN (II).

2.3.5. Função de Atingibilidade

Devido à representação em SAN ser de forma modular e o autômato global (equivalente à MC) ser composto pela combinação de todos os autômatos do modelo, se faz necessária a especificação de uma função que defina quais são os estados atingíveis deste autômato global que representam a SAN.

A definição de quais destes estados podem ser *atingidos* ou *alcançados* em SAN é dada pela *função de atingibilidade*. Ela é definida usando as mesmas regras adotadas para a definição de taxas e probabilidades funcionais.

Vamos tomar como exemplo a SAN descrita na Figura 2.11, que apresenta 2 autômatos, 4 eventos locais, um evento sincronizante e uma transição funcional. Ela possui seus eventos e taxas de probabilidade descritos na tabela 2.2.

Assumindo que os estados 0 de $A^{(1)}$ e 0 de $A^{(2)}$ como atingíveis (Figura 2.11 (I)) podemos supor que, por exemplo, o autômato $A^{(2)}$ não pode se encontrar no estado 1 se o autômato $A^{(1)}$ estiver no estado 1, e vice-versa. Para tanto, deve-se definir a seguinte *função de atingibilidade*:

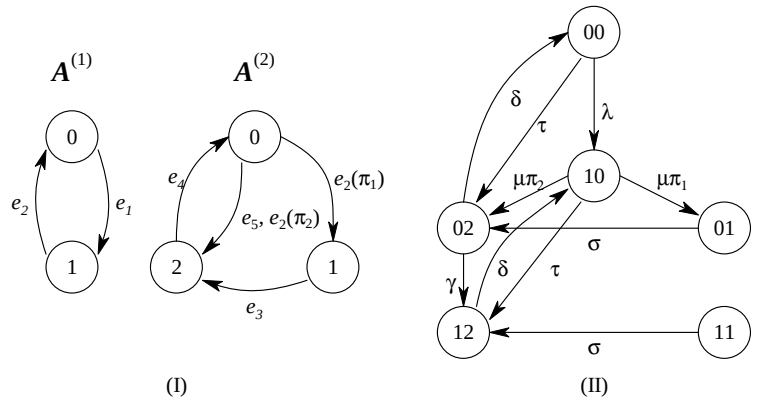


Figura 2.11 - SAN com estados inatingíveis (I) e seu autômato global equivalente (II).

$$reachability = !\ ((st\ A^{(1)} == 1)\ \&\&\ (st\ A^{(2)} == 1))$$

Através da função de atingibilidade podemos construir o autômato equivalente (Figura 2.11 (II)), onde é possível visualizar os estados inatingíveis da SAN. Dessa forma, os estados globais atingíveis dessa SAN seriam: 00, 01, 02, 10 e 12, visto que o estado 11 nunca é atingido, partindo de qualquer outro estado.

2.3.6. Funções de Integração

A proposta das *funções de integração* é obter a probabilidade do modelo encontrar-se em um determinado estado da SAN. Essas funções podem ser definidas em qualquer modelo SAN. Com isso, pode-se compor funções de integração que levem em conta a probabilidade do modelo se encontrar em um conjunto de estados, podendo assim obter índices de desempenho e confiabilidade do modelo. A avaliação dessas funções é realizada sobre o *vetor de probabilidades* que contém a probabilidade do modelo se encontrar em cada um de seus possíveis estados.

Por exemplo: se considerando a SAN apresentada na Figura 2.11 (II), a função de integração u que avaliaria a probabilidade do autômato $A^{(1)}$ estar no estado 0, seria dada da seguinte forma:

$$u = (st\ A^{(1)} == 0)$$

Via de regra, todas as funções são modeladas em SAN da mesma forma. O que as diferenciam é como elas são empregadas no modelo.

O próximo capítulo apresenta uma revisão sobre o estado da arte da geração de casos de teste, suas aplicações, eventuais limitações, destacando as informações que sustentam o presente trabalho, buscando assim evidenciar a aplicabilidade dessa pesquisa.

3. TRABALHOS RELACIONADOS

Esse capítulo apresenta uma revisão sobre alguns estudos relacionados a geração de casos de testes. Com isso, busca-se destacar as aplicações das técnicas estudadas, as aplicações visadas por essas, salientar as eventuais limitações que cada abordagem pode possuir. Como isso, almeja-se destacar as detalhes que sustentem a presente pesquisa, buscando dessa forma evidenciar a aplicabilidade do presente trabalho.

Vários estudos já foram efetuados sobre geração de casos de teste de *software*, enfatizando a sua aplicabilidade, analisando ainda a sua eficácia em comparação a outras técnicas convencionais. A seguir são descritas algumas metodologias para a geração de casos de teste.

3.1. Técnicas de teste de *software*

Diferentes abordagens sobre a fonte para a geração de casos de teste, bem como as técnicas para a geração desses teste, são alvo de vários estudos. Nessa seção são apresentados algumas dessas pesquisas, salientando alguns detalhes e vantagens de cada abordagem.

3.1.1. Teste Randômico

Segundo a tese de mestrado de Zhang [Zha00], testes randômicos são baseados no domínio de entrada e geralmente aplicado sobre o método da *caixa preta*. Os testes são randomicamente selecionados de todo o domínio de entrada, não considerando qualquer informação sobre a estrutura ou especificação do programa. Nesse trabalho foi citado o estudo de Girard e Rault (apud [GR73]), os quais propuseram a utilização de testes randômicos como um valioso esquema para a geração de casos de teste de *software*, especialmente na etapa final do testes de *software*.

Testes randômicos provém uma maneira simples de gerar casos de teste sobre *softwares* que não apresentam uma análise profunda sobre a sua confiabilidade. Com a esse tipo de teste é gerado um grande número de casos de teste, provendo assim uma maneira automática de geração da casos de teste. Uma desvantagem do teste randômico é destacada nesse trabalho: como não utiliza qualquer informação sobre a especificação do *software*, somente algumas entradas são selecionadas, não sendo possível garantir se efetividade dessa seleção.

3.1.2. Teste Estatístico

A emprego de MC para modelar a utilização de um *software*, e daí extrair informações para gerar casos de teste, foi sugerido (não inicialmente) por Whittaker e Poore em [WP93]. O estudo afirma que em um teste estatístico, os eventos de interesse são seqüências de estímulos que representam uma execução do *software*. Uma descrição estatística das seqüências é obtidas pela definição de variáveis randômicas que descrevem o perfil do conjunto total de seqüências usadas na verificação do *software*. Ainda, um modelo estocástico é definido para guiar a geração de casos de teste e calcular as estatísticas de uso do *software*.

No referido trabalho são propostas duas fases de construção da MC. Na *fase estrutural* os estados e os arcos são definidos, e na *fase estatística* as probabilidades das transições são especificadas. Nesse trabalho também foi referenciada a tese de mestrado de Sexton (apud [Sex88]), onde é mencionada a necessidade da seleção randômica dos casos de teste para estar de acordo com a *distribuição de uso*.

Em sua pesquisa [AL93], Avritzer descreve uma nova técnica de teste denominada *Deterministic Markov State Testing*, relatando também a sua aplicação. Essa é uma técnica para a geração e execução de casos de teste para um *software* de telecomunicação, baseado no seu perfil operacional, onde esse é usado para construir uma cadeia de Markov que representa o comportamento desse *software*. Dentre as vantagens apresentadas por essa método, estão a provisão de informações precisas sobre estados do *software* para a análise em testes de carga, e pelo gerenciamento simples de execução distribuída de casos de teste. Em outra pesquisa [AW94], Avritzer apresenta três algoritmos para a geração automática de casos de teste. Apesar dessas técnicas terem sido desenvolvidas para o teste de *software* de telecomunicação, elas podem ser usadas em qualquer *software* que puder ser modelado que cadeias de Markov.

Whittaker e Thomason descrevem em [WT94] como as falhas de *software* podem ser avaliadas analiticamente através de MC.

3.1.3. Teste Estocástico

Whittaker descreve em seu trabalho [Whi97] um método para a seleção de casos de teste que possibilita uma alternativa de formalização de teste de *software*. Isso seria feito através da criação de modelos estocásticos, por parte dos testadores, que descrevessem o comportamento do *software* em alternativa à geração de casos de teste tradicional.

Nesse artigo é afirmado que apesar de um modelo escolhido poder possuir um número qualquer de processos estocásticos, MC foram usadas pois mostram ser confiáveis e possibilitam um *feedback* analítico. O trabalho não define as métricas utilizadas para extração das probabilidades de ocorrência dos eventos, sugerindo que isso pode ser extraído de uma análise de comportamento dos usuários. Ainda, o exemplo utilizado segue um processo de construção informal, baseado em ações do usuário e meramente ilustrativo.

3.2. Testes de *software* baseados em modelos

Nos testes baseados em modelos, esses provém a informação primária para o desenvolvimento de *test suites*, checando ainda a implementação do sistema. Gross relata em seu relatório [Gro03] o *encaixe perfeito* da utilização dos diagramas UML para a geração de casos de teste. Nesse trabalho são apresentadas, superficialmente, as definições dos diagramas UML e como cada um deles pode auxiliar na geração de casos de teste. Ao final é discutido os perfis de teste que a UML proporciona, podendo ser regido por aspectos estruturais ou comportamentais.

Em [RPG03, FL00] são descritos métodos para geração de casos de teste, baseados na análise comportamental de casos de uso, especificados em três notações diferentes oferecida pela UML

[Hur06b]. Uma breve descrição sobre esses diagramas, bem como a sua pertinência para a geração de casos de teste, é feita a seguir:

Diagramas de Atividade: interpretam os casos de uso como processos ramificados e são recomendados por alguns autores para a formalização de casos de uso. Porém esses diagramas apresentam duas desvantagens: eles modelam apenas o fluxo de controle do programa e não suportam a distinção entre os comportamentos normais e anormais que a análise dos casos de uso beneficia.

Diagramas de Interação: esses diagramas[1] podem ser usados para formalizar separadamente os cenários contidos nos casos de uso. Sendo que um caso de uso é uma coleção hierárquica de cenários, esses devem ser formalizados separadamente em diagramas de interação e então serem agrupados novamente em único diagrama de estados.

Diagramas de Estado: especificam o comportamento do sistema em reação aos eventos ocasionados pelos atores. Em contraste com os diagramas de interação, diagramas de estado visualizam múltiplos cenários, como no caso de cenários hierárquicos descritos pelos casos de uso.

Como mencionado, os diagramas citados são baseados nas especificações realizadas nos casos de uso. A seguir será descrita a estrutura de especificação de casos de uso utilizada como foco desse trabalho.

3.2.1. Template para especificação de Casos de Uso

Existem estruturas para a especificação dos casos de uso que auxiliam a construção dos diagramas UML que auxiliam na análise comportamental do sistema. A estrutura do caso de uso citada no processo unificado [JBR99] é baseada no template criado por Alistair Cockburn [**?**]. A Tabela 3.1 descreve essa estrutura

A granularidade da especificação do cenário define futuramente a granularidade do teste, ou seja, um passo do cenário que descreve um estímulo será transformado numa entrada atômica de teste, e um passo do cenário descrevendo uma resposta representará uma resposta atômica observável.

3.2.2. Casos de teste gerados a partir de diagramas de estado

Partindo da estrutura especificada anteriormente, é possível construir um diagrama de estados que modele o comportamento do caso de uso. Em [RPG03, FL00] foram descritos métodos para construção de diagramas de estado a partir da estrutura de especificação de casos de uso proposta por Cockburn [**?**], que mapeiam o comportamento do caso de uso da seguinte forma:

Cenário de Sucesso Principal: As mensagens enviadas pelo sistema serão tratadas como ações dos estados. Cada mensagem enviada por um ator é denotada como um evento.

[1] diagramas de seqüência e de colaboração

Tabela 3.1 - Template para especificação de casos de uso.

Nome	Identificação do caso de uso.
Meta	Descreve o propósito global do caso de uso.
Atores	Envolvidos no caso de uso.
Pré-condições	Condições necessárias para que a seqüência dos casos de uso seja executada.
Pós-condições	Condições disponibilizadas após a execução do caso de uso.
Cenário de Sucesso Principal	Descreve como o caso de uso pode atingir a meta com a enumeração de estímulos e respostas alternadas do sistema, iniciando com o estimulo disparado pelo caso de uso.
Variações	Cursos alternativos da ações que mantém o caso de uso dentro de seus padrões normais. A especificação das variações é feita através referenciando o respectivo passo do cenário de sucesso principal, sendo esse refinado com um ou mais passos alternativos.
Extensões	Um cenário de resposta a uma circunstância excepcional (um dado de entrada inválido, por exemplo). Esta especificação adota o mesmo formalismo descrito para as variações.
Casos de Uso Incluídos	Lista de outros casos de uso utilizados por este (usualmente estes são referenciados por arcos <<includes>> nos diagramas de caso de uso).

Variações: São alternativas de execução de um passo no caso de uso, modelados através de múltiplos caminhos conectando dois estados (estados intermediários podem ser necessários para modelar variações mais complexas).

Extensões: Acontece quando uma sub-meta (passo) falha, pois uma pré-condição não foi assegurada. Isso é feito com a utilização de sub-estados do estado correspondente do cenário de sucesso principal.

Pré e Pós-condições Pré-condições descrevem as condições que asseguram previamente a execução do caso de uso. Pós-condições são modeladas como restrições no ultimo estado representado no caso de uso.

Casos de usos subordinados: UML define um mecanismo para a inclusão de um caso de uso como uma sub-função em outro caso de uso.

Os referidos trabalhos descrevem os mecanismos de extração do perfil de teste a partir das informações contidas nos diagramas de caso de uso. Os casos de teste são gerados a partir de uma determinada seqüência de estados de ações, levando em conta a probabilidade de ocorrência dessas ações.

3.2.3. Casos de teste gerados a partir de diagramas de interação

Em seu trabalho [AO00], Abdurazik e Offutt apresentam um critério de teste baseado em diagramas de colaboração UML. Segundo eles, o critério desenvolvido define testes que verificam aspectos

estáticos e dinâmicos em níveis de especificação e de instância a partir dos diagramas de colaboração. O paper também apresenta um algoritmo que auxilia o testator a rastrear os caminhos possíveis no diagrama.

Os diagramas de interação contém essencialmente a mesma informação, mas disposta de forma diferente. Os diagramas de seqüência e de colaboração se equivalem e podem ser transcritos de um para o outro sem perda de informação [BRJ00]. Como visto anteriormente, a utilização de diagramas de interação para a geração de casos de teste não se mostra completa pois esses formalizam os cenários contidos nos casos de uso separadamente. Seria apropriado reagrupar a coleção de cenários modelados pelos diagramas de interação em um único diagrama de estados.

3.3. Técnicas para análise de cobertura

Técnicas de cobertura de teste exercem um papel significante no aperfeiçoamento da qualidade dos sistemas de *software*. A avaliação de cobertura de código consiste na identificação das partes do programa que não são atingidas em uma ou mais execuções do programa. Nesse sentido, a análise de cobertura de testes de *software* tem sido alvo de vários estudos, sendo citados alguns desses nessa seção.

3.3.1. Ferramentas de teste baseadas em cobertura

Diversas ferramentas monitoramento e análise de cobertura de teste são disponibilizadas no mercado. Gaffney demonstra em sua pesquisa [GTJ04] que apesar das informações fornecidas por uma ferramenta de cobertura ser útil e valiosa, elas não são suficientes em testes de fragmentos de código. Essa afirmação baseia-se no critério de teste disponibilizado pelas ferramenta, os quais podem não suportar a cobertura de determinadas nuances contidas no código do programas[2].

Em sua *survey* [YLW06], Yang apresenta um comparativo entre 17 ferramentas de teste baseadas em cobertura de teste. Adicionalmente, essas ferramentas podem oferecer algumas outras funções, sendo três dessas utilizadas como método comparativo entre as mesmas: a abrangência da cobertura, o critério de cobertura, e geração automática de casos de teste e customização de relatórios.

3.3.2. Técnicas para otimização da cobertura de testes

Agrawal apresentou em [Agr99] uma técnica para encontrar um pequeno sub-conjunto de sentenças e decisões de um programa (também denominado como *bloco*), onde a cobertura desse sub-conjunto implique na cobertura do restante do programa. Essa técnica é uma otimização de uma abordagem apresentada anteriormente pelo autor em [Agr94], onde são encontrados sub-conjuntos de nodos (*blocos*) de um gráfico de fluxo que satisfaçam a seguinte condição: um conjunto de testes que exercite todos os nodos de um sub-conjunto exercita todo os os nodos do gráfico de fluxo.

Em sua otimização, Agrawal introduziu a idéia de um *mega-bloco*, sendo esse um conjunto de blocos básicos que contém múltiplos procedimentos, onde a cobertura de um bloco básico implica

[2]detalhes sobre as diferentes abordagens de cobertura de teste são apresentados na subseção 2.2.3..

na cobertura de todos os blocos básicos similares. Ainda é apresentada a definição de uma estrutura de dados denominada *global dominator graph*, onde são expostas as relações entre os *mega-blocos*. Com essa estrutura, o testator precisaria apenas criar casos de teste que cobrissem os blocos básicos, implicando na cobertura do restante do programa através do mesmo conjunto de testes.

Os estudos efetuados por Agrawal serviram como base para a formalização de outras técnicas, mas todas trabalhando com a idéia de utilização de gráficos de fluxo, porém agregando novos métodos para a otimização da cobertura dos casos de teste gerados. Dentre essas técnicas, podemos citar o trabalho de Tikir [TH02] e de Baudry [BFT06].

3.3.3. SAN e a análise de cobertura

Como citado anteriormente, um caso de teste exercita um determinado caminho modelado em um caso de uso, não herdando e nem propagando erros. Esse caminho (trajetória) é uma seqüência de passos (*estado-transição-estado*) no estado global de uma SAN. Como em uma SAN cada transição está associada a mais de um evento, porém cada evento corresponde a pelo uma transição em MC. Conforme formalizado em [Ber05], um caso de teste é uma estrutura $(\mathcal{S}, \mathcal{E})$, onde $\mathcal{S} = \mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{N+1}$ corresponde a um conjunto ordenado de estados globais (SAN) ou estados (MC) e $\mathcal{E} = \mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_N$ corresponde a um conjunto de eventos (SAN) ou transições (MC).

Um *test suite* é definido como um conjunto de casos de teste. Nesse trabalho vamos considerar $\mathcal{TS} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_\mathcal{M}\}$ como um *test suite* composto por $\mathcal{M}$ casos de teste $\mathcal{T}_i$, com i variando de 1 a $\mathcal{M}$.

Como mencionado, uma determinada trajetória é composta por uma quantidade finita e conhecida de passos, e cada passo possui uma probabilidade de ocorrência σ. Essa probabilidade é a ocorrência de uma transição do $i - esimo$ estado para o $i - esimo + 1$, e é dada por:

$$\sigma_i = \frac{\pi_{i+1}}{\sum_{\forall k \ll i} \pi_k} \tag{3..1}$$

onde π_k é a probabilidade do $i - esimo$ estado da solução estacionária (ou transiente) de um modelo, e $\forall k \ll i$ significa a possibilidade de ir para todos os $k - esimos$ estados sucessores do $i - esimo$ estado.

A probabilidade de um caso de teste é um produtório de todas as probabilidades dos passos $\sigma_i (i = 1..N)$ do caso de teste, o qual corresponde uma trajetória $\mathcal{T}$. Dessa forma, a probabilidade de um caso de teste $P_\mathcal{T}$ é dada por:

$$P_\mathcal{T} = \prod_{i=1}^{N} \sigma_i \tag{3..2}$$

Sendo que um *test suite* pode conter muitos casos de teste replicados, é necessário considerar apenas casos de teste distintos para avaliar a cobertura de um *test suite*. O número de casos de teste distintos D pode ser menor ou igual ao número total de casos de teste M. Sendo assim, o percentual de cobertura C de um *test suite* TS é calculado pela soma da probabilidade de todos os casos distintos, ou seja:

$$C = \sum_{\mathcal{T}=1}^{D} P_{\mathcal{T}} \tag{3..3}$$

A partir de um modelo podem ser gerados infinitos casos de teste, definindo assim como assintótico o cálculo de cobertura C de um *test suite*, uma vez que o índice de cobertura pode ser próximo mas nunca igual a 1.

A qualidade da geração dos casos de teste é verificada pelo cálculo da *Distância Euclidiana* e do *Discriminante de Kullback* [Ber05, Far02], efetuando assim a comparação de dois vetores: o *Vetor de Probabilidades* (Φ) e o *Vetor de Freqüências dos Casos de Teste* (Θ). O primeiro é obtido a partir de um *test suite* e contém apenas as probabilidades dos casos de teste distintos, portanto, sendo composto por D probabilidades e com cardinalidade dada por $|\Phi| = D$ e $D \leq M$. O vetor de freqüências também é composto por D probabilidades, porém agora são computados inclusive os casos de teste replicados do *test suite*. Após a normalização, que é dada pela contagem dos casos de teste replicados sobre o total de casos de teste gerados, obtém-se o vetor de freqüência $\Theta(\sum_{i=1}^{D} \approx 1)$.

No próximo capítulo será especificada a transcrição dos diagramas UML para uma estrutura equivalente descrita pelo formalismo SAN.

4. Construção das Redes de Autômatos Estocásticos

Nesse capítulo são descritos os detalhes da transcrição dos diagramas UML que fornecem a base para a construção de uma rede de autômatos estocásticos, com a qual será possível fazer uma análise do modelo de uso, buscando a geração de casos de teste de *software*.

4.1. Utilização de artefatos fornecidos pelo Processo Unificado para a construção da SAN

Baseado em trabalhos relacionados ao corrente estudo, pode-se afirmar que é possível compor um *framework* para a construção de SAN a partir de especificações de casos de uso, guiados pelo *template* proposto por Cockburn [Coc06]. Isso seria feito de maneira similar ao descrito em [FL00, RPG03], porém o alvo seria a construção dos elementos de uma SAN e não de um diagrama de estados. Quando o Analista de Requisitos conclui a atividade de detalhar um caso de uso, é possível extrair as informações necessárias para a construção da SAN que modele esse caso de uso, a partir do *template* citado. As atividades associadas ao *workflow* de requisitos proposto pelo Processo Unificado são apresentadas na Figura 4.1, sendo destacada a atividade de descrição de casos de uso.

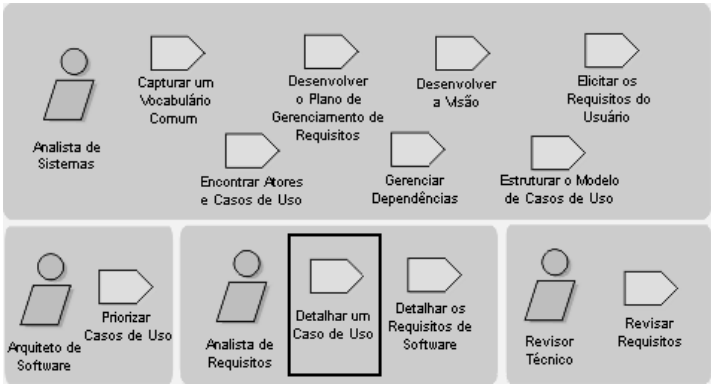


Figura 4.1 - Resumo das Atividades de Requisitos.

Sendo um dos objetivos desse trabalho a construção automática de arquivos .san a partir de informações extraídas de artefatos produzidos de acordo com o Processo Unificado, é necessário um volume maior de informações que sustentem a construção da SAN. Essas informações podem ser extraídas na fase de análise e projeto, conforme o *workflow* apresentado na Figura 4.2.

A atividade de projeto de casos de uso sugere a utilização de diagramas de estados para a descrição de caminhos alternativos para a realização de um caso de uso. Já no atividade de projeto de sub-sistemas, alguns desses podem ser altamente dependentes do estado em que se encontram, podendo representar uma ou mais *threads* de fluxo de controle. Nesses casos a utilização de diagramas de estado também é útil na descrição do comportamento do sub-sistema. Nesse contexto, esses diagramas são usados conjuntamente com os diagramas de atividade, buscando representar a decomposição das

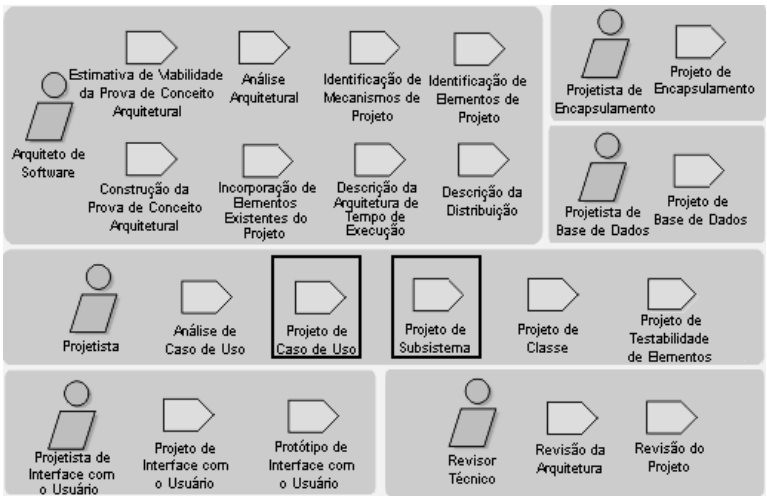


Figura 4.2 - Resumo das Atividades de Análise e Projeto.

threads de fluxo de controle. Sendo assim, esse trabalho busca integrar SAN ao Processo Unificado, por intermédio dos artefatos que esse disponibiliza. Com isso espera-se obter um modelo de uso do sistema a ser construído, sobre o qual é possível efetuar análises de comportamento do *software* como, por exemplo, a determinação da funcionalidade do sistema que possui uma maior probabilidade de uso. Busca-se ainda a utilização da SAN gerada pra a elaboração de casos de teste.

Com o estudo realizado para esse trabalho, é possível apresentar alternativas no processo de geração de casos de teste com a incorporação de SAN. Essa alternativa é destacada na Figura 4.3, que apresenta as atividades sugeridas para um processo de desenvolvimento de *software*.

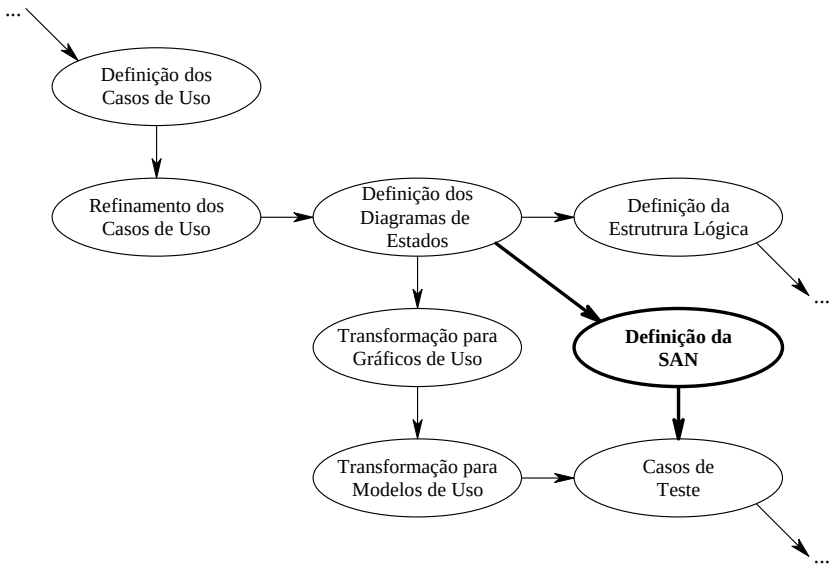


Figura 4.3 - Atividades sugeridas para o processo de desenvolvimento de *software*.

Os gráficos de uso são uma variação do diagrama de estados onde, dentre outras modificações, a estrutura hierárquica dos estados (estados aninhados) é substituída por uma estrutura plana, onde todos os estados possuem o mesmo nível hierárquico. Para a obtenção do modelo de uso, a distribuição de probabilidade do uso esperado do *software* precisa ser determinada. Detalhes sobre a construção dos diagramas e modelo de uso são descritos em [RPG03].

Dentre os diagramas que permitem a especificação comportamental dos casos de uso, optou-se

pela utilização de diagramas de estado. As razões que amparam essa decisão são:

- Os diagramas de estados abrangem todos os possíveis cenários para um dado objeto e é apropriado para a transcrição direta de um modelo de uso. De qualquer modo, o escopo resultante do modelo de uso será limitado pela classe correspondente, e ainda pelo teste unitário.

- Formam uma base compreensiva para a geração de código, sendo isomórfico ao programa, facilitando então a detecção de erros durante a análise do modelo dinâmico.

- Ainda em combinação com a geração de código, diagramas de estados permitem a *animação* das transições de estado capturada durante a execução de um modelo, aproximando-se de testes do tipo caixa-branca.

- Através da utilização de ferramentas específicas, diagramas de estados são a base para a automação de *model checking*, contanto que determinadas restrições possam ser definidas.

Portanto, o foco desse trabalho será a geração de SAN a partir de diagramas de estados da UML, buscando a geração de casos de teste de *software*. Nas próximas seções é feita a especificação do *framework* para a transcrição de diagramas de estados UML para uma estrutura equivalente em SAN, bem como a possível análise proporcionada com a utilização de SAN na geração de casos de teste de *software*, viabilizando a alternativa sugerida na Figura 4.3.

4.2. *Framework* para a Transcrição de Diagramas de Estado para SAN

Aqui serão descritos os termos e conceitos de um diagrama de estados contemplados na especificação da UML 2.0 [OMG06], os elementos utilizados na sua construção, bem como suas propriedades, e para cada um deles a estrutura equivalente em SAN, buscando assim fundamentar a transcrição de diagramas de estado para SAN, sem perda de informações necessárias para a geração de casos de teste de *software*.

Um diagrama de estados descreve uma máquina de estados, dando ênfase ao fluxo de controle de um estado para outro, especificando as seqüências de estados pelos quais um objeto passa durante seu tempo de vida em resposta a eventos, juntamente com suas respostas a esses eventos [BRJ00]. São utilizados para modelar os aspectos dinâmicos de um sistema, classe ou casos de uso. Dessa forma, a modelagem de objetos reativos[1], cujo comportamento é bem caracterizado pela sua resposta a eventos externamente ao seu contexto, se torna possível.

A descrição detalhada de cada um desses elementos, assim como o método de transcrição para um estrutura equivalente em SAN, será apresentada no decorrer dessa subseção.

[1] orientados por eventos

4.2.1. Transições

Uma transição é um relacionamento entre dois estados[2], indicando que o objeto do primeiro estado realizará certas ações e entrará no segundo estado quando um evento especificado ocorrer e as condições especificadas forem satisfeitas. Sendo assim, uma transição é composta por um evento de ativação, uma condição de guarda e uma ação, ligando assim um estado de origem a um estado destino. A sintaxe de uma transição é dada a seguir:

$$< \texttt{evento de ativação} > [< \texttt{condição de guarda} >] / < \texttt{ação} >$$

Tanto em diagramas de estado quanto em SAN, uma transição pode ter mais de um evento associado. No entanto, podem existir situações, em diagramas de estado, onde a discriminação do evento de ativação pode ser suprimida.

Segundo a definição da UML 2.0, podem existir três tipos de transições:

- Local: se uma transição ocorre entre estados internos de um estado composto, ela é dita local.
- Externa: sugere que a transição, caso seja disparada, tem com alvo (ou fonte) um estado externo ao estado composto, podendo o próprio estado composto ser esse alvo (autotransição).
- Interna: é um caso especial onde uma transição, caso seja disparada, ocorre sem causar uma mudança de estado. Ela pode ser modelada de forma que as transições ocorridas sejam explicitadas por um estado composto, através de transições locais.

Para modelar essas transições em SAN, partiremos do pressuposto que todas as transições internas serão modeladas como uma transição local, e que qualquer máquina de estados seja tratada como um estado composto, buscando assim atender às definições descritas. Isso é feito da seguinte forma: quando um diagrama de estados for utilizado para modelar um caso de uso, será criado um *super-estado* que englobará essa máquina, sendo esse super-estado denominado de acordo com o nome do caso de uso.

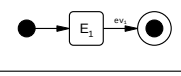
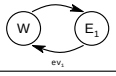
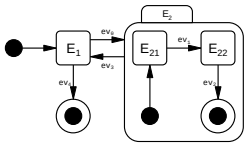
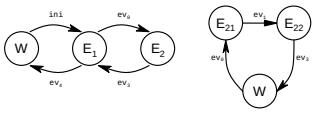
Essa condição garante a utilização de um caso de uso, como um caso de uso incluído. Sendo assim, sempre que um caso de uso incluído fosse requerido, esse apontaria para o super-estado que o modela. Como a ação executada pelo disparo da transição não é utilizada na análise da SAN, ela é abstraída no instante de nomear a transição da SAN. A Tabela 4.1 apresenta um resumo da notação utilizadas para descrever os tipos de transições disponibilizadas em diagramas de estados, as quais serão utilizadas nesse trabalho

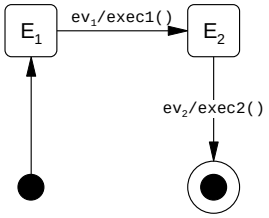
A seguir são descritos em detalhes os tipos de transições utilizadas na transcrição de diagramas de estados UML para SAN, bem como a transcrição propriamente dita, apresentando então a estrutura equivalente em SAN.

Transições locais As transições locais ocorrem internamente em um estado composto. Elas são tratadas como transições locais na SAN que modela esse estado composto. Isso é ilustrado na Figura 4.4

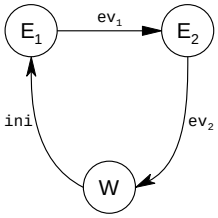
[2] a definição de estado e suas variantes, são apresentados na subseção 4.2.2.

Tabela 4.1 - Transições de um diagrama de estados e sua estrutura em SAN

Transição Local		
Transição Externa		



(I)



(II)

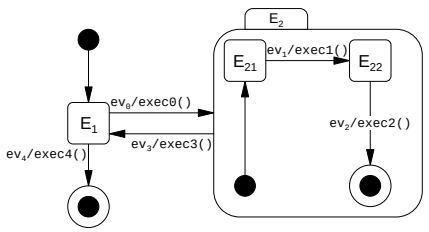
Tipo	Evento	Taxa
loc	ini	1
loc	ev_1	1
loc	ev_2	1
loc	reset	1

Figura 4.4 - Transições locais.

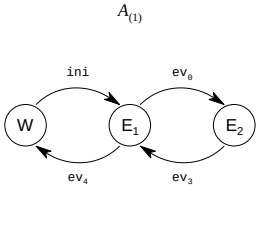
Para manter a estacionaridade do sistema, quando um estado final é atingido, essa transição é sincronizada com a transição externa que deixa esse estado composto. Na SAN, a transição que aponta para o estado final, na máquina de estados, aponta para o estado W (conforme descrito adiante) e nele permanece até uma nova execução.

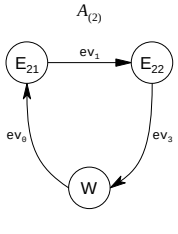
Transições externas As transições externas são modeladas com o auxílio de eventos sincronizantes em SAN. Conforme será descrito nesse relatório, para efeitos de legibilidade e modularização, um estado composto sempre será modelado em um autômato distinto, e toda a vez que uma transição tiver esse como alvo (ou fonte), sua modelagem em SAN será feita com o auxílio de eventos sincronizantes.

Para melhor elucidar essa idéia, um exemplo é apresentado na Figura 4.5. Nele é mostrado uma máquina de estados (I) que possui um estado composto (E_2). Quando o estado E_1 está apto a disparar uma transição pra E_2, ativada pelo evento ev_0, um evento sincronizante (ev_0) na SAN (II) modela essa transição.



(I)





Tipo	Evento	Taxa
syn	ini	1
syn	ev_0	α
loc	ev_1	1
loc	ev_2	1
syn	ev_3	1
loc	ev_4	α–1
loc	reset	1

(II)

Figura 4.5 - Transições externas.

O evento ev_0 ativa a transição do estado E_1 para E_2 (em $A_{(1)}$), e do estado W para E_{21} (em $A_{(2)}$), simultaneamente. Quando o autômato $A_{(1)}$ está em E_2, a execução desse estado composto é modelado pelo autômato $A_{(2)}$.

Como será descrito adiante, uma transição que parte de um estado inicial não possui um evento de ativação (estado transiente). Em SAN, a transição que parte de um estado inicial será sincronizada com o evento externo que ativa a execução do estado composto. No exemplo da Figura 4.5, o evento `ini` (autômato $A_{(1)}$) é sincronizado com o evento externo que ativa a execução da máquina de estados. Esse evento externo não é mostrado no exemplo.

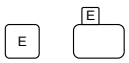
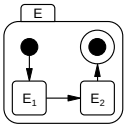
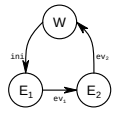
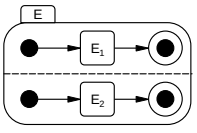
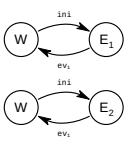
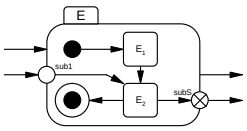
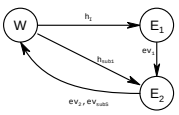
Sendo assim, a modelagem de uma transição externa para um estado composto em SAN, é feita com o sincronismo dessa transição externa com a transição que parte do estado inicial dessa estado composto. Esse comportamento é exemplificado na Figura 4.5, através do evento $\mathtt{ev}_0$.

De forma similar, quando o autômato $A_{(2)}$ dispara a transição do estado E_{22} para W, isso é feito com o auxílio de um evento sincronizante ($\mathtt{ev}_3$) que habilita a transição do estado E_2 para E_1 em $A_{(1)}$. Quando um estado composto é totalmente executado, o evento $\mathtt{ev}_3$ dispara a transição externa do estado E_2 para E_1.

4.2.2. Estados

Num diagrama de estado da UML, um estado é uma condição ou situação na vida de um objeto durante a qual ele satisfaz alguma condição, realiza alguma atividade ou aguarda algum evento. Um estado pode possuir um nome que o diferencia dos demais, ações de entrada e saída, transições internas que são manipuladas sem alterar o estado, e subestados (descritos em estados compostos). A Tabela 4.2 apresenta resumidamente as notações dos diferentes tipos de estados que podem compor um diagrama de estados UML e sua estrutura equivalente em SAN.

Tabela 4.2 - Estados de um diagrama de estados e sua estrutura em SAN

Estado Simples		
Estado Inicial		
Estado Final		
Estado Composto		
		
Estado Sub-máquina		

A seguir são descritos em detalhes os diferentes tipos de estados disponíveis em um diagrama de estados da UML, incluindo a descrição dos pseudo-estados, e o método utilizado para transcrever cada um desses em uma estrutura equivalente em SAN.

Estado Simples (simple state) É a forma elementar de representação de um estado. É aquele que não

contém subestados, ou seja, não possui regiões ou sub-máquinas de estado. A notação de um estado simples é feita por um retângulo com os cantos arredondados, podendo possuir seu nome escrito internamente ou em uma aba localizada na parte superior do estado, conforme ilustrado na Figura 4.6 (I).

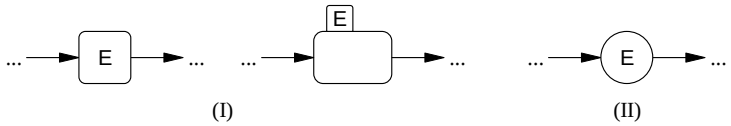


Figura 4.6 - Estado Simples.

Na mesma figura (II), temos a representação gráfica desse estado em SAN.

Estado Inicial (*initial state*) Esse estado indica que o início da execução de uma região foi acionada. Se uma máquina de estados está contida nessa região, significa que a mesma foi acionada, implicando no início do contexto do objeto da máquina de estados. Uma transição que parte de um estado inicial não possui um evento agregado a si, e deve sempre ser direcionada ao primeiro estado máquina. Num diagrama de estados, um estado inicial é denotado como um pequeno círculo preenchido, conforme observado na Figura 4.7 (I).

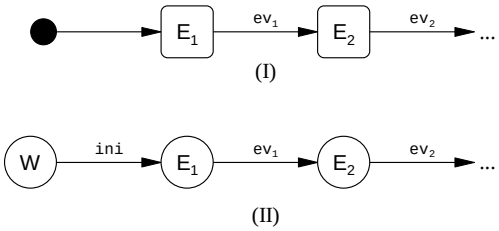


Figura 4.7 - Estado inicial.

Em SAN, um estado inicial será denotado como um estado W. Enquanto uma máquina de estados não é invocada, o autômato que modela essa máquina vai permanecer no estado W. Quando a execução dessa máquina de estados for habilitada, uma transição parte de W para o primeiro estado da máquina. A transição que parte de um estado W possui um evento que é sincronizado com a transição externa (evento `ini` nesse exemplo) que habilita a execução da máquina de estados. A Figura 4.7 (II) não explicita esse evento externo.

Estado Final (*final state*) Um estado final modela o encerramento de uma máquina de estados, indicando que a execução da mesma foi completada. Se uma máquina de estados está contida numa região, e todas as outras regiões desse mesmo estado também foram completadas (atingindo um estado final), significa que a máquina de estados foi encerrada, implicando no fim do contexto do objeto da máquina de estados. A Figura 4.8 (I) mostra um estado final, indicado como um círculo preenchido circunscrito. Como pode ser observado na mesma, mais de uma transição pode ser dirigida para um estado final.

Não há a necessidade da utilização de um estado que modele um estado final em SAN. Quando, no diagrama de estados, uma transição aponta para um estado final, o autômato

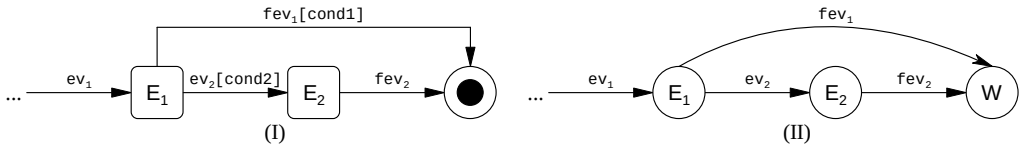


Figura 4.8 - Estado final.

que o modela possuirá um transição aponta para o estado W, tendo o seu evento sincronizado com o evento da transição externa que deixa essa máquina de estados. Essa configuração pode ser observada na Figura 4.8 (II).

Estado Composto (*composite state*) Um estado composto pode conter uma única região, contendo subestados disjuntos seqüencialmente ativos, ou ser decomposto em duas ou mais regiões ortogonais, cada uma possuindo subestados concorrentemente ativos. Esse segundo tipo permite especificar duas ou mais máquinas de estados que são executadas em paralelo no contexto do objeto que as contém. A Figura 4.9 mostra os possíveis arranjos de um estado composto (seqüencial (I) e composto (II)).

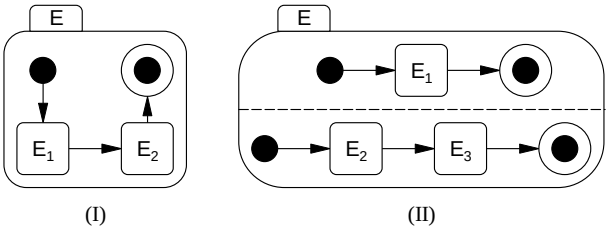


Figura 4.9 - Estado composto (seqüencial e concorrente).

Em determinadas situações pode ser conveniente esconder a decomposição de um estado composto. Esse é o caso onde um estado composto possui muitos estados aninhados, o que pode dificultar a legibilidade do diagrama. Nesses casos é inserido um símbolo no canto inferior direito, indicando que esse é um estado composto ocultado (Figura 4.10).

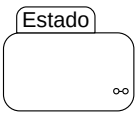


Figura 4.10 - Estado composto ocultado.

Como explanado anteriormente, por motivos de legibilidade e modularização, todo o estado composto deve ser modelado em um autômato distinto em SAN. As estruturas de autômatos que modelam os estados compostos da Figura 4.9 são mostradas na Figura 4.11 (seqüencial em (I) e concorrente em (II))

O autômato $A_{(1)}$ modela um estado composto seqüencial. Os eventos `ini` e $\texttt{ev}_2$ são sincronizados com os eventos das transições que chegam e partem desse estado composto, respectivamente. De maneira semelhante, os autômatos $A_{(2)}$ e $A_{(3)}$ modelam um estado composto concorrente. Da mesma forma, os eventos `ini` e $\texttt{ev}_2$ são sincronizados com os eventos das transições que chegam e partem desse estado composto, respectiva-

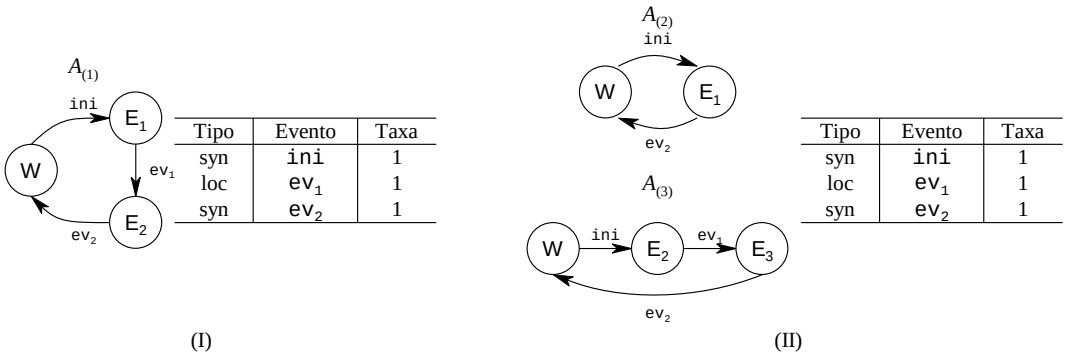


Figura 4.11 - Estrutura da SAN que modela um estado composto (seqüencial e concorrente).

mente, e também são sincronizados entre as diferentes regiões ortogonais desse estado. Respeitando as definições de SAN, quando uma transição externa alcança o referido estado composto, os eventos `ini` das regiões ortogonais são disparados simultaneamente. Finalmente, o estado composto só poderá habilitar uma transição externa quando todos os estados poderem disparar as transições para W.

Para amparar o sincronismo necessário ao finalizar diferentes regiões ortogonais, estabelece-se a seguinte condição: ao nomearmos os eventos da SAN que apontam para W, não há necessidade de manter a coerência com os nomes dos eventos do diagrama de estados, uma vez que o evento que sincroniza o encerramento das regiões do estado composto deve ser o mesmo nome. Como pode ser observado no exemplo anterior, os eventos que atingem os estados finais das regiões ortogonais podem possuir nomes diferentes, porém em SAN eles possuirão o mesmo nome ($\texttt{ev}_2$ na figura 4.11 (II))

Sobre os estados compostos ocultados: como esses são utilizados para abstrair um estado composto, facilitando assim legibilidade, não apresentam nenhuma informação pertinente para a construção de uma estrutura equivalente em SAN. Nesse caso, sugere-se a explicitação desse estado e que sua modelagem em SAN seja feita conforme descrito anteriormente.

Estado Sub-máquina (*submachine state*) Um estado do tipo sub-máquina contém a especificação da máquina de estados que é referenciada. Esse estado é semanticamente equivalente a um estado composto, porém podem conter pontos de entrada e saída inseridos como alvo e fonte desse tipo de estado. Esses pontos de entrada e saída são denotados na borda do estado sub-máquina, sendo essa desenhada como um estado normal, onde o nome do estado possui a seguinte sintaxe:

$$< \texttt{nome do estado} > : <$$
$$\texttt{nome da máquina de estados referenciada} >$$

Um estado sub-máquina pode ser invocado diversas vezes, podendo cada uma dessas chamadas acessar um diferente estado da sub-máquina, sendo esse acesso efetuado através

de distintos pontos de entrada. A Figura 4.12 ilustra essa situação. Ela modela um estado sub-máquina que pode ser acessada de diferentes formas e pode produzir diferentes eventos, de acordo com a sua execução.

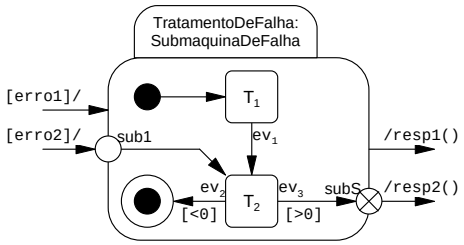


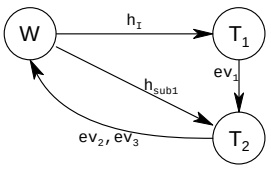
Figura 4.12 - Estado sub-máquina.

Esse estado sub-máquina (SubmaquinaDeFalha) modela o tratamento de um determinado erro. Nesse exemplo o estado pode ser acessado de acordo com a transição disparada para o mesmo. Caso o erro gerado seja do tipo 1, ele será encaminhado pela transição `erro1/`, e seu tratamento será feito a partir do estado inicial da máquina referenciada. Caso o erro gerado seja do tipo 2, o acesso ao estado é feito pelo ponto de entrada sub1, e o tratamento do erro é efetuado a partir estado T_2 da sub-máquina. Assim como um estado inicial, uma transição que parte de um ponto de entrada não possui um evento agregado.

De forma similar, se um determinado erro é tratado antes da execução completa da máquina de estados, a sub-máquina pode ser deixada através do ponto de saída subS. Note que a uma condição de guarda verifica se o erro foi tratado no estado T_2, habilitando a transição pelo disparo do evento ev_3, podendo então habilitar a transição externa (que executa `resp2()`) a partir de subS. Caso contrário, a sub-máquina é executada até seu estado final e a transição externa prevista nesse caso é disparada, executando `resp1()` nesse exemplo.

Para a modelagem de um estado do tipo sub-máquina em SAN, um estado W é utilizado como fonte de todas as transições iniciais da sub-máquina. Dessa forma, quando a sub-máquina de estados não estiver sendo executada, a SAN que a modela permanecerá no estado W. Dependendo da forma de acesso à essa sub-máquina, o estado W habilitará uma determinada transição, sendo essa sincronizada com a transição externa que acessa essa sub-máquina.

A utilização do estado W abstrai a necessidade da utilização de estados que modelem o início da execução da sub-máquina. A Figura 4.13 apresenta a estrutura da SAN que modela a sub-máquina descrita na Figura 4.12



Tipo	Evento	Taxa
syn	h_I	ρ
syn	h_{sub1}	τ
loc	ev_1	1
syn	ev_2	α
syn	ev_3	β

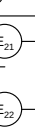
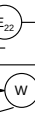
Figura 4.13 - SAN que modela a sub-máquina descrita na Figura 4.12.

Os eventos h_I e h_{sub1} são sincronizados com os eventos externos que alcançam a sub-máquina, sendo que suas respectivas taxas de ocorrência são funções que dependem dos eventos externos.

Os eventos ev_2 e ev_3 são sincronizadas com as transições externas que partem da sub-máquina. Esse eventos apontam para o estado W, e nele ficam bloqueados até que a sub-máquina seja acessada novamente.

Pseudo-estados (*pseudo-states*) Pseudo-estados são abstrações que abrangem diferentes tipos de vértices transientes de um diagrama de estados. Um resumo das notações dos pseudo-estados disponibilizados na UML, e sua estrutura equivalente em SAN, é apresentada na Tabela 4.3.

Tabela 4.3 - Estados de um diagrama de estados e sua estrutura em SAN

Pseudo-estado Inicial		
Estado de Histórico Superficial		Não possui estrutura equivalente em SAN
Estado de Histórico Profundo		Não possui estrutura equivalente em SAN
Ponto de Entrada		
Ponto de Saída		
Junção		
Escolha		
Finalização		
Fork		
Join		

A semântica de cada pseudo-estado depende de como esse atributo é utilizado, sendo esses descritos a seguir.

- Pseudo-estado inicial (*initial pseudo state*): representa um vértice padrão que inicia a transição para o primeiro estado de um estado composto. Possui a mesma semântica que rege um estado inicial convencional, conforme descrito anteriormente. Um pseudo-estado inicial é apresentado na Figura 4.14 (I) como um fragmento de um estado composto.

  Como descrito na definição de um estado inicial, a transição que parte de um pseudo-estado inicial não possui evento agregado. Em SAN esse evento é sincronizado com o evento externo que alcança o estado composto (Figura 4.14 (II)).

- Estado de histórico superficial (*shallow history*): permite a existência de um estado

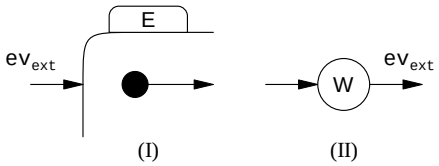


Figura 4.14 - Pseudo-estado inicial.

composto contendo subestados seqüenciais para lembrar o último estado nele ativo antes da transição do estado composto.

Um exemplo de aplicação desse tipo de estado é utilizado é apresentado na Figura 4.15. Se um estado composto que contém um estado de histórico superficial (estado H) é executado pela primeira vez, ou se na última execução desse estado composto o seu estado final foi atingido, não haverá histórico armazenado. Nesses casos, o estado de histórico funcionará como um pseudo-estado inicial.

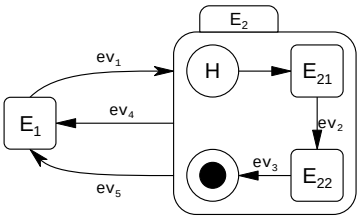


Figura 4.15 - Estado de histórico superficial.

Quando o estado final for alcançado, o evento $\mathtt{ev}_5$ é ativado. Se por algum motivo a execução do estado composto tiver que ser interrompida (para tratar alguma exceção, por exemplo) o estado estado interno em que se encontrava será armazenado e o evento $\mathtt{ev}_4$ será disparado. Na próxima vez que o estado composto for acessado, ele continuará a execução a partir do estado armazenado no histórico.

Em SAN não existe uma estrutura que modele um estado de histórico superficial. Nesse caso, sugere-se que a modelagem do estado composto que possui um estado de histórico possua mecanismos que possibilitem o acesso a um determinado estado, a partir do estado W da SAN, e que de alguma forma essa transição seja guiada em função do estado em que a SAN se encontrava anteriormente. Para isso, a utilização de autômatos auxiliares pode ser necessária.

- Estado de histórico profundo (*deep history*): similar ao estado de histórico superficial, porém esse permite lembrar o último estado ativo em qualquer nível de aninhamento, o que não é possível com o estado descrito anteriormente, o qual lembra somente o estado aninhado mais externo.

  A Figura 4.16 mostra um exemplo de utilização de um estado de histórico profundo (H$*$). A sistemática é a mesma empregada no estado de histórico superficial, porém são registrados inclusive os estados mais internos. Nesse exemplo, caso o evento $\mathtt{ev}_9$ seja disparado enquanto a máquina de estados se encontre em $\mathtt{E}_{22}$, esse estado será armazenado. Quando o estado $E2$ for acessado novamente, a execução da máquina

de estado continuará a partir de E_{22}. Se o estado E_2 alcançar seu estado final, o evento ev_{10} será disparado e nenhum estado será armazenado no histórico.

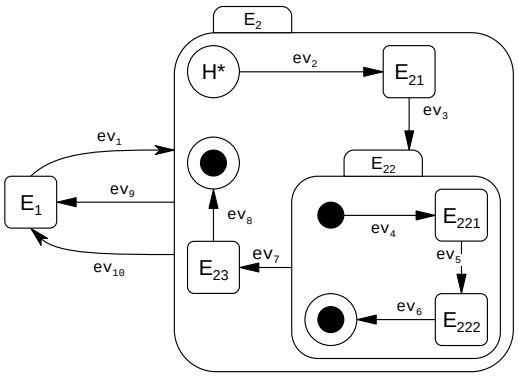


Figura 4.16 - Estado de histórico profundo.

Segundo a definição de estados de histórico, podem existir vários desses incluídos em diferentes níveis de aninhamento de um mesmo estado composto.

Assim como num estado de histórico superficial, não existe uma estrutura que modele um estado de histórico profundo em SAN. Nesse caso, sugere-se a mesma estratégia apresentada anteriormente.

- Ponto de entrada (*entry point*): é um pseudo-estado que modela a entrada em um estado composto. Um ponto de entrada é exemplificado na Figura 4.17 (I) como um fragmento de um estado composto.

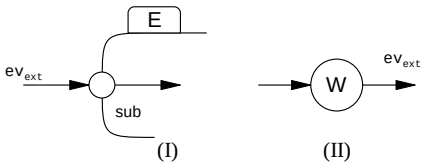


Figura 4.17 - Ponto de entrada.

Conforme descrito na definição de uma sub-máquina, um ponto de entrada não possui um evento de ativação na transição que ele habilita. Em SAN esse evento é sincronizado com o evento externo (ev_{ext}) que alcança o estado composto (Figura 4.17 (II)). Nesse caso é utilizado um estado W, e o evento que parte desse estado é sincronizado com o evento externo que atinge o ponto de entrada.

- Ponto de saída (*exit point*): é um pseudo-estado que modela a saída de um estado composto. Um ponto de saída é apresentado na Figura 4.18 (I) como um fragmento de um estado composto.

  Assim como descrito na definição de estados do tipo sub-máquina, para modelar o comportamento de um ponto de saída em SAN, a transição que chega no ponto de saída (sub_S no diagrama de estados desse exemplo) apontam para um estado W da SAN, e nele fica bloqueado até que a sub-máquina seja acessada novamente. O evento que dispara essa transição é sincronizado com o evento externo que deixa a

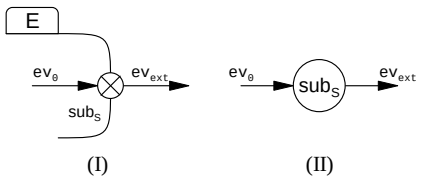


Figura 4.18 - Ponto de saída.

sub-máquina.

- Junção (*junction*): são vértices livres de semântica usados para juntar múltiplas transições. Eles constroem uma nova transição através da união de diferentes transições oriundas de diferentes estados (*merge*), cada qual com uma diferente condição de guarda, ou separam uma transição em várias (*branch*), onde cada qual possui uma determinada condição de guarda. Isso realiza uma ramificação condicional estática, onde uma transição é habilitada se todas as outras forem avaliadas como falsas. É apresentado na Figura 4.19 (I) um exemplo de junção.

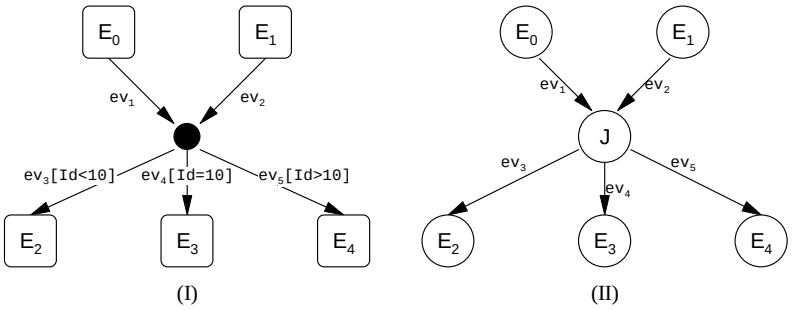


Figura 4.19 - Junção.

Em SAN um comportamento equivalente é obtido com a inserção de um estado (J) que auxilia na modelagem dessa situação. Isso pode ser observado na SAN exemplificada na figura acima (II).

- Escolha (*choice*): o alcance desse vértice resulta em uma ramificação condicional dinâmica, onde caso mais de uma condição seja avaliada como verdadeira, uma delas é escolhida arbitrariamente. A notação desse pseudo-estado é mostrada na Figura 4.20, bem como a sua estrutura equivalente em SAN, através da inserção de um estado que modele esse comportamento (estado CH).

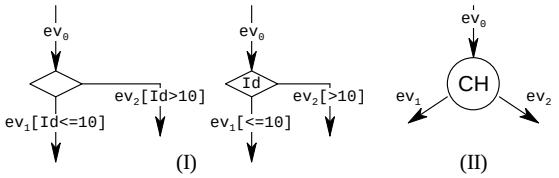


Figura 4.20 - Escolha.

- Finalização (*terminate*): implica no fim da execução da máquina de estado, de acordo com o contexto do objeto. A entrada nesse pseudo-estado é equivalente a uma chamada do tipo *DestroyObjectAction*.

Em SAN, não é necessaria a inserção de um estado especifico que modele uma finalização. Nesse caso, a transição que atinge o ponto de saída será sincronizada com a transição que deixa a máquina de estados. Para efeitos de modelagem, uma finalização em um diagrama de estados tem a mesmo comportamento de um estado final em SAN. Sendo assim, a transição que aponta para uma finalização em um diagrama de estados passa apontar para o estado W em SAN. Essa situação é exemplificada na Figura 4.21.

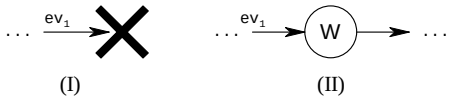


Figura 4.21 - Finalização.

- *Fork*: é uma notação opcional, utilizada para ramificar uma transição em várias, cada uma para diferentes regiões ortogonais de um estado composto. As transições originadas no *fork* não contém guardas ou eventos de disparo. Um exemplo dessa estrutura é apresentada na Figura 4.22 (I).

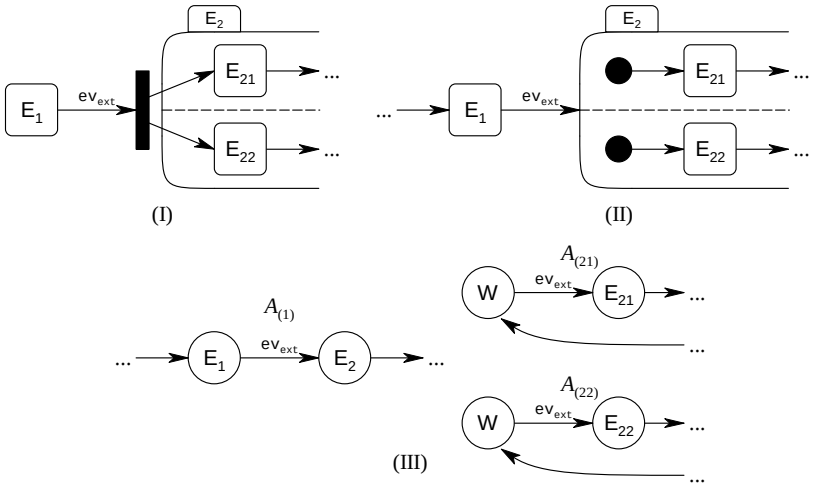


Figura 4.22 - *Fork*.

Esse pseudo-estado modela o início simultâneo de duas regiões ortogonais de um estado composto. Como visto anteriormente, isso pode ser modelado (e é sugerido) com a utilização de estados iniciais em cada região, conforme mostrado na Figura 4.22 (II).

Em SAN, um *fork* é modelado com a utilização de eventos sincronizantes que asseguram a execução simultânea de cada uma das regiões, sendo que cada região é modelada por um autômato distinto. Cada um desses autômatos possui um estado W no qual cada um desses se encontra antes da execução da região que esse modela. Essa estrutura é verificada na Figura 4.22 (III).

- *Join*: é utilizado para reunir diferentes transições originadas em regiões ortogonais de um estado composto. As transições que originam o *join* não contém guardas ou eventos de disparo. A Figura 4.23 mostra pseudo-estado do tipo *join*.

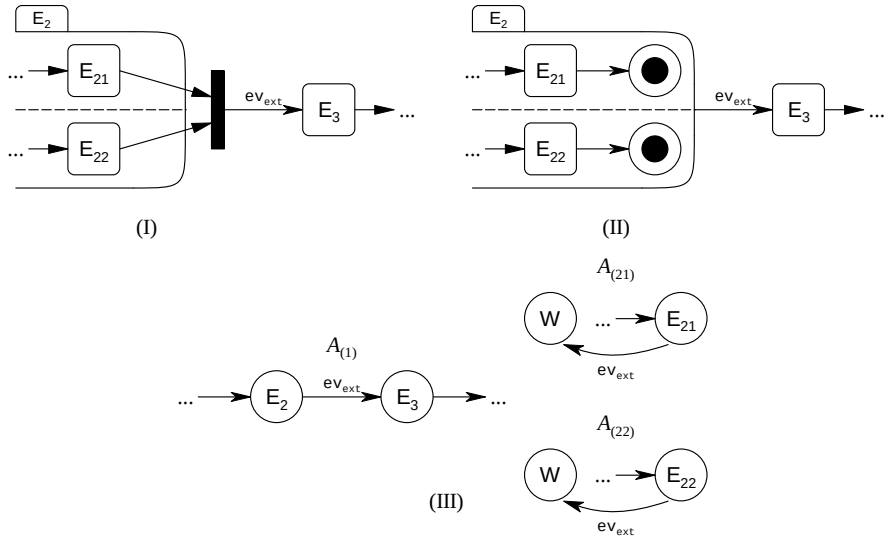


Figura 4.23 - *Join.*

Esse pseudo-estado modela o término simultâneo de duas regiões ortogonais de um estado composto. Conforme explanado anteriormente, isso pode ser modelado (e é sugerido) com a utilização de estados finais em cada região, como mostrado na Figura 4.23 (II).

Em SAN, um *join* é modelado com a utilização de eventos sincronizantes que asseguram o fim simultâneo de cada uma das regiões, sendo que cada região é modelada por um autômato distinto. Cada um desses autômatos possui um estado W, que também modela o início da maquina de estados, que é atingido pela transição que atinge o estado final de cada uma das regiões ortogonais. Esse comportamento é visto na Figura 4.22 (III).

Com a transcrição aqui formalizada pode-se traduzir um diagrama de estados da UML para uma estrutura equivalente em SAN. Para melhor elucidar o método proposto, um estudo de caso é apresentado no Capítulo 6.

4.2.3. Determinação das Taxas de Ocorrência

Não existe uma sistemática geral para a determinação das taxas de ocorrência de um evento derivado de um diagrama de estados. Sugere-se nesse caso a avaliação da estrutura da SAN, da especificação fornecida, que os detalhes do comportamento do sistema seja discutido com o cliente, ou que as taxas sejam atribuídas segundo a avaliação de um histórico de comportamento do sistema (ou de sistemas similares). Uma aproximação para a determinação sistemática de transições de probabilidade, usando funções objetivas de restrições derivadas da estrutura e uso do modelo de teste, é introduzido em [WP00]. Quanto mais fiel for essa avaliação maior será a qualidade do modelo de uso do sistema, que por sua vez infere na obtenção de casos de teste mais confiáveis.

4.2.4. Tratamento das condições de guarda

A presença de condições de guarda em determinadas transições sugere a inclusão de uma estrutura que avalie a restrição, para então permitir (ou não) o disparo dessa transição. Para melhor elucidar essa idéia, é apresentado na Figura 4.24 um trecho de um diagrama de estados que possui condições de guarda em suas transições:

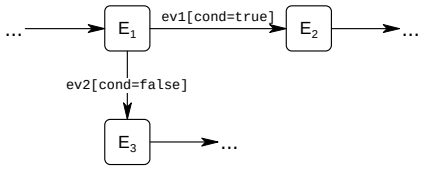


Figura 4.24 - Condição de guarda.

Para modelar essa situação em SAN, é necessário compor um autômato que simule as possíveis restrições impostas pelas condições de guarda. Essa situação é exposta na Figura 4.25. Nela, o autômato Aut_cond descreve o estado da restrição que é avaliada nas condições de guarda do diagrama de estados. Dependendo do estado desse autômato, a transição correspondente será habilitada de acordo a sua taxa funcional (*f1* 3 *f2* nesse exemplo).

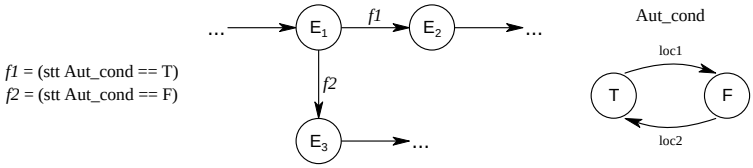


Figura 4.25 - SAN com avaliação de condição de guarda.

Como pode ser observado na figura, a transição que possui a taxa funcional *f1* somente será disparada se o autômato Aut_cond se encontrar no estado T. De forma similar, o disparo da transição entre E_1 e E_2 só ocorrerá se o autômato Aut_cond se encontrar em F.

4.3. Análises Propostas

De posse da SAN que representa o modelo de uso do sistema, é possível efetuar uma análise comportamental do sistema como um todo, ou de uma parte específica do mesmo. Essa análise é efetuada com a execução desse modelo na ferramenta PEPS2003, onde informações que possam auxiliar no planejamento do sistema, como alocação de recursos ou estimativa de tempo, podem ser extraídas.

Como os diagramas de estados são concebidos antes da fase de implementação e de testes, as informações extraídas na análise comportamental do sistema pode ser utilizada no auxílio de tomada de algumas decisões, dependendo do critério utilizado. Por exemplo, na avaliação da SAN pode ser determinado qual o autômato com maior probabilidade de execução, indicando qual o setor do sistema que será executado com maior freqüência. Se esse for um critério utilizado pra a determinação da alocação de recursos, o analista pode determinar que os melhores programadores (ou testadores)

sejam responsáveis por esse setor do sistema. Como relatado anteriormente, a SAN gerada representa o modelo de uso do sistema, o que serve de base para a geração de casos de teste.

4.3.1. Análise do Sistema

A análise do sistema é obtida com uma SAN que descreva o comportamento de todos os casos de uso do sistema, e a interação desses com os atores. Sendo assim, somente os diagramas de estados não são suficientes para compor essa análise. As informações sobre a interação dos atores com o sistema são obtidas no diagrama de casos do uso. Essa idéia é apresentada na Figura 4.26.

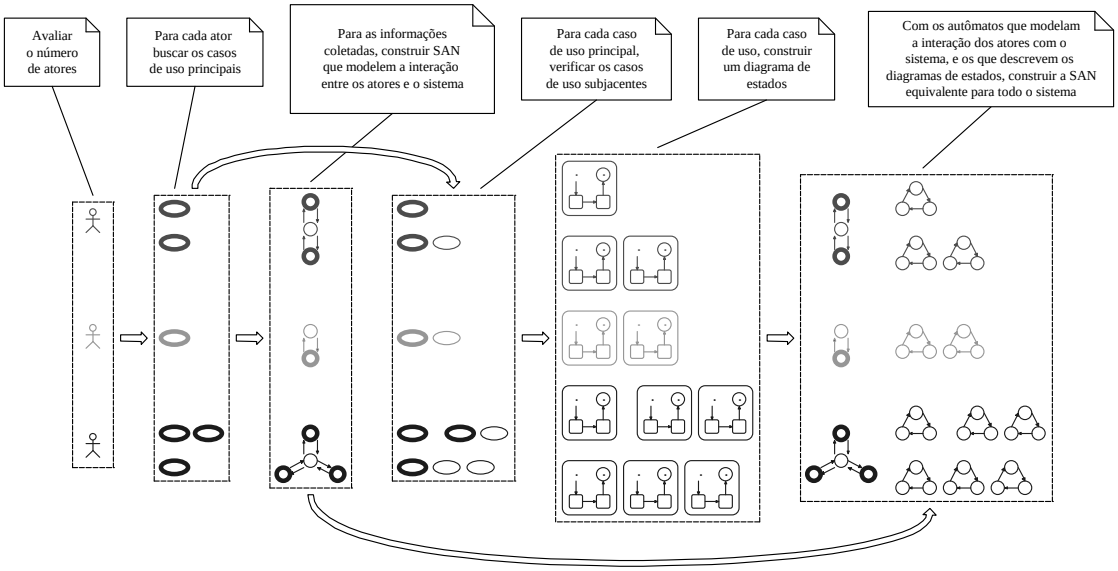


Figura 4.26 - Análise de sistema.

Inicialmente, é necessário identificar os atores do sistema. Com isso, busca-se os casos de uso principais, aqueles que possuem associação direta com um ator do sistema, indicando assim uma interação. A interação entre um ator e seus casos de uso é modelada por um autômato que traduza essa associação, conforme representado na Figura 4.27.

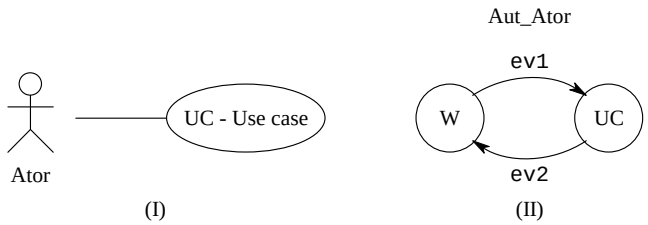


Figura 4.27 - Interação entre um ator e um caso de uso.

Nessa análise, toda a associação entre um ator e um caso de uso (figura (I)) é desdobrada em duas transições no autômato que modela essa associação (figura (II)), e essas transições possuem eventos sincronizantes que habilitam as transições que iniciam e concluem a execução do autômato que modela o caso de uso. Ou seja, o evento `ev1` do autômato Aut_Ator será sincronizado com o evento que parte do estado W do autômato que modela o caso de uso. De forma similar, o evento `ev2` será sincronizado com o evento que atinge o estado W do autômato que modela o caso de uso.

Dos casos de uso principais (que possuem associação direta com um ator) busca-se os casos de uso incluídos e estendidos dos mesmos. Com a determinação de todos os casos de uso do sistema, cada um será *explodido* em um diagrama de estados que descreve o seu comportamento. Com a especificação feita nas seções anteriores, a partir de cada um dos diagramas de estados será gerado um autômato equivalente. Por fim, esses autômatos serão integrados com os autômatos que modelam a interação entre os atores e o sistema, sendo possível assim obter uma SAN que modele o comportamento do sistema como um todo.

4.3.2. Análise do Componente

Definiremos aqui como análise do componente a análise proporcionada pela SAN que modela o comportamento de um caso de uso que possui interação direta com um ator, bem como seu casos de uso subordinados, incluídos e estendidos. Aqui, a interação entre o sistema e o ator não é analisada. Esse método é descrito na Figura 4.28.

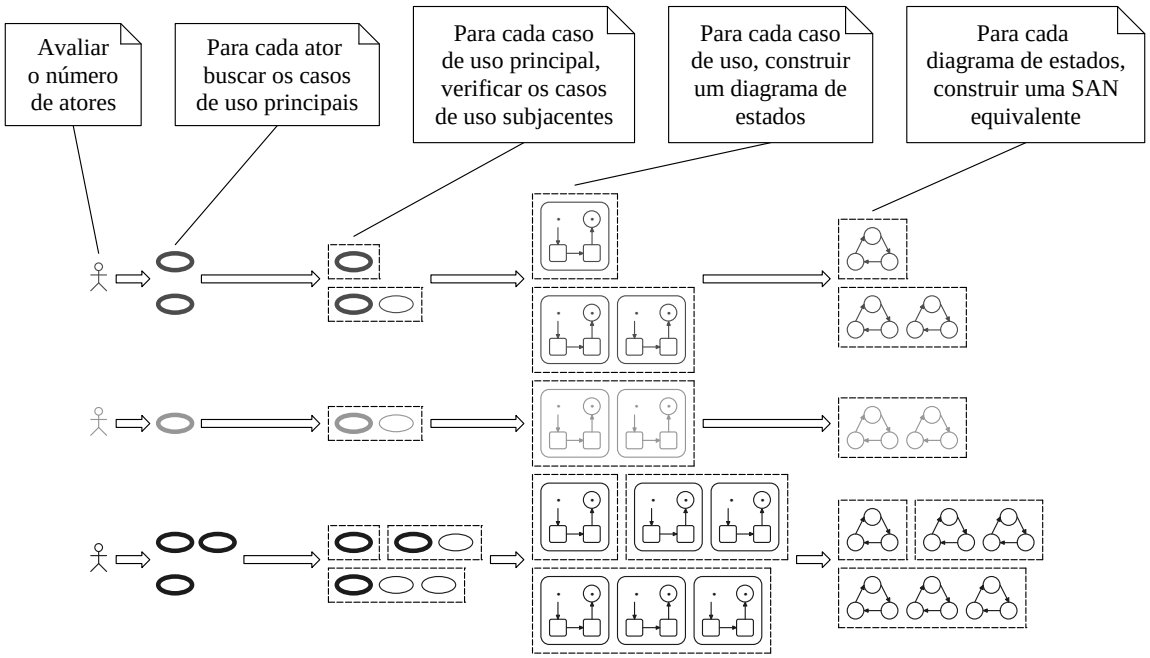


Figura 4.28 - Análise de componente.

Apesar de não serem utilizados nos cálculos que a análise viabiliza, os atores devem ser identificados com o intuito de encontrar os casos de uso principais. Cada caso de uso principal pode conter casos de uso subordinados, e o conjunto formado por esses irá compor uma SAN distinta, cada qual disponibilizando informações comportamentais de um componente do sistema. Os autômatos dessa SAN são construídos pela transcrição dos diagramas de estados que modelam o comportamento de cada um dos casos de uso do componente.

4.4. Simplificação da SAN Resultante

Durante a concepção desse trabalho, verificou-se que o modelo SAN construído pode apresentar um número de autômatos (e de estados) que inviabiliza a sua análise na ferramenta PEPS2003. Esse comportamento é observado na análise do sistema (apesar de certos componentes do sistema apresentarem um alto nível de complexidade), o que pode comprometer a análise pretendida.

Dessa forma, ao se obter a SAN que modela o comportamento do sistema, deve-se avaliar o espaço total de estados gerado. Isso é feito através do cálculo do produtório do número de estados do modelo SAN. Por exemplo: uma SAN que possui 5 autômatos, sendo que esses possuem 4, 3, 5, 3 e 5 estados, o espaço total de estados gerado será de 900 estados. O histórico de utilização da ferramenta PEPS2003 alerta para a inviabilidade de se analisar sistemas com número total de estados superior a 60.000.000 de estados, o que é comum em uma SAN gerada em uma análise de sistema.

Sendo assim, ao final da construção da SAN deve-se avaliar se o número total de estados gerados não ultrapassa o limiar suportado pela ferramenta PEPS2003. Caso isso aconteça, deve-se buscar uma SAN mais enxuta, com um número menor de autômatos e estados, mas que ainda possua informações que viabilizem a análise de cobertura do sistema. Essa simplificação é obtida da seguinte forma:

1. É mantida a modelagem dos atores que interagem com o sistema. Como a simplificação da SAN resultante será aplicada na análise do sistema, é necessário manter essa modelagem;

2. Serão mantidos os estados que influem diretamente na determinação do fluxo de dados do componente. Esses estados, que aqui serão denominados como *estados-chave*, são os seguintes:

   - Estado de espera (W): a partir desse estado é modelado o início da execução de uma funcionalidade do sistema. As transições que partem e chegam de um estado W possuem eventos sincronizantes, o que sugere a permanência desse estado em um modelo simplificado.
   - Estado de espera/sincronização (WS): esses estados são utilizados para modelar um bloco de sincronização, o qual se forma entre um *fork* e um *join*. Assim como em um estado W, as transições que partem e chegam em um estado WS possuem eventos sincronizantes, sendo então necessária a modelagem desse estado no novo modelo simplificado.
   - Estado de bifurcação: são estados de onde partem mais de uma transição, cada qual com uma possível condição de guarda. Como cada uma dessas possíveis transições inicia um trecho de código que precisa ser testado, pois compõe um caminho distinto dentro do código. Fica aqui definido que para toda a bifurcação presente em um autômato, haverá um estado que irá compor a confluência dos trechos iniciados na bifurcação. Essa confluência pode ocorrer, inclusive, no estado W.
   - Estado de confluência: é um estado que é atingido por mais de uma transição, determinando assim o fim de um trecho iniciado em um bifurcação.

- Estado que modela um caso de uso diferente: as transições que chegam e partem desse tipo de estado possui eventos sincronizantes, os quais modelam o inicio e o fim da execução do autômato que modela o caso de uso diferente. Essa situação sugere a manutenção desses estados no novo autômato em construção.

3. Primeiro nível de simplificação (N1): um trecho composto por estados seqüenciais, trecho esse iniciado após um estado-chave e que termina antes de um outro estado-chave, será substituído por um estado equivalente que representa o referido trecho. Em outras palavras, uma seqüencia de estados que apareça entre dois estados-chave será substituído por um estado equivalente. Para melhor ilustrar essa idéia, é apresentada a Figura 4.29.

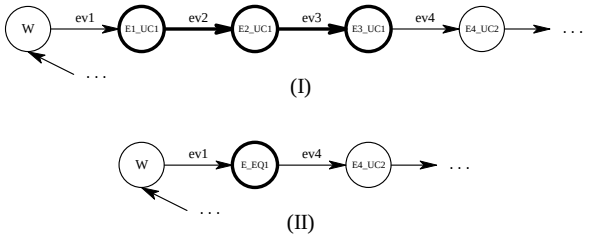


Figura 4.29 - Exemplo de primeiro nível de simplificação.

Essa figura exemplifica uma situação onde é aplicada uma simplificação de nível 1. Nesse trecho de um autômato, o qual modela a execução de UC1, podem ser observados dois estados-chave: W e E4_UC2 (um estado que modela um caso de uso diferente). Todos os estados seqüenciais entre esses estados-chave, em negrito na figura (I), serão substituídos por um estado equivalente, modelado como E_EQ1 na figura (II). Como também pode ser observado, os eventos anteriores e posteriores a esse *trecho seqüencial* são mantidos na nova estrutura.

Quase todas as combinações de estados-chave poderão marcar o início e o fim de um trecho seqüencial, o qual será substituído por um estado equivalente. Na Tabela 4.4 são mostradas as possíveis combinações de estados-chaves que delimitam o início (primeiro coluna da tabela) e o fim (primeira linha da tabela) de um trecho seqüencial.

Tabela 4.4 - Possíveis combinações de estados-chave que delimitam trechos seqüenciais.

		FIM				
		W	WS	Bifurcação	Confluência	UC≠
INÍCIO	W	✓	✓	✓	✗	✓
	WS	✓	✓	✓	✓	✓
	Bifurcação	✓	✓	✓	✓	✓
	Confluência	✓	✓	✓	✓	✓
	UC≠	✓	✓	✓	✓	✓

Como pode ser observado, a única combinação que não será composta é aquela que inicia um trecho seqüencial a partir de um estado W e termina antes de uma confluência. Isso é devido ao fato de, a partir um estado W, uma confluência sempre é precedida de uma bifurcação, em algum instante. Sendo assim, uma bifurcação sempre será encontrada antes de uma confluência, o que impede a composição de um trecho seqüencial entre um

estado W e uma confluência.

4. Segundo nível de simplificação (N2): aqui, os estados equivalentes serão eliminados, e uma transição direta entre os estados chave (transição equivalente) será composta. Essa simplificação é exposta na Figura 4.30.

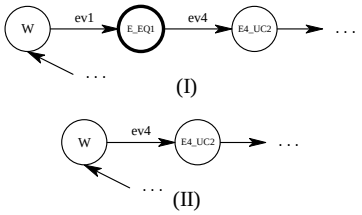


Figura 4.30 - Exemplo de segundo nível de simplificação.

A figura (I) mostra o estado equivalente gerado na Figura 4.30. Como foi explanado, uma nova transição é composta entre os estados-chave, sendo mantido o evento da transição que parte do estado equivalente (evento ev4, nesse exemplo).

Devido à nova estrutura gerada, algumas alterações adicionais podem ser necessárias. Na realidade, em um ultimo passo dessa simplificação, deve-se atentar para duas situações especiais, as quais precisam ser tratadas:

- Auto-transição: a eliminação de um estado equivalente, sendo esse substituída por uma única transição. Para melhor elucidar essa idéia, é apresentada a Figura 4.31.

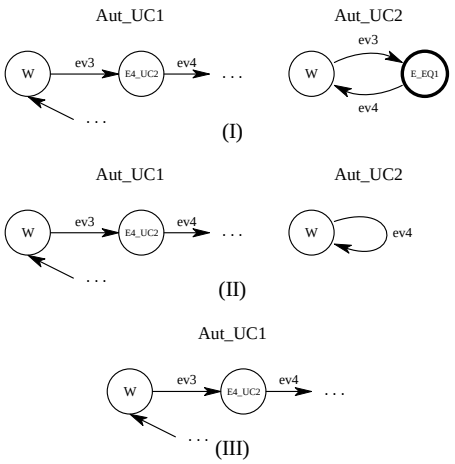


Figura 4.31 - Simplificação N2 - Tratamento de auto-transições.

Na figura (I), o autômato Aut_UC2 possui um estado equivalente que, ao ser eliminado na simplificação N2, produzirá uma auto-transição para o estado W, conforme pode ser observado na figura (II). Nessa situação, não há necessidade de manutenção do autômato Aut_UC2, sendo então possível eliminá-lo, assim como vislumbrado na figura (III). Nesse novo arranjo, os eventos ev3 e ev4, que antes era sincronizados com o início e o fim da execução de Aut_UC2, passam a ser eventos locais no Aut_UC1.

Nessa nova estrutura, o teste de execução do autômato Aut_UC2 passa a ser feito

pelo teste dos eventos ev3 e ev4. Nesses testes pode ser verificado, por exemplo, se a funcionalidade descrita por UC2 é corretamente invocada, e que essa produz um determinado resultado. Acredita-se que para um teste de sistema, essa abordagem seja suficiente, visto que testes mais refinados podem ser obtidos com autômatos construídos na análise de componente.

- Várias transições entre dois estados: outra situação que deve ser observada é aquela onde a simplificação N2, através da substituição dos estados equivalentes por transições, pode gerar múltiplas transições entre dois estados. Essa situação pode ser observada na Figura 4.32:

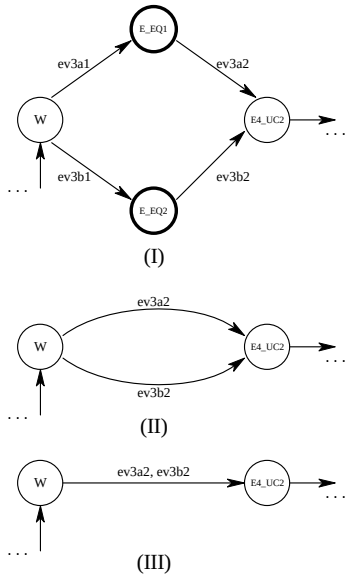


Figura 4.32 - Simplificação N2 - Tratamento de múltiplas transições entre dois estados.

A figura (I) mostra dois estados equivalentes que são eliminados na simplificação N2, dando origem ao arranjo apresentado na figura (II), onde duas transições distintas entre W e E4_UC2 são originadas. Essas duas transições devem ser substituídas por uma única, e essa deve conter os dois eventos que estavam associados às transições originais. Para testar essa transição, deve ser verificado se a mesma é disparada pela ativação de qualquer um dos dois eventos. Em outras palavras, um transição com mais de um evento deve possuir um teste para cada evento.

Seguindo os passos descritos, é possível construir uma SAN mais enxuta, mas que ainda apresenta as informações necessárias para a análise de cobertura de sistema como um todo.

4.5. Geração dos Casos de Teste

Com a concepção da SAN que representa o modelo de uso do sistema, que reflete uma descrição formal do comportamento do *software* a ser desenvolvido, é possível gerar testes estatísticos de *software*, conforme proposto por Bertolini [Ber05, BFF+04]. Para o seu trabalho foi aplicado o processo

de teste estatístico utilizado pelo Centro de Pesquisa em Teste de *Software* (CPTS) da PUCRS, onde um modelo formal do sistema, construído em MC ou SAN, é utilizado para gerar os casos de teste.

O CPTS desenvolveu um *framework* de automatização do processo de teste estatístico chamado *State Based Test Generator* (STAGE). Sua arquitetura é organizada em três módulos [BFF+04]:

STAGE-Model: é uma ferramenta que permite a modelagem de uma aplicação através da composição de um conjunto de estados e transições que remetam ao modelo comportamental da aplicação, sendo possível modelar o sistema em SAN, uma vez que a STAGE pode estar integrada a ferramenta PEPS2003. Cada passo do caso de teste é composto por um estado global da SAN, juntamente com o evento que dispara a mudança de estado.

Das restrições de modelagem que a ferramenta impõe [Ber05], a descrição comportamental do sistema (análise de sistema) composta a partir dos diagramas de estados não atribui um único estado para o início da execução do modelo, e STAGE necessita desse estado inicial. Sugere-se aqui que a atribuição desse estado seja feita manualmente pela interface do STAGE. Outras informações como entradas e saídas dos estados, não disponibilizada pela SAN, pode ser capturada na análise do diagrama de estados correspondente.

STAGE-Test: é responsável por gerar os casos de teste a partir dos dados fornecidos pela STAGE-Model. Como saída, a ferramenta disponibiliza uma *test suite*, a qual serve de base para a geração de *scripts* de execução automática de casos de teste.

STAGE-Script: com as informações disponibilizadas nos outros módulos do STAGE, são compostos *scripts* de teste, os quais podem ser inseridos no aplicativo *RobotJ* da Rational como um novo projeto, com o intuito de testar a aplicação.

Eventualmente, a SAN gerada pode não atender todos os requisitos necessários para a sua execução no STAGE. De qualquer forma, a adequação necessaria pode ser efetuada pela interface de edição do STAGE-Model, onde a definição do estado inicial e dos eventos do tipo mestre e escravo, requisitados pelo STAGE [Ber05].

Alternativamente, a geração de casos de teste de *software* pode ser dada pela execução de um caminho específico da SAN gerada, onde cada evento associado a uma transição deve ser testado. Para tanto, para cada uma das transições da SAN global deve ser atribuída a ação correspondente, indicada na transição do diagrama de estados.

No próximo capítulo é descrita o método utilizado para a construção do protótipo para geração de um arquivo SAN, a partir de informações fornecidas pelos diagramas construídos sob a abordagem o Processo Unificado, construídos através da ferramenta Rational Rose.

5. PROTÓTIPO UMLTOSAN

Esse capítulo descreve detalhes sobre a implementação do protótipo para a tradução dos diagramas UML gerados pela metodologia do Processo Unificado em uma estrutura equivalente em SAN. Essa descrição inicia com um apanhado geral sobre o contexto onde o protótipo vai atuar, sendo então especificado os detalhes da implementação do mesmo.

5.1. Descrição do Protótipo

Buscando a validação das atividades de pesquisa propostas nesse trabalho, foi proposta a construção de um protótipo para a construção de um arquivo .san a partir de um arquivo gerado pelo Rational Rose, seguindo a metodologia do Processo unificado, visando a análise do modelo de uso de um *software*, para a posterior geração de casos de teste.

A IBM disponibiliza em seu site um plug-in para a tradução dos arquivos gerados pelo Rose para uma estrutura XML (RoseXMLTools). Esse plug-in foi utilizado nessa pesquisa, possibilitando então a conversão de um arquivo gerado pelo Rose (*.mdl ou *.ptl) para uma estrutura XML equivalente, sendo esse utilizado para a construção da SAN. Visando a padronização da modelagem, o contorno de eventuais dualidades e a viabilização da construção da SAN, o padrão de construção de diagrama de estados UML, descrito no capítulo anterior, deve ser adotado.

Conforme relatado anteriormente, a análise de sistemas complexos pode ocasionar uma explosão de espaço de estados (acima de 60 milhões) na SAN gerada, o que inviabiliza a sua sua análise na ferramenta PEPS2003. Sendo que essa explosão pode ser tratada, seguindo os passos descritos anteriormente, essa ação será efetuada em um módulo diferente ao que traduz uma especificação UML para SAN. Em outras palavras, o protótipo proposto será dividido em dois módulos:

Módulo 1 - Tradução: nesse módulo ocorrerá a tradução do arquivo XML, sendo então construída uma SAN equivalente.

Módulo 2 - Simplificação: caso a tradução efetuada no Módulo 1 resulte em uma explosão do espaço de estados na SAN, a técnica de simplificação apresentada anteriormente deve ser aplicada.

5.1.1. Restrições de modelagem impostas pelo Rational e suas alternativas

Apesar de dispormos de toda a transcrição que traduz os elementos dos diagramas de estados da UML 2.0 para uma estrutura equivalente em SAN, a Rational Rose não disponibiliza todos esses elementos. Dessa forma, serão descritos aqui esses elementos, bem como uma alternativa de modelagem que assegure a construção

Pontos de entrada e saída em uma sub-máquinas de estados: esses pseudo-estados não são disponibilizados pelo Rational Rose. Para obtermos uma estrutura equivalente, é necessário efetuar uma pequena alteração no diagrama de estados que esta sendo construído. Uma

opção para um ponto de entrada é a utilização de um estado auxiliar, e dele será feita a transição desejada, dependendo da condição de guarda da transição que atinge a sub-máquina. Vale lembrar que apenas uma transição pode partir de um estado inicial, o que justifica a utilização de um estado auxiliar.

Para um ponto de saída, é sugerido que a transição que atingiria esse pseudo-estado aponte diretamente para o estado final da sub-máquina. A transição que partirá do pseudo-estado, que será habilitada a partir do alcance do estado final, dependerá do evento da transição que atinge o estado final. Em outras palavras, os eventos das transições que atingem o estado final da sub-máquina serão utilizados como condição de guarda das transições que partem da sub-máquina. Para melhor elucidar essa idéia, é apresentada a Figura 5.1.

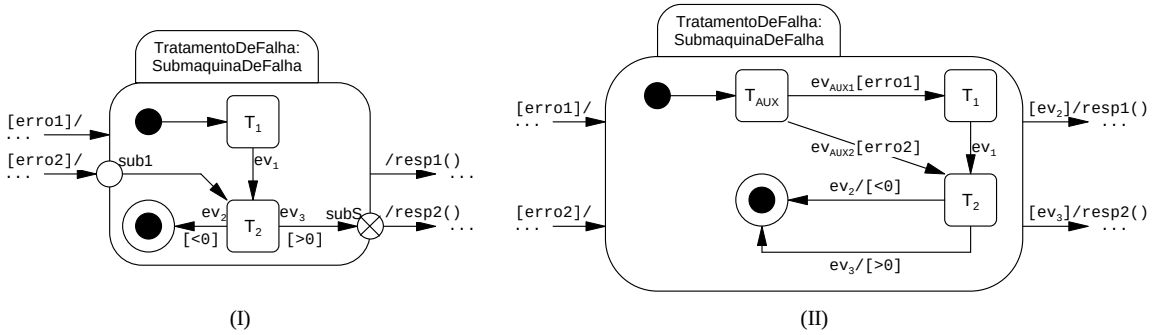


Figura 5.1 - Estrutura opcional para um estado do tipo sub-máquina.

Na nova configuração, as condições de guarda determinam qual a transição que vai apontar para o estado final (figura (II)), e o evento ocorrido nessa transição será utilizado como condição de guarda para a determinação de qual das duas transições será disparada pela sub-máquina. Dessa forma a sub-máquina continua sendo executada conforme a transição que a atinge e de acordo com a maneira como ela é finalizada.

Junção: apesar de constar na especificação da UML 2.0, o Rational Rose não disponibiliza pseudo-estados do tipo junção. Alternativamente, sugere-se que todas as transições que partem de uma junção sejam disparadas diretamente dos estados anteriores

à junção, sendo assim possível suprimir esse pseudo-estado, apesar do número de transições poder aumentar. Essa configuração é mostrada na Figura 5.2.

A junção apresentada na Figura (I) possui três transições (cada qual com uma condição de guarda específica) que são disparadas a partir dela. Com a supressão sugerida, essas transições serão disparadas de todos os estados anteriores à junção, ainda mantendo as condições de guarda especificadas anteriormente (Figura (II)).

Finalização: por também não ser disponibilizado um pseudo estado de finalização,

é sugerido aqui que todas as transições que apontam para uma finalização (indicando a destruição de um objeto, por exemplo) apontem para o estado final da máquina de estados. Essa situação é exemplificada na Figura 5.3.

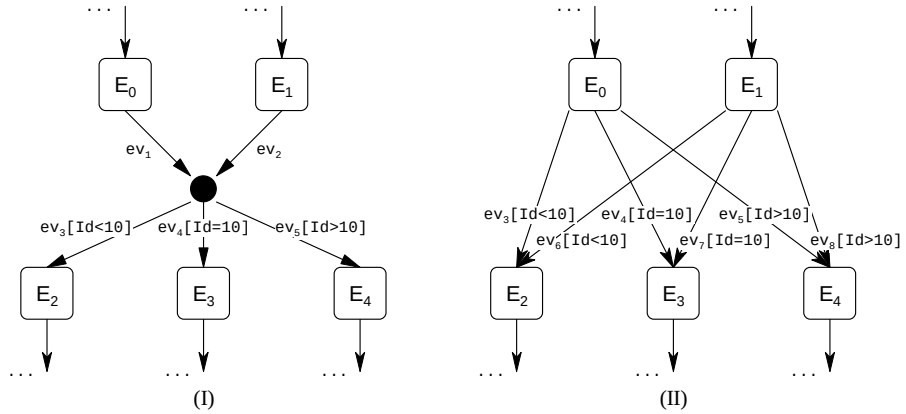


Figura 5.2 - Estrutura opcional para uma junção.

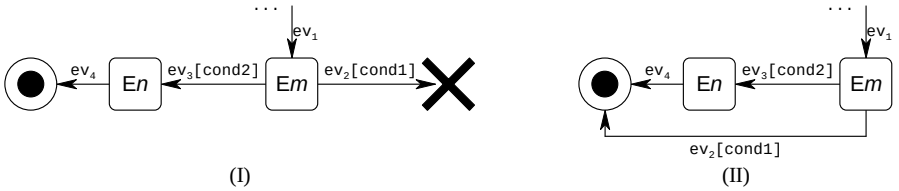


Figura 5.3 - Estrutura opcional para uma finalização.

Eventualmente, a utilização de um estado de finalização pode acarretar numa ação alternativa a que seria executada após o estado final, caso essa máquina de estados esteja agregada a uma outra. Esse fluxo alternativo deve ser assegurado de maneira similar ao apresentado no item que especifica as alternativas aos pontos de entrada e saída de uma sub-máquina de estados.

***Choice*:** apesar de ser disponibilizado pelo Rational Rose, sugere-se aqui

que esse pseudo-estado (*branch*) seja suprimido no instante da construção do diagrama de estados, conforme apresentado na Figura 5.4.

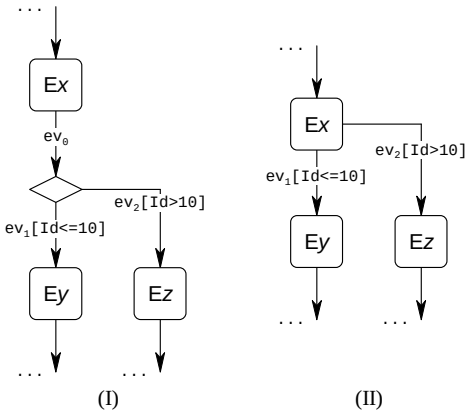


Figura 5.4 - Estrutura opcional para um pseudo-estado do tipo *choice*.

Nessa nova estrutura, o fluxo de execução prossegue de acordo com as avaliações requisitadas nas condições de guarda.

A seguir são descritos os detalhes do XML que disponibilizam as informações necessárias para a construção da SAN.

5.2. Informações pertinentes do XML para a construção da SAN

Na estrutura disponibilizada no arquivo XML gerado a partir do Rational Rose, contendo as informações sobre os diagramas de estados do sistema,

5.2.1. Diagrama de estados

Sendo que um diagrama de estados é utilizado para descrição comportamental de caso de uso, esses diagramas serão nomeados de forma que indique qual o caso de uso que está representando. Esses casos de uso possuirão identificadores padrão que os distinguirão dos demais. Pra isso, um número de identificação do tipo UC*n*, onde *n* é o número total de casos de uso, irá compor o nome do caso de uso juntamente com a sua descrição textual (opcional). Por exemplo: UC3 - Verificar saldo.

Seguindo a especificação descrita no Capítulo 4., cada diagrama de estados modelado no RUP, que modela um determinado caso de uso, deve estar contido em um super-estado. Cada um desses diagramas de estado estará contigo em um super-estado, sendo que esse será nomeado respeitando a seguinte sintaxe: E_UC1, E_UC2, E_UC3, ..., E_UC*n*, onde *n* é o número total de casos de uso.

Todos os estados de um diagrama descrevem o comportamento determinado caso de uso, possuindo a identificação desse caso de uso em seu nome, da seguinte maneira: E1_UC*n*, E2_UC*n*, E3_UC*n*, ..., Em_UC*n*, onde *m* é o número total de estados (além dos estados adicionais que modelam fluxos alternativos ao fluxo principal de dados) e *n* é o número total de casos de uso.

Segundo o padrão de modelagem especificado, os eventuais fluxos alternativos são nomeados com letras minúsculas seqüenciais, como segue: E*ma*_UC*n*, E*mb*_UC*n*, E*mc*_UC*n*, ..., E$m\alpha$_UC*n*, onde α é o número total de fluxos distintos que um estado pode seguir. Caso alguns desses fluxos alternativos sejam compostos por mais de um estado, cada um desses é nomeado da seguinte forma: Emα1_UC*n*, Emα2_UC*n*, Emα3_UC*n*, ..., Emαt_UC*n*, onde t é o numero total de estados que um determinado fluxo alternativo pode conter. A especificação descrita aqui é utilizada para nomear estados de um diagrama de estados. Logo, parte-se do princípio que essa especificação foi adotada na construção dos diagramas de estados, não sendo necessária qualquer modificação adicional para a construção dos estados da SAN, exceto pela inserção de estado auxiliares (W e WS) no arquivo .san, conforme descrito posteriormente.

Eventualmente, determinados estados de um diagrama pode conter o nome um caso de uso diferente do caso de uso que está sendo modelado. Isso vai indicar a execução de um outro diagrama de estados. Por exemplo, uma seqüência de estados do tipo <estado inicial> $\rightarrow$ E1_UC1 $\rightarrow$ E2_UC2) $\rightarrow$ E3_UC1) $\rightarrow$ <estado final> indica que quando a máquina de estados E_UC1 encontra-se no estado E2_UC2, a máquina de estados E_UC2 estará em execução.

De posse do arquivo XML com as informações sobre os diagramas de estado que descrevem o comportamento do sistema, sendo que esse arquivo seguiu a especificação descrita anteriormente, é possível construir o arquivo SAN para a análise do modelo de uso desse sistema, bem como a obtenção dos casos de teste para o mesmo.

Os diagramas de estados são possuem são identificados pela *tag* `<UML:StateMachine>` que a distingue dos demais objetos descritos. Conforme formalizado anteriormente, cada diagrama de es-

tados que descreve um caso de uso está encapsulado em um super-estado. Sendo assim, percorrermos o arquivo XML, será encontrada uma estrutura que segue o seguinte formato:

```
<UML:UseCase ...
   name = '<nome do caso de uso>'
   ... />
   <UML:StateMachine ...
      name = 'State/Activity Model'
      ...>
      <UML:CompositeState ...
         ... >
         <UML:CompositeState.subvertex>
            <UML:CompositeState ...
               name = '<nome do super-estado>'
               ... >
               <UML:CompositeState.subvertex>
               ...
```

Os nomes do caso de uso e do super-estado possuem o mesmo identificador (no formato UC*n*, onde *n* é o número total de casos de uso), que será utilizado adiante na especificação do nome do autômato correspondente na SAN. Prosseguindo na leitura do arquivo, será encontrada a estrutura do diagrama de estados (denominado E_UC1 nesse exemplo), conforme a seguir:

```
...
<UML:Pseudostate xmi.id = 'G.93'
   name = '' visibility = 'public' isSpecification = 'false'
   kind = 'initial'
   outgoing = 'G.94' />
<UML:FinalState xmi.id = 'G.95'
   name = '' visibility = 'public' isSpecification = 'false'
   incoming = 'G.100 G.106' />
<UML:SimpleState xmi.id = 'G.97'
   name = 'E1_UC1)' visibility = 'public' isSpecification = 'false'
   outgoing = 'G.98 G.100' incoming = 'G.109' />
<UML:SimpleState xmi.id = 'G.102'
   name = 'E2_UC2)' visibility = 'public' isSpecification = 'false'
   outgoing = 'G.103' incoming = 'G.98' />
<UML:SimpleState xmi.id = 'G.105'
   name = 'E3_UC1)' visibility = 'public' isSpecification = 'false'
   outgoing = 'G.106' incoming = 'G.103' />
<UML:SimpleState xmi.id = 'G.108'
   name = 'E4_UC1)' visibility = 'public' isSpecification = 'false'
   outgoing = 'G.109' incoming = 'G.94' /> ...
```

Nessa estrutura, cada estado possui um número próprio que o distingue dos demais (`xmi.id`), e esses estados podem ser classificados segundo as seguintes *tags*:

- `<UML:SimpleState>`: descreve um estado simples. Na sua estrutura, um estado simples possui atributos que descrevem a sua situação no contexto do diagrama, sendo os que interessam para os propósitos desse trabalho são:
  - `xmi.id`: número que identifica o estado corrente;
  - `name`: descreve o nome do estado corrente;
  - `outgoing`: informa o xmi.id do próximo objeto;
  - `incoming`: informa o xmi.id do objeto anterior
- `<UML:Pseudostate>`: descrevem os pseudo-estados que constituem o diagrama de

estado. Esses pseudo-estados são transientes, logo o sistema nunca permanece nesses. Os atributos necessários para a nossa análise são:

  - `xmi.id`: número que identifica o pseudo-estado corrente;
  - `kind`: descreve o tipo do pseudo-estado corrente, sendo esses:
    * `initial`: descreve um estado inicial de uma maquina de estado. Nesse tipo de pseudo-estado, não haverá incoming.
    * `fork`: descreve o disparo paralelo e sincronizado de varias transições.
    * `join`: descreve a sincronização de várias transições em uma única.
    * `branch`: descreve uma escolha entre duas ou mais transições, de acordo com a condição de guarda associada a cada transição
  - `outgoing`: informa o xmi.id do próximo objeto;
  - `incoming`: informa o xmi.id do objeto anterior.

  Os pseudo-estados não possuem nomes associados a si. De qualquer forma essa informação não é necessária para a construção da SAN.

- `<UML:FinalState>`: indica o fim da execução de uma máquina de estados. Nesse tipo de estado, não haverá outgoing.

Os `xmi.id` especificados em `outgoing` e `incoming` são usados na composição de uma transição, sendo que essa é composta com uma seqüência do tipo <xmi.id do estado/pseudo-estado> → <xmi.id da transição> → <xmi.id do estado/pseudo-estado>. Outras informações das transições, como condição de guarda, ações ou nome dos eventos (esses serão compostos no instante da construção da SAN), não são necessárias para a construção da SAN.

O Rational Rose não possui um objeto que designe diretamente um *fork* ou um *join*. Para isso, é disponibilizado um objeto denominado estado de sincronização (*synchronization state*) com o qual é possível compor estrutura do tipo *fork* e *join*. Porém, ao traduzir um arquivo gerado pelo Rose para uma estrutura XML, os identificadores *fork* e *join* são disponibilizados, como pode ser verificado no trecho abaixo:

```
...
<UML:Pseudostate xmi.id = 'G.239'
   name = '' ...
   kind = 'fork'
   outgoing = 'G.240 G.241' incoming = 'G.230' />
...
<UML:Pseudostate xmi.id = 'G.242'
   name = '' ...
   kind = 'join'
   outgoing = 'G.243' incoming = 'G.233 G.236' />
...
```

Apesar da UML não explicitar que para cada *fork* especificado em um diagrama, deverá existir um *join* correspondente, nesse trabalho será adotada essa premissa. Isso é necessário pois o disparo paralelo de transições precisa ocorrer em autômatos distintos, o que sugere a construção um novo autômato para cada uma das transições disparadas, na SAN que será construída. Dessa forma, também

é necessário compor novos eventos sincronizantes em cada um dos autômatos construídos, de forma que a execução retorne para o autômato que disparou as transições paralelas, imediatamente após a execução do referido disparo. Para melhor ilustrar essa idéia, é apresentada a Figura 5.5.

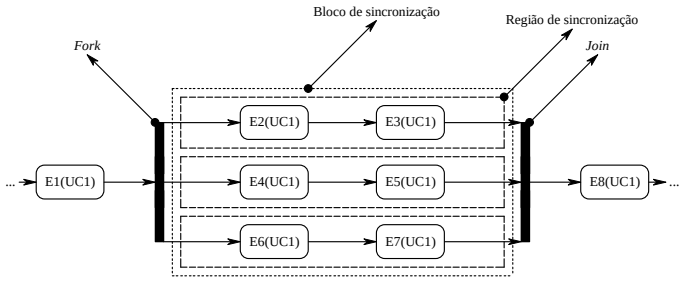


Figura 5.5 - Regiões de sincronização.

Essa foi a forma encontrada para tratar as regiões de sincronização, sendo que essas não são disponibilizadas pelo Rational Rose.

5.2.2. Interação entre atores e sistema

Conforme explanado anteriormente, as análises almejadas pela transcrição dos diagramas de estados UML para SAN, requerem informações adicionais além das fornecidas pelos diagramas de estados, referentes à identificação dos casos de uso com os quais os atores interagem. Essas informações são necessárias para integrar os casos de uso do sistema na composição da análise de sistema, e para a determinação de quais os casos de uso que irão compor cada uma das análises de componente.

No arquivo XML, a interação entre as entidades que compõem um diagrama de casos de uso é feita por uma estrutura denominada Association. Essa estrutura (identificada pela tag `<UML:Association>`) é composta por dois objetos do diagrama de casos de uso, cada um especificado em uma estrutura do tipo `AssociationEnd`. Para encontrar os casos de uso que possuem interação direta com os atores, é necessário seguir alguns passos, conforme especificado na Figura 5.6.

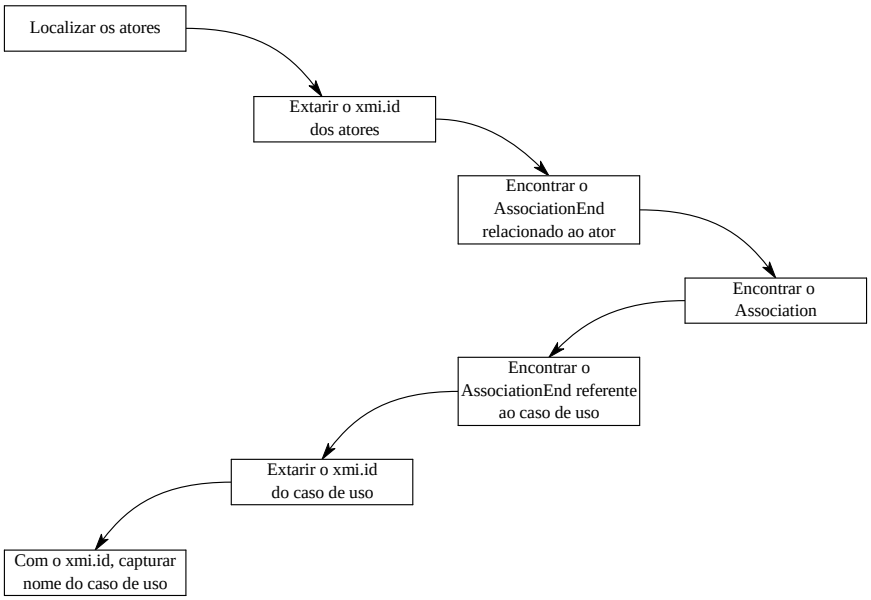


Figura 5.6 - Determinação dos casos de uso principais a partir dos atores.

A determinação dos atores e quais casos de uso que esses interagem é feita da seguinte forma:

- No arquivo XML, são localizados os atores do sistema, os quais são identificados pela *tag* <UML:Actor>. No arquivo XML, os atores estarão especificados da seguinte forma:

```
<UML:Actor xmi.id = '<xmi.id do ator1>'
   name = '<nome do ator1>' ...  />
<UML:Actor xmi.id = '<xmi.id do ator2>'
   name = '<nome do ator2>' ...  />
<UML:Actor xmi.id = '<xmi.id do ator3>'
   name = '<nome do ator3>' ...  />
```

- Para cada ator localizado, deve-se capturar o xmi.id do mesmo, estando esse especificado da seguinte forma:

```
<UML:Actor xmi.id = '<xmi.id do ator1>' ...
```

- Com o xmi.id de um ator, encontrar o AssociationEnd que compõe a associação com um determinado caso de uso. Um AssociationEnd é disposto no arquivo XML da seguinte forma:

```
<UML:AssociationEnd xmi.id = '<xmi.id do AssociationEnd1>'
   ...
   type = '<xmi.id do ator1>' >
```

- A partir do associationEnd que se refere ao ator, deve-se encontrar a outra extremidade da associação, a qual se refere ao caso de uso que interage diretamente com o ator. Esse caso de uso é especificado no outro associationEnd que compõe a referida associação (identificada pela *tag* <UML:Association>), conforme pode ser observado na seguinte estrutura:

```
<UML:Association xmi.id = '<xmi.id do Association>'
   ...  >
   <UML:Association.connection>
      <UML:AssociationEnd xmi.id = '<xmi.id do AssociationEnd1>'
         ...
         type = '<xmi.id do ator1>' >
         <UML:AssociationEnd.multiplicity>
            <UML:Multiplicity >
               <UML:Multiplicity.range>
                  <UML:MultiplicityRange xmi.id = '<xmi.id do MultiplicityRange1>'
                     ...  />
               </UML:Multiplicity.range>
            </UML:Multiplicity>
         </UML:AssociationEnd.multiplicity>
      </UML:AssociationEnd>
      <UML:AssociationEnd xmi.id = '<xmi.id do AssociationEnd2>'
         ...
         type = '<xmi.id do Caso de Uso X>' >
         <UML:AssociationEnd.multiplicity>
            <UML:Multiplicity>
               <UML:Multiplicity.range>
                  <UML:MultiplicityRange xmi.id = '<xmi.id do MultiplicityRange2>'
                     ...  />
               </UML:Multiplicity.range>
            </UML:Multiplicity>
         </UML:AssociationEnd.multiplicity>
      </UML:AssociationEnd>
   </UML:Association.connection>
</UML:Association>
```

- Ao localizar o outro AssociationEnd da associação, deve-se capturar o xmi.id do mesmo, que será a identificação do caso de uso que interage com o ator1, conforme pode exemplifica a seguinte estrutura:

```
<UML:AssociationEnd xmi.id = '<xmi.id do AssociationEnd2>'
   ...
   type = '<xmi.id do Caso de Uso X>' >
```

- com o xmi.id capturado, deve-se encontrar o caso de uso (identificado pela *tag* <UML:UseCase>) que é especificado com esse identificador, assim como descrito a seguir:

```
<UML:UseCase xmi.id = '<xmi.id do Caso de Uso X>' ...
```

- ao encontrar o referido caso de uso, é possível capturar o seu nome, sendo esse especificado na sua estrutura, como pode ser observado a seguir:

```
<UML:UseCase xmi.id = '<xmi.id do Caso de Uso X>'
   name = '<nome do Caso de Uso X>' ...
```

5.3. Módulo 1 - Tradução

O Módulo 1 do protótipo proposto refere-se à transcrição dos diagramas de estados especificados no arquivo XML, para uma estrutura equivalente em SAN, disponibilizando essa para a análise do modelo de uso do sistema, bem como a construção de casos de teste do tipo Bertolini [BFF+04, Ber05]. As Figuras 5.7 e 5.8 apresentam, respectivamente, os diagramas de classe modela a estrutura do Módulo 1 e o diagrama de seqüência da classe principal (UMLtoSAN).

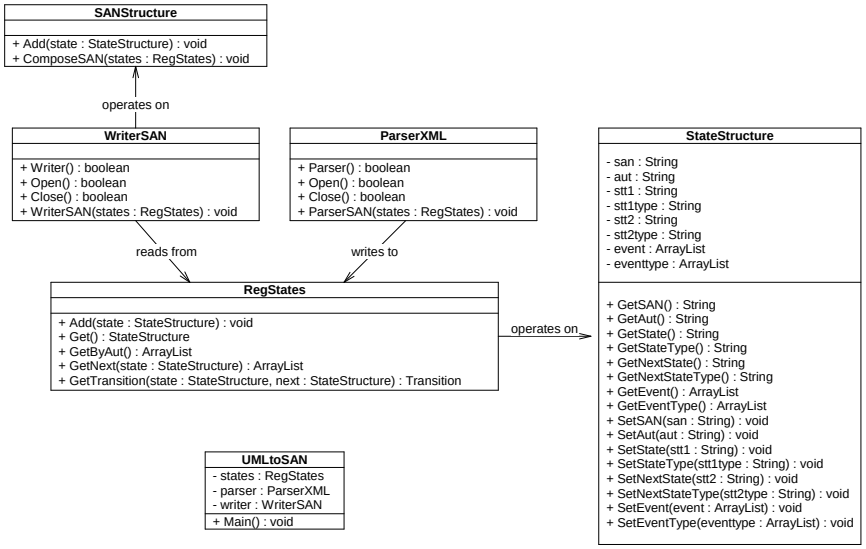


Figura 5.7 - Diagrama de classes do Módulo 1.

No diagrama de seqüência apresentado na Figura 5.8 não mostra as trocas de mensagens entre as classe UMLtoSAN e SANStructure, pois essa ultima interage diretamente com a classe WriterSAN, sendo essa interação especificada no diagrama de seqüência que modela as chamadas efetuadas pela WriterSAN (esse e os outros diagramas de seqüências não são mostrados nesse trabalho).

A seguir é apresentada uma descrição geral sobre o funcionamento de cada uma das classes do sistema:

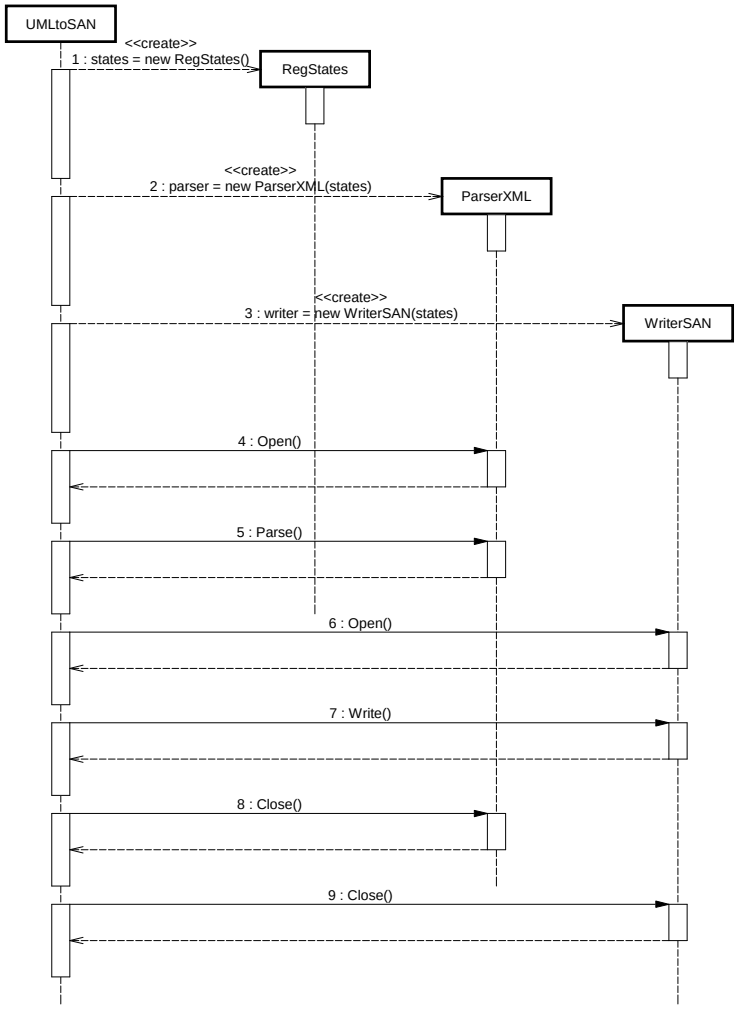


Figura 5.8 - Diagrama de seqüências do Módulo 1.

***ParserXML*:** efetua a leitura do arquivo XML e seleciona os dados necessários para a composição de uma estrutura com as informações necessárias para a construção da SAN.

***StateStructure*:** com a leitura feita pelo *ParserXML* é construída uma estrutura com as seguintes informações;

- SAN: especifica o nome da SAN que agrega o autômato e, por conseqüência, o estado que está sendo analisado. No caso de uma análise de componente, o nome da SAN será atribuído de acordo com o nome do arquivo XML, uma vez que essa análise compreende uma única SAN. No caso de um análise de componente, será construída uma SAN para cada caso de uso que interage diretamente com um ator, bem como os casos de uso subordinados a esse. O nome de cada uma dessas SAN será atribuído de acordo com o caso de uso que interage diretamente com o ator. Sua nomeação segue a seguinte sintaxe: SAN_<nome da SAN>.
- Autômato: os autômatos serão nomeados de acordo com o diagrama de estados (que possui o mesmo nome do caso de uso) que o mesmo modela. Sendo assim, o referido autômato será nomeado de acordo com a seguinte sintaxe: Aut_<nome do caso de uso>.

- Estado 1: esse é um estado, ou pseudo-estado, de onde parte uma transição. No caso dos estados, o nome será o mesmo especificado no arquivo XML. Para os pseudo-estados, os nomes será atribuídos de acordo com o seu tipo, uma vez que esses não possuem nomes especificados no arquivo XML. Sendo assim, o estado inicial de um diagrama de estado será nomeado como W, e os *forks* e *joins* como WS*n*, onde *n* é o número de blocos de sincronização que constituem um mesmo diagrama de estados, sejam essas seqüenciais ou aninhados. Um estado final nunca será especificado como "Estado 1" pois dele nunca partirá uma transição.
- Tipo do estado 1: o tipo de estado é determinado de acordo com a *tag* XML que o descreve, ou pelas informações adicionais que especificam um pseudo-estado (kind). Sendo assim, a Tabela 5.1 descreve os tipos dos estados e pseudo-estados, de acordo com a sua especificação:

Tabela 5.1 - Tipos de estados.

Tipo do estado/pseudo-estado	*Tag* XML	*Kind*
Inicial	<UML:Pseudostate>	*initial*
Simples	<UML:SimpleState>	-
Fork	<UML:Pseudostate>	*fork*
Join	<UML:Pseudostate>	*join*
Final	<UML:FinalState>	-

- Estado 2: esse é um estado, ou pseudo-estado, que é atingido por uma transição. A determinação do seu nome é feita da mesma forma conforme descrito para o "Estado 1", exceto pela nomeação de um estado final como W. Um pseudo-estado inicial nunca será especificado como "Estado 2", uma vez que nenhuma transição atinge um pseudo-estado inicial.
- Tipo do estado 2: o tipo do estado 2 é determinado da mesma forma especificado no "Tipo do estado 1", adotando as mesmas regras de nomeação descritas na Tabela 5.1.
- Evento: descreve o nome do evento agregado à transição descrita. Sua determinação será feita numa etapa posterior, durante a construção da SAN.
- Tipo do evento: o tipo do evento é determinado de acordo com o contexto onde a transição ocorre, conforme especificado adiante.

***SANStructure*:** organiza a informações necessárias para a construção da SAN, principalmente para a verificação dos eventos sincronizantes.

***WriterSAN*:** compõem o arquivo SAN.

Para melhor elucidar o funcionamento do protótipo, tomaremos como exemplo um sistema modelado com os diagramas descritos na Figura 5.9

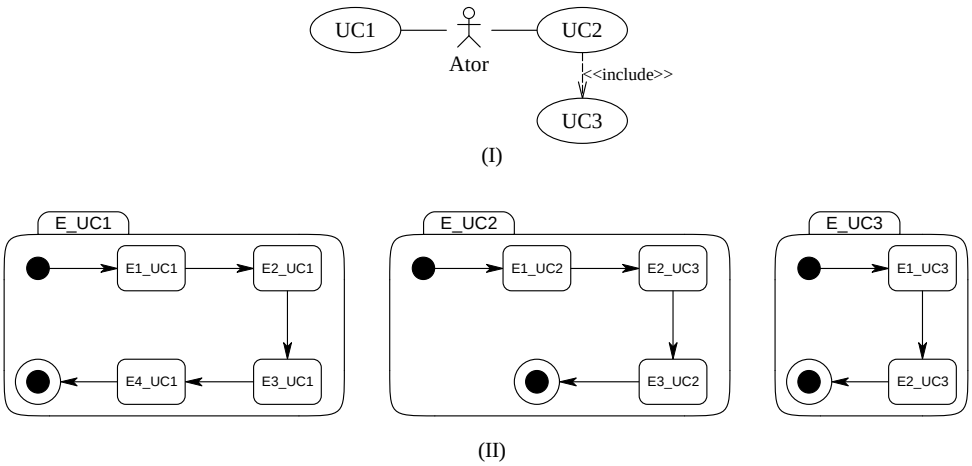


Figura 5.9 - Diagramas que modelam um sistema exemplo.

Caso a análise pretendida para esse *software* seja uma análise de sistema, a estrutura construída seria similar a observada na Tabela 5.2. Suponhamos que o arquivo gerado pelo Rational Rose tenha sido nomeado como Test.xml.

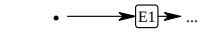
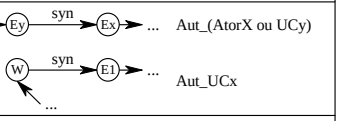
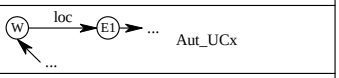
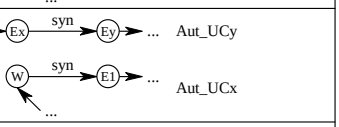
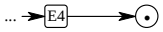
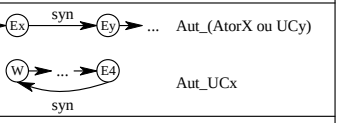
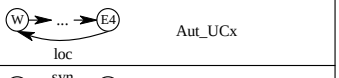
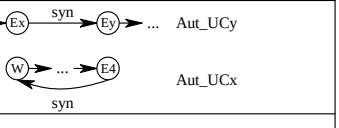
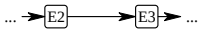
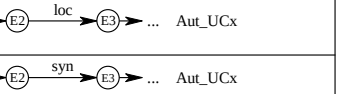
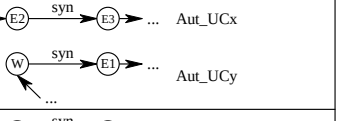
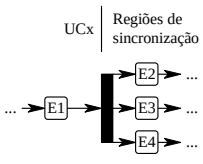
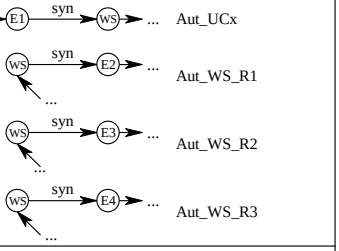
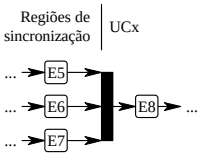
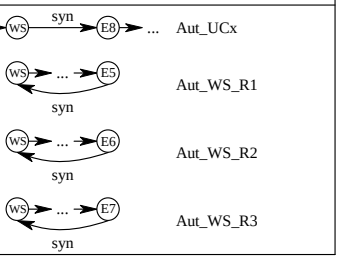
Tabela 5.2 - Tipos de estados.

SAN	**Automaton**	**Stt1**	**Stt1Type**	**Stt2**	**Stt2Type**	**Event**	**EventType**
SAN_Test	Aut_Ator	W	initial	UC1	simple	-	-
SAN_Test	Aut_Ator	UC1	simple	W	initial	-	-
SAN_Test	Aut_Ator	W	initial	UC2	simple	-	-
SAN_Test	Aut_Ator	UC2	simple	W	initial	-	-
SAN_Test	Aut_UC1	W	initial	E1_UC1	simple	-	-
SAN_Test	Aut_UC1	E1_UC1	simple	E2_UC1	simple	-	-
SAN_Test	Aut_UC1	E2_UC1	simple	E3_UC1	simple	-	-
SAN_Test	Aut_UC1	E3_UC1	simple	E4_UC1	simple	-	-
SAN_Test	Aut_UC1	E4_UC1	simple	W	final	-	-
SAN_Test	Aut_UC2	W	initial	E1_UC2	simple	-	-
SAN_Test	Aut_UC2	E1_UC2	simple	E2_UC3	simple	-	-
SAN_Test	Aut_UC2	E2_UC3	simple	E3_UC2	simple	-	-
SAN_Test	Aut_UC2	E3_UC2	simple	W	final	-	-
SAN_Test	Aut_UC3	W	initial	E1_UC3	simple	-	-
SAN_Test	Aut_UC3	E1_UC3	simple	E2_UC3	simple	-	-
SAN_Test	Aut_UC3	E2_UC3	simple	W	final	-	-

Por motivos de simplificação e pela necessidade de classificarmos os estados que modelam a interação entre o ator e o sistema, quando o ator não estiver interagindo com o sistema, o autômato que o modela se encontrará em um estado W do tipo *initial*. Quando o ator estiver interagindo com o sistema por intermédio de um determinado caso de uso, o autômato estará no estado que identifica esse caso de uso, e esse será classificado como *simple* na estrutura que especifica as transições.

A estrutura gerada possui informações capturadas diretamente do arquivo XML. Já as informações referentes aos eventos (nome e tipo) serão compostas a partir das informações já coletadas, de acordo com os estados envolvidos em cada transição, e em qual o contexto que essa transição ocorre. Para isso, a Tabela 5.3 foi construída para auxiliar na determinação dessas informações.

Tabela 5.3 - Determinação dos eventos da SAN.

Diagrama de estados	Contexto		Evento	SAN
	Análise de sistema		syn	 Aut_(AtorX ou UCy) Aut_UCx
	Análise de componente	SD associado a um ator	loc	 Aut_UCx
		SD associado a um SD	syn	 Aut_UCy Aut_UCx
	Análise de sistema		syn	 Aut_(AtorX ou UCy) Aut_UCx
	Análise de componente	SD associado a um ator	loc	 Aut_UCx
		SD associado a um SD	syn	 Aut_UCy Aut_UCx
	Estados que modelam o mesmo UC		loc	 Aut_UCx
	Estados que modelam UCs diferentes		syn	 Aut_UCx Aut_UCy
	Todos		syn	 Aut_UCx Aut_WS_R1 Aut_WS_R2 Aut_WS_R3
	Todos		syn	 Aut_UCx Aut_WS_R1 Aut_WS_R2 Aut_WS_R3

Utilizando regras, baseadas na Tabela 5.3, para a determinação do tipo de evento agregado a cada transição, é possível preencher a coluna **EventType** da Tabela 5.2. O componente SANStructure será responsável pela verificação dos eventos sincronizantes, uma vez que um mesmo evento estará agregado a mais de uma transição, atribuindo a esses eventos nomes temporários, que serão redefinidos no instante do preenchimento da coluna **Event**. Essa última coluna será composta pela nomeação dos eventos de forma seqüencial, obedecendo a sintaxe ev_*n*, onde *n* é o número total de eventos da SAN. Sendo assim, é possível compor a estrutura com as informações das transições que serão usadas para a construção da SAN, assim como exposto na Tabela 5.4:

De posse da estrutura completa, é possível construir o arquivo SAN com o qual é possível efetuar análises sobre o modelo de uso do sistema. Essa construção é feita pelo componente WriterSAN, que

Tabela 5.4 - Tipos de estados.

SAN	Automaton	Stt1	Stt1Type	Stt2	Stt2Type	Event	EventType
SAN_Test	Aut_Ator	W	initial	UC1	simple	ev_1	syn
SAN_Test	Aut_Ator	UC1	simple	W	initial	ev_2	syn
SAN_Test	Aut_Ator	W	initial	UC2	simple	ev_3	syn
SAN_Test	Aut_Ator	UC2	simple	W	initial	ev_4	syn
SAN_Test	Aut_UC1	W	initial	E1_UC1	simple	ev_1	syn
SAN_Test	Aut_UC1	E1_UC1	simple	E2_UC1	simple	ev_5	loc
SAN_Test	Aut_UC1	E2_UC1	simple	E3_UC1	simple	ev_6	loc
SAN_Test	Aut_UC1	E3_UC1	simple	E4_UC1	simple	ev_7	loc
SAN_Test	Aut_UC1	E4_UC1	simple	W	final	ev_2	syn
SAN_Test	Aut_UC2	W	initial	E1_UC2	simple	ev_3	syn
SAN_Test	Aut_UC2	E1_UC2	simple	E2_UC3	simple	ev_8	syn
SAN_Test	Aut_UC2	E2_UC3	simple	E3_UC2	simple	ev_9	syn
SAN_Test	Aut_UC2	E3_UC2	simple	W	final	ev_4	syn
SAN_Test	Aut_UC3	W	initial	E1_UC3	simple	ev_8	syn
SAN_Test	Aut_UC3	E1_UC3	simple	E2_UC3	simple	ev_10	loc
SAN_Test	Aut_UC3	E2_UC3	simple	W	final	ev_9	syn

captura as informação necessárias disponibilizadas pelos componentes StateStructure e SANStructure. O Arquivo gerado pela WriterSAN seria o seguinte:

```
events

   syn ev_1 (0.6)
   syn ev_2 (1)
   syn ev_3 (0.4)
   syn ev_4 (1)
   syn ev_5 (1)
   syn ev_6 (1)
   syn ev_7 (1)
   syn ev_8 (1)
   syn ev_9 (1)
   syn ev_10 (1)

reachability = 1;

network Test (continuous)

   Aut_Ator
      stt W
         to(UC1) ev_1
         to(UC2) ev_2
      stt UC1
         to(W) ev_3
      stt UC2
         to(W) ev_4

   Aut_UC1
      stt W
         to(E1_UC1) ev_1
      stt E1_UC1
         to(E2_UC1) ev_5
```

(continua...)

(continuação)

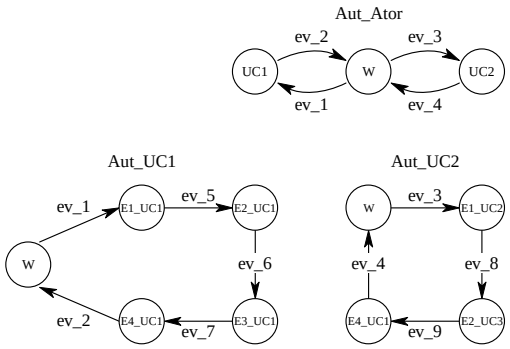
```
      stt E2_UC1
         to(E3_UC1) ev_6
      stt E3_UC1
         to(E4_UC1) ev_7
      stt E4_UC1
         to(W) ev_2

   Aut_UC2
      stt W
         to(E1_UC2) ev_3
      stt E1_UC2
         to(E2_UC3) ev_8
      stt E2_UC3
         to(E3_UC2) ev_9
      stt E3_UC2
         to(W) ev_4

   Aut_UC3
      stt W
         to(E1_UC3) ev_8
      stt E1_UC3
         to(E2_UC3) ev_10
      stt E2_UC3
         to(W) ev_9

results
```

Graficamente, a referida SAN seria estruturada conforme mostrado na Figura 5.10:



Tipo	Evento	Taxa
syn	ev_1	0.6
syn	ev_2	1
syn	ev_3	0.4
syn	ev_4	1
loc	ev_5	1
loc	ev_6	1
loc	ev_7	1
syn	ev_8	1
syn	ev_9	1
loc	ev_10	1

Figura 5.10 - SAN para o sistema exemplo.

Com a SAN gerada, é possível efetuar análises sobre o modelo de uso do sistema, podendo assim servir como instrumento auxilio para o analista de *software*, no que se refere, por exemplo, tomada de decisões para alocação de recursos (desenvolvedores e testadores) e para estimativas para tempo de implementação. Sendo que o exemplo aqui apresentado possui um objetivo meramente ilustrativo, ainda assim é possível obtermos algumas informações que pudessem auxiliar o analista de sistema na tomada de decisões. Por exemplo: a determinação do estado do autômato que possua uma maior probabilidade de execução, devido as taxas de ocorrência, poderia ser um indicativo que esse setor do sistema é um ponto crítico, sendo necessário que o mesmo seja robusto e confiável, sendo que o analista possa alocar o melhores desenvolvedores e testadores para esse setor.

Sobre a geração de casos de teste de *software*, tendo posse de uma SAN que descreva o comportamento do sistema como um todo, ou de parte dele, é possível gerar casos de teste conforme

proposto por Bertolini [BFF+04, Ber05]. As eventuais informações adicionais que não são disponibilizadas pela SAN podem ser inseridas manualmente na interface do STAGE-Model. Uma inserção automática desses eventuais dados pode ser feita pela composição de um novo módulo ao protótipo, sendo isso proposto aqui como um trabalho futuro.

5.3.1. Particularidades da análise de sistema

Apesar que já terem sido mencionadas, serão aqui ressaltadas algumas particularidades observadas (e esperadas) para a análise de sistema, proposta como produto do presente protótipo. Dentre elas, podemos destacar:

- a interação dos atores com os casos são modelados por autômatos específicos. cada um deles especifica a interação de um determinado ator os casos de uso com os quais esse ator interage;
- uma única SAN, nomeada conforme o nome do arquivo XML gerado pelo Rational Rose, será concebida para a análise de sistema;
- um eventual caso de uso incluído que seja subordinados a mais de um caso de uso, possuirá apenas um autômato correspondente na análise de sistema, uma vez que uma única SAN modela o sistema inteiro;
- nos casos onde SAN gerada possua um número de autômatos e estado que inviabilizem a sua análise na ferramenta PEPS2003, a execução do módulo 2 do presente protótipo fará alterações na presente SAN, efetuando simplificação que possibilitem a sua posterior análise, sendo que essa simplificação não compromete a análise pretendida.

5.3.2. Particularidades da análise de componente

Dentre as particularidades esperadas para a análise de componente, podemos destacar as seguintes:

- para cada componente, derivado de um caso de uso que interage com um ator do sistema, será concebida uma SAN específica para a análise do modelo de uso dessa funcionalidade do sistema.
- cada uma das SAN que modela uma determinada funcionalidade do sistema, funcionalidade essa disponibilizada para um ator, será nomeada de acordo com o caso de uso com o qual o ator interage.
- se um caso de uso incluído for subordinado a casos de uso que compõem SAN diferentes, o mesmo será modelado em cada uma dessas SAN.
- eventualmente, um componente pode possuir um numero de autômatos e estado que inviabilizam a sua análise na ferramenta PEPS2003. Nesses casos, a utilização do módulo 2 do sistema será empregada.

5.4. Módulo 2 - Simplificação

No intuito de executar a simplificação da SAN gerada na análise de sistema, foi implementado um módulo de simplificação, sendo que a sua execução inicia após a composição da SAN na análise de sistema. Para esse módulo, foi adicionada uma classe ao projeto UMLtoSAN, descrito anteriormente. Essa nova classe, SANSimplifier, possui um método responsável pela substituição de um bloco seqüencial de estados por um estado equivalente, e um método responsável por excluir os estados equivalentes e tratar os casos especiais (auto-transição e múltiplas transições).

Os métodos dessa classe operam sobre a estrutura construída pelo método RegStates(), utilizada para armazenar as informações referentes às transições, para a elaboração da SAN. Sendo assim, o método de simplificação (Simplify()) é executado da seguinte forma:

1. São capturados todos estados-chave nos autômatos da SAN, conforme descrito na seção 4.4..

2. A partir de cada estado-chave, é percorrido um possível caminho, guardando cada estado percorrido em um *array*.

3. Ao ser encontrado um novo estado chave, todos os estados e transições encontrados até então são substituídos por um estado equivalente.

Já o método Adjust(), responsável por excluir os estados equivalentes, e por tratar das eventuais auto-transições e múltiplas transições, é executado da seguinte forma:

1. São eliminados os estados equivalentes. Para isso, é composta uma transição que liga os estados-chave que se encontram antes e depois do estado equivalente. Essa nova transição possuirá o evento que, anteriormente, estava agregado à transição que partia do estado equivalente. Isso significa que haverá um teste para esse evento, de forma que seja verificada a execução do evento imediatamente anterior a um estado-chave é executado.

2. Caso algum autômato gerado no passo 1 possua apenas um estado, e esse possua uma auto-transição, esse autômato é excluído. Quando isso ocorrer, os eventos sincronizantes do outro autômato, aquele que determinava o início e o fim da execução do autômato excluído, serão definidos como eventos locais.

3. Caso algum autômato gerado no passo 1 possua múltiplas transições entre dois estados, são eliminadas essas transições, sendo então composta uma nova. Essa nova transição possui os eventos que, anteriormente, estavam agregadas às transições eliminadas.

Após a execução desses métodos, a classe WriterSAN é executada novamente, compondo assim a nova SAN.

5.5. Interface com o Usuário

Para a execução do protótipo UMLtoSAN, o usuário deve dispor do arquivo XML gerado pelo RoseXMLTools, arquivo esse que proverá os dados necessários para a construção da SAN. Quando o arquivo UMLtoSAN.exe é executado, o usuário opta pela análise de sistema ou pela análise de componente. Após a escolha da opção de análise, o usuário informa no nome do arquivo XML a ser analisado. Esse passo é exemplificado na Figura 5.11, aonde é mostrada uma análise de componente.

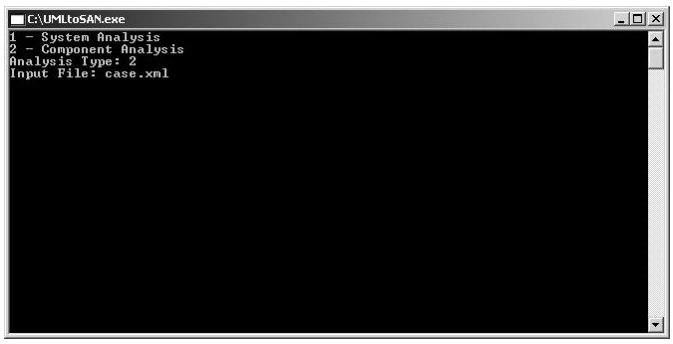


Figura 5.11 - Protótipo UMLtoSAN - Análise de Componente.

A Figura 5.12 ilustra o protótipo em execução, mostrando na tela alguns resultados da análise do arquivo XML, os cálculos iniciais para a construção da SAN, bem como os passos dessa construção. Como produto final, os arquivos `.san` são gerados e disponibilizados no diretório de execução, para a posterior análise na ferramenta PEPS2003.

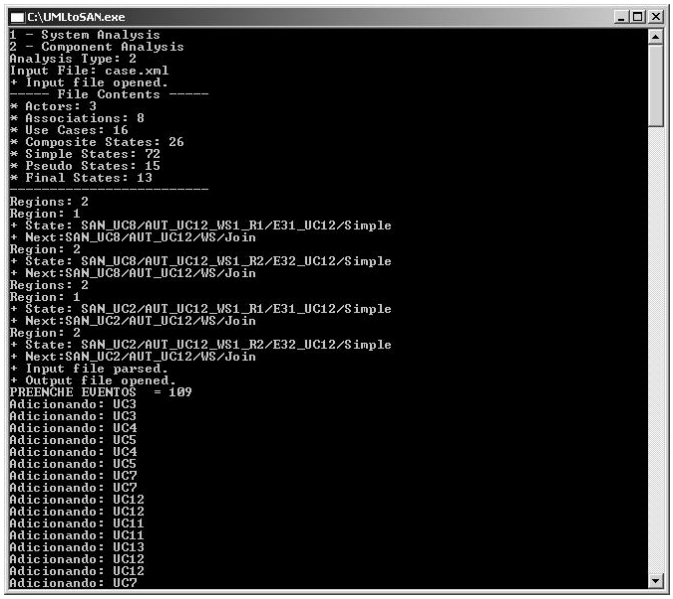


Figura 5.12 - Protótipo UMLtoSAN - Exemplo de execução.

No próximo capítulo será apresentado um estudo de caso, sendo esse utilizado para exemplificar o método proposto nessa dissertação. Será apresentada a descrição desse estudo de caso, juntamente com os diagramas e artefatos necessários para a composição da SAN.

6. ESTUDO DE CASO

Com intuito de verificar a eficácia da pesquisa efetuada, é proposto aqui um estudo de caso que englobe as terminologias descritas. Nesse capítulo, serão especificados os detalhes do sistema de estudo proposto, bem como a modelagem e a transcrição dos diagramas gerados para uma estrutura equivalente em SAN.

6.1. Descrição do Domínio

O estudo de caso proposto nesse relatório refere-se a um sistema de controle e automação de um bar. O referido sistema é responsável por gerenciar a venda de bebidas, o pagamento da contas e controle de estoque do estabelecimento. A automação das atividades citadas é viabilizada pela utilização de cartões magnéticos, que substituem as tradicionais ”comandas”. Ao chegar no estabelecimento, o cliente informa seu nome e CPF ao atendente, para eventual identificação em caso de perda ou extravio. Esses dados são vinculados a um cartão magnético, sendo esse entregue ao cliente para a sua posterior utilização.

Para efetuar uma compra, o cliente solicita o produto e entrega o cartão ao atendente. Nesse instante, o atendente passa o cartão na leitora e é verificado o status desse cartão, para certificar se esse cartão está apto para prosseguimento da operação de compra. Um cartão está inapto para a compra nos casos em que um cliente estivesse tentando efetuar uma compra com um cartão que tivesse sido dado como perdido, ou em casos onde o consumo atingisse um determinado valor. Esse último é utilizado de forma preventiva, impedindo que um cliente efetue compras acima de um valor considerado alto. Caso esse valor seja atingido, é solicitado o pagamento desse valor para que o cartão fique apto novamente.

Caso o cartão esteja apto para compra, o atendente informa no termina qual o produto e a quantidade solicitada. Nesse instante seria feita uma verificação no estoque sobre a disponibilidade do produto. Sendo confirmada a disponibilidade, a quantidade solicitada é decrementada do estoque e o valor resultante dessa operação (valor unitário do produto multiplicado pela quantidade) seria vinculada ao cartão. Ressalta-se aqui que a quantidade e valor operados sobre esse cartão fica armazenada em uma base de dados, sendo que o cartão apenas vincula o cliente ao seu consumo, não possuindo qualquer informação sobre valores ou quantidades consumidas.

Para possibilitar o controle de consumo por parte do cliente, o estabelecimento possui terminais para a verificação de valores. Para a sua utilização, o cliente passa o cartão no terminal que logo após informa no visor o valor consumido vinculado a esse cartão. Nessa operação também é efetuada a verificação do status do cartão, e caso o mesmo apresente alguma irregularidade, o terminal mostra uma mensagem para que o cliente dirija-se ao caixa para regularizar a situação do cartão.

O sistema possui um administrador responsável por cadastrar os produtos, juntamente com suas quantidades e valores unitários. No instante do cadastramento, o administrador informa uma quantidade crítica de estoque de cada produto, e caso essa quantidade seja atingida, o administrador é informado para que efetue as providencias para a reposição do produto.

6.2. Diagrama de casos de uso

Para a modelagem do sistema descrito anteriormente, propõe-se a construção de três componentes que permitam a interação dos atores (Atendente, Cliente e Administrador) com o sistema. O Atendente estará apto a receber o pagamento mediante aos cálculos requisitados e de acordo com a forma de pagamento, poderá inserir os dados do cliente na base de dados no intuito atribuir as informações sobre consumação (efetuada ao mesmo), poderá desabilitar um cartão dado como perdido e ainda estará apto a proceder um pedido efetuado pelo cliente.

O cliente, por sua vez, pode interagir com o sistema ao consultar o saldo do valor consumido até um determinado instante. Já o administrador irá cadastrar os produtos a serem disponibilizados aos clientes. Essa visão geral do sistema pode ser melhor vislumbrada através do seguinte diagrama de casos de uso (Figura 6.1):

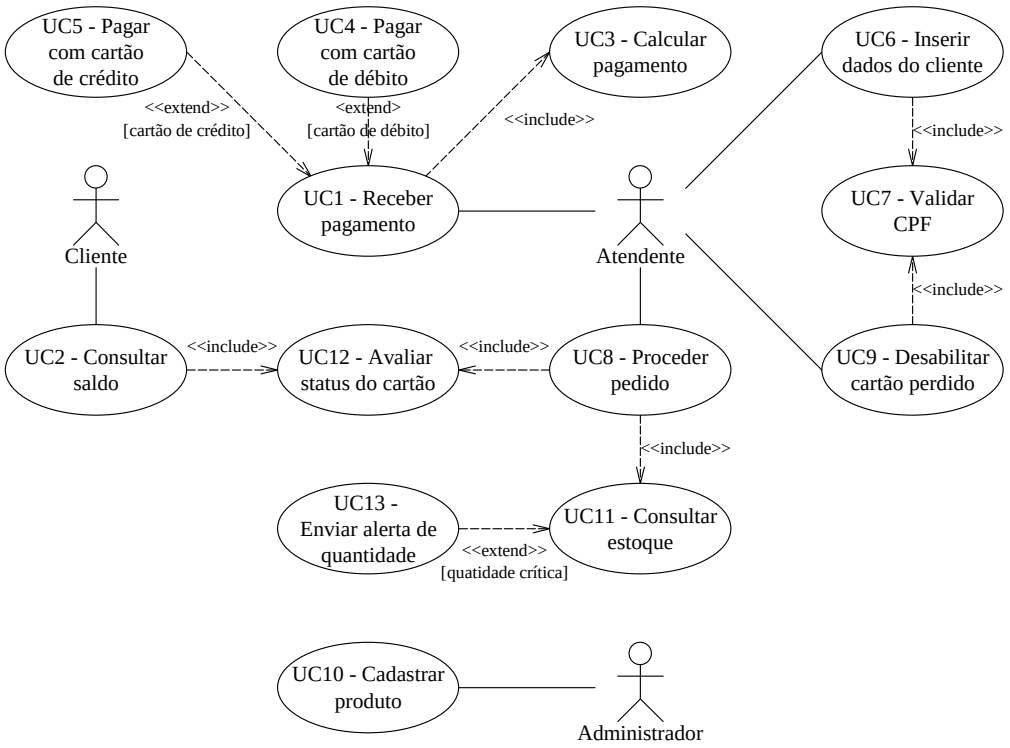


Figura 6.1 - Diagrama de casos de uso para o sistema proposto.

Cada um dos casos de uso apresentados nesse diagrama pode ser especificado através do template proposto por Cockburn [**?**], e desse é possível construir um diagrama de estados que formalize o comportamento do referido caso de uso.

6.3. Descrição dos casos de uso e os diagramas de estados correspondentes

De acordo com o diagrama de casos de uso descrito na seção anterior, é possível especificar cada um dos casos de uso, para então construir o diagrama de estados correspondente para cada desses. Sendo assim, a seguir é feita essa especificação, sendo ainda

UC1 - Receber pagamento: esse caso de uso trata do recebimento do valor consumido do cliente pelo atendente. Para isso, o atendente deve receber o cartão que o cliente utilizou, e pela identificação do seu ID deve-se buscar os produtos, quantidades e valores totais consumidos. Ao ser feita a identificação do cartão, é avaliado se o mesmo não foi dado como perdido, o que pode ocasionar a retenção do cartão. Caso nada seja constatado, o procedimento normal prossegue normalmente. Duas variações podem ocorrer no instante da escolha da forma de pagamento (em dinheiro, cartão de crédito ou cartão de débito). A descrição comportamental do caso de uso 1 é mostrada na Tabela 6.1.

Tabela 6.1 - UC1 - Receber pagamento.

Nome	UC1 - Receber pagamento
Meta	Efetuar os procedimentos para o pagamento da conta
Atores	Atendente
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - capturar ID do cartão 2 - solicitar cálculo da conta 3 - informar o valor da conta 4 - escolher forma de pagamento 5 - confirmar pagamento 6 - acessa a base de dados 7 - armazenar informações da operação na tabela de caixa 8 - remover informações da tabela controle 9 - mostra tela de confirmação da operação
Variações	4 - forma de pagamento 4a - cartão de débito 4b - dinheiro 4b - cartão de crédito
Extensões	4a - cartão de débito 4a1 - Pagamento com cartão de débito (UC4 - Pagar com cartão de débito) 4c - cartão de crédito 4b1 - Pagamento com cartão de crédito (UC5 - Pagar com cartão de crédito)
Casos de Uso Incluídos	UC3 - Calcular pagamento

Para a descrição efetuada, é possível construir um diagrama de estados que modela o comportamento desse caso de uso. Esse diagrama é mostrado na Figura 6.2, onde é possível observar que o fluxo de execução pelos estados E3_UC3, E4a_UC4 e E4c_UC5 modelam a execução de outras máquinas de estados (E_UC3, E_UC4 e E_UC5, respectivamente). Os diagramas que modelam essas máquinas de estados serão apresentados adiante.

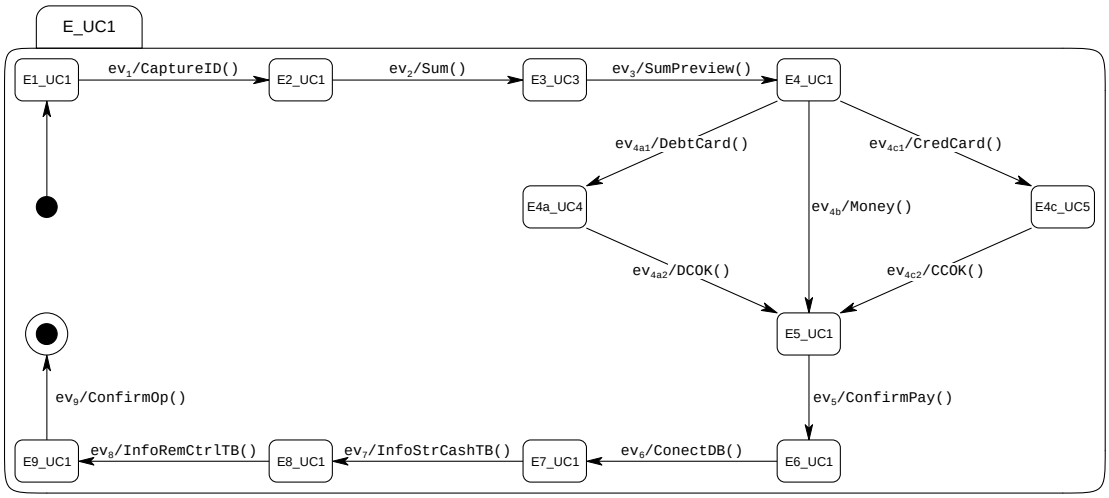


Figura 6.2 - Diagrama de estados que modela UC1.

UC2 - Consultar saldo: esse caso de uso descreva a utilização de um terminal de consulta saldo por parte do atendente. Nessa operação, o *status* do cartão também é avaliado antes de fornecer as informações desejadas. Caso o cartão esteja apto para a operação, é feita uma consulta na base de dados para determinar o valor consumido pelo cliente. A descrição desse caso uso é mostrada na tabela 6.2.

Tabela 6.2 - UC2 - Consultar saldo.

Nome	UC2 - Consultar saldo
Meta	Informar o valor consumido pelo cliente
Atores	Cliente
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - capturar id do cartão 2 - avaliar status do cartão 3 - acessar a base de dados 4 - selecionar e somar todos os valores consumidos nesse cartão 5 - enviar mensagem com o valor para o visor
Variações	-
Extensões	-
Casos de Uso Incluídos	UC12 - Avaliar *status* do cartão

Para esse descrição, o seguinte diagrama de estados é gerado (Figura 6.3):

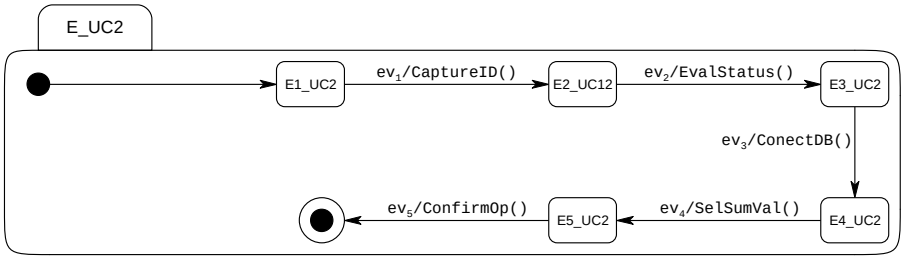


Figura 6.3 - Diagrama de estados que modela UC2.

Nesse diagrama, quando a máquina de estados estiver no estado E2_UC12, a máquina de estados EUC12 estará em execução.

UC3 - Calcular pagamento: esse é um caso de uso incluído do caso de uso 1, que trata do recebimento do pagamento. Nesse caso de uso são efetuadas as operações que calculam os valores consumidos pelo cliente. A Tabela 6.3 especifica o comportamento desse caso de uso.

Tabela 6.3 - UC3 - Calcular pagamento.

Nome	UC3 - Calcular pagamento
Meta	Calcular o valor consumido pelo cliente
Atores	Atendente
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - recebe id do cartão 2 - acessar a base de dados 3 - selecionar e somar todos os valores consumidos nesse cartão 4 - retornar o valor total consumido
Variações	-
Extensões	-
Casos de Uso Incluídos	-

O diagrama de estados que é gerado a partir da descrição feita, é apresentado na Figura 6.4.

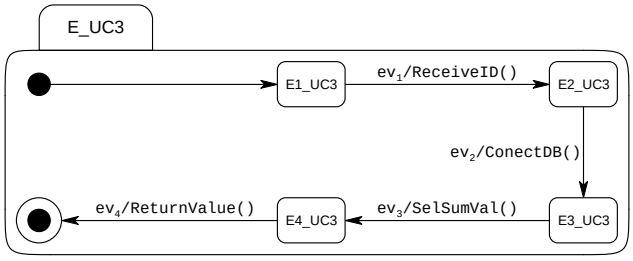


Figura 6.4 - Diagrama de estados que modela UC3.

Esse diagrama não possui estados que modelam a execução de outra máquina de estados.

UC4 - Pagar com cartão de débito: como um caso de uso incluído do caso de uso 1, esse trata da operação de pagamento com o cartão de crédito. No início do processamento dessa funcionalidade, o limite do cartão é consultado. Caso o saldo disponível seja menor que a conta a ser paga, a operação será finalizada. Caso contrário, o procedimento normal prossegue, conforme descrito na Tabela 6.4.

A Figura 6.5 mostra o diagrama de estados gerado a partir da descrição do caso de uso 4.

Como o caso de uso que esse diagrama modela não possui casos de uso subordinados, seus estados tratam somente do comportamento do caso de uso 4.

Tabela 6.4 - UC4 - Pagar com cartão de débito.

Nome	UC4 - Pagar com cartão de débito
Meta	Coletar informações necessárias para o pagamento com cartão de crédito
Atores	Cliente
Pré-condições	Disponibilidade de limite
Pós-condições	-
Cenário de Sucesso Principal	1 verificar limite do cartão de débito 2 - mostrar tela de inserção de senha 3 - digitar senha 4 - mostrar tela de confirmação da operação
Variações	1a - limite ultrapassado 2a1 - terminar a operação
Extensões	-
Casos de Uso Incluídos	-

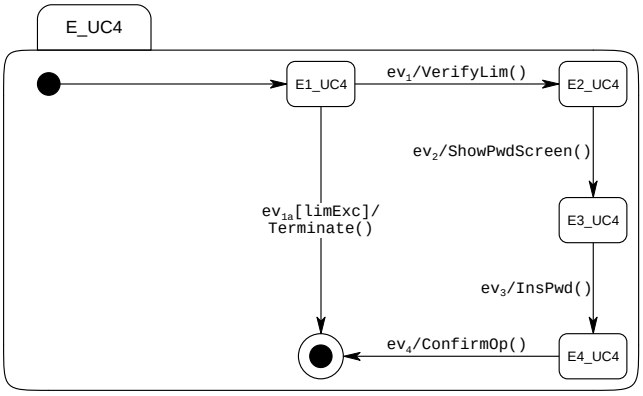


Figura 6.5 - Diagrama de estados que modela UC4.

UC5 - Pagar com cartão de crédito: assim como nó item anterior, esse também é um caso de uso subordinado ao caso de uso 1, e trata do pagamento com cartão de crédito. Além da consulta do limite, é possível escolher as condições de pagamento. A Tabela 6.5 apresenta os detalhes desse caso de uso 5.

Tabela 6.5 - UC5 - Pagar com cartão de crédito.

Nome	UC5 - Pagar com cartão de crédito
Meta	Coletar informações necessárias para o pagamento com cartão de crédito
Atores	Cliente
Pré-condições	Disponibilidade de limite
Pós-condições	-
Cenário de Sucesso Principal	1 - verificar limite do cartão de crédito 2 - escolher forma de pagamento 3 - mostrar tela de inserção de senha 4 - digitar senha 5 - mostrar tela de confirmação da operação
Variações	2a - limite ultrapassado 2a1 - terminar a operação 3a - forma de pagamento 3a1 - pagamento em parcela única 3a2 - pagamento em duas parcelas 3a3 - pagamento em três parcelas
Extensões	-
Casos de Uso Incluídos	-

O diagrama de estados que modela o caso de uso que trata do pagamento com cartão de crédito é exposto na Figura 6.6.

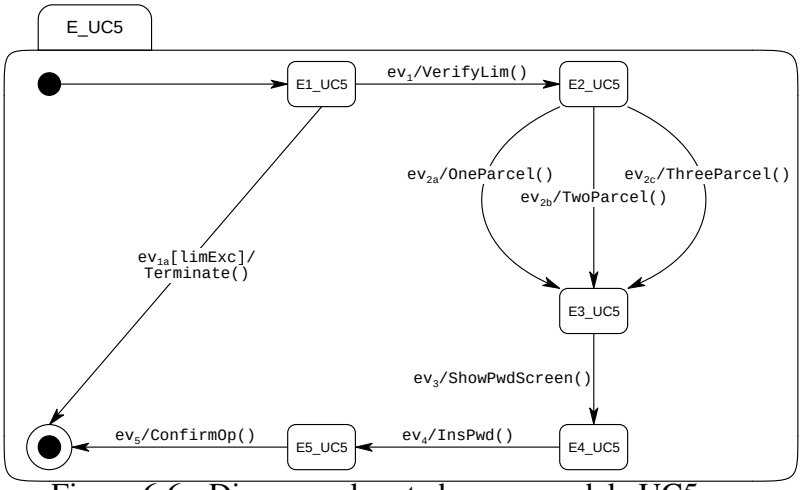


Figura 6.6 - Diagrama de estados que modela UC5.

Assim como no ítem anterior, os estados desse diagrama modelam apenas a execuções referentes ao caso de uso 5.

UC6 - Inserir dados do cliente: a tarefa de inserção dos dados do cliente no sistema é atribuída ao caso de uso 6. A execução dessa funcionalidade passa pela validação do CPF do cliente, tarefa que será executada pelo caso de uso 7. Caso o CPF do cliente seja inválido, a execução dessa funcionalidade não prossegue, sendo então finalizada. Esse comportamento é descrito na Tabela 6.6.

Tabela 6.6 - UC6 - Inserir dados do cliente.

Nome	UC6 - Inserir dados do cliente
Meta	Cadastrar dados necessários do cliente para a vinculação ao cartão
Atores	Atendente
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - captura id do cartão 2 - Insere CPF 3 - Valida CPF 4 - Insere nome 5 - acessa a base de dados 6 - insere dados na tabela de controle 7 - mostra tela de confirmação da operação
Variações	-
Extensões	-
Casos de Uso Incluídos	UC7 - Validar CPF

O template que descreve o comportamento do caso de uso 5 habilita a construção do diagrama de estado que modela esse comportamento. Sendo assim, o referido diagrama é mostrado na Figura 6.7.

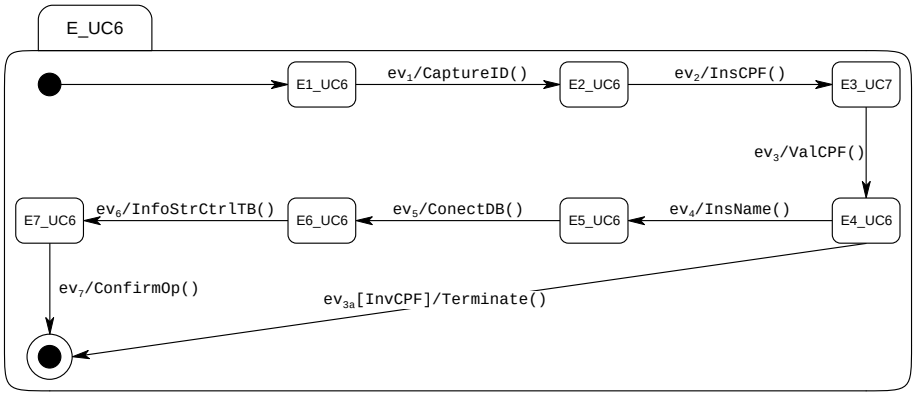


Figura 6.7 - Diagrama de estados que modela UC6.

Conforme dito aqui, a tarefa de validar o CPF informado pelo cliente será executado em um caso de uso subordinado. Dessa forma, quando a máquina de estados estiver em E3_UC7, a máquina de estados que modela a tarefa de verificação do CPF (EUC7) estará sendo executada.

UC7 - Validar CPF: esse é o caso de uso que trata da validação do CPF, relatado no ítem anterior. Essa funcionalidade é especificada na Tabela 6.7.

A Figura 6.8 exibe o diagrama de estados que representa o comportamento esperado para o caso de uso que trata da validação do CPF do cliente.

Como pode ser observado nesse diagrama de estados, os estados referenciam apenas os possíveis comportamentos do caso de uso 7.

Tabela 6.7 - UC7 - Validar CPF.

Nome	UC7 - Validar CPF
Meta	Verificar e validar o CPF informado
Atores	-
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - captura CPF 2 - avalia CPF 3 - retorna a validação do CPF
Variações	-
Extensões	-
Casos de Uso Incluídos	-

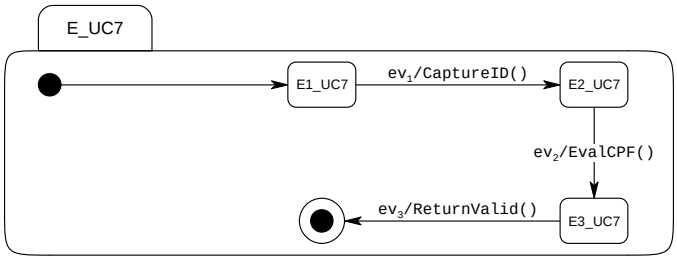


Figura 6.8 - Diagrama de estados que modela UC7.

UC8 - Proceder pedido: nesse caso de uso são executados os procedimentos de processamento e entrega de um pedido feito por um cliente. Como pode ser observado na Tabela 6.8, esse caso de uso possui duas extensões que indicam a execução de sub-funcionalidades necessárias para a tarefa de venda de um produto a um cliente. A primeira refere-se à verificação do status do cartão utilizado na compra e a segunda à consulta de quantidade do produto no estoque, para a certificação de que o produto solicitado está disponível.

No diagrama de estados construído (Figura 6.9) é possível visualizar os estados que modelam a execução das máquinas de estados EUC11 e ECU12.

Assim como em outros diagramas de estados, estados que possuem a identificação de um caso de uso diferente do que está sendo executado (E2_UC12 e E6_UC11 nesse diagrama), indica a execução de outra máquina de estados.

Tabela 6.8 - UC8 - Proceder pedido.

Nome	UC8 - Proceder pedido
Meta	Efetuar pedido do cliente
Atores	Atendente
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - capturar id do cartão 2- avaliar status do cartão 3 - insere o produto 4 - insere a quantidade 5 - acessa a base de dados 6 - consulta estoque 7 - insere pedido na tabela de controle 8- decrementa quantidade da tabela de estoque 9 - mostra tela de confirmação da operação
Variações	7 - Inserir pedido 7a - Terminar a operação
Extensões -	
Casos de Uso Incluídos	UC12 - Avaliar *status* do cartão UC11 - Consultar estoque

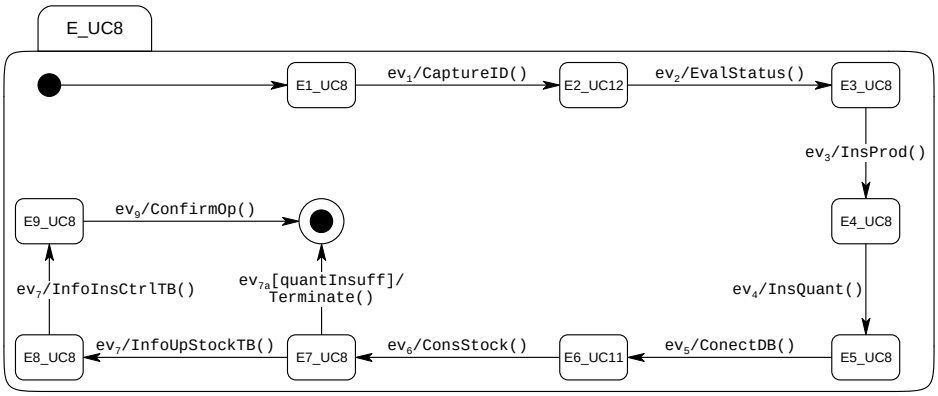


Figura 6.9 - Diagrama de estados que modela UC8.

UC9 - Desabilitar cartão perdido: esse funcionalidade do sistema é executada quando um cliente informa ao atendente que o cartão ques estava utilizando foi perdido. O atendente busca a identificação do cartão pelo número do CPF do cliente, sendo então necessário validar o número informado (caso de uso incluído). Ao obter-se a identificação desse cartão, o mesmo ficará inapto para novas compras. Mais detalhes sobre essa operação são apresentados na Tabela 6.9.

O diagrama de estados correspondente a esse especificação de caso de uso é exposto na Figura 6.10.

Assim como no caso de uso responsável pela inserção dos dados do cliente no sistema (UC6), esse também possui o caso de uso 7 subordinado a execução do caso de uso corrente. Sendo assim, quando a máquina de estados estiver no estado E2_UC7, indicará

Tabela 6.9 - UC9 - Desabilitar cartão perdido.

Nome	UC9 - Desabilitar cartão perdido
Meta	Inabilitar a utilização de um cartão dado como perdido
Atores	Atendente
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - Digita CPF 2 - valida CPF 3 - acessa a base de dados 4 - altera o status do cartão 5 - mostra tela de confirmação da operação
Variações	-
Extensões	-
Casos de Uso Incluídos	UC7 - Validar CPF

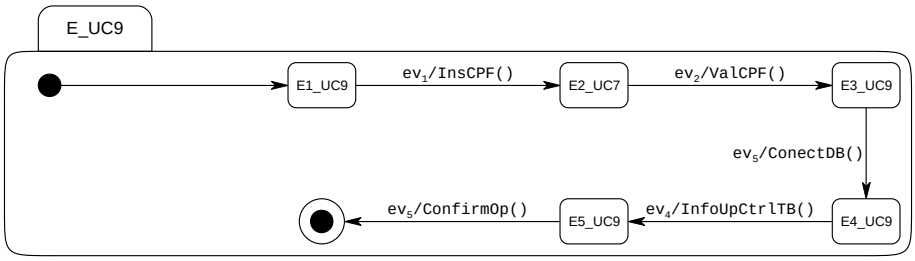


Figura 6.10 - Diagrama de estados que modela UC9.

que a máquina de estados que modela UC7 está sendo executada.

UC10 - Cadastrar produto: esse tarefa é executada pelo administrador, sendo essa responsável pelo cadastramento de produtos no sistema, informando para os mesmos a quantidade disponível em estoque e a quantidade crítica de produto, que acarretará no envio de uma mensagem de alerta quando essa quantidade for atingida. O comportamento desse caso de uso é especificado na Tabela 6.10.

O diagrama de estados descrito na Figura 6.11 pôde ser construído a partir das informações fornecidas pela Tabela 6.10.

Por não possuir casos de uso subordinados, esse diagrama modela apenas o comportamento do caso de uso 10.

Tabela 6.10 - UC10 - Cadastrar produto.

Nome	UC10 - Cadastrar produto
Meta	Inserir informações sobre produtos disponíveis para consumo
Atores	Administrador
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - entra no sistema 2 - insere id do produto 3 - insere quantidade do produto 4 - informa quantidade crítica de produto 5 - acessa a base de dados 6 - insere dados na tabela de estoque 7 - mostra tela de confirmação da operação
Variações	-
Extensões	-
Casos de Uso Incluídos	-

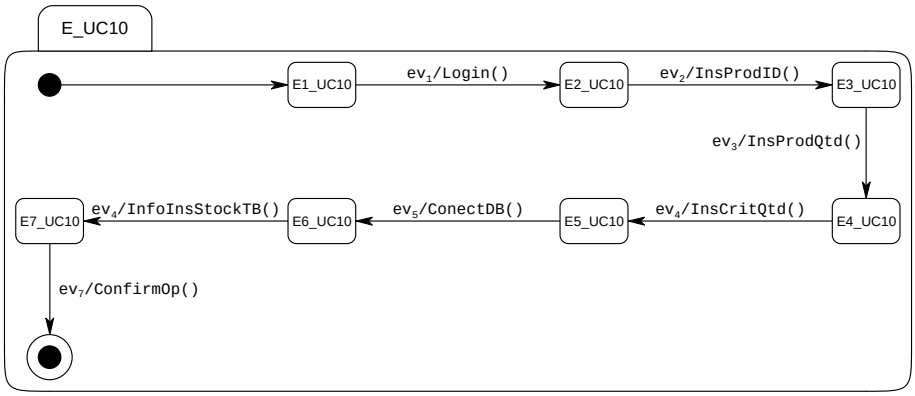


Figura 6.11 - Diagrama de estados que modela UC10.

UC11 - Consultar estoque: essa funcionalidade do sistema é acionada quando um atendente efetua um pedido solicitado pelo cliente. Durante a sua execução, ocasionalmente pode ser constatado que a quantidade disponível em estoque atingiu o ponto crítico, quantidade essa informada pelo administrador no instante do cadastro do produto. Essa situação é descrita na Tabela 6.11.

O envio de um alerta de quantidade, descrito pela extensão 4a da tabela, é executado em outra máquina de estados, enquanto a máquina de estados que modela UC11 (Figura 6.12) permanecer no estado E4a_UC13.

Tabela 6.11 - UC11 - Consultar estoque.

Nome	UC11 - Consultar estoque
Meta	Verificar a quantidade de produto disponível
Atores	-
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - recebe produto 2 - acessa a base de dados 3 - consulta a quantidade do produto na tabela de estoque 4 - retorna disponibilidade do produto
Variações	-
Extensões	4a - quantidade crítica 4a1 - envia alerta de quantidade (UC13 - Enviar alerta de quantidade)
Casos de Uso Incluídos	-

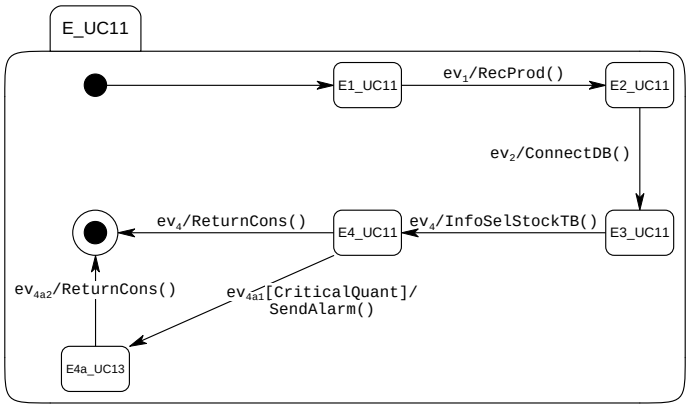


Figura 6.12 - Diagrama de estados que modela UC11.

UC12 - Avaliar *status* do cartão: o status do cartão será avaliado no instante da consulta de um saldo consumido ou quando for efetuado um pedido de produto. Essa avaliação executa duas operações em paralelo: a primeira trata da impossibilidade do uso do cartão devido a uma perda informada anteriormente, e a segunda remete ao alcance de um valor limite de consumo (o que inabilita o consumo até que parte do valor consumido seja quitado). Essa configuração é descrita na Tabela 6.12.

A execução paralela das consultas do *status* do cartão e do valor consumido são modelado com o uso de um *fork*, como pode ser observado na Figura 6.13.

Por não disponibilizar de regiões de sincronização, a modelagem dessas estruturas na ferramenta Rose da Rational deve ser composta por pares de *forks* e *joins*, ou seja, para cada *fork* especificado deve possuir um *join* que finalize a região de sincronização.

Tabela 6.12 - UC12 - Avaliar *status* do cartão.

Nome	UC12 - Avaliar *status* do cartão
Meta	Verifica se o cartão está apto a ser utilizado
Atores	-
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - recebe id do cartão 2 - acessa a base de dados 3.1 - consulta o *status* do cartão na tabela de controle 3.2 - consulta o saldo consumido na tabela de controle 4 - retorna as informações solicitadas
Variações	-
Extensões	-
Casos de Uso Incluídos	-

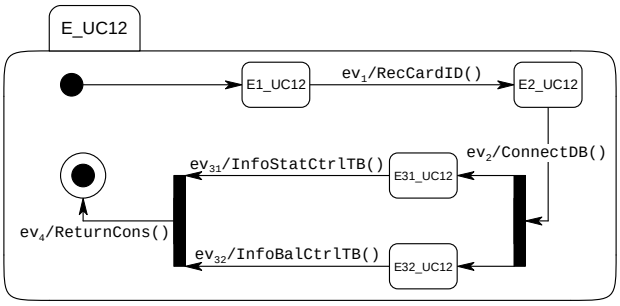


Figura 6.13 - Diagrama de estados que modela UC12.

UC13 - Enviar alerta de quantidade: essa funcionalidade do sistema (descrita na Tabela 6.13) será executada quando for detectado que a quantidade de um produto atingiu um nível crítico, sendo essa associada ao caso de uso que trata da consulta de estoque (UC11).

Tabela 6.13 - UC13 - Enviar alerta de quantidade.

Nome	UC13 - Enviar alerta de quantidade
Meta	Avisar o administrador que o produto atingiu uma quantidade crítica
Atores	-
Pré-condições	-
Pós-condições	-
Cenário de Sucesso Principal	1 - recebe solicitação de alarme 2 - envia mensagem para o administrador 3 - mostra tela de confirmação da operação
Variações	-
Extensões	-
Casos de Uso Incluídos	-

A presente descrição possibilita a construção do diagrama de estados apresentado na Figura 6.14.

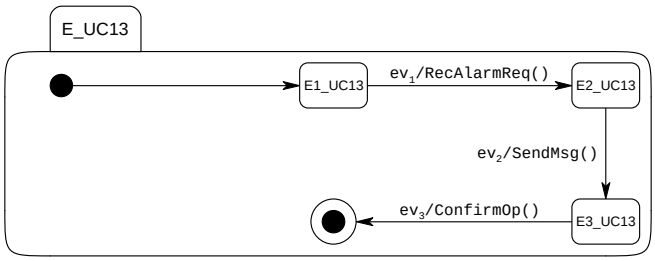


Figura 6.14 - Diagrama de estados que modela UC13.

Como o caso de uso modelado pelo diagrama de estados EUC13 não possui casos de uso subordinados, os estados modelam apenas o comportamento esperado para o caso de uso UC13.

Com a especificação dos diagramas de estado que modelam o comportamento dos casos de uso de sistema, é possível aplicar o método descrito no capítulo anterior para construir a SAN equivalente para o sistema, ou para componentes do mesmo, conforme a análise desejada.

6.4. SAN Resultante

Conforme formalizado no capítulo anterior, a tradução dos diagramas de estados que descrevem o comportamento dos casos de uso do sistema pode disponibilizar uma análise para o sistema como um todo (contando aqui com a interação dos atores com o sistema) ou dos componentes do mesmo, sendo um componente uma funcionalidade disponibilizada para uma ator. A seguir serão expostas a análises proporcionadas através do método proposto nesse trabalho, para os diagramas de estados construídos nesse estudo de caso.

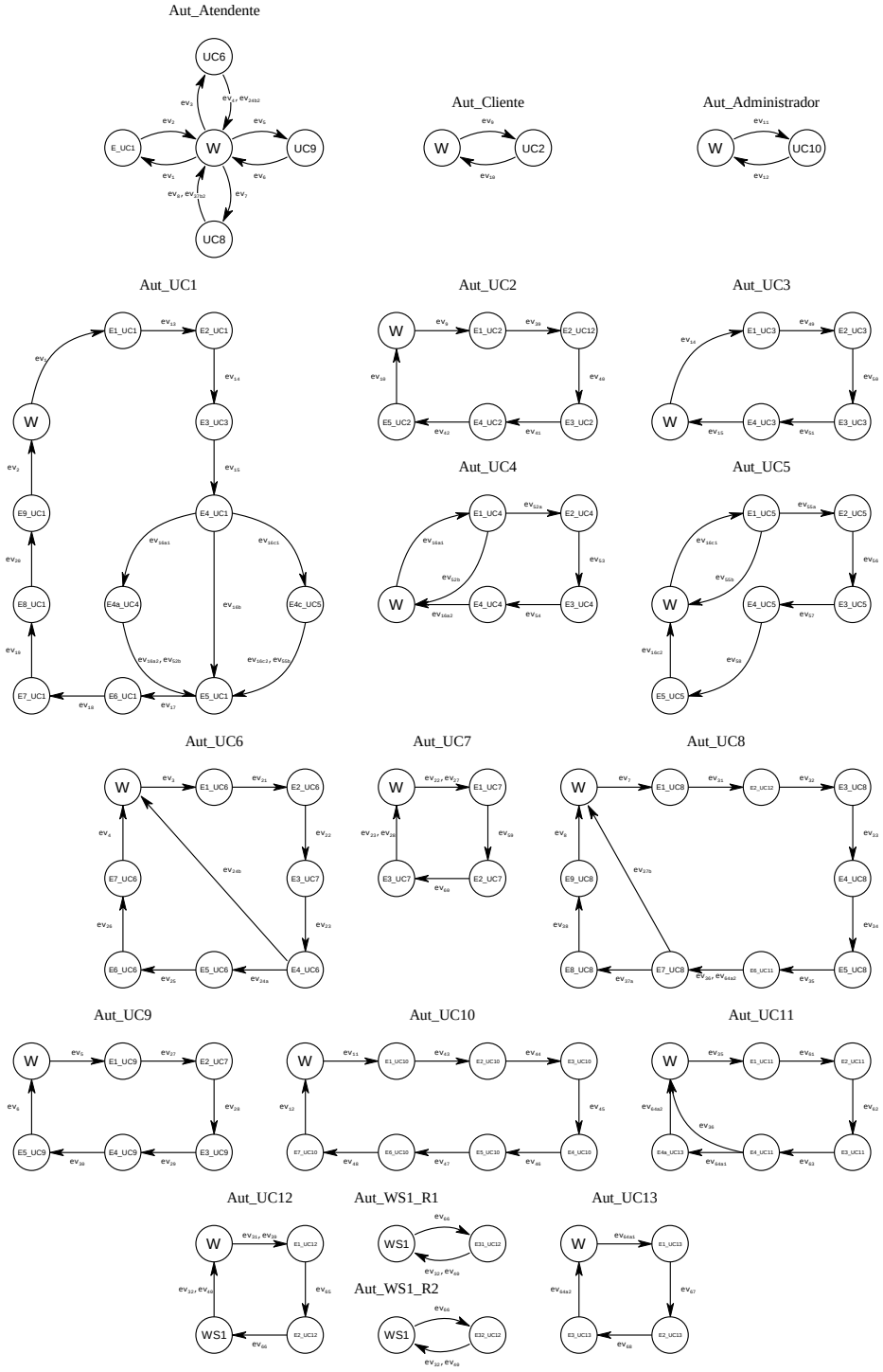
6.4.1. Análise do Sistema

Com os diagramas de estados que descrevem o comportamento esperado para o estudo de caso proposto, é possível construir uma única SAN que modele o comportamento geral do sistema, levando em conta a interação do mesmo com seus atores. Para o estudo de caso proposto, a SAN resultante que modela esse comportamento é apresentada na Figura 6.15.

Com essa SAN é possível extrair casos de teste que sejam direcionados à validação do sistema como um todo, levando em conta a interação dos atores com todos os componentes do sistema.

Nesse instante da pesquisa foi verificado que apesar da modelagem de um sistema inteiro em SAN ser possível, sua análise na ferramenta PEPS2003 [BBF+03] seria inviável. Apesar do sistema proposto não ser complexo, a quantidade de autômatos e de estados obtida inviabiliza a análise do modelo de uso do sistema. Para termos uma idéia, caso fôssemos modelar esse sistema em cadeias de Markov, o autômato resultante possuiria 1.274.019.840.000 estados.

Nesse ponto partiu-se para uma nova estratégia. A análise dos casos de uso que possuem interação direta com um ator será feita conforme descrito até aqui, uma vez que a quantidade de autômatos e estados viabiliza a análise, podendo então ser fonte para testes de componente e integração, dependendo do nível de granularidade da especificação dos casos de uso. Para a análise do sistema como



Tipo	Evento	Taxa
syn	ev_{1}	0.25
syn	ev_{2}	1
syn	ev_{3}	0.25
syn	ev_{4}	1
syn	ev_{5}	0.01
syn	ev_{6}	1
syn	ev_{7}	0.49
syn	ev_{8}	1
syn	ev_{9}	1
syn	ev_{10}	1
syn	ev_{11}	1
syn	ev_{12}	1
loc	ev_{13}	1
syn	ev_{14}	1
syn	ev_{15}	1
syn	ev_{16a1}	0.5
syn	ev_{16a2}	1
loc	ev_{16b}	0.3
syn	ev_{16c1}	0.2
syn	ev_{16c2}	1
loc	ev_{17}	1
loc	ev_{18}	1
loc	ev_{19}	1
loc	ev_{20}	1
loc	ev_{21}	1
syn	ev_{22}	1
syn	ev_{23}	1
loc	ev_{24a}	0.85
syn	ev_{24b}	0.15
loc	ev_{25}	1
loc	ev_{26}	1
syn	ev_{27}	1
syn	ev_{28}	1
loc	ev_{29}	1
loc	ev_{30}	1
syn	ev_{31}	1
syn	ev_{32}	1
loc	ev_{33}	1
loc	ev_{34}	1
syn	ev_{35}	1
syn	ev_{36}	0.95
loc	ev_{37a}	0.7
syn	ev_{37b}	0.3
loc	ev_{38}	1
syn	ev_{39}	1
syn	ev_{40}	1
loc	ev_{41}	1
loc	ev_{42}	1
loc	ev_{43}	1
loc	ev_{44}	1
loc	ev_{45}	1
loc	ev_{46}	1
loc	ev_{47}	1
loc	ev_{48}	1
loc	ev_{49}	1
loc	ev_{50}	1
loc	ev_{51}	1
loc	ev_{52a}	0.9
syn	ev_{52b}	0.1
loc	ev_{53}	1
loc	ev_{54}	1
loc	ev_{55a}	0.8
syn	ev_{55b}	0.2
loc	ev_{56}	1
loc	ev_{57}	1
loc	ev_{58}	1
loc	ev_{59}	1
loc	ev_{60}	1
loc	ev_{61}	1
loc	ev_{62}	1
loc	ev_{63}	1
loc	ev_{64a1}	1
syn	ev_{64a2}	0.05
loc	ev_{65}	1
syn	ev_{66}	1
loc	ev_{67}	1
loc	ev_{68}	1

Figura 6.15 - SAN resultante para o sistema proposto.

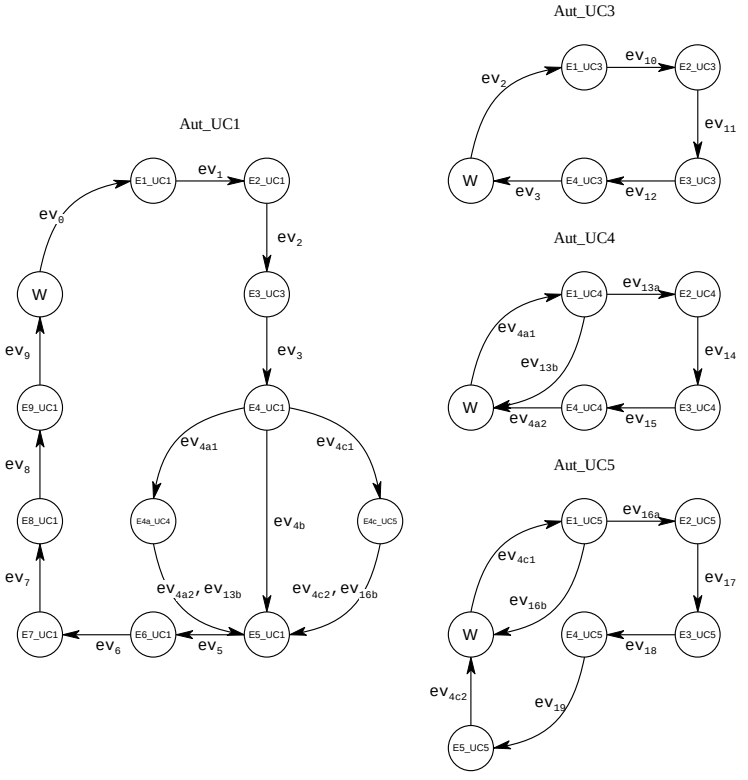
um todo, é necessário um processo de simplificação dos autômatos. Esse processo será descrito na próxima seção.

6.4.2. Análise do Componente

Para cada caso de uso relacionado diretamente com um ator, uma SAN deve ser construída, buscando assim modelar o comportamento do sistema quando o mesmo interage com um ator. No sistema proposto, o atendente interage diretamente com quatro casos de uso, enquanto o cliente com um e o administrador com outro.

Sendo assim, nosso sistema pode ser modelado com seis SAN, sendo que casos de usos compartilhados são modelados em cada uma das SAN que utilizem-no. Se por ventura mais de um ator compartilharem o mesmo caso de uso, não há necessidade da modelagem de uma SAN distinta para cada ator, uma vez que o caso de uso é o mesmo.

A Figura 6.16 modela a interação entre o caso de uso que trata do recebimento do pagamento com os seus casos de uso subordinados. O atendente interage diretamente com esse caso de uso.



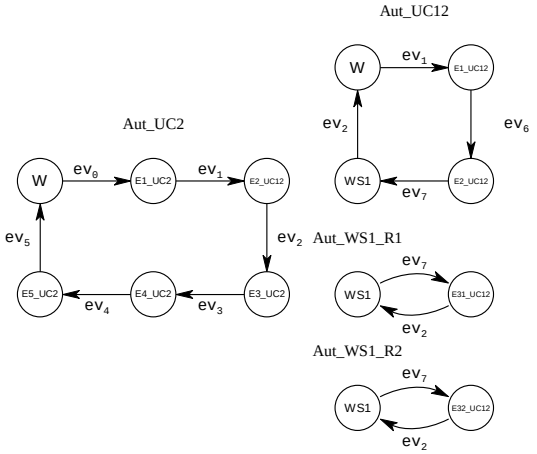
Tipo	Evento	Taxa
loc	ev_0	1
loc	ev_1	1
syn	ev_2	1
syn	ev_3	1
syn	ev_{4a1}	0.5
syn	ev_{4a2}	1
loc	ev_{4b}	0.3
syn	ev_{4c1}	0.2
syn	ev_{4c2}	1
loc	ev_5	1
loc	ev_6	1
loc	ev_7	1
loc	ev_8	1
loc	ev_9	1
loc	ev_{10}	1
loc	ev_{11}	1
loc	ev_{12}	1
loc	ev_{13a}	0.9
syn	ev_{13b}	0.1
loc	ev_{14}	1
loc	ev_{15}	1
loc	ev_{16a}	0.8
syn	ev_{16b}	0.2
loc	ev_{17}	1
loc	ev_{18}	1
loc	ev_{19}	1

Figura 6.16 - SAN resultante para o caso de uso ”Receber pagamento”.

O componente do sistema que possibilita o cliente consultar o saldo dos produtos consumidos é modelado pela SAN descrita na Figura 6.17. Essa SAN é composta por dois autômatos, cada um modelando um dos casos de uso que constituem esse componente (UC2 e UC12).

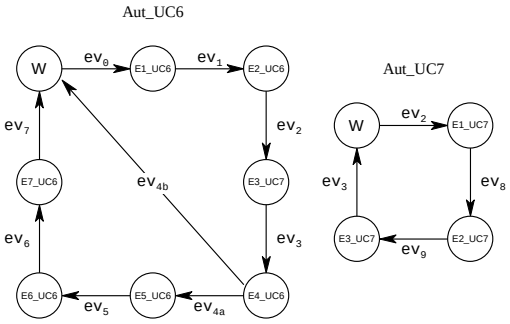
A tarefa de inserção dos dados do cliente, efetuada pelo atendente, é modelada pela SAN exposta na Figura 6.18. Nela pode ser observados dois autômatos, cada qual nomeado de acordo com o caso de uso que modela, sendo esses constituintes do componente que trata da inserção de dados.

A funcionalidade responsável pelo processamento dos pedidos do cliente, também de respons-



Tipo	Evento	Taxa
loc	ev_0	1
syn	ev_1	1
syn	ev_2	1
loc	ev_3	1
loc	ev_4	1
loc	ev_5	1
loc	ev_6	1
syn	ev_7	1

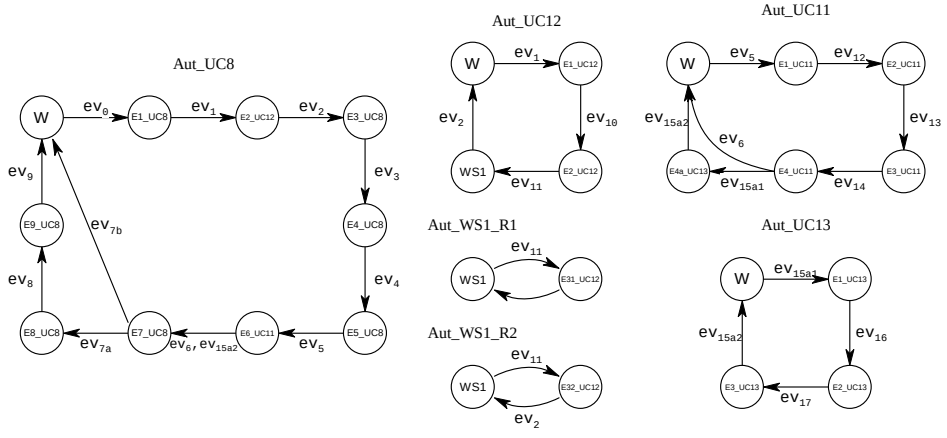
Figura 6.17 - SAN resultante para o caso de uso ”Consultar saldo”.



Tipo	Evento	Taxa
loc	ev_0	1
loc	ev_1	1
syn	ev_2	1
syn	ev_3	1
loc	ev_{4a}	0.85
loc	ev_{4b}	0.15
loc	ev_5	1
loc	ev_6	1
loc	ev_7	1
loc	ev_8	1
loc	ev_9	1

Figura 6.18 - SAN resultante para o caso de uso ”Inserir dados”.

abilidade do atendente, a qual é composta por quatro casos de uso. A SAN descrita na Figura 6.19 é composta por seis autômatos: quatro descrevem os casos de uso do componente e outros dois modela as regiões de sincronização derivadas do disparo sincronizante efetuado pelo *fork* do caso de uso UC12.

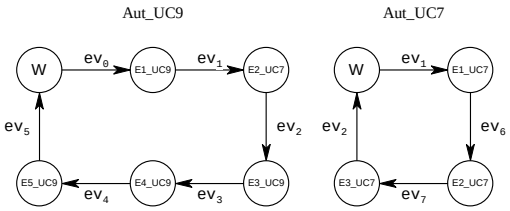


Tipo	Evento	Taxa
loc	ev_0	1
syn	ev_1	1
syn	ev_2	1
loc	ev_3	1
loc	ev_4	1
syn	ev_5	1
syn	ev_6	0.95
loc	ev_{7a}	0.7
loc	ev_{7b}	0.3
loc	ev_8	1
loc	ev_9	1
loc	ev_{10}	1
syn	ev_{11}	1
loc	ev_{12}	1
loc	ev_{13}	1
loc	ev_{14}	1
syn	ev_{15a1}	0.05
syn	ev_{15a2}	1
loc	ev_{16}	1
loc	ev_{17}	1

Figura 6.19 - SAN resultante para o caso de uso ”Proceder pedido”.

A modelagem do componente responsável por desabilitar um cartão dado como perdido, tarefa também executada pelo atendente, é mostrada na Figura 6.20.

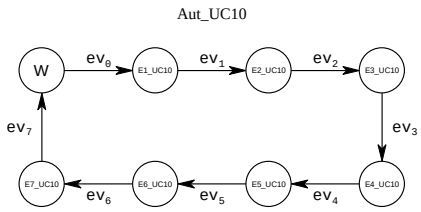
Pos fim, na Figura 6.21 é modelada a funcionalidade do sistema responsável pelo cadastramento



Tipo	Evento	Taxa
loc	ev_0	1
syn	ev_1	1
syn	ev_2	1
loc	ev_3	1
loc	ev_4	1
loc	ev_5	1
loc	ev_6	1
loc	ev_7	1

Figura 6.20 - SAN resultante para o caso de uso ”Desabilitar cartão perdido”.

dos produtos a serem disponibilizados ara os clientes, tarefa essa executada pelo administrador. Essa SAN é composta por um único autômato que descreve o único caso de uso que constitui esse componente



Tipo	Evento	Taxa
loc	ev_0	1
loc	ev_1	1
loc	ev_2	1
loc	ev_3	1
loc	ev_4	1
loc	ev_5	1
loc	ev_6	1
loc	ev_7	1

Figura 6.21 - SAN resultante para o caso de uso ”Cadastrar produto”.

6.5. Simplificação da SAN Resultante

Seguindo os passos descritos no capítulo anterior, é possível construir uma SAN mais enxuta, mas que ainda apresenta as informações necessárias para a análise de cobertura. Com a execução do módulo 2, é possível obter uma nova SAN com menos autômatos e estados, porém mantendo as características que viabilizem a composição de testes de sistema. Para ilustrar o passos da simplificação aplicados para o estudo de caso proposto nesse trabalho, é apresentada a Figura 6.22.

A figura (I) mostra a SAN gerada a partir da análise de sistema, sobre os diagramas de estados do estudo de caso, sendo destacados os estados e transições que serão substituídos por um estado equivalente. A referida substituição é mostrada na figura (II), conforme previsto na simplificação N1, sendo mantidas as transições e eventos que atingem e partem do trecho seqüencial original.

A simplificação N2 é mostrada na figura (III), onde os estados equivalentes são substituídos por uma única transição entre os estados-chave dos autômatos. Por fim é verificada a existência de autômatos que possuem apenas um estado, sendo que esse possua uma auto transição, e a existência de múltiplas transições entre dois estados. Nessas situações, os autômatos com um único estado serão eliminados, e os eventos que sincronizavam a execução desse autômato serão substituídos por eventos locais. Ainda, as múltiplas transições serão substituídas por uma única transição, sendo que essa passa a conter os eventos das transições originais. O resultado dessas transformações são apresentadas na figura (IV).

Com a aplicação das simplificações efetuadas na execução do módulo 2 do protótipo, uma nova SAN é concebida, sendo essa descrita na Figura 6.23, onde também são descritos os eventos, seus tipos e as taxas de ocorrência correspondentes.

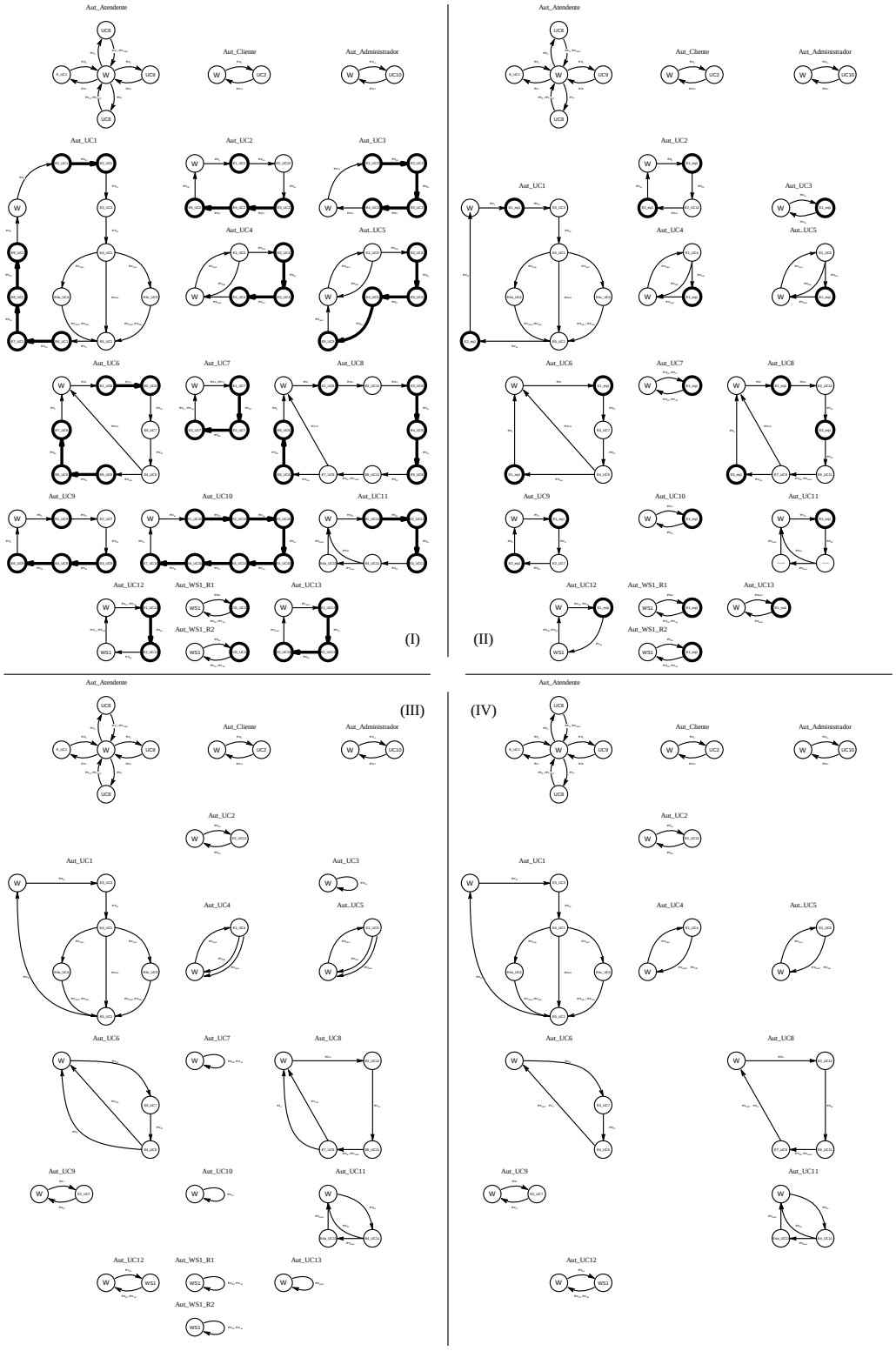
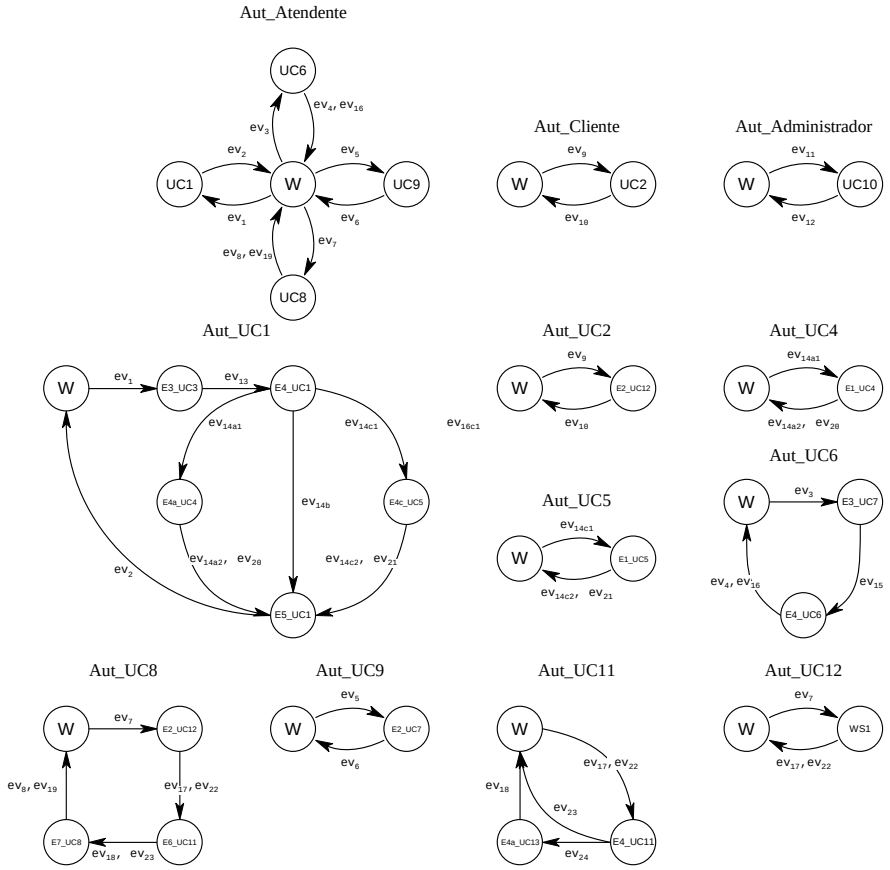


Figura 6.22 - Simplificação da SAN.



Tipo	Evento	Taxa
syn	ev_1	0.25
syn	ev_2	1
syn	ev_3	0.25
syn	ev_4	0.85
syn	ev_5	0.01
syn	ev_6	1
syn	ev_7	0.49
syn	ev_8	0.7
syn	ev_9	1
syn	ev_{10}	1
loc	ev_{11}	1
loc	ev_{12}	1
loc	ev_{13}	1
syn	ev_{14a1}	0.5
syn	ev_{14a2}	1
loc	ev_{14b}	0.3
syn	ev_{14c1}	0.2
syn	ev_{14c2}	1
loc	ev_{15}	1
syn	ev_{16}	0.15
syn	ev_{17}	1
syn	ev_{18}	0.05
syn	ev_{19}	0.3
syn	ev_{20}	1
syn	ev_{21}	1
syn	ev_{22}	1
loc	ev_{23}	0.05
loc	ev_{24}	0.95

Figura 6.23 - SAN simplificada.

O espaço total de espaços gerados nessa nova SAN é de 138.240 estados, o que torna viável a análise desse modelo na ferramenta PEPS2003.

No próximo capítulo são apresentadas as conclusões extraídas nesse trabalho, sendo retomado o contexto onde as soluções propostas são aplicadas. Ainda serão expostas as limitações que a presente solução possui, sendo também sugeridas alternativas para essas limitações (trabalhos futuros).

7. CONCLUSÃO

O intuito do presente trabalho é agregar o formalismo SAN ao processo unificado de desenvolvimento de *software*, visando a geração de casos de teste. A escolha da metodologia do processo unificado como base para a estruturação da presente proposta, deve-se a sua ampla utilização, por ser um processo que utiliza interações para evitar o impactos de mudanças, por ser dirigido a casos de uso, por utilizar os elementos da UML no preparo dos seus artefatos, e por definir os testes de *software* como uma importante fase do projeto.

7.1. Resultados prévios revistos nessa dissertação

Dentre os estudos relacionados, alguns relatam emprego de diagramas de estados para descrever o comportamento do sistema, utilização essa sugerida no processo unificado. A partir desses diagramas é possível compor o modelo de uso do sistema, sendo esse usado como base para a construção de casos de teste de *software*.

Outras técnicas utilizam métodos estatísticos para a análise, e também como base para a construção de casos de teste. Dentre essas técnicas, é descrita a utilização do perfil operacional do sistema, de onde são capturadas informações sobre probabilidade de uso e taxas de ocorrência, para a posterior construção de cadeias de Markov que representam o comportamento do *software*.

Apesar das notórias vantagens apresentadas pelo formalismo de cadeias de Markov, o mesmo pode apresentar limitações referentes ao espaço total de estados gerados, podendo não oferecer uma precisão necessária para a descrição de sistemas complexos. Além disso, um modelo descrito por cadeias de Markov pode apresentar dificuldades para a legibilidade do sistema, sob a ótica de um observador humano.

O uso de SAN para a construção de casos de teste de *software* mostrou-se eficaz em estudos anteriores, principalmente no que tange o ganho obtido na análise de cobertura dos casos de teste construídos. Além de propiciarem a modularização do modelo de uso do sistema, e por possibilitar a modelagem de sincronismo e paralelismo, a análise viabilizada por esse formalismo permite ainda fornecer informações que possam ser utilizadas como auxílio em tomadas de decisões, por exemplo.

7.2. Contribuição central desse dissertação

Sendo assim, os diagramas de estados foram empregados como fonte de informações para a composição de autômatos constituintes de uma SAN. Dessa forma, foi formalizada a transcrição dos elementos dos diagramas de estados UML, para uma estrutura equivalente em SAN.

Para formalizar a tradução dos componentes de um diagrama de estados UML para um estrutura equivalente em SAN, foi desenvolvido um *framework* de transcrição. Com essa estrutura, buscou-se detalhar a sintaxe dos elementos que constituem um diagrama de estados da UML 2.0, o comportamento que esse elemento descreve, os detalhes da SAN que refletem esse comportamento, bem como a estrutura resultante da SAN.

Buscando obter agilidade na concepção de SAN, possibilitando assim a geração de casos de teste, foi construído um protótipo que traduz automaticamente os diagramas de estados UML, produzidos pela ferramenta Rational Rose, para uma estrutura equivalente em SAN. O referido sistema utiliza como fonte arquivos XML, nos quais estão contidas as informações necessárias para a tradução. A possibilidade de construção de SAN que, devido ao elevado número de autômatos e estados, sejam inviáveis para a análise na ferramenta PEPS2003, levou à composição de um módulo para simplificação das referidas SAN, onde as características comportamentais e a interação entre funcionalidades do sistema foram mantidas.

Tendo como meta demonstrar a referida tradução, foi proposto um estudo de caso de caso, onde fosse possível utilizar os elementos constituintes de um diagrama de estados para a sua descrição comportamental. Esse sistema foi especificado desde a descrição do seu domínio, passando pela especificação dos casos de uso, bem como pelos diagramas de estados que descrevessem o seu comportamento, culminando na construção de SAN que representasse o modelo de uso do sistema. Para esse exemplo, foi exposta a possibilidade da composição de modelos SAN que descrevam o comportamento do sistema como um todo (análise de sistema), onde a interação dos atores incorpora informações sobre a utilização do sistema. Outra análise evidenciada é aquela onde cada funcionalidade do sistema, disponibilizada diretamente para um ator, possibilita a construção de uma SAN específica (análise de componente).

7.3. Limitações

Sobre as limitações do presente estudo, verificou-se a ausência de determinados elementos dos diagramas de estados por parte da ferramenta Rational Rose, sendo que esses são especificados na UML 2.0. Dessa forma, tradução desses elementos não foi incluída no protótipo, apesar dessa transcrição ser formalizada no *framework* descrito nesse trabalho. Outra limitação refere-se aos tipos de casos de teste viabilizados, tendo como fonte de informações os diagramas de estados. Logo, a eficácia dos teste gerados depende diretamente do nível de granularidade usado na especificação dos diagramas. Ainda assim, não é assegurada a elaboração casos de testes que cubram todos os trechos do código, o que sugere a utilização de outros diagramas (ou outros artefatos) que forneçam tais informações.

7.4. Trabalhos futuros

No que se refere a possíveis trabalhos futuros, sugere-se a inclusão de um módulo que disponibilize automaticamente eventuais informações não contempladas na SAN, sendo essas necessárias para a construção dos casos de teste na ferramenta STAGE-Model, que atualmente são inseridas manualmente. Sugere-se também a inclusão de mecanismos para o tratamento de eventuais condições de guarda existentes em transições dos diagramas de estados. Isso seria feito através da inclusão de autômatos na SAN que modelem as possíveis restrições impostas pelas condições de guarda, sendo que uma transição só seja habilitada mediante a avaliação do autômato que modela essas restrições (utilização de taxas funcionais). Por fim, sugere-se ainda um estudo sobre a agregação de outros dia-

gramas da UML que possibilitem a construção de casos de teste mais refinados, incrementando assim a cobertura dos mesmos.

REFERÊNCIAS BIBLIOGRÁFICAS

[AFJ+99] C. Alegretti, P. Fernandes, R. Jungblut-Hessel. "Avaliação de Desempenho de Sistemas Através de Redes de Autômatos Estocásticos". In: Escola Regional de Informática (ERI), 1999, pp. 159–184.

[Agr94] H. Agrawal. "Dominators, Super Blocks, and Program Coverage". In: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1994, pp. 25–34.

[Agr99] H. Agrawal. "Efficient Coverage Testing Using Global Dominator Graphs". In: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), 1999, pp. 11–20.

[AL93] A. Avritzer, B. Larson. "Load Testing Software Using deterministic State Testing". ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 1993, pp. 82–88.

[AO00] A. Abdurazik, J. Offutt. "Using UML Collaboration Diagrams for Static Checking and Test Generation". In: UML 2000 – The Unified Modeling Language. Advancing the Standard. Third International Conference, 2000, pp. 383–395.

[AW94] A. Avritzer, E. Weyuker. "Generating Test Suites for Software Load Testing". In: ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 1994, pp. 44–57.

[BBF+03] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, W. Stewart. "The PEPS Software Tool". In: Computer Performance Evaluation / TOOLS 2003, 2003 , pp. 98–115.

[Bei90] B. Beizer. "Software Testing Techniques". Van Nostrand Reinhold, 1990, 503p.

[Ber05] C. Bertolini. "Análise de Casos de Teste Estatisticamente Relevantes Através da Descrição Formal de Programas", Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação - PUCRS, 2005, 88p.

[BFF+04] C. Bertolini, A. Farina, P. Fernandes, F. Oliveira. "Test Case Generation Using Stochastic Automata Networks: Quantitative Analysis". In: Software Engineering and Formal Methods (SEFM), 2004, pp. 251–260.

[BFS04] L. Brenner, P. Fernandes, A. Sales. "Avaliação de Desempenho de Sistemas Paralelos". In: Escola Regional de Alto Desempenho (ERAD), 2004, 24p.

[BFT06] B. Baudry, F. Fleurey, Y. Traon. "Improving Test Suites for Efficient Fault Localization". In: 28th International Conference on Software Engineering (ICSE), 2006, pp. 82–91.

[BPL06] D. Buchs, L. Pedro, L. Lúcio. "Formal Test Generation from UML Models", Technical Report, Genève, Switzerland, 2006, 626p.

[BRJ00] G. Booch, J. Rumbauch, I. Jacobson. "UML, Guia do Usuário". Addison-Wesley, 2000, 500p.

[Coc06] A. Cockburn. "Structuring Use Cases with Goals". Capturado em: http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm, Maio 2006.

[Cor06] S. Cornett. "Software Test Coverage Analysis". Capturado em: http://www.bullseye.com/coverage.html, Setembro 2006.

[Far02] A. Farina. “Uso de Redes de Autômatos Estocásticos na Representação de Modelos de Uso para Teste de Software”, Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação - PUCRS, 2002, 63p.

[FFO02] A. Farina, P. Fernandes, F. Oliveira. “Representing Software Usage Models with Stochastic Automata Networks”. In: 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), 2002, pp. 401–407.

[FL00] P. Fröhlich, J. Link. “Automated Test Case Generation from Dynamic Models”. In: 14th European Conference on Object-Oriented Programming (ECOOP), 2000, pp. 472–492.

[GR73] E. Girard, J-C. Rault. “A Programming Technique for Software Reliability”. In: IEEE Symposium on Computer Software Reliability, 1973, pp. 45–50.

[Gro03] L-G. Gross. “Testing and the UML - A Perfect Fit”, Technical Report, Fraunhofer IESE, Kaiserslautern, Germany, 2003, 110p.

[GTJ04] C. Gaffney, C. Trefftz, P. Jorgensen. “Tools for Coverage Testing: Necessary but not Sufficient”, *Journal of Computing Sciences in Colleges*, Vol. 20-1, Out 2004, pp. 27–33.

[Hur06] R. Hurlbut. “The Three R’s of Use Case Formalisms: Realization, Refinement, and Reification”. Capturado em: http://www.iit.edu/ rhurlbut/xpt-tr-97-06.html, Junho 2006.

[JBR99] I. Jacobson, G. Booch, J. Rumbauch. “The Unified Software Development Process”. Addison-Wesley, 1999, 463p.

[MM63] J. Miller, C. Maloney. “Systematic Mistake Analysis of Digital Computer Programs”, *Communications of the ACM*, Vol. 6-2, Fev 1963, pp. 58–63.

[Mye04] G. Myers. “The Art of Software Testing”. John Wiley & Sons, Inc., 2004, 256p.

[Neu05] F. Neuwald. “Técnica para Obtenção de Redes de Autômatos Estocásticos Baseada em Especificações de Software em UML”, Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação - PUCRS, 2005, 91p.

[Nta88] S. Ntafos. “A Comparison of Some Structural Testing Strategies”, *IEEE Transactions on Software Engineering*, Vol. 14-6, Jun 1988, pp. 868–874.

[OMG06] OMG. “Unified Modeling Language: Superstructure”. Capturado em: http://www.uml.org, Maio 2006.

[Pla84] B. Plateau. “De l’Evaluation du Parallélisme et de la Synchronisation”, Tese de Doutorado, Paris-Sud, Orsay, 1984, 188p.

[Pre06] R. Presoto. “Otimizações para Multiplicação Vetor-Descritor Através do Algoritmo Slice”, Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação - PUCRS, 2006, 115p.

[Rop94] M. Roper. “Software Testing”. McGraw-Hill Book Company, 1994, 149p.

[RPG03] M. Riebisch, I. Philippow, M. Götze. “UML-Based Statistical Test Case Generation”. In: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World (NODe), 2003, pp. 394–411.

[Sex88] B. Sexton. “Statistical Testing of Software”, Dissertação de Mestrado, University of Tennessee, Knoxville, 1988, 112p.

[SK99] M. Schader, A. Korthaus. “The Unified Modeling Language: Technical Aspects and Applications”. Springer-Verlag New York, Inc., 1999, 282p.

[TH02] M. Tikir, J. Hollingsworth. “Efficient Instrumentation for Code Coverage Testing”. In: ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2002, pp. 86–96.

[Whi97] J. Whittaker. “Stochastic Software Testing”, *Annals of Software Engineering*, Vol. 4, Jan 1997, pp. 115–131.

[WP93] J. Whittaker, J. Poore. “Markov Analysis of Software Specifications”, *ACM Transaction on Software Engineering and Methodology*, Vol. 2-1, Jan 1993, pp. 93–106.

[WP00] G. Walton, J. Poore. “Generating Transition Probabilities to Support Model-Based Software Testing”, *Software - Practice and Experience*, Vol. 30-10, Ago 2000, pp. 1095–1106.

[WPT95] G. Walton, J. Poore, C. Trammell. “Statistical Testing of Software Based on a Usage Model”, *Software - Practice and Experience*, Vol. 25-1, Jan 1995, pp. 97–108.

[WT94] J. Whittaker, M. Thomason. “A Markov Chain Model for Statistical Software Testing”, *IEEE Transactions on Software Engineering*, Vol. 20-10, Out 1994, pp. 812–824.

[YLW06] Q. Yang, J. Li, D. Weiss. “A Survey of Coverage Based Testing Tools”. In: International Workshop on Automation of Software Test (AST), 2006, pp. 99–103.

[Zha00] Y. Zhang. “Random Testing with Log File Analysis”, Dissertação de Mestrado, University of Western Ontario, London, 2000, 110p.