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

MÉTODO DE CONVERSÃO DE DIAGRAMA DE ATIVIDADES UML PARA SAN E GERAÇÃO DE CASOS DE TESTE DE SOFTWARE

Toni Amorim de Oliveira

Orientador: Dr. Paulo Henrique Lemelle Fernandes

Porto Alegre, Brasil
2010

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

MÉTODO DE CONVERSÃO DE DIAGRAMA DE ATIVIDADES UML PARA SAN E GERAÇÃO DE CASOS DE TESTE DE SOFTWARE

Toni Amorim de Oliveira

Dissertação de Mestrado apresentada como requisito para obtenção do título de Mestre em Ciência da Computação pelo Programa de Pós-graduação da Faculdade de Informática. Área de concentração: Ciência da Computação.

Orientador: Dr. Paulo Henrique Lemelle Fernandes

Porto Alegre, Brasil
2010

Dados Internacionais de Catalogação na Publicação (CIP)

O48m Oliveira, Toni Amorim de

Método de conversão de diagrama de atividades UML para SAN e geração de casos de teste de software / Toni Amorim de Oliveira. – Porto Alegre, 2009.

105 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. 5. UML (Informática). I. Fernandes, Paulo Henrique Lemelle. II. Título.

CDD 005.1

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 "Método de Conversão de Diagrama de Atividades UML para SAN e Geração de Casos de Teste de Software", apresentada por Toni Amorim de Oliviera, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído , aprovada em 27/01/10 pela Comissão Examinadora:

Prof. Dr. Paulo Henrique Lemelle Fernandes PPGCC/PUCRS

Orientador

Prof. Dr. Duncan Dubugras Alcoba Ruiz PPGCC/PUCRS

Dr. Afonso Henrique Corrêa de Sales Bolsista PNPD FACIN/PUCRS

Homologada em 01/06/2010, conforme Ata No. 09/2010 pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

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

a Raquel, minha esposa
a Verônica, minha filha
ao Gabriel, meu filho
a Marta, minha mãe

Agradecimentos

Agradeço primeiramente a Deus por ter me dado forças para concluir esse trabalho, pois sem ele sei que nada seria possível.

Gostaria de agradecer aos meus colegas do Minter pela ajuda em todos os momentos em que precisei. Não tinha a intenção de citar nomes, pois todos tiveram sua parcela de contribuição durante esta longa caminhada, mas é impossível não agradecer a alguns deles, dentre eles o Tales, o Benevid, o Ivan e o Maicon, com quem compartilhei a maior parte das longas viagens entre Colider e Barra do Bugres e que foram companheiros na maior parte dos trabalhos realizados durante o mestrado.

Agradeço aqui também ao Everton, ao Shimazu e ao Alberto, pois compartilhamos várias discussões a respeito de SAN. E em relação a SAN agradeço também aos colegas do PEG (Performance Evolution Group), em especial a Thais, ao Czekster e ao Dione por toda a ajuda.

Quanto a Thais e ao Czekster, posso afirmar com certeza que a concretização deste trabalho se deve em grande parte a ajuda de vocês dois. Só me resta pedir que Deus vos pague por toda ajuda e dedicação que empregaram neste trabalho.

Gostaria de agradecer ao meu orientador Paulo Fernandes pela orientação e ajuda, pois eu sei o quanto foi difícil concluir este trabalho.

Agradeço ainda a minha esposa e minha filha pela compreensão e paciência que sempre tiveram durante esses anos, sabendo entender as ausências. Agradeço ainda a minha mãe, aos meus irmãos e a minha sobrinha pelo apoio nesses anos todos.

Gostaria de agradecer ao meu amigo e companheiro de trabalho na Unemat, o prof. Max Robert Marinho pela contribuição dada a este trabalho.

Encerro agradecendo a todos que contribuíram para a conclusão deste trabalho, seja com uma idéia, um incentivo ou um momento de descontração e que em muitas vezes me deram forças para continuar.

MÉTODO DE CONVERSÃO DE DIAGRAMA DE ATIVIDADES UML PARA SAN E GERAÇÃO DE CASOS DE TESTE DE SOFTWARE

RESUMO

O processo de desenvolvimento de software é uma tarefa que envolve um conjunto de atividades a serem realizadas, e em muitos casos, por equipes grandes que podem se encontrar geograficamente dispersas. Isso exige do desenvolvedor a utilização de métodos que proporcionem uma visão de todas as etapas desse processo de desenvolvimento. A UML (*Unified Modeling Language*) é uma linguagem de modelagem que possibilita essa visão através do uso de diagramas que demonstram graficamente a estrutura do software em desenvolvimento. O diagrama de atividades é utilizado para modelar o comportamento do sistema, através dos fluxos de execução de cada atividade desempenhada pelo mesmo. Com o objetivo de obter um modelo comportamental de sistemas computacionais apresentamos neste trabalho uma proposta de conversão de diagramas de atividades para SAN (*Stochastic Automata Networks*), um formalismo matemático que possibilita a modelagem de sistemas em geral, a partir do qual é possível extrair índices probabilísticos relacionados aos estados, permitindo aos responsáveis pelo processo de desenvolvimento tomar decisões sobre os recursos alocados no projeto. Com o intuito de demonstrarmos como executar a conversão, usamos uma versão simplificada dos elementos do diagrama de atividades da UML, para descrever os parâmetros e regras utilizadas para a conversão proposta. Apresentamos ainda os resultados obtidos a partir do modelo SAN gerado.

Palavras-chave: Redes de Autômatos Estocásticos, Teste de Software, UML (*Unified Modeling Language*)

METHOD FOR CONVERSION OF UML ACTIVITY DIAGRAM FOR SAN AND GENERATION OF CASES OF SOFTWARE TESTING

ABSTRACT

The process of software development is a task that involves a set of activities to be performed and, in many cases, by large and geographically dispersed teams. This requires the utilization of methods that renders a broad vision of all stages of the development process. The UML (Unified Modeling Language) is a modeling language that enables this vision through the use of diagrams that graphically demonstrates the software structure being developed. The activity diagram is used to model the behavior of the system with executing flows for every defined activity In order to obtain a behavioral model of computational systems, this work presents a proposal to converter activity diagrams to SAN (Stochastic Automata Networks), a mathematical formalism for modeling probabilistic index extraction related to the states, allowing decisions making for project resources allocated. To demonstrate how to execute the conversion , we use a simplified version of the elements from the UML activity diagrams to describe both parameters and rules associated to de conversion. We also present the results generated from the SAN model.
Keywords: Stochastic Automata Networks, Software Testing, UML (*Unified Modeling Language*)

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

LISTA DE SIGLAS

UML	Unified Modeling Language
SPN	Stochastic Petri Nets
OOSE	Object Oriented Software Engineer
OMT	Object Modeling Technical
OMG	Object Manager Group
CVDS	Ciclo de Vida de Desenvolvimento de Software
FSM	Finite State Machine
PEPS	Performance Evaluation of Parallel Systems
GSPN	Generalized Stochastic Petri Nets
GREATSPN	GRaphical Editor and Analyzer for Timed and Stochastic Petri Net
MOF/QVT	Meta Object Facility/Query View Transformation
LGSPN	Labeled Generalized Stochastic Petri Nets
IPN	Inteligência de Processos de Negócios
PPGCC	Programa de Pós-Graduação em Ciência da Computação
STAGE	State Based Test Generator
CTPS	Centro de Pesquisa e Teste de Software

Sumário

1. Introdução

O processo de desenvolvimento de software é uma tarefa que envolve um conjunto de atividades a serem realizadas, e que em muitos casos, podem envolver equipes grandes. Esse processo exige muitas vezes do desenvolvedor, a utilização de métodos que proporcionem uma visão de todas as etapas do ciclo de vida de desenvolvimento do sistema.

A UML (*Unified Modeling Language*) é uma linguagem de modelagem, desenvolvida por Rumbaugh, Booch e Jacobson [BRJ05], que possibilita essa visão através do uso de diagramas que demonstram graficamente a estrutura do software em desenvolvimento. Dentre os diagramas UML, o diagrama de atividades é um dos que permite modelar o comportamento do sistema, através dos fluxos de execução de cada atividade desempenhada pelo mesmo.

Segundo Bernardi et al [BDM02], por falta de uma semântica formal não é possível aplicar, diretamente, técnicas matemáticas de analíse em modelos UML com o objetivo de validar o sistema em desenvolvimento. Assim, para alcançar este objetivo, algumas pesquisas, dentre elas [BDM02, LGMC04, Bar06, LSP+08, Neu08], propõem o uso de formalismos como redes de Petri estocásticas (SPN - *Stochastic Petri Nets*)

Os modelos construídos para essa análise, em pesquisas como as de Lopéz-Grao et al em [LGMC04], Bernardi et al em [BDM02], Lachtermacher et al em [LSP+08] e Neuwald em [Neu08], Barros em [Bar06] baseiam-se em diagramas UML que descrevem o comportamento do sistema e proporcionam, no caso da pesquisa feita por Barros, a geração de casos de teste de software.

Um modelo matemático é uma representação ou interpretação simplificada da realidade, ou uma interpretação de um fragmento de um sistema, segundo uma estrutura de conceitos mentais ou experimentais, que apresenta apenas uma visão ou cenário de um fragmento do todo [Rio86].

Os modelos matemáticos baseados em processos Markovianos são aplicados em diversas áreas, tais como biologia, física, ciências sociais, processos de modelagem de negócios e engenharias. Um modelo matemático, baseado em um processo Markoviano, é modelado como um conjunto de estados onde o sistema só pode ocupar um e somente um desses estados em um dado espaço de tempo [Ste94].

Com base em modelos matemáticos gerados, é possível executar testes de software. O teste de software é definido por Pezzé e Youg em [PeY08] como atividade que tem por objetivo ou avaliar a qualidade do software ou possibilitar melhorias no software revelando defeitos.

Na tentativa de obter um modelo comportamental de sistemas computacionais apresentamos neste trabalho um método de conversão de diagramas de atividades para SAN, um formalismo matemático que possibilita a modelagem de sistemas em geral.

A partir dos modelos criados, é possível extrair índices probabilísticos relacionados aos estados do modelo SAN gerado, permitindo a geração de cenários de testes de software que possibilitem uma maior qualidade ao produto desenvolvido.

1.1 Objetivos do trabalho

É fato conhecido que o custo para a correção de erros nas especificações de requisitos no desenvolvimento de sistemas é muito mais baixo quando encontrado nas fases iniciais do projeto, do que quando encontrados na fase de execução. Assim, a validação de especificações de requisitos em fases iniciais de desenvolvimento é muito importante.

O uso de notações como UML, através do uso de diagramas, proporciona uma visão comportamental do sistema em desenvolvimento, porém, apenas o uso da UML não permite uma análise matemática do sistema em desenvolvimento. Assim, visando associar a UML ao formalismo de modelagem SAN, de modo que tal ação seja possível, destacamos os objetivos específicos deste trabalho:

- desenvolver um método para transformar diagramas de atividades em redes de autômatos estocásticos (SAN), a partir do mapeamento dos elementos que compõem o diagrama de atividades, para uma estrutura equivalente em SAN que represente o modelo comportamental do sistema;
- definir as regras para a conversão de diagramas de atividades da UML para SAN;
- gerar cenário de teste de software e análise dos índices de cobertura dos testes gerados com base no método proposto;
- executar a aplicação do método proposto, através da elaboração de um estudo de caso, que tem por finalidade demonstrar a utilização do método em processos de desenvolvimento de sistemas.

1.2 Escopo e contribuição do trabalho

Essa dissertação tem o foco principal na conversão de diagramas de atividades da UML para SAN e geração de cenários de teste de software. Para isso, definiremos as regras de conversão a serem utilizadas para a conversão dos diagramas de atividades e posterior geração dos modelos. A versão UML utilizada nesta pesquisa será a versão 1.5, homologada em 2002, que consideramos ser suficientemente capaz de gerar os resultados esperados para este trabalho.

Na pesquisa desenvolvida por Barros [Bar06], foram obtidos resultados, dentre eles, a construção de um modelo de uso do sistema analisado e a definição de índices de cobertura para a geração de casos de teste de software, associando o formalismo SAN a UML, porém baseados em outros diagramas. Assim esta dissertação baseia-se em diagramas de atividades para a obtenção dos modelos SAN e posterior geração dos cenários de teste de software.

Esta dissertação fornece contribuições para a área de Engenharia de Software e avaliação de desempenho contribuindo para a melhoria do processo de desenvolvimento de software. Portanto, entre os benefícios da nossa proposta, podemos citar:

- obtenção de índices de desempenho, tais como probabilidade e execução de atividades descritas no diagrama, a partir dos modelos gerados pela conversão;

- possibilidade de identificação de atividades que podem demandar mais tempo para a execução durante o desenvolvimento.

1.3 Metodologia utilizada

A revisão bibliográfica foi a primeira etapa na elaboração deste trabalho, onde foram feitos levantamentos acerca dos assuntos pertinentes a esta pesquisa. Inicialmente foram revisados os conceitos sobre UML (*Unified Modeling Language*), dando-se ênfase ao diagrama de atividades. Ainda nessa primeira etapa foram pesquisados os principais conceitos sobre SAN (*Stochastic Automata Networks*) e teste de software.

Na segunda etapa, foram realizados estudos sobre o processo de conversão de diagramas UML, dentre eles diagramas de atividades para redes de Petri e SAN, que serviram de base para a formulação do método proposto nesta dissertação, buscando identificar padrões de mapeamento dos elementos UML durante a conversão, quando da existência desses, que pudessem ser adotados também neste trabalho.

Os padrões buscados nessa etapa podem ser exemplificados como tratamento de condições de guarda. Uma condição é uma expressão booleana que precisa ser satisfeita para habilitar uma transição, seja em um diagrama de estados ou em um diagrama de atividades.

Na terceira etapa, foram gerados os primeiros modelos de redes de autômatos estocásticos gerados a partir de diagramas de atividades, com o objetivo de modelar o comportamento do sistema em estudo, tais modelagens, inicialmente, levaram em consideração apenas algumas características do diagrama de atividades, uma vez que, os primeiros modelos gerados não levaram em consideração a existência de condições de guarda no diagrama modelado.

Finalmente, para a conclusão deste trabalho, foi realizada a aplicação do método aqui proposto, em dois exemplos, onde os diagramas elaborados foram convertidos em um modelo SAN que serviu de base para a geração dos casos de teste.

1.4 Estrutura do trabalho

Após a introdução definida como sendo o Capítulo 1, este trabalho está estruturado em cinco capítulos e uma conclusão, a qual sumariza as contribuições e faz propostas de trabalhos futuros referentes ao tema.

O Capítulo 2 trata de temas como UML e teste de software, apresentando os principais conceitos sobre os dois temas, descrevendo sua evolução e estrutura da UML, porém, dando ênfase aos diagramas de atividades, um dos focos principais desta pesquisa. São descritos ainda no segundo capítulo, os conceitos fundamentais sobre teste de software, tanto estruturais quanto funcionais, bem como as principais técnicas usadas em cada tipo de teste.

O Capítulo 3 apreesenta os principais conceitos a respeito do formalismo de rede de autômatos estocásticos e apresenta ainda os trabalhos, baseados em formalismos Markovianos, os quais servirão de base para a eleaboração do método proposto neste trabalho.

O Capítulo 4 descreve em detalhes o processo de conversão dos elementos do diagrama de atividades em um modelo SAN, apresentando os passos a serem executados para a obtenção do mesmo.

O Capítulo 5 apresenta dois exemplos de conversão de um digrama de atividades para uma SAN equivalente, bem como os casos de teste gerados e a análise dos resultados obtidos. O referido capítulo traz ainda o algoritmo utilizado para a geração dos casos de teste de software com base na SAN obtida no processo de conversão.

Para finalizar, o Capítulo 6 apresenta as considerações finais e os trabalhos futuros propostos com base nos resultados obtidos com este trabalho.

2. UML e teste de software

A Engenharia de Software é uma disciplina da engenharia relacionada com todos os aspectos da produção de software, desde os estágios iniciais de especificação do sistema até sua manutenção depois do mesmo entrar em operação, abordando tanto aspectos técnicos quanto de gerenciamento de processo [MNF01].

Do ponto de vista formal, pode-se dizer que a Engenharia de Software busca conciliar custo baixo e qualidade no processo de desenvolvimento de software, associando as melhores práticas para a obtenção de um software confiável, seguro e eficiente.

Em muitos projetos, o desenvolvimento de software envolve tanto a modelagem quanto o uso de linguagens orientadas a objeto, buscando empregar sempre os elementos da Engenharia de Software, dentre eles a UML (*Unified Modeling Language*) [Som07].

Um processo de software define o conjunto de atividades que serão conduzidas no contexto do projeto, os recursos necessários (software, hardware e pessoas), os artefatos (insumos e produtos) e os procedimentos a serem adotados na realização de cada uma das atividades.

Atualmente, desenvolver software de qualidade, com elevada produtividade, dentro do prazo estabelecido e sem necessitar de mais recursos do que os alocados, têm sido o grande desafio da Engenharia de Software, seja pela melhoria de processos ou pelo desenvolvimento de ferramentas de gerenciamento dos mesmos.

Nesse sentido a UML foi desenvolvida com o objetivo de descrever qualquer tipo de sistema, em termos de diagramas orientados a objetos e de padronizar a forma de especificação de sistemas, desde as fases iniciais até a fase de testes e manutenção.

A padronização do processo de desenvolvimento, proporcionada pela Engenharia de Software seja com técnicas ou através de seus paradigmas de desenvolvimento e programação, abrange todas as fases do desenvolvimento. A fase de testes, dentro de um processo de desenvolvilmento, é considerada uma das fases que demandam mais tempo e consomem mais recursos do projeto.

Nas próximas seções será detalhada a composição da UML, um dos focos principais desta dissertação, dando ênfase ao diagrama de atividades demonstrando um exemplo de aplicação da UML com base no uso desse diagrama, demonstrando-se um exemplo de aplicação do diagrama de atividades.

Aborda-se ainda, dentre os assuntos descritos nesse capítulo, a definição e classificação do teste de software.

2.1 UML - *Unified Modeling Language*

As linguagens de modelagem orientadas a objeto sugiram entre meados dos anos 70 e final dos anos 80 como metodologia aplicada à análise e ao projeto de software. O número de métodos orientados a objeto aumentou de menos de 10 para mais de 50 durante o período entre 1989 e 1994. Porém poucos métodos se destacaram, como foi o caso dos métodos Booch proposto por

Booch [BRJ99], OOSE (*Object Oriented Software Engineer*) [BRJ99] proposto por Jacobson e OMT (*Object Modeling a Tecnical*) proposto por Rumbaugh [BRJ05].

Em outubro de 1994 a UML foi oficialmente homologada pela OMG (*Object Manager Group*), após Rumbaugh se juntar a Booch na empresa *Rational*, com o objetivo de unificar os métodos Booch e o OMT, dando origem, em outubro 1995, a versão 0.8 da UML que representava o método unificado.

Pouco tempo depois, Jacobson passou a integrar o projeto, que incorprou a metodologia OOSE (*Object Oriented Software Engineer*). Essa união deu origem a versão 0.9 da UML, lançada em junho 1996.

Após o lançamento da versão 0.9, um consórcio de empresas como a *Digital Equipament Corporation*, a *Hewlett-Packard*, a *iLogix*, a *Intell corp*, a *IBM*, a *MCI Systemhouse*, a *Microsoft*, a *Oracle*, a *Rational*, a *Texas Instruments* e a *Unisys* passou a contribuir para uma definição forte e completa da UML. Esta colaboração conduziu à UML 1.0, uma linguagem de modelagem mais expressiva, poderosa, e aplicável a um maior número de problemas decorrentes do ciclo de vida de desenvolvimento de software (CVDS).

A versão 1.0 foi oferecida à OMG, para a padronização, em janeiro 1997, em resposta ao seu pedido de elaboração de uma linguagem de modelagem padrão. Uma versão revisada da UML, a 1.1, foi oferecida à OMG para a padronização em julho 1997 e aprovada em setembro 1997, sendo então adotada pela OMG em 14 de novembro do mesmo ano. Em junho de 1998 foi liberada a versão 1.2 e em dezembro do mesmo ano OMG homologou a versão 1.3 da UML [Pen04].

Em 2001 a OMG homologou a versão 1.4 da UML e em 2002 a versão 1.5, mas foi a versão 2.0, lançada em 2003, que trouxe mais alterações à UML.

Em sua versão 2.0, a UML possui 13 diagramas classificados em estruturais (diagrama de classes, diagrama de objetos, diagrama de componentes e diagrama de implantação) e comportamentais (diagrama de casos de uso, diagrama de sequência, diagrama de atividades, diagrama de colaboração e diagrama de estados), sendo o diagrama de atividades, foco desta pesquisa, empregado para modelar aspectos dinâmicos do sistema através de etapas sequênciais e, em alguns casos, concorrentes em um processo computacional.

A UML tem por objetivos principais [FoS08, BRJ05, Pen04]:

- oferecer aos modeladores uma linguagem expressiva e visual para o desenvolvimento de modelos significativos;
- fornecer mecanismos de extensibilidade e especialização de conceitos de base;
- descrever modelos de sistema - do mundo real e de software - baseado em conceitos de orientação a objetos;
- fornecer uma linguagem de modelagem orientada a objeto, visual, fácil e pronta para uso, permitindo amplas facilidades de modelagem;
- admitir conceitos de desenvolvimento de componentes, framework e padrões;
- integrar melhores práticas em desenvolvimento de sistemas orientados a objeto.

2.1.1 Diagramas

A UML utiliza-se de diagramas para representar o sistema, ou de parte dele, em desenvolvimento modelando seu comportamento em diferentes fases. A UML 2.0 possui 13 diagramas, classificados em estruturais e comportamentais, que podem ser descritos da seguinte forma [Pen04]:

1. **Estruturais:**

Diagrama de Classes: define a estrutura das classes utilizadas pelo sistema determinando seus atributos e métodos;

Diagrama de Objetos: fornece uma visão dos valores armazenados pelos objetos de um diagrama de classes em um determinado momento da execução de um processo;

Diagrama de Componentes: representa os componentes do sistema, tais como bibliotecas, códigos fonte, módulos executáveis, os quais serão implementados. Pode ser utilizado para modelar interfaces;

Diagrama de Implantação: define as necessidades de hardware do ambiente onde o sistema será executado.

2. **Comportamentais:**

Diagrama de Estrutura Composta: modela as colaborações entre os componentes do sistema. Uma colaboração descreve um conjunto de entidades, as quais interagem para a execução de uma função específica;

Diagrama de Sequência: modela a ordem temporal em que as mensagens são trocadas entre os objetos envolvidos em um determinado processo;

Diagrama de Comunicação: modela os vínculos existentes entre os objetos e quais as mensagens trocadas entre eles, independente da ordem temporal em que elas acontecem;

Diagrama de Estados: baseia-se no diagrama de caso de uso, e representa o comportamento do sistema;

Diagrama de Atividades: descreve os passos a serem percorridos para a conclusão de uma atividade específica que altera o estado de um sistema;

Diagrama de Pacotes: descreve os subsistemas englobados por um sistema de forma a determinar as partes que o compõem.

Diagrama de Interação Geral: descreve uma visão ampla do sistema ou processo de negócios e pode englobar diversos diagramas de interação a fim de atingir os seus objetivos;

Diagrama de Pacotes: tem o objetivo de representar os subsistemas que compõem um sistema, determinando assim as partes que o compõem;

Diagrama de Tempo: descreve a mudança no estado ou condição de uma instância de uma classe ou seu papel durante um tempo.

Apesar de fazermos uma descrição suscinta de todos os diagramas UML, neste trabalho utilizaremos apenas o diagrama de atividades, o qual descreveremos com mais detalhes na próxima seção. Um estudo mais aprofundado dos demais diagramas pode ser feito em ”UML a bíblia”[Pen04].

A UML possui cinco diagramas utilizados para a modelagem de aspectos dinâmicos e comportamentais de um sistema, são eles: diagrama de atividades, diagrama de colaboração, diagrama de sequência, diagrama de estados e o diagrama de caso de uso.

Cada um dos diagramas comportamentais modela uma característica em relação ao comportamento do sistema, sendo empregado em uma ou mais etapas do processo de desenvolvimento de um sistema. O uso de diagramas comportamentais para a obtenção de modelos matemáticos analíticos se deve a sua capacidade de representar um ou mais estágios de uso ou execução do sistema em diferentes fases [BRJ05].

Essa característica comportamental desses diagramas UML os torna passíveis de serem representados por formalismos matemáticos como redes de Petri, usadas em [BDM02, LSP+08, LGMC04], cadeias de Markov usadas em [YJH04] e SAN (Redes de Autômatos Estocásticos) usadas em [Bar06, Neu08].

O diagrama de atividades é um dos cinco diagramas UML utilizados para a modelagem de aspectos dinâmicos e comportamentais de um sistema, que em alguns casos envolve atividades sequenciais e possivelmente concorrentes em processos computacionais.

Essa característica de modelagem de execuções em forma de atividades e fluxos que as relacionam, e ainda a possibilidade de representação sequencial dessas execuções, confere ao diagrama de atividades um caráter significativo, o que torna sua semântica semelhante as utilizadas em formalismos como cadeias de Markov (CM), redes de autômatos estocásticos (SAN) e redes de Petri estocásticas (SPN).

A escolha deste diagrama como foco deste trabalho deve-se as características anteriormente citadas que possibilitam a conversão dos mesmos para modelos criados com base no formalismo SAN, o que viabiliza uma análise matemática dos diagramas. Um exemplo de conversão de UML para SAN pode ser encontrado na pesquisa desenvolvida por Barros [Bar06], onde diagramas de estados da UML, que também são diagramas comportamentais foram convertidos para modelos SAN, demonstrando a viabilidade de análise matemática dos mesmos.

2.1.2 Estrutura do Diagrama de Atividades

Um diagrama de atividades representa o fluxo de uma atividade para outra, descrevendo os passos a serem percorridos para a conclusão de um método ou algoritmo específico. Os elementos que compõem a estrutura de diagrama de atividades podem ser descritos da seguinte forma [Pen04, BRJ05, FoS08]:

Estado Inicial: estabelece o início da execução de uma atividade. É representado por um círculo totalmente preenchido;

Atividade: uma atividade é uma execução de um processamento não-atômico em uma máquina de estados, de modo que cada atividade modela o processamento de uma ação. Uma atividade é representada por um retângulo de cantos arredondados;

Decisão: tem por função definir a direção do fluxo, permitindo fluxos de controle alternativos condicionados por expressões booleanas, denominadas condição de guarda, que deve ser verifi-

cada para que a transição seja ativada. Esse elemento é representado por um losango. Ainda sobre as condições de guarda, Rumbaugh, Jacobson e Booch em [**?**] definem que a condição de guarda é uma expressão booleana de valores de atributo que permitem que a transição ocorra somente se a condição assumida pela expressão é verdadeira. Uma condição de guarda é avaliada uma vez quando o evento de disparo sobre a transição ocorre. Se ela é falsa, a transição não é acionada e a condição é reavaliada. Dentro da expressão booleana, é possível incluir as condições sobre o estado de um objeto [BRJ05];

***Merge*:** é representado por um losango com até três transições de entrada e uma única transição de saída. É usado, assim como o *join*, para unir diferentes fluxos.

Um ponto de união ou de fusão funde um conjunto de fluxos (transições) em um único. O ponto de união não tem conotação de sincronização, mas apenas de reunir um conjunto de elementos sintáticos em um único fluxo. Não possui semântica associada, apenas acrescenta informação ao diagrama.

Um diagrama de atividades pode conter fluxos de controle concorrentes, isso significa que pode haver dois ou mais fluxos sendo executados simultaneamente, ou seja, de forma paralela. Para sincronizar dois ou mais fluxos, barras de sincronização são utilizadas. Essas barras de sincronização podem ser de dois tipos:

***Fork*:** é uma barra de bifurcação que estabelece os fluxos concorrentes, ou seja, recebe um fluxo de entrada e cria dois ou mais fluxos de saída;

***Join*:** é uma barra de junção que recebe dois ou mais fluxos de entrada unindo-os em um único fluxo. Esta barra tem o objetivo de sincronizar os fluxos de controle, concorrentes ou não, criados pelo elemento *fork*. Os fluxos de saída da barra de sincronização join só são disparadas quando todos os fluxos de entrada tiverem sido disparados.

O diagrama de atividades ainda possui os seguintes elementos:

Objeto: representa um objeto de uma classe que participa de um fluxo de trabalho representado pelo gráfico de atividades. O objeto pode ser a saída de uma atividade e a entrada de muitas outras atividades. O mesmo objeto pode ser manipulado por um número de atividades consecutivas que mudam o estado do objeto. O objeto pode ser exibido várias vezes em um gráfico de atividades, cada uma representando um estado diferente da sua vida e o estado do objeto a cada ponto pode ser colocado entre parênteses e anexado ao nome da classe. O objeto é representado por um retângulo;

Fluxo de Objeto: o fluxo de objeto é um tipo de fluxo de controle com entrada ou saída. O símbolo de fluxo de objeto representa a existência de um objeto em um determinado estado, não só o próprio objeto;

Estado Final: estabelece o final da execução de uma atividades, finalizando o diagrama de atividade. É representado por um circulo semi preenchido.

Swimlanes são mecanismos para organizar visualmente as atividades dentro do diagrama, demonstrando quem ou que parte do sistema é responsável pela execução de uma determinada atividade. Um *Swimlanes* pode ser representado por barras verticais.

A Tabela 2.1 mostra a representação gráfica referente aos elementos anteriormente descritos.

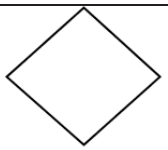
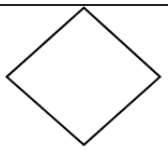
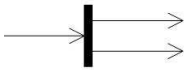
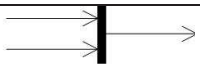
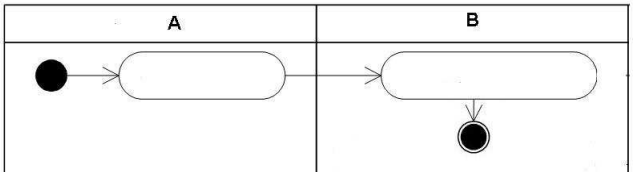
Elemento	Representação gráfica
estado inicial	
atividade	
decisão	
merge	
fork	
join	
objeto	
fluxo de objeto	
estado final	
swimlanes	

Tabela 2.1: Representação gráfica dos elementos UML

Exemplo de descrição através do diagrama de atividades

A Figura 2.1 mostra um exemplo de aplicação do diagrama de atividades. Nesse exemplo é apresentado um processo de controle de estoque.

O diagrama apresentado descreve um processo integrante de um sistema de controle de estoque que é iniciado quando o usuário efetua o *login* no sistema. O sistema valida os dados de login fornecidos pelo usuário, caso haja inconsistência em algum dos dados fornecidos pelo usuário, o sistema emite mensagem de erro e solicita novos dados para a validação. Essas ações são representadas pelas atividades *efetuar login e validar usuário e senha* mostrados no diagrama.

Após a validação, o usuário informa os dados do cliente e o sistema consulta se o cliente está ou não cadastrado, caso não esteja, é necessário efetuar o cadastramento, onde os dados necessários a realização do cadastro devem ser fornecidos. Essas ações são representadas pelas atividades *cadastrar* e *consultar* clientes, que estão relacionadas entre si pelo elemento decisão, que determina a execução de uma ou outra atividade mostrada no diagrama.

Após a confirmação da existência do cliente ou da efetivação do cadastro do mesmo no sistema, é executada a abertura do pedido, representado pelo elemento atividade com o mesmo nome, procedendo a consulta dos itens cadastrados, representado também pelo elemento com mesmo nome, caso o produto não esteja cadastrado ou não disponível em estoque, o sistema solicita ao cliente que informe um novo item a ser consultado. Essas ações são representadas pelo elemento decisão e os fluxos a ele relacionado.

Quando o item é localizado ele é adicionado ao pedido do cliente que informa se deseja adicionar mais produtos ou fechar o pedido e posteriormente emitir a nota fiscal. Tais elementos são representados pelos elementos atividade nomeados com o mesmo nome e relacionados também por um elemento decisão.

O próximo passo executado pelo sistema é confirmar pedido e separar produtos, tais ações são executadas separadamente, e são representadas por elementos atividades e por uma barra de sincronização *fork*. Posteriormente o sistema procede a baixa nas mercadorias selecionadas no pedido.

O sistema permite o cancelamento da nota fiscal e na sequência do pedido. Caso o pedido seja confirmado, o sistema encerra a compra. Tais ações são representadas por elementos atividade, elemento decisão e um elemento de junção *join*. O elemento estado final representa o encerramento das atividades demonstradas no diagrama.

2.2 Teste de software

Os sistemas de informação passaram a fazer parte do ambiente das empresas, seja por meio de convergência, canais multimídia, múltiplos fatores interligados e negócios cada vez mais dependentes de softwares e computadores.

Isso ocasiona uma crescente demanda por produtos de software com qualidade, o que faz com que as empresas invistam em profissionais, ferramentas e técnicas que proporcionem a melhoria do processo de desenvolvimento de produtos de software. Em muitos casos, a obtenção de melhorias advem da adoção de técnicas de teste de software.

2.2.1 Definição

Durante o processo de desenvolvimento de software há diversas atividades que tem por objetivo garantir a qualidade do produto a ser entregue ao cliente. Porém problemas ainda podem aparecer, assim sendo, os testes são executados para garantir a identificação e resolução desses problemas.

Segundo Bartié [Bar02], os custos decorrentes da correção de um problema detectado nas fases inicias do desenvolvimento são consideravelmente inferiores aos custos do mesmo problema,

quando detectado após a entrega do produto ao cliente. Nessa fase, os custos podem ser bem mais que financeiros, pois podem atingir também a imagem e a credibilidade da empresa perante o cliente.

Os testes podem ser considerados como o processo de executar ações visando encontrar e revelar a presença de erros no sistema, ou seja, consiste na verificação dinâmica do funcionamento de um determinado programa, baseado em um conjunto finito de casos de testes, cuidadosamente relacionados dentro de um domínio infinito de entradas contra seu funcionamento esperado [PeY08].

Os testes são amplamente utilizados e bem aceitos para a avaliação e aceitação de um sistema de software e podem ser considerados como uma forma de se fazer uma revisão completa do sistema, avaliando e apontando os erros, desde o projeto até a implementação e podem ser classificados em teste estrutural ou teste funcional [Bar02, BRC+07, PeY08].

Teste estrutural (*White Box*)

Segundo Bartié [Bar02], o teste estrutural, também chamado de teste de caixa-branca, consiste em examinar a estrutura interna do programa testando a lógica do mesmo através da análise do código fonte e da elaboração de teste, cobrindo todos os caminhos do programa.

Esses testes devem garantir que todas as linhas de códigos e condições estejam corretas. Os testes devem exercitar:

- todos os caminhos independentes dentro de um módulo, ao menos uma vez;
- as decisões lógicas para verdadeiro e falso;
- todos os laços em suas fronteiras e dentro de seus limites;
- as estruturas de dados internas para garantir a sua validade.

Os critérios pertencentes a estas técnicas são classificados:

- com base no fluxo de controle que utilizam características do controle de que são necessárias;
- com base no fluxo de dados que associam ao grafo de fluxo de dados de controle, informações sobre o fluxo de dados do programa;
- com base na complexidade que utiliza informações sobre a complexidade do programa para determinar os requisitos de teste.

Estes testes são divididos em diferentes categorias que por sua vez possibilitam a detecção de falhas no sistema sob diferentes perspectivas, sendo que devido à baixa criatividade de certos sistemas, algumas categorias podem ser planejadas em conjunto.

Teste funcional (*Black Box*)

O teste funcional testa o software com uma função que mapeia um conjunto de valores de entrada em um conjunto de valores de saída, sem levar em conta a forma como esse mapeamento foi implementado.

O teste funcional tem por objetivo garantir que os requisitos do sistema foram plenamente atendidos pelo algoritmo que compõem a estrutura do software. O teste funcional preocupa-se em identificar quais funcionalidades estão sendo executadas, e não como elas são executadas. Este tipo de teste baseia-se exclusivamente na especificação de requisitos para determinar que tipo de saídas são esperadas para um certo conjunto de entradas. Neste tipo de teste são definidos os critérios de teste como:

Particionamento de equivalência: essa técnica de teste, também conhecida como classe de equivalência é uma técnica introduzida por Myers em 1979 [Mye79] para reduzir o número de casos de teste a um nível controlável, mas mantendo uma cobertura razoável de teste. Basicamente a técnica considera que uma vez que não se pode testar todas as possibilidades de execução de um sistema, é possível dividí-lo em classes, de modo que casos de teste dentro de cada classe sejam equivalentes;

Análise do valor limite: esta técnica valida os valores limites do domínio de entrada de um determinado sistema, os casos de teste selecionados são os valores das fronteiras. Essa técnica geralmente é usada para complementar a técnica de particionamento de equivalência, pois permite testar valores nos limites de cada classe.

2.2.2 Níveis de teste

Segundo Pezzé e Young [PeY08] e Bastos et al [BRC+07], os testes são aplicados de acordo com o ciclo de desenvolvimento do sistema e são classificados da seguinte forma:

Teste de unidade: também chamado de teste unitário, concentra-se no esforço de validação da menor unidade de projeto de software e tem por objetivo garantir que a lógica do programa esteja completa e correta;

Teste de integração: verifica basicamente se as unidades testadas de forma individual comportam-se de maneira adequada quando são colocadas juntas, isto é, integradas;

Teste de sistema: é usado para demonstrar que o sistema inteiro está correto, para tanto, esse teste coloca o sistema para funcionar junto com outros sistemas e elementos de hardware, nas mesmas condições em que será utilizado pelo cliente;

Teste de aceitação: é realizado pelo cliente quando partes significativas, ou o sistema como um todo, são consideradas concluídas após a realização dos testes. Os testes de aceitação podem ser do tipo alfa, quando realizado no mesmo ambiente onde o software foi desenvolvido, ou beta realizado no ambiente onde o sistema será implantado.

2.2.3 Cenários e casos de teste

Um cenário de teste de software é uma história hipotética que visa ajudar a solucionar um problema complexo, recriando um caminho a ser seguido ou situação a ser testada, podendo ser descrito com base em especificações como UML (*Unified Modeling Language*) [BRC+07]. Um cenário de teste de software é composto por casos de teste, que incluem os dados de entrada, os resultados esperados, as ações e as condições gerais para a execução dos testes.

Um caso de teste é uma especificação mais detalhada do teste com informações sobre itens do sistema, estabelecendo quais informações serão empregadas durante os testes do cenário e quais serão os resultados esperados.

Em Engenharia de Software, um "*test suítes*" é uma coleção de casos de teste que se destina a ser utilizada como base para um programa demonstrar que ele possui um conjunto de especificações para avaliar o comportamento do sistema.

Um *test suíte* geralmente contém instruções detalhadas ou metas para cada conjunto de casos de teste e informações sobre a configuração do sistema a ser utilizado durante o ensaio, podendo conter também condições, estados, etapas e as descrições dos testes seguintes [Som07].

A geração dos casos de teste pode usar inúmeras metodologias, e dentre as metodologias utilizadas, destacam-se o:

- **Teste Randômico:** são aplicados sobre o método da caixa-preta e é com esse tipo de teste que é gerado o maior número de caso de teste. Mas o mesmo seleciona apenas algumas entradas, não garantindo a efetividade dessa solução;
- **Teste Estatístico:** nesse teste os eventos de interesse são sequências de estímulos que representam uma execução do software, onde uma descrição estatística das sequências é obtida pela definição de variáveis randômicas que caracterizam o perfil do conjunto total de sequências usadas na verificação do software;
- **Teste Estocástico:** este é um método para seleção de casos de teste que possibilita uma alternativa de formalização de teste de software, através da criação de modelos estocásticos, por parte dos testadores, que descrevem o comportamento do sistema em alternativa a geração de casos de teste de software;
- **Teste Baseado em Modelos:** o teste baseado em modelos consiste em usar ou derivar modelos do comportamento esperado para produzir especificações de casos de teste (*test suítes*) que podem revelar discrepâncias entre o comportamento real do programa e o modelo.

Os modelos podem ser derivados de modelos formais tais como máquinas de estados finitos (FSM - *Finite State Machine*) ou gramáticas, ou informais, como diagramas UML *(Unified Modeling Language*), especialmente dos diagramas comportamentais como diagrama de estados, diagrama de atividades e diagrama de sequência.

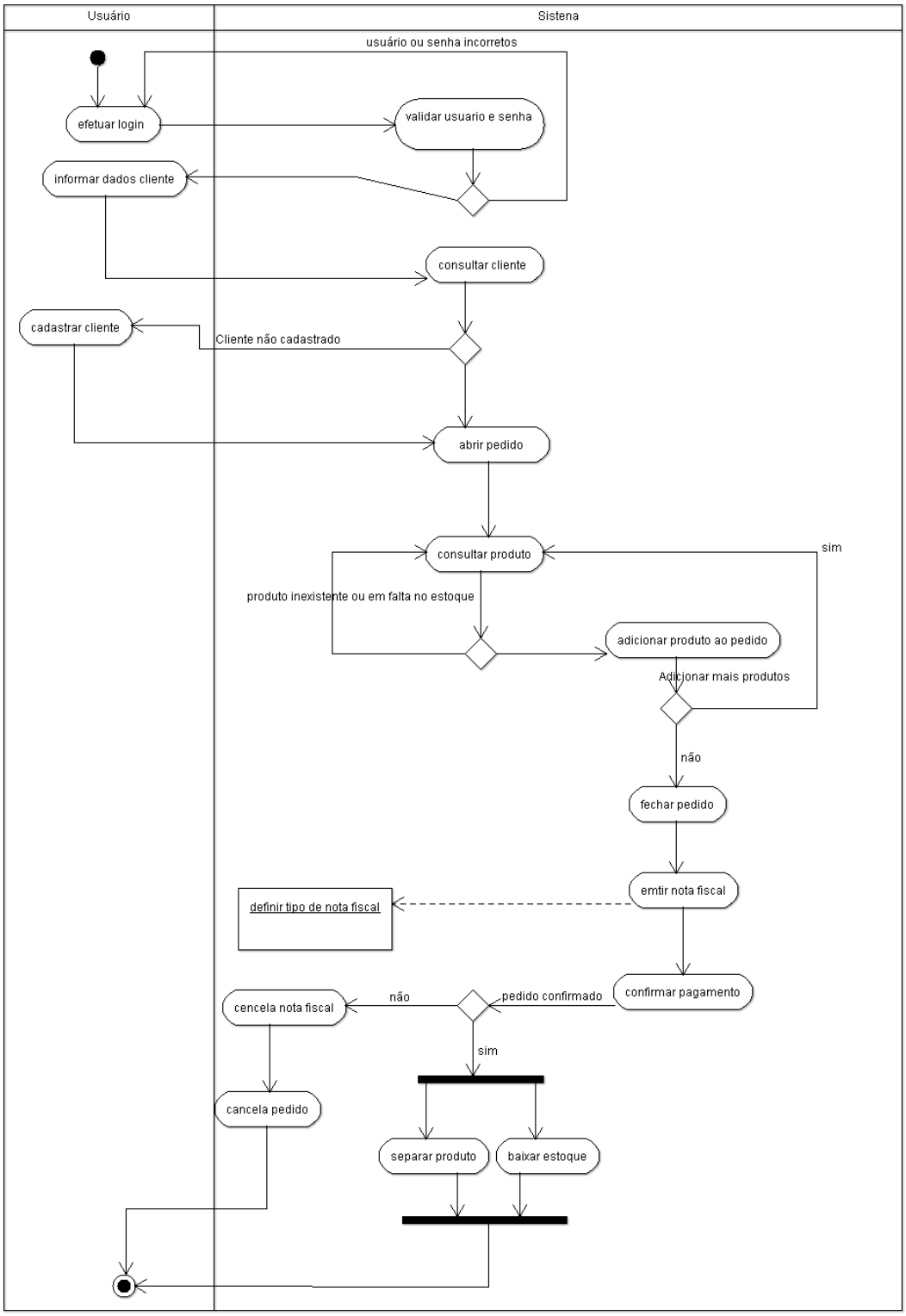


Figura 2.1: Exemplo de diagrama de atividades

3. Redes de Autômatos Estocásticos

Neste capitulo é apresentada uma definição informal do formalismo de Redes de Autômatos Estocásticos (SAN - *Stochastic Automata Networks*), a partir da descrição das primitivas utilizadas pelo formalismo.

3.1 Redes de Autômatos Estocásticos

Redes de Autômatos Estocásticos, do inglês - SAN *(Stochastic Automata Networks)*, é um formalismo proposto por Plateau [PIA91] nos anos 80, para modelagem analítica de sistemas com um grande espaço de estados, equivalente a cadeia de Markov, dividido em subsistemas.

Essa modularização característica do formalismo SAN, permite o armazenamento e a solução de sistemas complexos, onde o problema de explosão de estados que ocorre com o uso do formalismo de cadeias de Markov, com o qual SAN possui equivalência, é atenuado [Sal03].

Uma SAN é composta por um conjunto finito de estados e um conjunto finito de transições entre os estados. As transições entre os estados são disparadas por eventos que podem ser do tipo local ou sincronizante e onde cada evento tem uma taxa de ocorrência associada a ele [PIA91].

O estado de um autômato representa toda a informação referente a seu passado, assim, para um determinado conjunto de estados, um sistema poderá assumir somente um estado a cada momento, sendo este denominado como estado local. O estado local do sistema modelado com o uso do formalismo SAN é o estado individual de cada autômato do modelo. O estado global de um modelo do SAN é definido como a combinação de todos os estados locais de cada autômato componente da mesma [Fer98].

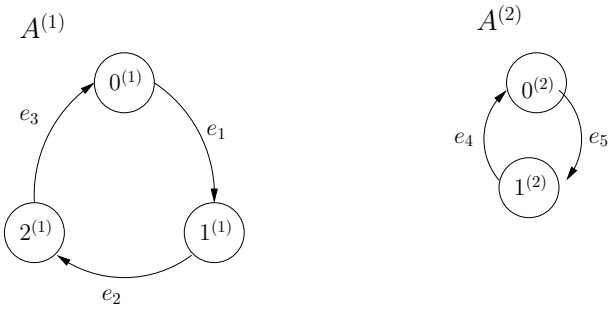
3.1.1 Eventos

Em uma SAN, um evento é uma entidade responsável por disparar uma transição que altera o estado global de um modelo. Cada transição possui um ou mais eventos associados a ela e esta é disparada pela ocorrência de qualquer um dos eventos. Os eventos podem ser de dois tipos: eventos locais ou eventos sincronizantes.

3.1.2 Eventos Locais

Os eventos locais mudam o estado global da SAN alterando apenas o estado de um único autômato que passa de um estado para outro. O evento local permite que vários autômatos tenham um comportamento paralelo, trabalhando independentemente, sem que haja interações entre eles [PIA91, BFF+04]. A Figura 3.1 apresenta dois autômatos que possuem apenas eventos locais [Sal03].

O modelo SAN apresentado na Figura 3.1 é composto por dois autômatos, o $A^{(1)}$ que possui três estados $0^{(1)}$, $1^{(1)}$ e $2^{(1)}$, já o autômato $A^{(2)}$ possui dois estados $0^{(2)}$ e $1^{(2)}$. Os autômatos apresentados nesse exemplo possuem apenas eventos locais, os quais são e_1, e_2, e_3, e_4 e e_5,



Tipo	Evento	Taxa
loc	e_1	α_1
loc	e_2	α_2
loc	e_3	α_3
loc	e_4	α_4
loc	e_5	α_5

Figura 3.1: SAN com eventos locais

sendo que os eventos e_1, e_2, e_3 são modelados no autômato $A^{(1)}$ e os eventos e_4 e e_5 no autômato $A^{(2)}$

A Figura 3.2 apresenta a CTMC (Continuous Time Markov Chain) equivalente ao modelo SAN da Figura 3.1.

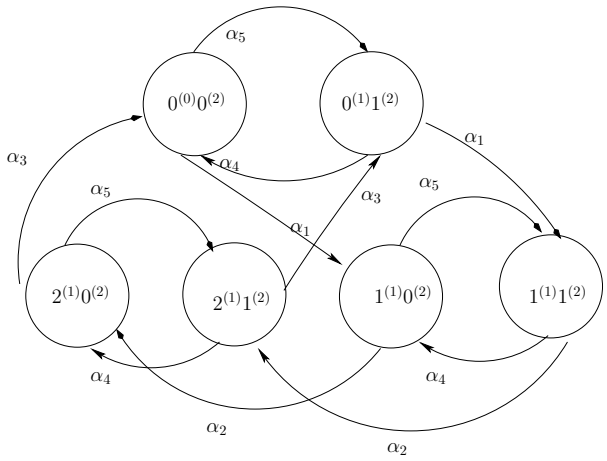
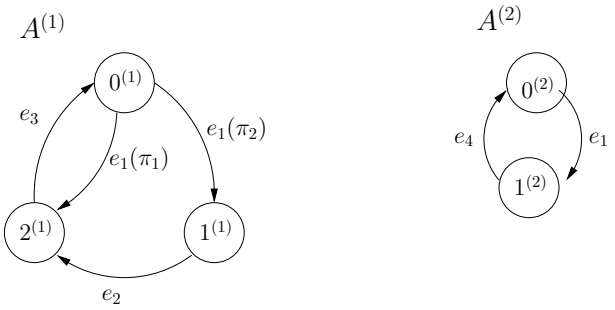


Figura 3.2: Cadeia de Markov equivalente ao modelo SAN da Figura 3.1

3.1.3 Eventos Sincronizantes

Os eventos sincronizantes disparam transições que mudam o estado local de mais de um autômato, isto é, dois ou mais autômatos podem mudar seus estados locais simultaneamente, ocasionando assim, também uma mudança no estado global da SAN. A Figura 3.3 apresenta um autômato com eventos locais e sincronizantes.

Na Figura 3.3, o autômato $A^{(1)}$ possui três estados $0^{(1)}$, $1^{(1)}$ e $2^{(1)}$, já o autômato $A^{(2)}$ possui dois estados $0^{(2)}$ e $1^{(2)}$. O modelo apresenta ainda quatro eventos, sendo três locais e_2, e_3, e_4, e um sincronizante e_1. O evento e_1 possui uma probabilidade de ocorrência de π_1 e π_2 respectivamente. O evento sincronizante e_1 altera os estados dos autômatos $A^{(1)}$ e $A^{(2)}$ de $0^{(1)}$ para $1^{(1)}$ e $0^{(2)}$ para $1^{(2)}$, respectivamente, com uma probabilidade π_1 e também altera os estados dos autômatos $A^{(1)}$ e $A^{(2)}$ de $0^{(1)}$ para $1^{(1)}$ e $0^{(2)}$ para $1^{(2)}$, respectivamente, com uma probabilidade π_2.



Tipo	Evento	Taxa
syn	e_1	α_1
loc	e_2	α_2
loc	e_3	α_3
loc	e_4	α_4

Figura 3.3: SAN com eventos locais e sincronizantes

A Figura 3.6 apresenta a CTMC (*Continuous Time Markov Chain*) equivalente ao modelo SAN da Figura 3.3.

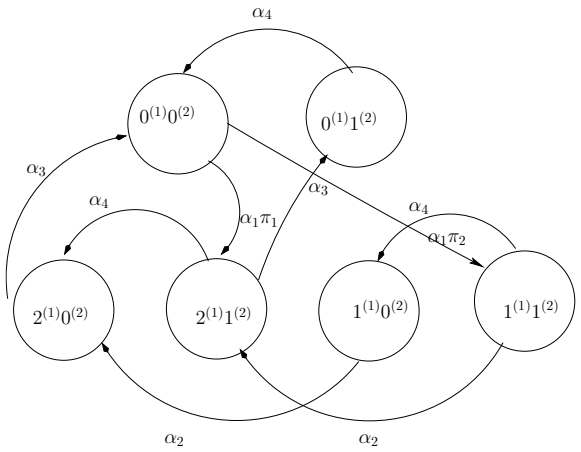


Figura 3.4: Cadeia de Markov equivalente ao modelo SAN da Figura 3.3

Cada evento deve possuir uma taxa de ocorrência e uma probabilidade associada ao mesmo. Tanto a taxa quanto a probabilidade de ocorrência podem ter valores constantes ou valores funcionais, os quais assumem valores diferentes de acordo com os estados de outros autômatos do modelo [Sal03].

A classificação de um evento como local ou sincronizante é feita pelo identificador do evento e no conjunto de eventos de um autômato. Caso o identificador apareça em um único autômato, esse evento é classificado como local, caso ele apareça em mais de um autômato ele será denominado como sendo sincronizante [Sal03].

3.1.4 Taxas funcionais

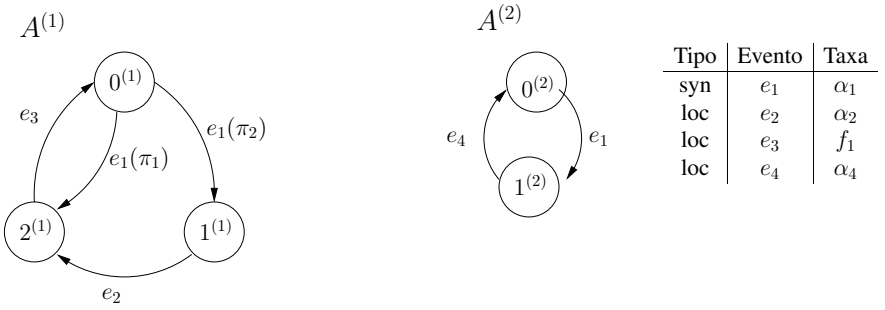
As taxas funcionais são usadas quando as taxas de ocorrência dos eventos que disparam as transições, não forem constantes, ou seja, tem-se uma função do estado local de outros autômatos da SAN, avaliada conforme os estados atuais do modelo. As taxas funcionais podem ser atribuídas a eventos locais ou sincronizantes para definir se um evento pode ou não ocorrer dado o estado de outros autômatos da rede.

As transições disparadas por eventos aos quais existam taxas funcionais associadas baseiam-se nos estados atuais do modelo, podendo variar seu valor conforme os estados em que se encontram os autômatos envolvidos na função.

Por exemplo, se no autômato $A^{(1)}$ quiséssemos representar uma transição disparada por um evento local ao qual existe uma taxa funcional associada, teríamos que definir uma função $f1$, conforme é demonstrado abaixo:

$$f1 = \begin{cases} \lambda_1 & se\ A^{(2)}\ esta\ no\ estado\ 1^{(2)} \\ 0 & se\ A^{(2)}\ nao\ esta\ no\ estado\ 1^{(2)} \end{cases}$$

A Figura 3.5 mostra uma SAN que possui um evento ao qual foi atribuída taxa funcional, mostrada anteriormente. A taxa atribuída determina que a transição disparada pelo evento e_3 ocorrerá somente quando autômato $A^{(2)}$ estiver no estado $1^{(2)}$, caso contrário a transição não ocorrerá.



Tipo	Evento	Taxa
syn	e_1	α_1
loc	e_2	α_2
loc	e_3	f_1
loc	e_4	α_4

Figura 3.5: SAN com taxas funcionais

A figura 3.6 apresenta a CTMC (Continuous Time Markov Chain) equivalente ao modelo SAN da Figura 3.5.

3.1.5 Função de atingibilidade

Através do uso da ferramenta PEPS *(Performance Evaluation of Parallel Systems)* [BMF+03], podemos utilizar outras duas funções, a função de atingibilidade e a função de integração.

Uma função de atingibilidade é uma função booleana que determina os estados atingíveis do modelo, dentro do espaço total de estados de um modelo SAN. Quando a avaliação desta função for igual a "*true*", pode-se afirmar que os estados do modelo avaliados por "*true*"são atingíveis.

Porém, na maioria dos modelos, não é o que acontece, pois quando se modela um sistema do mundo real é possível identificar-se condições para que determinados eventos ocorram ou não, o que significa dizer que quando estas condições não forem cumpridas, alguns estados globais terão probabilidade nula de ocorrerem, ou seja, serão considerados como estados inatíngiveis. A função de atingibilidade para o modelo apresentado na Figura 3.5, com base na linguagem defnida para a ferramenta PEPS2003 [BMF+03], pode ser escrita da seguinte forma:

reachability = 1

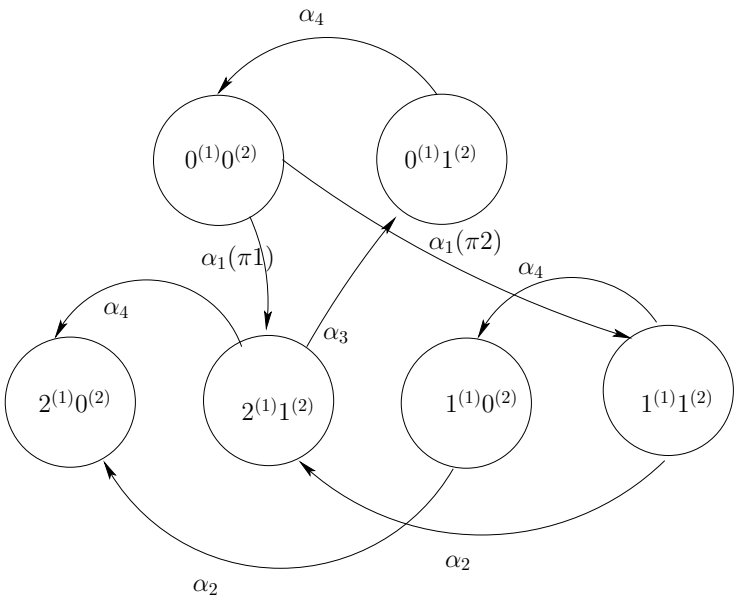


Figura 3.6: Cadeia de Markov equivalente ao modelo SAN da Figura 3.5

A função de atingibilidade pode ser melhor compreendida quando imaginamos um modelo com um número N de clientes disputando uma quantidade R de recursos e que a quantidade de recursos é inferior a quantidade de recursos disponíveis.

Para o exemplo aqui descrito, adaptado do trabalho de [Nas09], é apresentado um modelo onde existem C clientes que disputam R recursos, como demonstrado na Figura 3.7. Um autômato que descreve os clientes pode ser modelado com dois estados, aguardando ou usando, quando o cliente está aguardando o recurso, significa dizer que autômato se encontra no estado $0^{(i)}$ e quando o cliente está de posse do recurso o autômato se encontra no estado $1^{(i)}$.

A função de atingibilidade, com base na ferramenta PEPS [BMF+03], para o exemplo descrito é escrita da seguinte forma:

$$\text{reachability} = (\sum_{i=1}^{C} \text{st } (C^{(i)}) \leq R^{(i-1)})$$

Para esse exemplo a atingibilidade é possível caso a somatória dos estados do autômato $C^{(i)}$, que representa a quantidade de clientes, seja menor ou igual à quantidade de recursos disponíveis, representado pelo autômato $R^{(i)}$.

3.1.6 Função de integração

As funções de integração são definidas para a obtenção de resultados numéricos sobre o modelo SAN visando avaliar qual a probabilidade do modelo estar em um determinado estado. 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 [BBF+04].

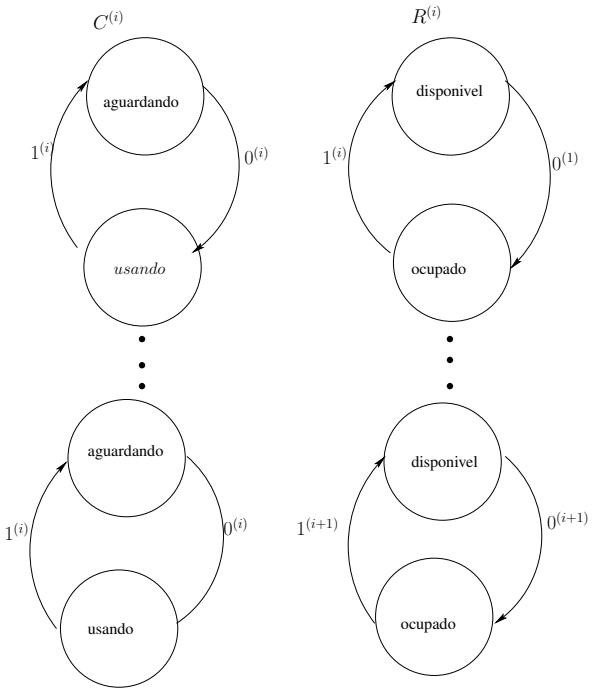


Figura 3.7: Modelo SAN para a função de atingibilidade

As funções de integração não fazem parte do formalismos SAN, sendo apenas uma implementação da ferramenta PEPS [BMF+03] mas que se mostra bastante útil para a obtenção de resultados, a partir dos modelos gerados.

As funções de integração são avaliadas sobre o vetor de probabilidades, o qual contém a probabilidade do modelo de se encontrar em cada um dos estados pertencente a ele. Por exemplo, podemos compor a seguinte função de integração para o modelo da Figura 3.5, onde *X* representa a probabilidade do autômato $A^{(2)}$ se encontrar no estado $1^{(2)}$ [Web03]:

$$X = st(A^{(2)} == 1)$$

3.2 Trabalhos relacionados ao formalismo Markoviano

Nesta seção são descritos alguns métodos de conversão de diagrama de atividades da UML para formalismos como Redes de Petri e SAN. Tais trabalhos servirão de base para a elaboração do método proposto nesta dissertação.

Três dos trabalhos aqui descritos baseiam-se em redes de Petri. Os métodos baseados em redes de Petri são: o desenvolvido por Bernardi et al [BDM02], o método de conversão desenvolvido por Lachtermacher et al [LSP+08], o método de conversão desenvolvido por López-Grao et al [LGMC04] e adicionalmente o método de conversão de diagrama de estados da UML para SAN, desenvolvido por Barros [Bar06] também é apresentado.

Redes de Petri é uma ferramenta de modelagem aplicável a uma série de sistemas, especialmente aqueles com eventos concorrentes. Elas foram criadas por C. A. Petri, que na sua tese de

doutorado apresentou um tipo de grafo orientado e bipartido com estados associados, com o objetivo de estudar a comunicação entre autômatos [Pet62].

De acordo com Reisig [Rei85], as redes de Petri são formadas por dois tipos de componentes: um ativo denominado transição, correspondente a uma ação realizada dentro do sistema, e outro passivo. Uma rede de Petri recebe uma ou mais marcas (ou *tokens*) que podem estar em mais de um lugar ao mesmo tempo. O disparo de uma transição consome marcas, fazendo com que elas mudem de lugar [Sal03].

Os trabalhos aqui apresentados não esgotam as discussões sobre o tema, mas elencam algumas contribuições significativas para esta pesquisa.

3.2.1 Método de conversão de diagrama de sequência e de estados em redes de Petri

No método de conversão apresentado por Bernardi et al [BDM02], é descrito um estudo sobre a utilização de diagramas de sequência e diagramas de estados da UML para a validação e avaliação do desempenho de sistemas, a partir da conversão dos mesmos para uma rede de Petri. Para tal Bernardi et al [BDM02] assume que um sistema é especificado como um conjunto de estados (sequência de gráficos e diagramas), os quais são usados para representar “Execuções de Interesse”.

Segundo Bernardi et al em [BDM02], a conversão deve-se ao fato da UML não possuir uma semântica formal que possibilite a aplicação de uma forma direta de análise matemática no sistema modelado com o uso dessa linguagem. Assim a utilização de formalismos, dentre eles as redes de Petri, possibilita a análise matemática do sistema.

A pesquisa propõe a conversão automática de diagramas de estados e diagramas de sequência em GSPN (*Generalized Stochastic Petri Nets* - Redes de Petri Estocásticas Generalizadas) [ABC$^+$91], para que seja possível a aplicação de métodos matemáticos de avaliação de desempenho de sistemas. Cabe ressaltar que os autores não detalham em seu trabalho, quais índices de desempenho ou probabilidades de ocorrência dos estados foram obtidos a partir da conversão dos diagramas.

De acordo com Bernardi et al [BDM02], a pesquisa tinha por objetivo analisar a coerência entre as duas descrições e avaliar estocasticamente se o comportamento do sistema é consistente com os padrões de interação descritos pelos diagramas de sequência e de estados.

A proposta de conversão não leva em consideração a totalidade de elementos dos diagramas de sequência e de estados. Nos diagramas de sequência, os recursos não considerados são: iterações, temporalidade e restrições. Já no diagrama de estados, não são considerados: estados compostos, *pseudoestados* e histórico dos estados.

Maiores detalhes sobre os elementos que compõem os diagramas de estados e de sequência podem ser encontrados em ”UML: guia do usuário”[BRJ05]. As análises do modelo obtido são feitas segundo uma das duas técnicas denominadas por Bernardi et al em [BDM02], como *The full case* e *The constrained case*.

A primeira técnica é baseada na construção de um modelo completo do diagrama de atividades e do diagrama de sequência para a geração da LGSPN (*Labeled Generalized Stochastic Petri Nets*

- Redes de Petri Estocásticas Generalizadas Rotuladas) resultante. Na segunda técnica, Bernardi et al [BDM02] define os possíveis caminhos na rede de Petri, a partir de alterações na sua estrutura, assim, os resultados obtidos são analisados na pesquisa, de acordo com um comportamento do sistema, definido pelos pesquisadores.

Nos dois casos, os modelos foram submetidos, afim de se obter os índices de desempenho do modelo, a uma ferramenta denominada GREATSPN (*GRaphical Editor and Analyzer for Timed and Stochastic Petri Net*), ferramenta desenvolvida pelo departamento de Informática da Universidade de Torino, para a modelagem, validação e avaliação de desempenho de sistemas distribuídos, usando Redes de Petri Estocásticas Generalizadas e sua extensão colorida[1] [Dep08].

Essa pesquisa contribuiu para a definição dos primeiros passos para a conversão de diagramas de atividades, pois com base nessa pesquisa defini-se que as atividades seriam convertidos em estados de um modelo SAN. Outra contribuição dessa pesquisa, foi a definição de que não seria possível utilizar todos os elementos do diagrama de atividades no momento da conversão. Como contribuição, cita-se também a definição de transições entre os estados bem como os eventos que disparam essas transições.

3.2.2 Método de conversão de diagrama de atividades para redes de Petri com base em QVT

O método proposto por Lachtermacher et al em [LSP+08] tem por objetivo converter diagramas de atividades da UML em redes de Petri, baseado em uma especificação definida pela OMG (*Object Modeling Group*), denominada de MOF/QVT (*Meta Object Facility/Query View Transformation*) [OMG08b], que visa a conversão de um modelo UML em um modelo matemático. O processo de conversão proposto é executado considerando os seguintes aspectos [LSP+08]:

- o elemento *Atividade* no diagrama de atividades é convertido em *transição* em uma rede de Petri;
- o elemento *Estado Inicial* do diagrama de atividades e o elemento *Decisão* são convertidos em elemento *Lugar*, na rede de Petri, sem nenhuma marca;
- os elementos *Fork* (separação) e *Join* (junção) são mapeados como uma *transição* na rede de Petri;
- os elementos *Final de Fluxo* e *Final de Atividade* são mapeados para um lugar que não tem saída, chamado de *Sink*;
- o elemento *Objeto* é mapeado para um *lugar* com uma *marca* que representa a mudança de estado do objeto.

[1] Redes de Petri coloridas permitem que *tokens* individualizados (coloridos) representem diferentes processos ou recursos em uma mesma sub-rede.

Em sua pesquisa, Lachtermacher et al [LSP+08] afirma que para atender a especificação da OMG foram desenvolvidos dois metamodelos, um contendo as definições para a composição de um diagrama de atividades e o outro contendo as definições para a composição de uma rede de Petri.

Um metamodelo é uma descrição, ou seja, a definição de regras para criação de novos modelos, a partir de um modelo padrão. As regras para a conversão, nas quais o metamodelo se baseia, foram especificadas com o uso da linguagem (*Object Constraint Language*) [OMG08a] e o uso de uma ferramenta *opensourse* denominada MediniQVT [IKV08].

O objetivo da pesquisa foi realizar um estudo sobre QVT, visando auxiliar o desenvolvedor de software a utilizar o diagrama de atividades para finalidades as quais ele foi estruturado para realizar, que é a de demonstrar o fluxo de execução de processos ou de um algoritmo que compõem um sistema [FoS08].

Para alcançar o objetivo proposto, Lachtermacher et al [LSP+08] realizou a conversão de modelos UML para redes de Petri, afim de demonstrar a viabilidade de conversão de modelos. A pesquisa não teve por objetivo extrair nenhum índice de desempenho ou probabilidades a partir dos modelos gerados. As contribuições dessa pesquisa para a construção do método proposto podem ser descritas da seguinte forma:

O processo de transcrição do elemento atividade é o mesmo tanto na pesquisa de [LSP+08], onde o referido elemento é transcrito como um lugar em uma rede de Petri, quanto na dissertação aqui apresentada, pois esse elemento representa um estado em uma SAN. O elemento decisão e união são mapeado como um *lugar* sem marcas na rede de Petri. No método aqui proposto, esses elementos são mapeados como sendo transições entre estados do modelo SAN.

Outra contribuição deste trabalho é em ralação ao elemento *fork*, que tanto neste trabalho relacionado, quanto no método proposto é mapeado como sendo uma transição.

3.2.3 Método de conversão de diagrama de atividades para redes de Petri

Na pesquisa apresentada por López-Grao et al [LGMC04] é descrito um método de conversão de diagramas de atividades em uma rede de Petri, adotando uma semântica formal que possibilite transformar cada elemento de um diagrama de atividades em um elemento equivalente em LGSPN (*Labeled Generalized Stochastic Petri Nets*).

O resultado da composição é uma nova LGSPN que representa um modelo para avaliação de desempenho do sistema com um razoável grau de expressividade para lidar com a descrição e avaliação da dinâmica de sistemas grandes e complexos.

O objetivo do método de conversão proposto por López-Grao é formalizar uma semântica que permita traduzir diagramas de atividades em modelos de Redes de Petri Estocáticas Generalizadas analisáveis.

A pesquisa buscou uma tradução do digrama de atividades em um modelo estocástico baseado em redes de Petri, o qual permita verificar propriedades lógicas, bem como calcular os índices de desempenho. Os autores não informam, no material pesquisado, quais índices de desempenho podem ser extraídos a partir do modelo gerado.

Na pesquisa de López-Grao et al [LGMC04], apresenta-se uma breve descrição de um protótipo

da ferramenta que implementa o método de conversão proposto pelos autores. A conversão de cada um dos elementos do diagrama de atividades para a rede de Petri pode ser resumida como um processo executado em três etapas, obedecendo regras de transição definidas com o uso da linguagem OCL [OMG08a] e podendo ser descrita da seguinte forma:

- inicialmente as transições são identificadas e para cada tipo de transição uma regra é aplicada;
- posteriormente é gerada a rede de Petri correspondente ao diagrama;
- finalmente, em um terceiro passo, uma nova LGSPN é gerada considerando-se, caso existam, pseudo estados como *fork* e *join*.

López-Grao et al relata em [LGMC04] que com o objetivo de automatizar o processo de conversão foi desenvolvida uma ferramenta, utilizando-se a linguagem Java, que coleta informações em arquivo .xmi. O XMI combina os benefícios da XML para definição, validação e compartilhamento de formatos de documentos com os benefícios da linguagem de modelagem visual UML para especificação, visualização, construção e documentação de objetos construídos também a partir de modelos UML [w3c08].

Os arquivos .xmi citados por Lopéz-Grao et al em [LGMC04] foram gerados com o uso da ferramenta ArgoUML [Tig08], que segundo os autores, apresenta limitações de modelagem que foram compensadas durante o desenvolvimento da ferramenta proposta. A rede de Petri resultante dessa pesquisa tem seus índices de desempenho extraídos com o uso da ferramenta GreatSPN [Dep08].

As contribuições dessa pesquisa para a construção do método proposto podem ser descritas da seguinte forma:

- o elemento decisão é substituído por transições de saída equivalente a estados de ação, preservando as propriedades inerentes ao desempenho da rede gerada. No método proposto esse elemento também é transcrito da mesma forma;
- o elemento *merge* tanto na pesquisa de [LGMC04] quanto no método proposto são transcritos como transição entre os estados;
- o elemento *fork*, tanto na pesquisa quanto no método proposto, é mapeado como transições que no caso de SAN é feita entre os estados;

3.2.4 O método de conversão de diagrama de estados para SAN

A pesquisa apresentada por Barros [Bar06] teve como objetivo obter um modelo de uso do sistema a ser construído, baseado em diagrama de estados da UML, sobre o qual, com o uso de SAN seria possível efetuar análises de comportamento do software.

Na pesquisa de Barros [Bar06], inicialmente define-se um método de conversão dos elementos do diagrama de estados para uma estrutura correspondente em SAN e as taxas de transição a serem utilizadas nesse mesmo modelo. As informações referentes aos eventos (nome e tipo) foram

compostas a partir de informações coletadas com base nos estados envolvidos em cada transição, considerando-se o contexto em que essa transição ocorre.

O processo de transcrição do diagrama UML para o modelo SAN ocorre considerando-se cada estado do diagrama como sendo um estado de um autômato. As transições existentes no diagrama são modeladas como eventos locais, ou eventos sincronizantes, podendo ser associadas taxas funcionais aos eventos que disparam essa transição, quando da existência de condições de guarda associadas ao diagrama de estados.

Como resultado final de sua pesquisa Barros [Bar06] apresenta uma ferramenta que executa o processo de conversão de diagramas UML para SAN, a partir de informações extraídas de um arquivo .xmi [w3c08], gerando um arquivo com extensão .san, que posteriormente é submetido a ferramenta PEPS (*Performance Evaluation of Parallel Systems*) [BMF+03].

As contribuições dessa pesquisa para a construção do método proposto, podem ser descritas da seguinte forma:

- o elemento *fork*, em SAN, 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. No método proposto cada transição oriunda de um elemento *fork* também é modelada como um autômato distinto;

- o tratamento de condições de guarda que tanto nesta pesquisa quanto no método são tratadas de maneira que a presença de condições de guarda em determinadas transições sugere a inclusão de uma estrutura (um autômato), que avalie a restrição, para então permitir (ou não) o disparo dessa transição.

No capítulo seguinte, apresentamos o método de conversão de diagramas de atividades para Redes de Autômatos Estocásticos (SAN).

4. Método de conversão de diagramas de atividades em SAN

Neste capítulo são apresentadas as regras criadas para a conversão de diagramas de atividades da UML (*Unified Modeling Language*) para um modelo SAN (*Stochastic Automata Networks*), buscando descrever as ações executadas em cada passo da conversão.

4.1 Descrição das regras de conversão

Nesta seção, serão descritas as regras de conversão de diagramas de atividades da UML para um modelo SAN. Inicialmente cabe ressaltar que um dos principais desafios do uso de diagramas UML para a avaliação de desempenho é a escolha de uma semântica de forma adequada, que não seja nem demasiadamente restritiva, permitindo ao modelador um bom grau de expressividade, nem muito permissiva a ponto de não representar o comportamento adequado do sistema a ser modelado.

Assim, segundo Bernardi et al [BDM02] o uso de métodos de conversão de UML para formalismos matemáticos Markovianos implica em tomadas de decisão por parte do desenvolvedor sobre a interpretação do diagrama.

O método que se propõem neste trabalho é uma conversão manual de diagramas de atividades da UML para um modelo SAN que possibilite realizar análises de comportamento, tais como, determinar a funcionalidade do sistema que possui uma maior probabilidade de uso. Pretende-se ainda, com base no modelo SAN obtido, gerar casos de teste de software.

A seguir são descritos os passos para a obtenção de uma SAN a partir de um diagrama de atividades da UML. O método de conversão aqui apresentado baseia-se nos métodos propostos por:

- Bernardi et al [BDM02], para a conversão de diagramas de atividades em rede de Petri;
- Lachtermacher et al [LSP+08] método de conversão de diagrama de atividades para rede de Petri com base em QVT;
- López-Grao et al [LGMC04] o método de conversão de diagrama de atividades para rede de Petri;
- Barros [Bar06] método de conversão de diagrama de estados para SAN.

Ressalta-se que nos modelos aqui apresentados, tanto na descrição dos passos de conversão quanto nos exemplos demonstrados, não se está levando em consideração as taxas de ocorrência dos eventos atribuindo-se a todas elas o valor 1. Tal decisão se deve ao fato de estarmos focando apenas a execução do método, visando demonstrar sua viabilidade.

4.2 Passos para conversão de diagramas UML para SAN

O processo de conversão pode ser descrito em sete passos, conforme apresentado nas subseções a seguir.

4.2.1 Conversão dos elementos estado inicial e estado final

Os diagramas de atividades mostram uma sequência de ações, e deverão indicar o ponto de partida da sequência, e para tal, é utilizado o elemento estado inicial. O estado inicial é desenhado como um sólido círculo com uma linha de transição (seta) que o conecta a primeira atividade modelada no diagrama.

Embora a especificação UML não defina uma localização específica para o estado inicial no diagrama de atividades, ele é normalmente colocado antes da primeira atividade, no canto superior esquerdo do diagrama. Outra característica desse elemento é a de que só pode existir um único estado inicial e apenas uma linha de transição que liga o mesmo a uma atividade.

O fluxo de atividade termina quando a linha de transição da última ação no diagrama se conecta a um estado final, símbolo que é representado por um círculo em torno de um círculo menor, sendo este totalmente preenchido. Todo diagrama de atividades deve ter pelo menos um símbolo do estado final, podendo possuir vários [Bel09].

Passo 1: inicia-se com a identificação dos elementos que compõem o diagrama de atividades e que serão convertidos em estados ou transições de um ou mais autômatos. Assim os elementos $estado - inicial$ e $estado - final$ são convertidos como sendo um autômato com dois estados nomeados como I e F e duas transições disparadas por dois eventos sincronizantes. A criação desse autômato se deve ao fato de que a semântica da UML, para o diagrama de atividades, define que deva existir um elemento $estado - inicial$ e um elemento $estado - final$.

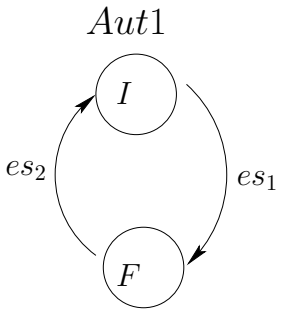
A Figura 4.1 traz a representação gráfica de um diagrama de atividades que possui os elementos $estado - inicial$ e $estado - final$.

Figura 4.1: Diagrama de atividades com elemento estado-inicial e estado-final

A Figura 4.2 traz a representação gráfica do autômato obtido com base na Figura 4.1. O autômato mostrado na Figura 4.2, possui dois estados denominados I, que representa o elemento $estado - inicial$ e ao qual foi associada uma transição disparada pelo evento sincronizante es_1 e

um estado denominado F que representa o elemento $estado - final$ e que possui uma transição disparada pelo evento sincronizante es_2.

Neste exemplo, a atividade demonstrada na Figura 4.1 não foi convertida, aqui foram considerados apenas os elementos $estado - inicial$ e $estado - final$. Quando um diagrama de atividades possuir mais que um estado final, estes serão convertidos em apenas um estado denominado F, conforme descrito anteriormente.



Tipo	Evento	Taxa
syn	es_1	1
syn	es_2	1

Figura 4.2: Autômato que representa a Figura 4.1

4.2.2 Conversão do elemento atividade

Um elemento atividade em um diagrama de atividades é indicado por um objeto retangular com as bordas arredondadas e o texto em seu interior indica uma ação. O nome atribuído a uma atividade deve ser formado pela combinação de um verbo e um substantivo. A descrição de uma atividade deve definir uma única instância, exprimindo o que ela faz e como ela o faz [Bel09].

Passo 2: Cada atividade de um diagrama é mapeada como sendo um estado de um autômato nomeado como E_i onde i representa a identificação do estado, com i variando de $1..n$ onde n representa a quantidade de atividades existentes no diagrama. Afim de melhor representar cada elemento pode-se utilizar como forma de identificação de cada estado do modelo SAN a terminologia UC_i (*Use Case*), com i variando de $1..n$, juntamente com a nomenclatura anteriormente citada. Tal definição leva em conta o fato de que os diagramas de atividades são criados com base nas especificações definidas no diagrama de caso de uso.

A Figura 4.3 traz a representação gráfica do elemento atividade em UML ao passo que a Figura 4.4 traz a representação gráfica do modelo SAN obtido com base na Figura 4.3.

O modelo SAN obtido possui dois autômatos, denominados $Aut1$ e $Aut2$, sendo que o primeiro autômato representa a execução do passo 1 descrito anteriormente e o segundo representa a execução do passo 2. O autômato $Aut2$ possui os estados 0 que possui uma transição disparada pelo evento E_1 e é utilizado para representar o momento em que nenhuma atividade está sendo executada. O autômato traz ainda os estados E_1 que possui uma transição disparada pelo evento local el_1, e o estado E_2 que possui uma transição disparada pelo evento sincronizante es_2. Ambos os estados são resultantes da conversão das atividades 1 e 2 respectivamente.

A Figura 4.5 traz a representação gráfica da cadeia de Markov equivalente ao modelo SAN resultante da execução do passo 2.

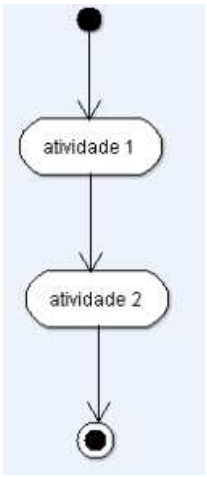
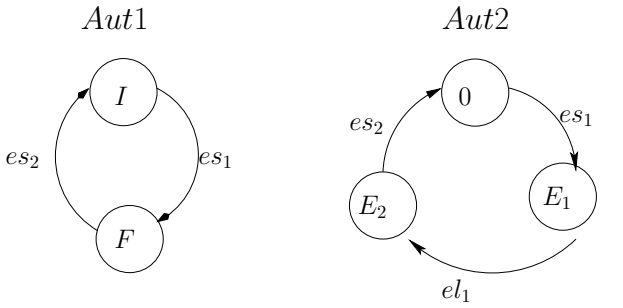


Figura 4.3: Diagrama de atividades com elementos atividade



Tipo	Evento	Taxa
syn	es_1	1
syn	es_2	1
loc	el_1	1

Figura 4.4: SAN obtida a partir da conversão da Figura 4.3

4.2.3 Conversão do elemento transição

A transição representa o relacionamento entre duas atividades e são chamadas de transição não-ativadas, por não representarem um espaço de tempo. Quando um estado de ação ou de atividade se completa, o fluxo de controle passa imediatamente para o próximo estado de ação ou de atividade conectados via uma transição. O fluxo que representa o elemento transição é indicado por uma seta unidirecional [Bel09].

Passo 3: As transições entre as atividades do diagrama são mapeadas como transições entre os estados. Essas transições são disparadas por eventos locais ou sincronizantes, de acordo

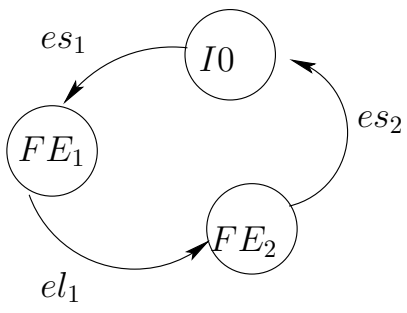


Figura 4.5: Cadeia de Markov equivalente ao modelo SAN da Figura 4.4

com as características do diagrama. As transições em SAN, assim como as transições em UML, são representadas por uma seta unidirecional.

4.2.4 Conversão do elemento decisão

Normalmente algumas decisões precisam ser tomadas durante uma atividade, dependendo do resultado de uma ação específica prévia. Para tal utiliza-se o elemento decisão representado por um diamante em um diagrama de atividades.

Uma vez que a decisão terá pelo menos dois resultados diferentes, o elemento decisão terá duas linhas de transição conectado a duas ações diferentes. Cada linha de transição envolvida em um ponto de decisão deve ser rotulada com um texto que indica uma condição de guarda, comumente abreviado como guardas. A condição de guarda define quando uma ação pode ou não ser executada.

Passo 4: O elemento decisão é mapeado como transições disparadas por eventos locais ou sincronizantes. As condições de guarda existentes nesse elemento são tratadas utilizando-se um autômato que possui dois estados denominados T e F respectivamente e possuem duas transições disparadas por eventos locais. As transições resultantes da conversão do elemento decisão têm suas taxas de ocorrência definidas com o uso de taxas funcionais [Bar06] que determinam que o disparo da transição ocorrerá conforme o estado do autômato que representa a condição de guarda.

A Figura 4.6 traz a representação gráfica de um diagrama de atividades que possui um elemento decisão em UML ao passo que a Figura 4.7 traz a representação gráfica do modelo SAN obtido com base na Figura 4.6.

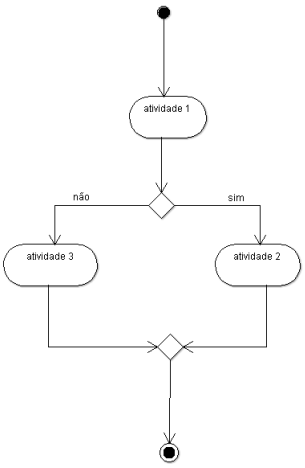
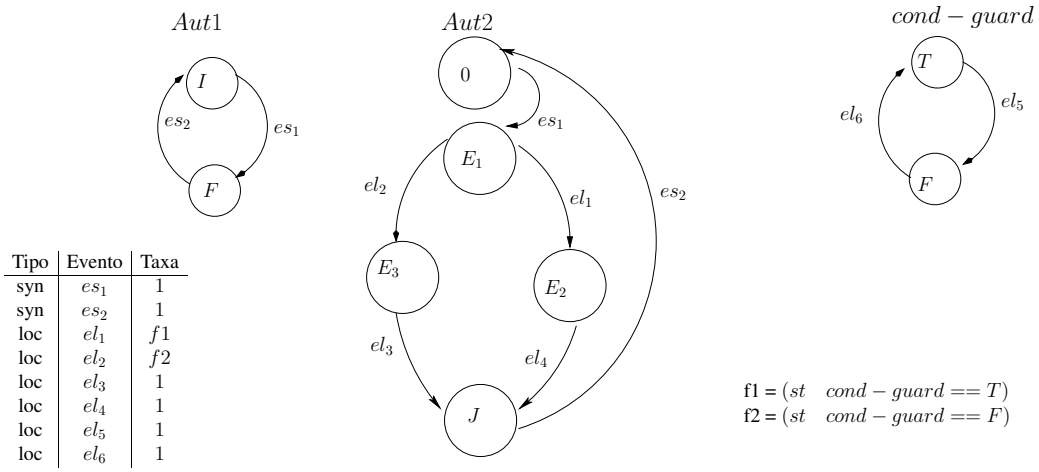


Figura 4.6: Diagrama de atividades com elemento decisão



Tipo	Evento	Taxa
syn	es_1	1
syn	es_2	1
loc	el_1	$f1$
loc	el_2	$f2$
loc	el_3	1
loc	el_4	1
loc	el_5	1
loc	el_6	1

Figura 4.7: Modelo SAN obtido na conversão da Figura 4.6

O modelo SAN obtido possui três autômatos definidos como $Aut1$, $Aut2$ e $cond - guard$, sendo que o primeiro autômato representa a execução do passo 1 descrito anteriormente e possui dois estados denominados I e F com duas transições disparadas pelos eventos sincronizantes es_1 e es_2, o segundo e o terceiro autômatos representam a execução do passo 4.

O autômato $Aut2$ possui 3 estados denominados 0 que possui uma transição disparada pelo evento sincronizante es_1, que representa a não execução de qualquer atividade, o estado E_1 que representa a atividade 1 e que possui uma transição disparada pelo evento local el_1 e uma transição disparada pelo evento el_2 que levam aos estados E_2 e E_3 respectivamente. Para esses eventos foi definida uma taxa de ocorrência da transição, através de uma taxa funcional que determina que essas transições só poderão ocorrer se o autômato $cond - guard$ estiver no estado T para haja a transição disparada pelo evento el_1 ou no estado F para o disparo do evento el_2.

Os estados E_2 e E_3 possuem uma transição cada disparadas pelos eventos locais el_4 e el_3 respectivamente, que levam ao ao estado J que representa o final da execução de todas as atividades e que possui uma transição disparada pelo evento sincronizante es_2 que leva ao estado 0. O autômato $cond - guard$ possui dois estados denominados T que possui uma transição disparada pelo evento el_5 que leva ao estado F que também possui uma transição disparada pelo evento el_6 que leva ao estado T. Esse autômato é utilizado para a avaliação da condição de guarda existente no diagrama de atividades, tal avaliação é feita com o uso da taxa funcional descrita anteriormente.

O estado J existente nos autômatos $Aut2$ e $Aut3$ representam o elemento *join* que será descrito posteriormente.

A Figura 4.8 traz a representação gráfica da cadeia de Markov equivalente ao modelo SAN resultante da execução do passo 4.

4.2.5 Conversão do elemento *fork*

Diagramas de atividades da UML podem representar paralelismos, ou seja, duas ou mais atividades podem ser executadas ao mesmo tempo [FoS08], tal ação é sempre representada por um elemento *fork*, o qual possui um fluxo de entrada e dois ou mais fluxos de saída. Assim, na aplicação

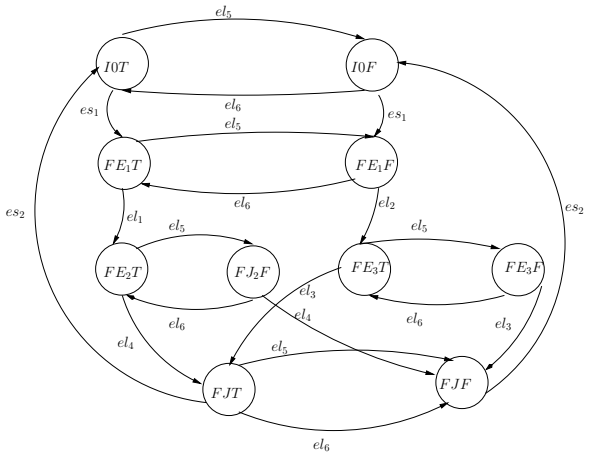


Figura 4.8: Cadeia de Markov equivalente ao modelo SAN da Figura 4.7

do método aqui proposto, deve-se converter o diagrama de atividades considerando-se cada fluxo de saída do elemento *fork*, ou seja, cada execução como um autômato, relacionando-os através de eventos sincronizantes de acordo com cada diagrama.

Passo 5: O elemento *fork* é mapeado como sendo uma ou mais transições disparadas em autômatos que representam as atividades executadas, uma vez que cada execução das atividades paralelas do diagrama é mapeada como sendo um autômato onde cada estado componente desse autômato representa uma atividade do diagrama.

A Figura 4.9 traz a representação gráfica de um diagrama de atividades que possui um elemento *fork*.

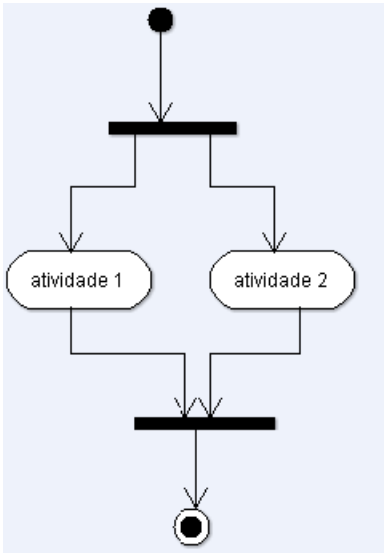


Figura 4.9: Diagrama de atividades com um elemento *fork*

A Figura 4.10 traz a representação gráfica do modelo SAN obtido com base na Figura 4.9.

Tipo	Evento	Taxa
syn	es_1	1
syn	es_2	1
loc	el_1	1
loc	el_2	1

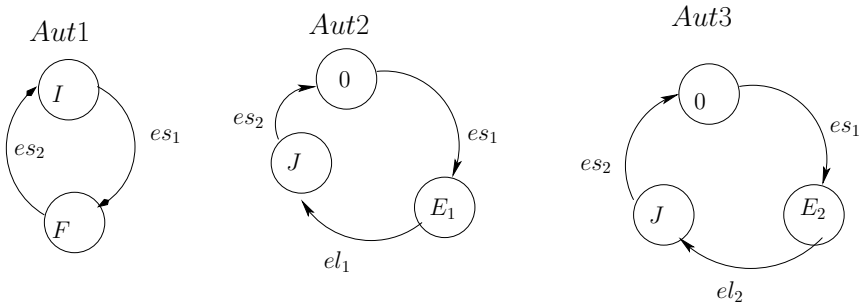


Figura 4.10: Modelo SAN obtido na conversão da Figura 4.9

O modelo SAN obtido possui três autômatos definidos como $Aut1$, $Aut2$ e $Aut3$, sendo que o primeiro autômato representa a execução do passo 1 descrito anteriormente e possui dois estados denominados I e F com duas transições disparadas pelos eventos sincronizantes es_1 ees_2, o segundo e o terceiro autômatos representam a execução do passo 5.

Tanto o autômato $Aut2$ quanto o $Aut3$ possuem a mesma estrutura, a qual possuem 3 estados denominados 0 que possui uma transição disparada pelo evento sincronizante es_1, que representa a não execução de qualquer atividade, o estado E_1 que possui uma transição disparada pelo evento local el_1 no autômato $Aut2$ e el_2 no autômato $Aut3$ e que representa a atividade 2, o estado J que possui uma transição disparada pelo evento es_2. O estado J representa o elemento *join* que será descrito posteriormente.

De acordo com Jacobson, Booch and Rumbaugh em [BRJ99], a utilização do elemento *fork*, requer a utilização do elemento *join*, uma vez que esse elemento é o responsável pela união dos fluxos gerados. A conversão do elemento *join* é detalhada no passo seguinte. Cabe ressaltar que devido a essa exigência da UML as Figuras 4.9 e 4.12 utilizadas para demonstrar a representação gráfica dos elementos *fork* e *join* são as mesmas.

A Figura 4.11 traz a representação gráfica da cadeia de Markov equivalente ao modelo SAN resultante da execução do passo 5.

4.2.6 Conversão do elemento *join*

Quando o diagrama de atividades representa um comportamento paralelo definido por um elemento *fork*, se faz necessário uma sincronização dos fluxos gerados por esse elemento. Essa sincronização ou junção é executada pelo elemento *join* que além de desempenhar a função anteriormente citada determina que a transição seguinte será efetuada somente quando todos os estados nas transições de entrada tenham completado suas atividades [Bel09].

Passo 6: O elemento $join$ é mapeado como sendo um estado nomeado como j_i, com i variando de $1..n$, em cada um dos autômatos que representam as atividades paralelas executadas pelo diagrama. O estado j_i é associado a quantos estados gerados de acordo com a quantidade de atividades existentes no diagrama. Tal mapeamento visa garantir a preservação de uma das características desse elemento, a qual determina que a transição que leva ao estado final do diagrama só pode ocorrer quando todas as atividades paralelas tiverem sido executadas.

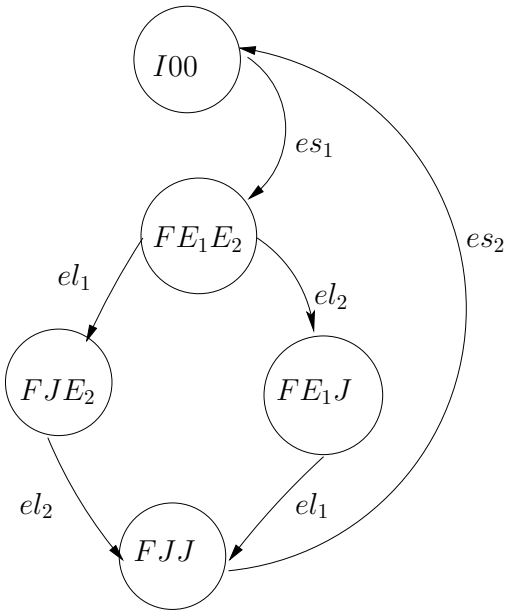


Figura 4.11: Cadeia de Markov equivalente ao modelo SAN da Figura 4.10

Assim é possível atribuir ao estado que representa esse elemento, uma transição disparada por um evento sincronizante. Os estados gerados nesse passo são nomeados como J_i com i variando de $1...n$. A forma de associação ao diagrama de caso de uso, citada no passo 1 também pode ser aplicada quando da execução desse passo.

A Figura 4.12 traz a representação gráfica de um diagrama de atividades que possui um elemento $join$, enquanto a Figura 4.13 traz a representação gráfica do modelo SAN obtido com base na Figura 4.12.

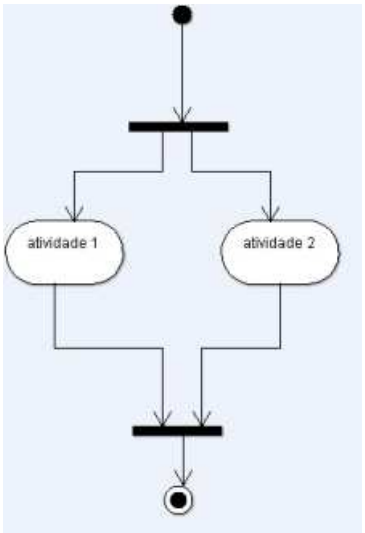


Figura 4.12: Diagrama de atividades com um elemento $join$

O modelo SAN obtido possui três autômatos definidos como $Aut1$, $Aut2$ e $Aut3$, sendo que o primeiro autômato representa a execução do passo 1 descrito anteriormente e possui dois esta-

Tipo	Evento	Taxa
syn	es_1	1
syn	es_2	1
loc	el_1	1
loc	el_2	1

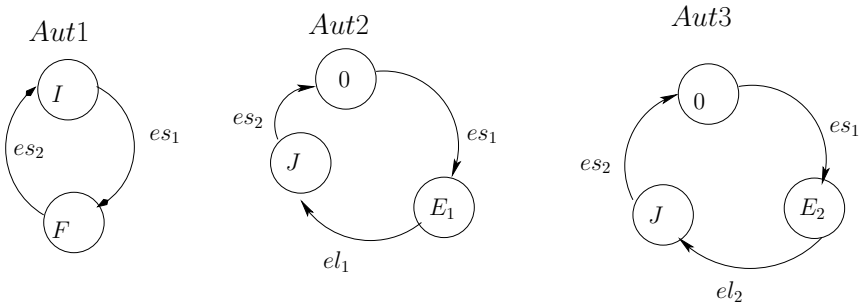


Figura 4.13: Modelo SAN obtido na conversão da Figura 4.12

dos denominados I e F com duas transições disparadas pelos eventos sincronizantes es_1 e es_2, o segundo e o terceiro autômatos representam a execução do passo 6.

Tanto o autômato $Aut2$ quanto o $Aut3$ possuem a mesma estrutura, a qual possuem três estados denominados: 0 que possui uma transição disparada pelo evento sincronizante es_1, que representa a não execução de qualquer atividade, o estado E_1 que possui uma transição disparada pelo evento local el_1 no autômato $Aut2$ e el_2 no autômato $Aut3$ e que representa a atividade 2, o estado J que possui uma transição disparada pelo evento sincronizante es_2, que leva ao estado 0.

A criação dos estados J visa garantir a semântica existente no diagrama de atividades da UML que determina que quando o elemento *join* é utilizado, podem ocorrer situações onde as atividades modeladas são finalizadas em instantes diferentes, o que só habilita a transição para o elemento final quando todas as atividades se encerram e tem seu fluxo direcionado ao elemento *join*.

A Figura 4.14 traz a representação gráfica da cadeia de Markov equivalente ao modelo SAN resultante da execução do passo 6.

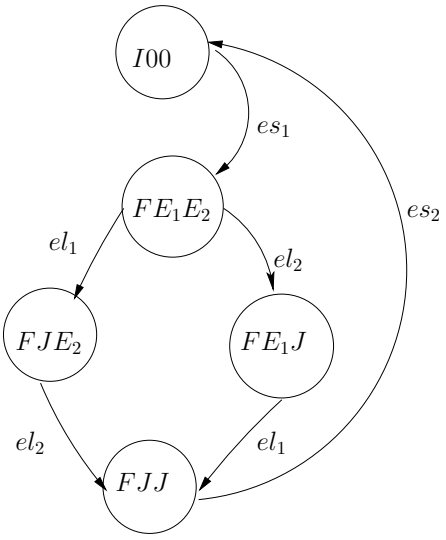


Figura 4.14: Cadeia de Markov equivalente ao modelo SAN da Figura 4.13

4.2.7 Conversão do elemento *merge*

Em alguns casos, o fluxo processual de um caminho decisão pode ligar de volta para outro caminho decisão [Bel09]. Nestes casos, podemos conectar dois ou mais caminhos de ação juntamente com o ícone do mesmo diamante com vários caminhos apontando para ele, mas com apenas uma linha de transição que vem de fora. Isso não indica um ponto de decisão, mas sim uma forma de unir diferentes fluxos, essa característica é modelada por um elemento denominado *merge*.

Passo 7: O elemento *merge*, assim como o elemento decisão, é mapeado como transições disparadas por eventos locais ou sincronizantes conforme a utilização do elemento. Cabe ressaltar que diferentemente do elemento decisão, este elemento não possui nunca uma condição de guarda associada a ele, pois o mesmo tem apenas a função de unir diferentes fluxos.

A Figura 4.15 traz a representação gráfica de um diagrama de atividades que possui um elemento *merge*, ao passo que a Figura 4.16 traz a representação gráfica do modelo SAN obtido com base na Figura 4.15.

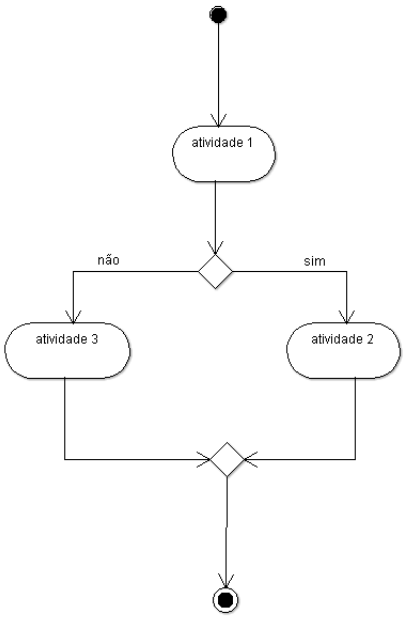
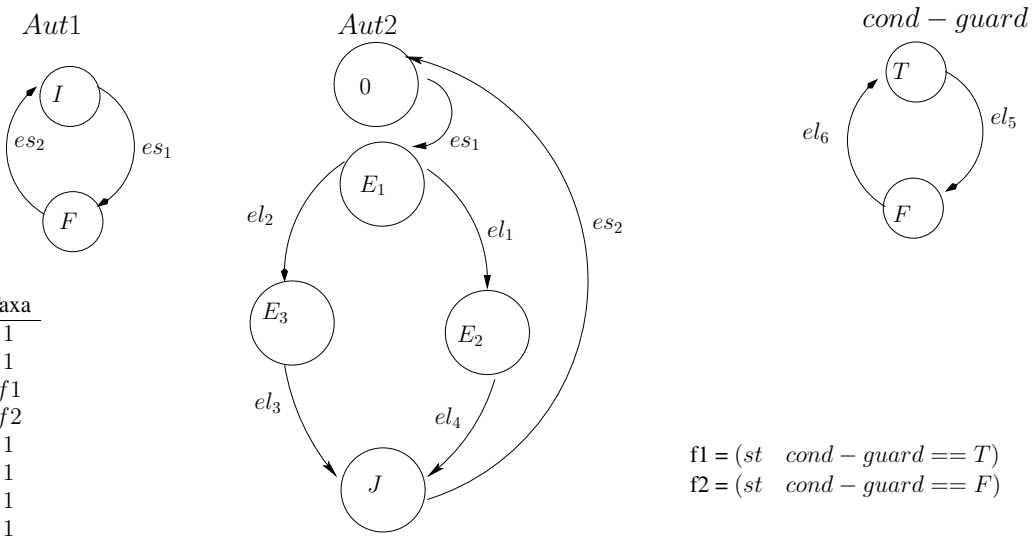


Figura 4.15: Diagrama de atividade com um elemento *merge*

O modelo SAN obtido possui três autômatos definidos como $Aut1$, $Aut2$ e $Aut3$, sendo que o primeiro autômato representa a execução do passo 1 descrito anteriormente e possui dois estados denominados I e F com duas transições disparadas pelos eventos sincronizantes es_1 ees_2, o segundo e o terceiro autômatos representam a execução do passo 7. No exemplo apresentado o diagrama de atividades possui um elemento *merge* que tem a função de unir os fluxos vindos das



Tipo	Evento	Taxa
syn	es_1	1
syn	es_2	1
loc	el_1	$f1$
loc	el_2	$f2$
loc	el_3	1
loc	el_4	1
loc	el_5	1
loc	el_6	1

Figura 4.16: Modelo SAN obtido na conversão da Figura 4.15

atividades 1 e 2. Na conversão para o modelo SAN, apresentada aqui, o elemento *merge* foi convertido na transição disparada pelo evento que levam do estado E_2, para o estado J disparada pelo evento local el_1 e do estado E_3 para o estado J disparada pelo evento el_2. Os demais elementos da SAN apresentada neste passo seguem a descrição apresentada anteriormente no passo 4.

A Figura 4.16 traz a representação gráfica da cadeia de Markov equivalente ao modelo SAN resultante da execução do passo 7.

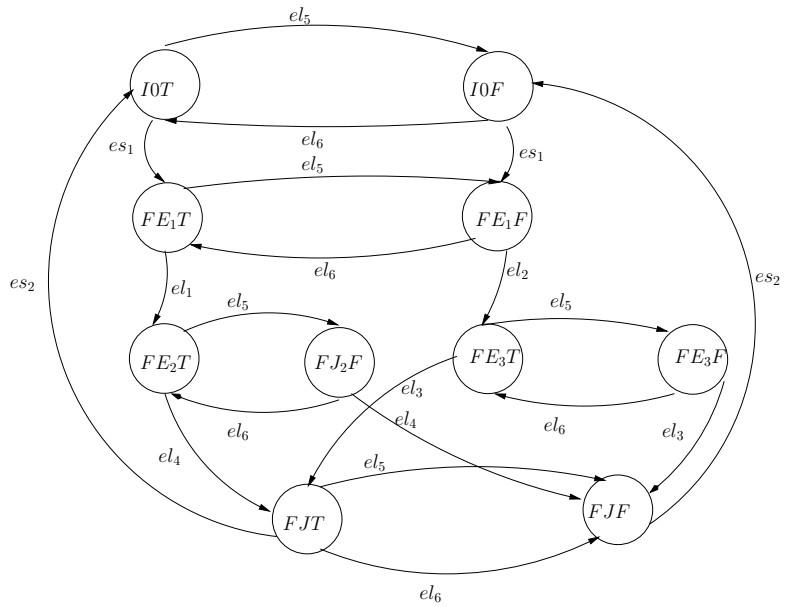


Figura 4.17: Cadeia de Markov equivalente ao modelo SAN da Figura 4.16

4.2.8 Elementos não convertidos para o modelo SAN

Nesta seção, apresentamos alguns elementos que compõem um diagrama de atividades, mas que não foram considerados no momento da conversão de UML para um modelo SAN. Tal decisão se deve ao fato de que o uso dos elementos a seguir descritos não altera as características do diagrama e também não traz nenhuma informação adicional sobre o comportamento do sistema que está sendo modelado. As definições a seguir se baseiam no trabalho de Bell [Bel09].

- ***Swimlanes* (raias):** Em diagramas de atividades, muitas vezes, é útil para o modelo da atividade processual fluxo de controle entre os objetos (pessoas, organizações ou outros entidades responsáveis) que realmente executar a ação. Para isso, é possível adicionar um elemento denominado *swimlanes* (raias) para o diagrama de atividades. Cabe ressaltar que, embora o uso de raias melhore a clareza de um diagrama de atividades, todas as regras que regem a criação dos diagramas de atividades devem ser obedecidas, assim o uso ou não desse elemento não altera as características nem a função do diagrama.

- **Objeto:** O objeto em símbolo do estado é o retângulo que possui um texto sublinhado que representa a informação sobre aquele objeto. A inclusão de um objeto não altera a forma como o diagrama de atividades é lido, apenas fornece informações adicionais sobre a execução da atividade a qual o objeto está relacionado.

- **Fluxo de objeto:** Um fluxo de objeto é similar a uma linha de transição, mas é mostrado como uma linha tracejada, em vez de uma sólida. Um fluxo de objeto linha está ligado a um objeto no estado símbolo, e uma outra linha de fluxo objeto conecta o objeto em símbolo do estado para a próxima ação.

No capítulo seguinte serão demonstrados dois exemplos de conversão de um diagrama de atividades para um modelo SAN, seguindo os passos anteriormente descritos. O capítulo apresenta ainda o processo de geração de casos de teste baseado no modelo SAN obtido.

5. Conversão de diagrama de atividades UML para SAN

Neste capítulo serão apresentados dois exemplos de conversão de diagramas de atividades UML (*Unified Modeling Language*) para SAN *(Stochastic Automata Networks)*. O capítulo traz ainda a descrição do processo de geração de casos de teste de software baseado no modelo SAN obtido.

5.1 Exemplos de conversão de diagrama de atividades UML para SAN

O primeiro exemplo aqui apresentado foi obtido do livro de Fowler e Scott [FoS08] e é representado pela Figura 5.1:

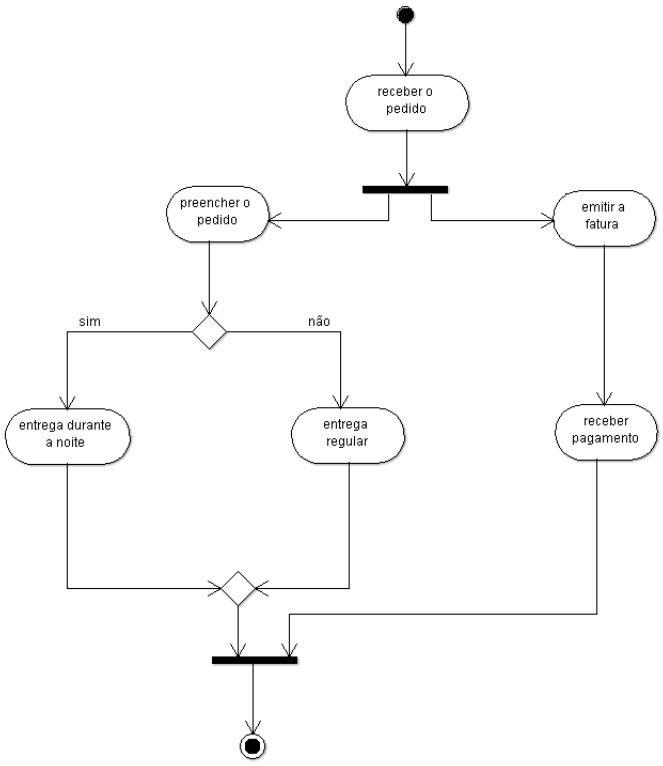


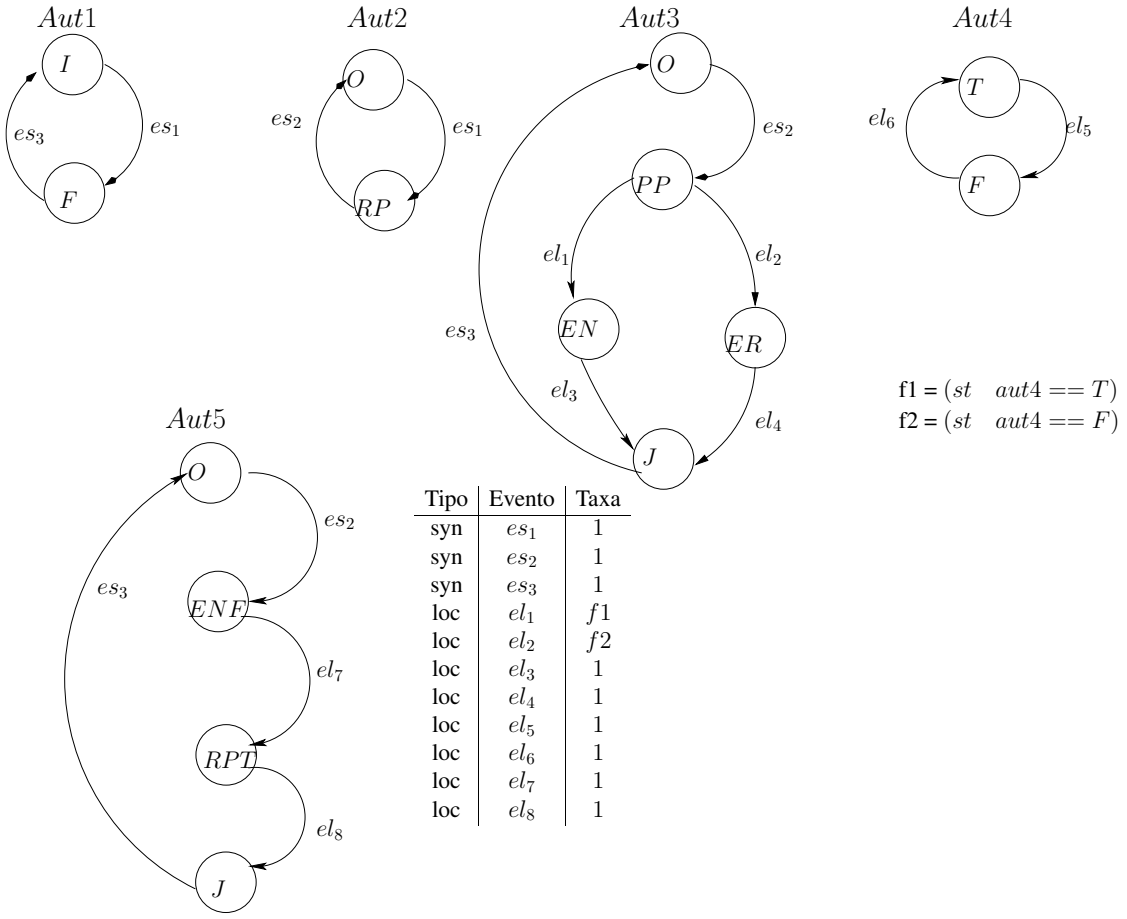
Figura 5.1: Diagrama de atividades a ser convertido para um modelo SAN

O diagrama apresentado aqui descreve um processo de venda de produtos que tem início com o *recebimento do pedido*. Na sequência duas atividades são executadas paralelamente, tais atividades são *preencher pedido* e *emitir nota fiscal*.

Após a atividade *preencher pedido*, é modelada a escolha entre duas atividades que podem vir a ser executadas, são elas, entrega durante a noite que pressupõem urgência na execução da tarefa, ou *entrega regular*.

O processo de emissão de nota fiscal é caracterizado por duas atividades, *emitir nota fiscal* e *receber pagamento*. Após a execução dessas atividades, o diagrama remete a um elemento $join$ que simboliza o aguardo do término de todas as atividades executadas em paralelo.

O método de conversão consiste em mapear os elementos de um diagrama de atividades para um modelo SAN. A Figura 5.2 apresenta o modelo SAN obtido no processo de conversão.



Tipo	Evento	Taxa
syn	es_1	1
syn	es_2	1
syn	es_3	1
loc	el_1	$f1$
loc	el_2	$f2$
loc	el_3	1
loc	el_4	1
loc	el_5	1
loc	el_6	1
loc	el_7	1
loc	el_8	1

Figura 5.2: SAN resultante da conversão da Figura 5.1

Com base no diagrama apresentado na Figura 5.1, o elemento estado inicial e final do diagrama de atividades é convertido para os estados *I* e *F* do autômato $A^{(1)}$. Para esse autômato foram definidas duas transições, disparadas pelos eventos sincronizantes es_1 e es_3 respectivamente.

O autômato $A^{(2)}$ possui dois estados, o estado 0 representa a espera pela execução da primeira atividade do diagrama, esse estado possui uma transição que leva ao estado RP disparada pelo evento sincronizante es_1. O outro estado do autômato é denominado RP e representa a atividade *Receber Pedido* com uma transição que leva ao estado 0 e é disparada pelo evento sincronizante es_2. Essa transição representa o elemento *fork* que possui duas atividades paralelas que originam os autômatos $A^{(3)}$ e $A^{(5)}$.

O autômato $A^{(3)}$ possui cinco estados sendo um denominado 0 representa a espera pela execução da atividade *Preencher Pedido*, esse estado possui uma transição que leva ao estado PP e é disparada pelo evento sincronizante es_2. Esse estado possui duas transições disparadas pelos eventos

locais *el*$_1$ e *el*$_2$ repectivamente, que levam ao estado *Entrega Durante a Noite*, denominado *EN* ou para o estado *Entrega Regular*, denominado *ER*.

O estado EN possui uma transição disparada pelo evento local el_3 que leva ao estado J e o estado ER também possui uma transição disparada pelo evento local el_4 que leva ao estado J. O Estado J representa o elemento *join* e possui uma transição disparada pelo evento sincronizante es_3 que leva ao estado 0.

A condição de guarda existente nesse diagrama é tratada com o uso do autômato $A^{(4)}$ que possui dois estados definidos como T que possui uma transição disparada pelo evento el_5 que leva a estado F que por sua vez possui uma transição disparada pelo evento el_6 e que leva ao estado T.

Para a validação da condição de guarda, foi atribuída uma taxa funcional aos eventos el_1 e el_2 ambas no autômato $A^{(3)}$. A taxa funcional do evento el_1 determina que a transição do estado PP para o estado EN só poderá ocorrer quando o autômato $A^{(4)}$ se encontrar no estado T, já a taxa funcional do evento el_2 determina que a transição do estado PP para o estado ER só poderá ocorrer quando o autômato $A^{(4)}$ se encontrar no estado F.

O autômato $A^{(5)}$ modela a segunda tarefa paralela representada pelo diagrama de atividades e possui os estados 0 representa a espera pela execução da atividade *Emitir Fatura*, esse estado possui uma transição que leva ao estado ENF e é disparada pelo evento sincronizante es_2. O estado *ENF* representa a atividade *Emitir Fatura* o qual possui uma transição disparada pelo evento local *el*$_7$ que leva ao estado *RPT* que representa a atividade *Recebe Pagamento* e que possui uma transição disparada pelo evento local *el*$_8$ que leva ao estado *J* que representa o elemento *join*, assim como no autômato $A^{(3)}$ e que possui uma transição disparada pelo evento es_3 e leva ao estado 0.

5.2 Geração dos casos de teste

Nesta seção apresentamos uma descrição do processo de geração de casos de teste, baseado em modelos Markovianos, e o algoritmo desenvolvido com base na descrição feita por Copstein et al [COR04] para a geração dos mesmos. O algoritmo aqui proposto tem por finalidade gerar casos de teste a partir de um modelo SAN.

Segundo Pezzé e Young em [PeY08] os testes têm por objetivo detectar o maior número possível de erros, e para se alcançar tal objetivo, deve-se executar o programa de maneira a forçar certos limites para que os erros possam ser encontrados.

Em alguns programas, o número de caminhos possíveis a serem executados aumenta de maneira a tornar o teste inviável e exaustivo. Assim, é necessário definir os casos de teste de maneira que os caminhos possíveis possam ser testados.

A geração de casos de teste aqui apresentada baseia-se nos trabalhos de Copstein et al [COR04], Bertolini et al [BFF$^+$04] e Barros [Bar06], e tem como objetivo identificar as probabilidades dos estados atingíveis do modelo, ou seja, a partir de um estado denominado inicial pode se calcular a probabilidade de se chegar a outro estado, definindo-se assim um passo, ou trajeto, com relação ao funcionamento do sistema.

Na proposta, considera-se $TS = T1, T2, ..., TM$, como um *test suite* composto de M casos de

teste, onde T_i é o caso de teste de índice i. Um caso de teste é uma sequência composta de N transições de estados do modelo, ou seja, é uma sequência de uso do software modelado.

Segundo Bertolini et al e Barros [BFF+04, Bar06], uma 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 $i - esimo + 1$ é dada por:

$$\sigma_i = \frac{\pi_{(i+1)}}{\sum_{\forall k \ll i} \pi_k}$$

De acordo com a fórmula acima, π_κ é a probabilidade do *i-esimo* estado da solução estacionária (ou transiente) de um modelo, e $k \ll i$ significa a possibilidade de ir para todos os $k - esimos$ estados sucessores do *i-esimo* estado.

A probabilidade de um caso de teste é o produtório de todas as probabilidades dos passos σ_i $(i = 1..N)$ do caso de teste, o qual compreende uma trajetória T. Assim, a probabilidade de um caso de teste Pt é dada por [BFF+04, Bar06]:

$$Pt = \prod_{i=1}^{N} \sigma_i$$

Casos de teste (ou trajetórias) são basicamente representados por uma sequência de estado-transição-estado (passo), partindo do estado inicial de um passo é possível encontrar seus estados finais através da execução dos eventos, dito isto, conclui-se que casos de teste são caminhadas aleatórias, através dos estados atingíveis do modelo [Whi97].

Cada vez que um estado ou arco é percorrido, um caso de teste é executado. Assim em um modelo estocástico, os estados atingíveis são considerados casos de teste executados com sucesso.

A geração de casos de teste é um processo que tem como finalidade percorrer caminhos possíveis até chegar a um estado final, partindo de um estado inicial. Esse percurso é traçado de forma aleatória, pois existem estados que podem possuir mais de uma transição para chegar a outro estado, e entre essas transições, uma delas deverá ser sorteada. Um caso de teste é concluído quando ocorrer uma transição para um estado final.

Antes de iniciar a geração de casos de teste, é feita uma procura por estados iniciais em todos os autômatos da rede. Os autômatos que não possuírem estados iniciais terão todos os seus estados como possíveis estados iniciais.

Após essa etapa, é escolhido um estado global do modelo SAN, a ser denominado estado inicial da rede. Esse estado global será selecionado aleatoriamente dentro de um grupo de estados globais, que são formados por estados iniciais e por possíveis estados iniciais dos autômatos.

A partir do estado inicial da rede, é iniciado o processo de geração de casos de teste, em que o primeiro *step* (passagem de um estado da rede a outro) de cada caso de teste ocorre em função

de um evento *start* no estado inicial da rede. O evento do tipo start é um evento criado para setar o primeiro *step* da rede [BFF+04].

Os *steps* seguintes são realizados a partir da escolha de eventos candidatos com base no atual estado da rede. É escolhido um evento candidato para que ocorra a mudança para outro estado global. Os passos são realizados até ocorrer o evento do tipo *quit*, criado para finalizar o caso de teste [COR04].

5.2.1 Algoritmo proposto para a geração dos casos de teste

Com base na descrição feita na subseção anterior, é apresentado aqui o Algoritmo 5.1 desenvolvido para a geração de casos de teste. O algoritmo foi desenvolvido com base no trabalho de Copstein et al [COR04]:

Algoritmo 5.1 Algoritmo para a geração de casos de teste de software

```
1: Inserir ESTADOFINAL (QUIT)
2: while CASOTESTE ≤ ESTADOFINAL do
3:    Enumere o ESTADOGLOBAL da SAN
4:    Escolha aleatória do ESTADOINICIAL (START)
5:    Adicione o ESTADOINICIAL para CASOTESTE
6:    while EVENTOATUAL ≠ EVENTOFINAL do
7:       Enumere o evento CANDIDATO (QT) a sair
8:       Calcular a probabilidade relativa do CANDIDATO
9:       Selecione CANDIDATO (QT) a sair (usando probabilidade computacional)
10:      Adicione o EVENTO selecionado ao CASOTESTE
11:      if EVENTO = LOCAL then
12:         Compute o próximo estado do autômato correspondente
13:         Atualize o estado global com o estado do autômato local
14:      else
15:         Encontre o autômato afetado pelo evento
16:         Compute o próximo estado de cada autômato afetado pelo evento
17:         Atualize o estado global com o estado do autômato afetado
18:         Adicione o Estado Global atualizado para o CASOTESTE
19:      end if
20:      Adicione o CASOTESTE para a lista de CASOTESTE
21:   end while
22: end while
23: INI = 1
24: while INI ≤ CONTADOR do
25:    imprime CASOTESTE
26:    INI = INI+1
27: end while
```

Inicialmente há a inserção do estado final onde é definido o primeiro estado do modelo SAN para dar início a geração dos casos de teste. Logo após é feita a comparação entre os casos de teste gerado e o estado final criando-se um *loop*.

Posteriormente o estado global do modelo é enumerado e o estado inicial é escolhido de maneira aleatória para geração do caso de teste. É criado um *loop* que compara se o estado inicial selecionado é diferente do estado final, caso não seja são definido os estados possíveis para a transição a partir do estado inicial e é calculada a probabilidade de cada uma das transições e essa probabilidade é adiciona ao caso de teste.

Caso o evento que dispara a transição seja um evento local é feita mudança de estado no autômato e o estado global do modelo é alterado. Caso contrario, é feita a busca pelos autômatos do modelo que são alterados e o estado global do modelo é atualizado e o caso de teste é adicionado a lista de casos de teste gerados.

Para finalizar é criado um contador para armazenar os casos de teste gerados que posteriormente serão impressos.

5.2.2 Resultados obtidos

Nesta seção apresentamos os resultados obtidos com a aplicação do algoritmo apresentado anteriormente e com o uso da ferramenta PEPS [BMF+03].

Segundo Whittaker em [Whi97] a aplicação da sequência de testes pode ser automática ou manual dependendo do ambiente de testes e do suporte dado a automação dos testes, assim para o modelo SAN gerada (Figura 5.2), inicialmente apresentamos na Tabela 5.2 as probabilidade do modelo se encontrar em um dos estados atingíveis, essas probabilidades para a geração dos casos de teste a soma dos valores apresentado na Tabela 5.2 é igual a 1.

Cabe ressaltar que os resultados obtidos são baixos devido aos valores atribuídos as taxas de ocorrência dos eventos que foram todos definidos com valor 1, pois o objetivo desse trabalho é apenas demonstrar a conversão de diagramas de atividades da UML para SAN e a posterior geração de casos de teste é viável.

A Tabela 5.1 demonstra a equivalência entre a nomenclatura utilizada pela ferramenta PEPS e a atribuída ao modelo gerado. Os elementos apresentados na tabela variam de acordo com o modelo a ser descrito.

Nome dos estados no PEPS	Nomenclatura usada no modelo SAN
0	Representa o estado inicial dos autômatos $A^{(1)}$, $A^{(2)}$, $A^{(3)}$, $A^{(4)}$ e $A^{(5)}$ podendo ser nomeado como I,0 ou T conforme o autômato
1	Representa o segundo estado dos autômatos $A^{(1)}$, $A^{(2)}$, $A^{(3)}$, $A^{(4)}$ e $A^{(5)}$ podendo ser nomeado como F,RP,PP,F, ou ENF conforme o autômato
2	Representa o teceiro estado dos autômatos $A^{(3)}$ e $A^{(5)}$ podendo ser nomeado como EN ou RPT conforme o autômato
3	Representa o quarto estado dos autômatos $A^{(3)}$ e $A^{(5)}$ podendo ser nomeado como EN ou RPT conforme o autômato nomeado como ER ou J conforme o autômato
4	Representa o quinto estado do autômato $A^{(3)}$ podendo ser nomeado também como J

Tabela 5.1: Descrição dos estados do modelo gerado para o exemplo 1 - Figura 5.1

A nomeclatura usada para os estados atingíveis obtidos com a função de atingibilidade e descrita a seguir seguem a definição feita na Tabela 5.1:

0 0 0 0 0: Esse estado é atingível pois representa a parte inicial do diagrama de atividades onde nenhuma atividade está sendo executada.

0 0 0 1 0: Esse estado é atingível, pois assim como o estado anterior, representa que nenhuma atividade está sendo executada, porém, como o autômato $Aut4$ possui apenas eventos locais, o mesmo pode mudar seu estado sem depender dos demais autômatos do modelo.

1 0 1 0 1: Esse estado é atingível, pois representa a execução do elemento *fork* que por sua vez dispara as atividades paralelas *preencher pedido* e *emitir fatura*.

1 0 1 0 2: Esse estado é atingível, pois representa a execução da atividade *preencher pedido*, uma das execuções paralelas modeladas pelo diagrama e a execução da atividade *receber pagamento* que faz parte da segunda execução paralela executada.

1 0 1 0 3: Esse estado é atingível, pois representa a execução da atividade *preencher pedido*, e faz a avaliação da condição de guarda existente no diagrama. A combinação desses estados atingíveis define que a entrega do pedido deve ser feita de forma regular.

1 0 1 1 1: Esse estado é atingível e pode ser descrito como o estado 10101 porém o autômato $aut4$, que trata a condição de guarda, estará no estado F e não mais no estado T.

1 0 1 1 2: Esse estado é atingível, pois representa a execução da atividade *preencher pedido*, uma das execuções paralelas modeladas pelo diagrama e a execução da atividade *receber pagamento*, que faz parte da segunda execução paralela executada.

Estado Global	Probabilidade
0 0 0 0 0	0,086956521
0 0 0 1 0	0,086956521
1 0 1 0 1	0,043478261
1 0 1 0 2	0,021739130
1 0 1 0 3	0,021739131
1 0 1 1 1	0,043478261
1 0 1 1 2	0,021739130
1 0 1 1 3	0,021739131
1 0 2 0 1	0,016304348
1 0 2 0 2	0,014945652
1 0 2 0 3	0,026721015
1 0 2 1 1	0,005434783
1 0 2 1 2	0,006793478
1 0 2 1 3	0,016757246
1 0 3 0 1	0,005434783
1 0 3 0 2	0,006793478
1 0 3 0 3	0,016757246
1 0 3 1 1	0,016304348
1 0 3 1 2	0,014945652
1 0 3 1 3	0,026721015
1 0 4 0 1	0,021739131
1 0 4 0 2	0,043478261
1 0 4 0 3	0,086956522
1 0 4 1 1	0,021739131
1 0 4 1 2	0,043478261
1 0 4 1 3	0,086956522
1 1 0 0 0	0,086956521
1 1 0 1 0	0,086956521

Tabela 5.2: Probabilidade de estados atingiveis do modelo gerado para o exemplo 1 - Figura 5.1

1 0 1 1 3: Esse estado é atingível, pois representa a execução da atividade *preencher pedido*, uma das execuções paralelas modeladas pelo diagrama e o final da execução da atividade *receber pagamento*, que faz parte da segunda execução paralela executada. Nesse momento, o diagrama de atividades dispara uma transição que remete ao elemento *fork*, onde a execução da segunda execução paralela aguarda até que a primeira termine.

1 0 2 0 1: Esse estado é atingível, pois representa a execução da atividade *entrega durante a noite*, uma das execuções paralelas modeladas pelo diagrama e a execução da atividade emitir fatura que faz parte da segunda execução paralela executada.

1 0 2 0 2: Esse estado é atingível, pois representa a execução da atividade entrega durante a noite,

uma das execuções paralelas modeladas pelo diagrama e a execução da atividade *receber pagamento*, que faz parte da segunda execução paralela executada.

1 0 2 0 3: Esse estado é atingível, pois representa a execução da atividade *entrega durante a noite*, uma das execuções paralelas modeladas pelo diagrama e o final da execução da atividade *receber pagamento*, que faz parte da segunda execução paralela executada. Nesse momento, o diagrama de atividades dispara uma transição que remete ao elemento *fork*, onde a execução da segunda execução paralela, aguarda até que a primeira termine.

1 0 2 1 1: Esse estado é atingível, e pode ser descrito como estado 10201, porém, o autômato $Aut4$, que trata a condição de guarda estará no estado F e não mais no estado T.

1 0 2 1 2: Esse estado é atingível, e pode ser descrito como estado 10202, porém, o autômato $Aut4$, que trata a condição de guarda estará no estado F e não mais no estado T.

1 0 2 1 3: Esse estado é atingível, e pode ser descrito como estado 10203, porém o autômato $Aut4$, que trata a condição de guarda estará no estado F e não mais no estado T.

1 0 3 0 1: Esse estado é atingível, pois representa a execução da atividade *entrega regular*, uma das execuções paralelas modeladas pelo diagrama e a execução da atividade *emitir fatura*, que faz parte da segunda execução paralela executada.

1 0 3 0 2: Esse estado é atingível, pois representa a execução da atividade *entrega regular*, uma das execuções paralelas modeladas pelo diagrama e a execução da atividade receber pagamento, que faz parte da segunda execução paralela executada.

1 0 3 0 3: Esse estado é atingível, pois representa a execução da atividade entrega regular, uma das execuções paralelas modeladas pelo diagrama e o final da execução da atividade *receber pagamento*, que faz parte da segunda execução paralela executada. Nesse momento, o diagrama de atividades dispara uma transição que remete ao elemento *fork*, onde a execução, segunda execução paralela aguarda o término da primeira.

1 0 3 1 1: Esse estado é atingível, e pode ser descrito como estado 10301, porém, o autômato $Aut4$, que trata a condição de guarda estará no estado F e não mais no estado T.

1 0 3 1 2: Esse estado é atingível, e pode ser descrito como estado 10302, porém, o autômato $Aut4$, que trata a condição de guarda estará no estado F e não mais no estado T.,

1 0 3 1 3: Esse estado é atingível, e pode ser descrito como estado 10303, porém, o autômato $Aut4$, que trata a condição de guarda estará no estado F e não mais no estado T.

1 0 4 0 1: Esse estado é atingível, pois representa o final da execução da atividade *entrega durante a noite*, uma das execuções paralelas modeladas pelo diagrama o qual remete ao elemento *join*, que simboliza o aguardo do término das atividades que compõem a outra execução paralela, que nessa combinação de estados, representa a execução da atividade *emitir fatura*.

1 0 4 0 2: Esse estado é atingível, pois representa o final da execução da atividade *entrega durante a noite*,uma das execuções paralelas modeladas pelo diagrama, o qual remete ao elemento *join*, que simboliza o aguardo do término das atividades que compõem a outra execução paralela, que nessa combinação de estados representa, executa a atividade *receber pagamento*.

1 0 4 0 3: Esse estado é atingível, pois representa o final da execução da atividade *entrega durante a noite*, uma das execuções paralelas modeladas pelo diagrama,o qual remete ao elemento *join*, que simboliza o aguardo do término das atividades que compõem a outra execução paralela. Nessa combinação de estados, ambas atividades paralelas foram executadas, o que habilita o disparo da transição que leva ao elemento estado final.

1 0 4 1 1: Esse estado é atingível, pois representa o final da execução da atividade *entrega regular*, uma das execuções paralelas modeladas pelo diagrama, o qual remete ao elemento *join*, que simboliza o aguardo do término das atividades que compõem a outra execução paralela, que nessa combinação de estados representa a execução da atividade *emitir fatura*.

1 0 4 1 2: Esse estado é atingível, pois representa o final da execução da atividade *entrega regular*, uma das execuções paralelas modeladas pelo diagrama, o qual remete ao elemento *join*, que simboliza o aguardo do término das atividades que compõem a outra execução paralela, que nessa combinação de estados, representa a execução da atividade *receber pagamento*.

1 0 4 1 3: Esse estado é atingível, pois representa o final da execução da atividade *entrega regular* uma das execuções paralelas modeladas pelo diagrama, o qual remete ao elemento *join*, que simboliza o aguardo do término das atividades que compõem a outra execução paralela. Nessa combinação de estados, ambas atividades paralelas foram executadas, o que habilita o disparo da transição que leva ao elemento estado final.

1 1 0 0 0: Esse estado é atingível, pois representa a execução da atividade *receber pedido*. Nessa combinação de estados, o autômato $Aut4$ está no estado F.

1 1 0 1 0: Esse estado é atingível, pois representa término das execuções paralelas modeladas pelo diagrama. Nessa combinação de estados o autômato Aut4 está no estado T.

Para a geração dos casos de teste, utiliza-se os dados apresentados na coluna estado global da Tabela 5.2 de modo que partindo-se do primeiro estado atingível do modelo, que na geração do caso de teste, é denominado como *start* (st) e atingindo-se o estado final denominado como *quit* (qt), obtêm um caso de teste. Se existirem mais que dois estados atingíveis, os estados entre o estado inicial st e o estado final qt serão denominados s que representam os passos percorridos entre o estado inicial e final, assim para o modelo gerado um dos casos de teste seria st(0 0 0 0 0) qt (0 0 0 1 0) aos demais seriam acrescidos os estados até se atingir todos os estados globais do modelo gerado. Com base nos estados atingíveis do modelo gerado obteremos as seguintes probabilidades de ocorrência para os casos de teste:

Número	Caso de teste	Probabilidade de teste (Pt)
1	st 0 0 0 0 0, qt 0 0 0 1 0	0,007561437
2	st 0 0 0 0 0, s 0 0 0 1 0 qt 1 0 1 0 1	0,003780718
3	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 qt 1 0 1 0 2	0,000945180
4	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s, 1 0 1 0 2, qt 1 0 1 0 3	0,000472590
5	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s, 1 0 1 0 2, 5, s 1 0 1 0 3, qt 1 0 1 1 1	0,000945180
6	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s, 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, qt 1 0 1 1 2	0,000945180
7	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1, s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, qt 1 0 1 1 3	0,000472590
8	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s, 1 0 1 0 2 s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3,qt 1 0 2 0 1	0,000354442
9	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s, 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, qt 1 0 2 0 2	0,000243679
10	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s, 1 0 1 0 2 s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, qt 1 0 2 0 3	0,000399363
11	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s, 1 0 1 0 2 s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, qt 1 0 2 1 1	0,000145223
12	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, qt 1 0 2 1 2	3,69211E-05
13	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, qt 1 0 2 1 3	0,000113840
14	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, qt 1 0 3 0 1	9,10720E-05

Tabela 5.3: Probabilidade de ocorrência dos casos de teste - exemplo 1 - Figura 5.1 parte I

Número	Caso de teste	Probabilidade de teste (Pt)
15	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2 s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3,s 1 0 3 0 1, qt 1 0 3 0 2	3,69211E-05
16	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, qt 1 0 3 0 3	0,000113840
17	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2 s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, qt 1 0 3 1 1	0,000273216
18	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2 s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, s 1 0 3 1 1, qt 1 0 3 1 2	0,000243679
19	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2 s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, s 1 0 3 1 1, s 1 0 3 1 2, qt 1 0 3 1 3	0,000399363
20	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2 s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, s 1 0 3 1 1, s 1 0 3 1 2, s 1 0 3 1 3, qt 1 0 4 0 1	0,000580892

Tabela 5.4: Probabilidade de ocorrência dos casos de teste - exemplo 1 - Figura 5.1 - parte II

Número	Caso de teste	Probabilidade de teste (Pt)
21	st 0 0 0 0 0 0, s 0 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 0 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, s 1 0 3 1 1, s 1 0 3 1 2, s 1 0 3 1 3, s 1 0 4 0 1, qt 1 0 4 0 2	0,000945180
22	st 0 0 0 0 0 0, s 0 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, s 1 0 3 1 1, s 1 0 3 1 2, s 1 0 3 1 3, s 1 0 4 0 1, s 1 0 4 0 2,qt 1 0 4 0 3	0,003780718
23	st 0 0 0 0 0 0, s 0 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3,s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, s 1 0 3 1 1, s 1 0 3 1 2, s 1 0 3 1 3, s 1 0 4 0 1, s 1 0 4 0 2, s 1 0 4 0 3, qt 1 0 4 1 1	0,001890359
24	st 0 0 0 0 0 0, s 0 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, s 1 0 3 1 1, s 1 0 3 1 2, s 1 0 3 1 3, s 1 0 4 0 1, s 1 0 4 0 2, s 1 0 4 0 3, s 1 0 4 1 1, qt 1 0 4 1 2	0,000945180
25	st 0 0 0 0 0 0, s 0 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, s 1 0 3 1 1, s 1 0 3 1 2, s 1 0 3 1 3, s 1 0 4 0 1, s 1 0 4 0 2, s 1 0 4 0 3, s 1 0 4 1 1 s 1 0 4 1 2, qt 1 0 4 1 3,	0,003780718

Tabela 5.5: Probabilidade de ocorrência dos casos de teste - exemplo 1 - Figura 5.1 - parte III

Número	Caso de teste	Probabilidade de teste (Pt)
26	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1, s 1 0 2 1 2, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, s 1 0 3 1 1, s 1 0 3 1 2, s 1 0 3 1 3, s 1 0 4 0 1, s 1 0 4 0 2, s 1 0 4 0 3, s 1 0 4 1 1, s 1 0 4 1 2, s 1 0 4 1 3, qt 1 1 0 0 0	0,007561437
27	st 0 0 0 0 0, s 0 0 0 1 0 s 1 0 1 0 1 s 1 0 1 0 2, s 1 0 1 0 3, s 1 0 1 1 1, s 1 0 1 1 2, s 1 0 1 1 3, s 1 0 2 0 1, s 1 0 2 0 2, s 1 0 2 0 3, s 1 0 2 1 1,s 1 0 2 1 2, s 1 0 3 1 3, s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, s 1 0 3 0 3, s 1 0 3 1 1, s 1 0 4 0 1, s 1 0 4 0 2, s 1 0 4 0 3, s 1 0 4 1 1, s 1 0 4 1 2, s 1 0 4 1 3, s 1 1 0 0 0, qt 1 1 0 1 0	0,007561437

Tabela 5.6: Probabilidade de ocorrência dos casos de teste - exemplo 1 - Figura 5.1 - Parte IV

Como forma de demonstrar, de maneira detalhada, a aplicação do método proposto neste trabalho será apresentado nesta seção o segundo exemplo de conversão de diagramas de atividades para SAN.

5.3 Um sistema de gerenciamento de festas

Com o objetivo de aplicar o método proposto nesta pesquisa, neste capítulo é apresentado um estudo de caso. Apresentamos aqui os detalhes do sistema de estudo proposto, bem como a modelagem e a transcrição dos diagramas gerados para uma estrutura equivalente em SAN. O exemplo utilizado neste trabalho foi proposto por [VaH02], assim como o diagrama de atividades apresentado na Figura 5.1, foi desenvolvido pelo professor Duncan Dubugras Alcoba Ruiz na disciplina IPN (Inteligência de Processos de Negócios), ofertada pelo PPGCC/FACIN (Programa de Pós-Graduação em Ciência da Computação da Faculdade de Informática da PUCRS).

5.3.1 Descrição do exemplo

Um grupo de estudantes quer formar uma agência que organiza festas.

O cliente indica a quantidade de dinheiro que quer gastar, o número de pessoas que a festa vai receber e a área onde a festa vai ser oferecida. Com estas informações, a agência procura por uma localização adequada e cuida do resto. A localização pode ser em um lugar fechado ou aberto. Se a localização é em um lugar fechado, um salão deve ser alugado. Caso a localização seja em um local aberto, uma tenda e um terreno devem ser arrumados, e provavelmente deve permitir que se possa fazer barulho (música). Há dois tipos de música: ao vivo ou som mecânico. A escolha é feita pela agência. A música ao vivo é preferida, mas é cara, então a maioria das festas é feita com som mecânico. O som mecânico também é escolhido se não se tem tempo para se contratar uma banda. Se som mecânico é escolhido, um sistema de som deve ser arrumado. No caso de música ao vivo, porém as coisas são mais complicadas. Primeiro, uma banda é selecionada. Então é enviada para esta banda uma carta convidando para tocar na festa. Se a banda não retorna a mensagem dentro de uma semana, uma nova banda é selecionada e o procedimento é repetido. Se existe retorno, novamente existem duas possibilidades: eles estão interessados ou eles não estão. No último caso uma nova banda é selecionada e o processo é repetido. No primeiro caso, porém, a banda não é contratada imediatamente. Primeiro a agência precisa ver e ouvir a banda para ver se eles são bons o suficiente. Porque os estudantes contratam apenas os melhores, cerca de 30% das bandas são consideradas boas o suficiente, para os outros 70%, uma nova banda é selecionada, etc. Se os estudantes não conseguem rapidamente uma banda, eles optam pelo som mecânico. É claro que as bandas que já foram selecionadas alguma vez não precisam ser novamente avaliadas, elas são contratadas imediatamente. Após cuidarem da localização e da música, eles então cuidam da comida e da bebida. No caso de uma banda, eles compram comida e bebida extra para os músicos. Para ter certeza que tudo está certo, os estudantes supervisionam a festa quando ela está começando. Após isto, a conta é enviada para o cliente. O diagrama de atividades construído pelo professor Dr Duncan Ruiz, com base na descrição acima, é apresentado na Figura 5.1:

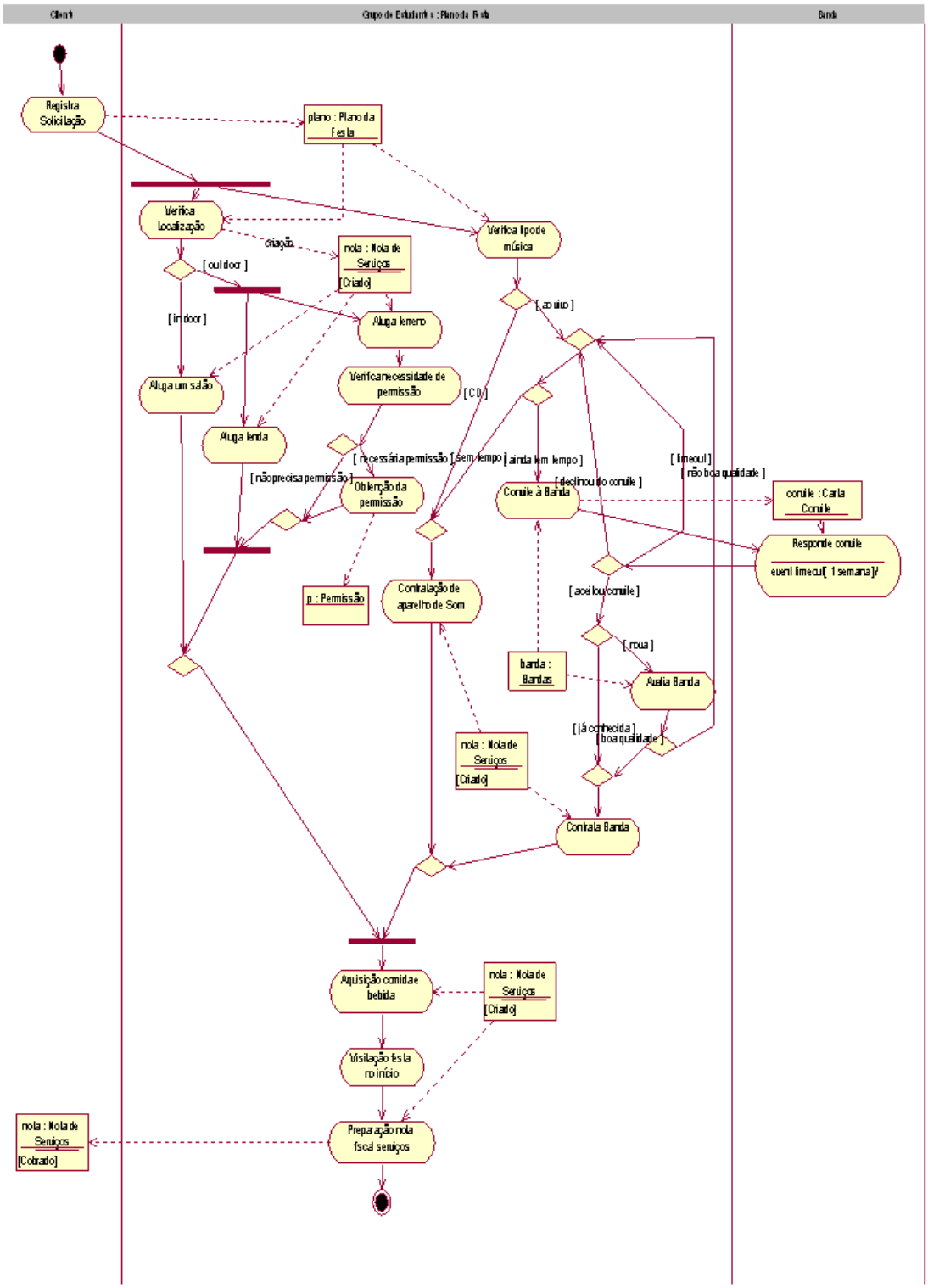


Figura 5.3: Diagrama de atividades a ser convertido para um modelo SAN

Assim como descrito na Seção 3.1, alguns elementos do diagrama de atividades não foram convertidos em elementos de umo modelo SAN. Assim o modelo SAN obtido no processo de conversão é apresentado na Figura 5.4

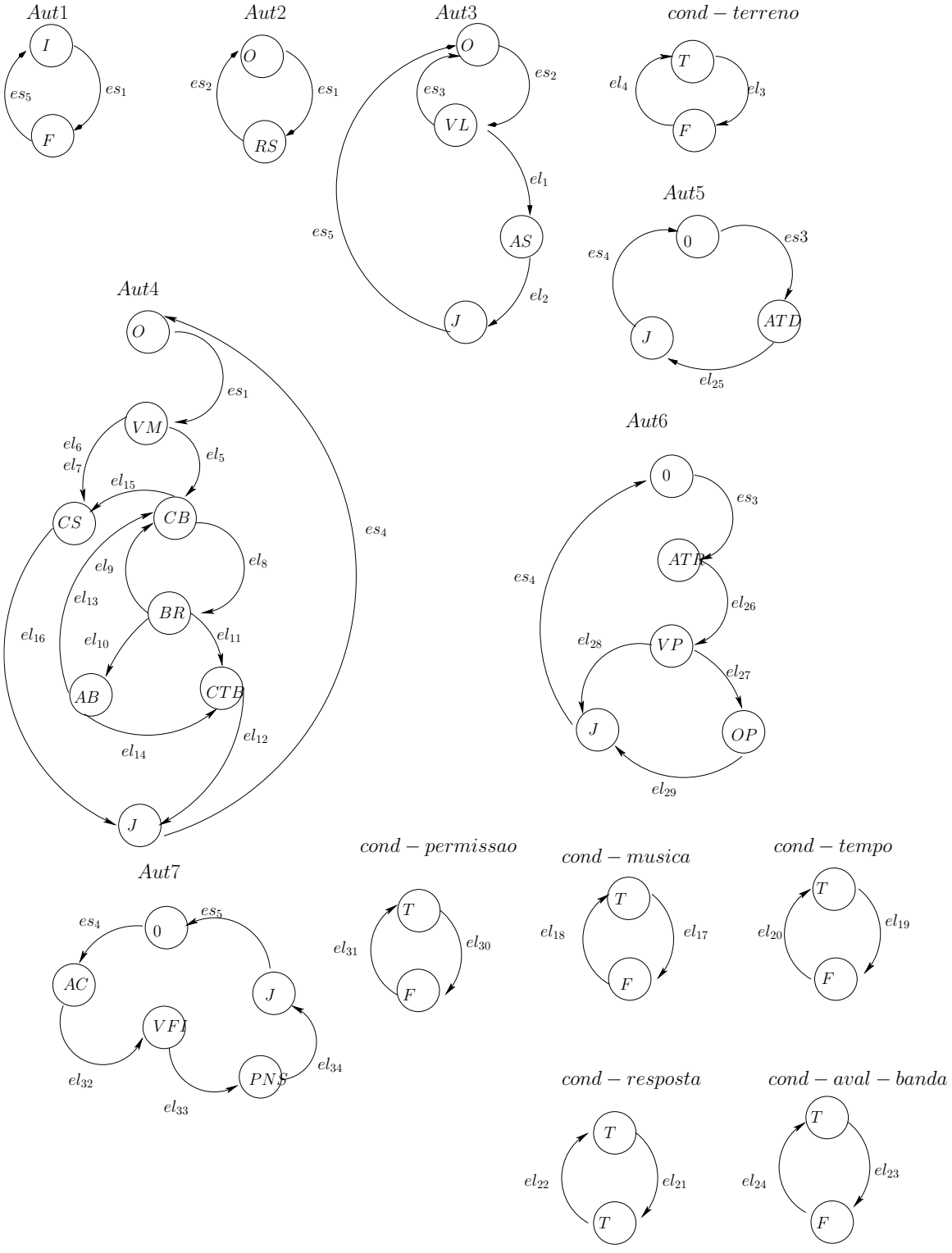


Figura 5.4: SAN resultante da conversão da Figura 5.3

Tipo	Evento	Taxa	Tipo	Evento	Taxa
syn	es_1	1	syn	es_2	1
syn	es_3	$f1$	syn	es_4	1
syn	es_5	$f1$	loc	el_1	$f1$
loc	el_2	$f2$	loc	el_3	1
loc	el_4	1	loc	el_5	$f4$
loc	el_6	$f5$	loc	el_7	$f6$
loc	el_8	1	loc	el_9	$f7$
loc	el_{10}	$f6$	loc	el_{11}	$f7$
loc	el_{12}	1	loc	el_{13}	$f8$
loc	el_{14}	$f9$	loc	el_{15}	$f10$
loc	el_{16}	1	loc	el_{17}	1
loc	el_{18}	1	loc	el_{19}	1
loc	el_{20}	1	loc	el_{21}	1
loc	el_{22}	1	loc	el_{23}	1
loc	el_{24}	1	loc	el_{25}	1
loc	el_{26}	1	loc	el_{27}	$f11$
loc	el_{28}	$f12$	loc	el_{29}	1
loc	el_{30}	1	loc	el_{31}	1
loc	el_{32}	1	loc	el_{33}	1

Tabela 5.7: Taxas dos eventos do modelo SAN da figura 5.4

O modelo SAN obtido é apresentado na Figura 5.4, o elemento estado inicial e final do diagrama de atividades é convertido para os estados *I* e *F* do autômato $Aut1$. Para esse autômato foram definidas duas transições disparadas pelos eventos sincronizantes denominados es_1 e es_5.

O autômato $Aut2$ possui dois estados, um denominado 0, que representa a não execução de nenhuma atividade, que possui uma transição disparada pelo evento sincronizante es_1 e que leva ao estado *RS*, que representa a atividade *registra solicitação* e possui uma transição que leva ao estado 0 e é disparada pelo evento sincronizante es_2.

O autômato $Aut3$ possui estados 0, que representa a não execução de nenhuma atividade, que possui uma transição disparada pelo evento sincronizante es_2 e que leva ao estado VL, que representa o elemento atividade *verifica localização*, que possui duas transições, uma disparada pelo evento local el_1, que leva ao estado *AS*, resultante do elemento *aluga salão* e a outra disparada pelo evento sincronizante es_3, que leva ao estado 0. Além desses estados, existe ainda o estado J, que representa o elemento *join* e possui uma transição disparada pelo evento es_5, que leva ao estado 0.

Aos eventos es_3 e el_1, foram associadas taxas funcionais que condicionam o disparo de cada uma das transições ao estado do autômato, denominado $cond-terreno$, que por sua vez possui dois estados nomeados com T e F. Tal associação tem por base o passo 4 proposto para o tratamento de condições de guarda existentes no diagrama modelado.

Todos os autômatos citados ao longo desta subseção e usados para atender as condições de guarda existentes no diagrama modelado possuem a mesma estrutura. Essa estrutura é composta

de dois estados nomeados como T e F, ambos com uma transição disparadas por eventos locais.

O autômato $Aut4$ possui o estado 0 que representa a não execução de nenhuma atividade, que possui uma transição disparada pelo evento sincronizante es_1 e que leva ao estado VM, o qual representa o elemento atividade *verifica musica*. Esse estado possui duas transições, uma disparada pelo evento el_5, que leva ao estado CB, que representa elemento atividade convida banda e outra disparada pelos eventos el_6 e el_7.

A esses eventos também foram associadas taxas funcionais, que condicionam o disparo das mesmas ao estado dos autômatos $cond - musica$ e $cond - tempo$, como forma de garantir o atendimento da condição de guarda existente.

O estado CB possui duas transições, disparadas pelos eventos locais el_{15} e el_8 que levam ao estado CS que representa o elemento atividade contrata aparelho de som e ao estado BR, que representa o elemento atividade banda responde e que possui três transições disparadas pelos eventos locais el_9 que leva ao estado CB, el_{10} que leva ao estado AB que representa o elemento atividade avalia banda e el_{11} e a última que leva ao estado CTB que representa a atividade contrata banda.

Os eventos el_9, el_{10} e el_{11} possuem taxas funcionais a fim de que se possa garantir o tratamento das condições de guarda. As taxas funcionais desses eventos estão relacionadas ao estado dos autômatos $cond - resposta$ e $cond - aval - banda$.

O estado AB possui duas transições disparadas pelos eventos locais el_{13}, que leva ao estado CB e el_{14} que leva ao estado CTB que representa o elemento atividade contrata banda. Os eventos citados também possuem taxas funcionais que condicionam o seu disparo ao autômato $cond-aval-banda$.

Para finalizar, o autômato $Aut4$ possui um estado denominado J que possui uma transição disparada pelo evento sincronizante es_4 que leva ao estado 0.

O $Aut5$ possui três estados, o estado 0 representa a não execução de nenhuma atividade, que possui uma transição disparada pelo evento sincronizante es_3 e que leva ao estado ATD, que representa o elemento atividade *aluga tenda*, que possui uma transição disparada pelo evento local el_{25} que leva ao estado J resultante do elemento *join* e que possui uma transição disparada pelo evento sincronizante es_4 que leva ao estado 0.

O $Aut6$ possui cinco estados, o estado 0 representa a não execução de nenhuma atividade, que possui uma transição disparada pelo evento sincronizante es_3 e que leva ao estado ATR que representa o elemento atividade *aluga terreno*, que possui uma transição disparada pelo evento local el_{26} que leva ao estado VP que representa o elemento atividade $verificapermissao$ e que possui duas transições disparadas pelos eventos locais el_{27}, que leva ao estado OP, que representa o elemento atividade obtêm permissão e el_{28} que leva ao estado J. Esses eventos também possuem taxas funcionais que condicionam o disparo das transições ao estado do autômato $cond-permissao$.

O estado OP possui uma transição disparada pelo evento local el_{29}, que leva ao estado J, que por sua vez, possui uma transição disparada pelo evento sincronizante es_4 que leva ao estado 0.

O $Aut7$ possui cinco estados 0 que representa a não execução de nenhuma atividade, que possui uma transição disparada pelo evento sincronizante es_4 e que leva ao estado AC, que representa o

elemento atividade *adquiri comida e bebida*, que possui uma transição disparada pelo evento local el_{32}, que leva ao estado VFI que representa o elemento atividade $verificafestanoinicio$ e que possui uma transição disparada pelo evento local el_{33}, que leva ao estado PNS, que representa o elemento atividade prepara nota de serviço. Esse estado, por sua vez, possui uma transição disparada pelo evento local el_{34}, que leva ao estado J, que possui uma transição disparada pelo evento sincronizante es_5, que eleva ao estado 0.

O modelo apresentado na Figura 5.4, possui 39754 estados atingíveis, definidos com base na função de atingibilidade e representam a execução de um conjunto de atividades ou de apenas uma atividade de acordo com a parte do diagrama de atividades que foi modelado. Um estado atingível pode representar além das atividades, a avaliação de uma determinada condição de guarda ou um elemento *join* do diagrama.

A Tabela 5.8 demonstra a equivalência entre a nomenclatura utilizada pela ferramenta PEPS e a atribuída ao modelo gerado. Os elementos apresentados na tabela variam de acordo com o modelo a ser descrito.

Nome dos estados no PEPS	Nomenclatura usada no modelo SAN
0	Representa o estado inicial dos autômatos $A^{(1)}$, $A^{(2)}$, $A^{(3)}$, $A^{(4)}$, $A^{(5)}$,$A^{(6)}$, $A^{(7)}$, $A^{(8)}$, $A^{(9)}$, $A^{(10)}$, $A^{(11)}$, $A^{(12)}$ e $A^{(13)}$ podendo ser s 1 0 2 1 3, s 1 0 3 0 1, s 1 0 3 0 2, nomeado como I,0 ou T conforme o autômato s 1 0 4 0 3, s 1 0 4 1 1, s 1 0 4 1 2, s 1 0 4 1 3, s1 1 0 0 0, qt 1 1 0 1 0
1	Representa o segundo estado dos autômatos $A^{(1)}$, $A^{(2)}$, $A^{(4)}$, $A^{(5)}$, $A^{(6)}$, $A^{(7)}$, $cond-terreno$, $cond-permissao$, $cond-tempo$, e $cond-musica$, $cond-resposta$ e $aval-banda$ podendo ser nomeado como F,RS,VL,VM, ATD,ATR,F ou AC conforme o autômato
2	Representa o teceiro estado dos autômatos $A^{(3)}$, $A^{(4)}$, $A^{(5)}$, $A^{(6)}$ e $A^{(7)}$ podendo ser nomeado como AS, J, CB, VP ou VFI, conforme o autômato
3	Representa o quarto estado dos autômatos $A^{(3)}$, $A^{(4)}$, $A^{(6)}$ e $A^{(7)}$ podendo ser nomeado como J, BR, OP ou PNS conforme a o autômato
4	Representa o quinto estado dos autômato $A^{(4)}$, $A^{(6)}$ e $A^{(7)}$ podendo ser nomeado também como J ou AB
5	Representa o sexto estado do autômato $A^{(4)}$ podendo ser nomeado como CTB
6	Representa o sétimo estado do autômato $A^{(4)}$ podendo ser nomeado como J

Tabela 5.8: Descrição dos estados do modelo gerado para o exemplo 2 - Figura 5.4

Como exemplo desses estados atingíveis, a partir de uma escolha aleatória, cita-se os seguintes:

1 1 1 1 1 0 1 1 0 0 2 4 1 4: Esse estado é atingível pois representa a execução da atividade recebe

solicitação que é a primeira atividade do diagrama, a atividade verifica localização para a qual uma condição de guarda é avaliada e que pelo autômato $cond - terreno$ ser falsa, isto é, estar no estado F o evento es_3 é disparado. Essa combinação de estados indica que a atividade aluga tenda está sendo executada e também a atividade obtêm permissão uma vez que o autômato cond-permissão está no estado F o que habilita o disparo do evento el_{27}. Os autômatos, cond-musica, cond-tempo, cond resposta, cond-aval-banda e por serem autômatos que possuem apenas eventos locais podem se encontrar em qualquer de seus dois estados possíveis, ou seja, T ou F uma vez que para execução dessas atividades os mesmo não são avaliados.

1 1 2 0 0 0 0 0 0 2 4 0 4: Esse estado é atingível pois representa a execução da atividade recebe solicitação que é a primeira atividade do diagrama. Essa combinação de estados indica que as demais atividades encontram-se no elemento *join* aguardando o término de outras atividades do diagrama assim como as demais atividades do diagrama.

1 1 3 0 5 0 1 1 1 2 4 0 0: Esse estado é atingível pois representa a execução da atividade recebe solicitação que é a primeira atividade do diagrama, a atividade avalia banda esta sendo executada. Essa atividade possui uma condição de guarda a ser avaliada, o que é feito pelo autômato cond-aval-banda que nessa combinação de estados está no estado F o que dispara o evento el_{13}. As demais atividades encontram-se no elemento *join* aguardando o término de outras atividades do diagrama assim como as demais atividades do diagrama.

Com relação ao exemplo dois, apresentado nessa dissertação, obteve-se uma grande quantidade de estados atingíveis, conforme citado anteriormente. Assim, as probabilidades desses estados, bem como os casos de teste gerados não serão apresentados neste capítulo.

Os exemplos aqui demonstrados não têm por objetivo esgotar as possibilidades de aplicação do método proposto, mas sim elucidar a maior quantidade de dúvidas que possam existir a respeito do mesmo.

6. Considerações Finais

A área de Engenharia de software, ao longo dos anos, tem buscado disciplinar o processo de desenvolvimento de software através do uso das mais diferentes técnicas de gerenciamento de projeto, mas uma das etapas do desenvolvimento que mais consome tempo e recursos é a de teste de software. Assim, com o intuito de melhorar os processos de teste de software e baratear os custos de execução dos mesmos, busca-se aplicar técnicas de avaliação de desempenho de sistemas no referido processo.

Este trabalho teve por objetivo apresentar um método de conversão de diagramas de atividades da UML para SAN e a geração de casos de teste de software. Para tal, foram apresentados os principais conceitos sobre UML, dando-se ênfase ao diagrama de atividades. Descreveram-se também as principais técnicas de teste de software, tais como teste estrutural e teste funcional. Abordaram-se aqui os conceitos sobre o formalismo SAN *(Stochastic Automata Networks)*, buscando demonstrar as principais características do mesmo.

O método apresentado baseou-se nos trabalhos aqui relacionados e descritos, que contribuiram para a construção da proposta. Dentre os trabalhos relacionados destaca-se o trabalho desenvolvido por Barros, em virtude da semelhança com o método apresentado neste trabalho.

Cabe ressaltar, que as pesquisas desenvolvidas para a elaboração desta dissertação demonstraram em sua maioria métodos de conversão de diagrama de atividades para redes de Petri, dentre eles [BDM02, LSP+08, LGMC04], o que faz com que o método aqui apresentado seja único.

Como forma de demonstrar a aplicação do método, foram apresentados neste trabalho dois exemplos de conversão de diagramas de atividades UML para SAN. Foram apresentados também os casos de teste gerados para os dois exemplos.

A seguir, são apresentadas as conclusões e considerações feitas a partir do desenvolvimento desse trabalho. Para finalizar, são propostos trabalhos futuros que podem ser desenvolvidos a partir deste estudo.

6.1 Limitações

Uma das principais limitações do método de conversão desenvolvido nessa dissertação é a baixa probabilidade de execução dos casos de teste gerados, que pode ocorrer em alguns casos. Acredita-se que a integração do diagrama de atividades com outros diagramas da UML *(Unified Modeling Language*) possa proporcionar melhores resultados.

Outra limitação do método é a forma de tratamento das condições de guarda, que deve ser feita com a criação de um autômato para cada condição de guarda existente no diagrama a ser convertido. Uma vêz que o estado global de uma SAN é definido como a combinação de todos os estados locais de cada autômato componente da SAN, com isso cada autômato criado para tratar essa característica do elemento decisão impacta no total de estados do modelo.

6.2 Trabalhos futuros

A partir desta dissertação, surgem alguns trabalhos futuros no sentido de complementar e dar continuidade ao trabalho realizado, dentre eles, relatamos alguns.

Um dos principais trabalhos futuros é a melhoria no método que proporcione a redução no estado global dos modelos SAN gerados, uma vez que os exemplos apresentados aqui, com base em diagramas com poucas atividades apresentaram um espaço de estados relativamente grande nos modelos gerados, especialmente em se tratando do exemplo dois.

Acredita-se que a conversão de um número maior de diagramas, gerará um modelo, cujo espaço de estados deva ser superior a 65.000.000, o que inviabilizaria o uso da ferramenta PEPS *(Performance Evolution of Parallel Systems)*.

Como trabalho futuro, espera-se encontrar outras formas de tratar as condições de guarda, visando obter modelos SAN com um tamanho menor, o que possibilitaria analisar sistemas mais complexos.

Outro estudo futuro é o desenvolvimento de um software, com base nos algoritmos aqui apresentados, visando automatizar o método de conversão e a geração dos casos de teste. Tal proposta se fundamenta no fato de que a ferramenta STAGE, desenvolvida pelos CTPS (Centro de Pesquisa e Teste de Software), é restrita aos membros do referido projeto.

Também como estudo futuro, propõe-se a adaptação do método à UML 2.0, pois o mesmo foi desenvolvido para o padrão UML 1.5. O poder de modelagem proporcionada pela UML 2.0 exige alterações no método desenvolvido, pois se espera que tal adaptação propicie outras formas de aplicação do referido método.

Referências Bibliográficas

[ABC+91] Ajmone-Marsan M., Balbo G., Chiola G., Conte G., Donatelli S., Franceschinis, G. An Introduction to Generalized Stochastic Petri Nets. journal Microelectronics and Reliability, pages 699–725, vol 3, november 1991.

[Bar02] Bartié A. Garantia da qualidade de software: adquirindo maturidade organizacional. Rio de Janeiro RJ, Elsevier, 2002, 328p.

[Bar06] Barros A. Integração de Redes de Autômatos Estocásticos ao Processo Unificado, Visando a Geração de Casos de Teste de Software. Master's thesis, PUCRS-FACIN-PPGCC, Porto Alegre, Dezembo 2006 123p.

[BBF+04] Benoit A., Brenner L., Fernandes P., Plateau B. Aggregation of stochastic automata networks with replicas, Linear Algebra and its Applications, vol. 386, Julho 2004, pp. 111–136.

[Bel09] Bell D. UML basics - Part II: The activity diagram. capturado de http://www.ibm.com/developerworks/ration/fumlbasicsdb.pdf,08/2009

[BDM02] Bernardi S., Donatelli S., Merseguer J. From UML sequence diagrams and statecharts to analysable petri net models. In WOSP '02: Proceedings of the 3rd international workshop on Software and performance, pages 35–45, New York, NY, USA, 2002. ACM.

[BFF+04] Bertolini C., Farina A. G., Fernandes P., Oliveira. F.M. Test Case Generation using Stochastic Automata Networks: Quantitative Analysis. In Proceedings of the Second IEEE International Conference on Software Engineering and Formal Methods, pages 251–260, Washington, DC, USA, September 2004. IEEE Computer Society.

[BFR+04] Baldo L., Fernandes L. G., Roisenberg P., Velho P. e Webber T. Parallel PEPS Tool Performance Analysis using Stochastic Automata Networks, Lecture Notes in Computer Science, vol. 3149, Agosto/Setembro 2004, pp. 214–219.

[BMF+03] Benoit A., Brenner L., Fernandes P., Plateau B. e Stewart. W.J. The PEPS Software Tool, Lecture Notes in Computer Science, vol. 2794, Setembro 2003, pp. 98–115.

[BRC+07] Bastos A., Rios E., Cristalli R., Moreira T. Base de conhecimento em teste de software. São Paulo, SP, Martins, 2007 264p.

[BRJ05] Booch G, Rambaugh J, Jacobson I. UML guia do usuário. Rio de Janeiro. RJ, Elsevier Editora, 2ªed., 2005, 474p.

[BRJ99] Booch G, Rambaugh J, Jacobson I. The unified software development process. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 1999, 474p.

[Coc05] Cockburn A. Escrevendo Casos de Uso Eficazes - um guia prático para desenvolvedores. Porto Alegre RS, Bookman, 2005, 254p.

[COR04] Copstein B., Oliveira F., Reginato L. STAGE: an Integrated Environment for Statistical Test Script Generation. In V Workshop de Testes e Tolerância a Falhas, pages 77–88, 2004.

[Dep08] Departamento de Informática.Universidade de Torino. GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. capturado de http://www.di.unito.it/greatspn/index.html#GreatManual, 12/2008.

[Fer98] Fernandes P. Méthodes numériques pour la solution de systèmes Markoviens à grand espace d'états, Tese de Doutorado, Institut National Polytechnique de Grenoble, France, 1998, 262p.

[FoS08] Fowler M, Scott K. UML Essencial: um breve guia para a linguagem padrão de modelagem de objetos. Porto Alegre - RS, Bookman, 2ª ed, 2004,162p.

[IKV08] IKV++ Tecnologies ag. Medini QVT:Uma Ferramenta para a transformação de modelos baseada em QVT. Capturado em http://projects.ikv.de/qvt, 12/2008.

[LGMC04] López-Grao J.P. , Merseguer J., Campos.J. From UML activity diagrams to stochastic Petri nets: Application to software performance engineering. In Proceedings of the Fourth International Workshop on Software and Performance (WOSP'04), pages 25–36, Redwood City, California, USA, January 2004. ACM. Also in ACM SIGSOFT Software Engineering Notes, Vol. 29, no. 1, January 2004.

[LSP+08] Lachtermacher L., Silveira D.S., Paes R.B., Lucena C.J.P. Transformando o diagrama de atividades em uma Rede de Petri. Technical Report 0103-9741, PUCRIO, Rio de Janeiro, 2008.

[MNF01] Mian P.G , Natali A.C.C., Falbo R.A.D. Ambientes de Desenvolvimento de Software e o Projeto ADS. 2001. Universidade Federal do Espirito Santo. capturado de http://www.inf.ufes.br/7Efalbo/download/pub/RevistaCT072001.pdf,12/2008

[Mye79] Myers G.J. The art of software testing. Ontario, Canada, John Wiley & Sons, Inc., 1979, 177p.

[Nas09] Nascimento E.R. Análise do Processo de Descoberta de Rotas e Avaliação do Protocolo de Roteamento Reativo DSR em Redes Wireless Ad Hoc Utilizando Redes de Autômatos Estocásticos - SAN. Master's thesis, PUCRS-FACIN-PPGCC, Porto Alegre, Dezembro 2009, 127p.

[Neu08] Neuwald F.B. Técnica para Obtenção de Redes de Autômatos Estocásticos Baseada em Especificações de Software em UML. Master's thesis, PUCRS-FACIN-PPGCC, Porto Alegre, Outubro 2008, 91p.

[OMG08a] OMG Object Manager Group. OCL Object Constraint Language OMG Available Specification. capturado de http://www.omg.org/cgi-bin/doc?formal/2006-05-01, 12/2008.

[OMG08b] OMG Object Manager Group. MOF MetaObject Facility. capturado de http://www.omg.org/mof/. 12/2008.

[Pen04] Pender T. UML a Biblia. Rio de Janeiro RJ, Campus, 2004, 752p.

[PlA91] Plateau B., Atif K. Stochastic Automata Networks for modeling parallel systems, IEEE Transactions on Software Engineering, vol. 17-10, Outubro 1991, pp. 1093–1108.

[Pet62] Petri C.A. Communication with automata. DTIC Research Report AD0630125, 1966, vol. 1-1, 97p.

[PeY08] Pezzé M., Young M. Teste e análise de software: processo, princípios e técnicas. Porto Alegre RS, Bookman, 2008, 830p.

[Rei85] Reisig W. Petri nets: an introduction. New York, NY, USA, Springer-Verlag Heidelberg, 1985, 161p.

[Rio86] Rios J.L.P. Modelos Matemáticos em Hidráulica e no Meio Ambiente. In Simpósio Luso-Brasileiro sobre Simulação e Modelagem em Hidraulica, pp. 193–210, Lisboa, Portugal, 1986. APH-LNEC.

[Sal03] Sales A. Formalismos Estruturados de Modelagem para Sistemas Markovianos Complexos. Master's thesis, PUCRS-FACIN-PPGCC, Porto Alegre, December 2003, 138p.

[Som07] Sommerville I. Engenharia de Software. São Paulo- SP, Pearson Addison Wesley, 8ª ed., 2007, 568p.

[Ste94] Stewart W.J. Introduction to the numerical solution of Markov chains. Princeton University Press, 1994, 539p.

[Tig08] Projeto tigris.org. Web Site ArgoUML Project. capturado de http://argouml.tigris.org/,12/2008.

[VaH02] Van W.D.A., Hee V.H. Workflow Management: Models, Methods, and Systems (Cooperative Information Systems. Massachusetts, USA, The MIT Press, 2002, 384p.

[w3c08] w3c World Wide Web Consortium. XML Extensible Markup Language. capturado de http://www.w3c.org./xml., 12/2008.

[Web03] Webber T. Alternativas para o Tratamento Numérico Otimizado da Multiplicação Vetor-Descritor. Master's thesis, PUCRS-FACIN-PPGCC, Porto Alegre, December 2003, 98p.

[Whi97] Whittaker J.A. Stochastic software testing. Ann. Softw. Eng., 4:115–131, 1997.

[YJH04] Jiong Y., Ji W., Huo-wang C. Automatic generation of markov chain usage models from real-time software uml models. In *QSIC '04: Proceedings of the Quality Software, Fourth International Conference*, pages 22–31, Washington, DC, USA, 2004. IEEE Computer Society.

A. Exemplo 1 (.SAN)

Aqui é apresentado o arquivo .san desenvolvido para o exemplo 1.

A.1 Exemplo 1 (.SAN)

```
{

identifiers
i=1;
f=1;
O=1;
rp=1;
f=1;
pp=1;
en=1;
er=1;
enf=1;
  rpt=1;
t=1;
j=1;
  f1=(st a4==t_e)*1;
f2=(st a4==f_e)*1;

events
syn  events1   (i);
  syn  events2   (i);
syn  events3   (i);
loc  eventl1   (f1);
loc  eventl2   (f2);
loc  eventl3   (i);
loc  eventl4   (rp);
loc  eventl5   (rp);
loc  eventl6   (rp);
  loc  eventl7   (pp);
loc  eventl8   (rp);
loc  eventl9   (rp);
```

```
partial reachability  = (st a1 == i_e) && (st a2 == O_e) && (st a3 == O_e)
                        && (st a4 == t_e) && (st a5 == O_e);

network ativsan (continuous)

aut a1

    stt i_e
          to (f_e)   events1
    stt f_e
      to (i_e)  events3

aut a2

    stt O_e
          to (rp_e)  events1
    stt rp_e
      to (O_e)   events2

aut a3

stt O_e
to (pp_e) events2

stt pp_e
to (en_e) eventl1
to (er_e) eventl2

stt en_e
to (j_e) eventl3

stt er_e
to (j_e) eventl4

stt j_e
to (O_e) events3

  aut a4

          stt t_e
```

```
              to (f_e) eventl5
        stt f_e
              to (t_e) eventl6

  aut a5

        stt O_e
              to (enf_e) events2

       stt enf_e
              to (rpt_e) eventl7

      stt rpt_e

to (j_e) eventl8

     stt j_e
            to (O_e) events3
results

caminho1 = (st a1 == i_e && st a2 == rp_e && st a3 == pp_e
&& st a4 == t_e && st a5 == enf_e);
caminho2 = (st a1 == i_e && st a2 == rp_e && st a3 == en_e
&& st a4 == t_e && st a5 == enf_e);
caminho3 = (st a1 == i_e && st a2 == rp_e && st a3 == er_e
&& st a4 == t_e && st a5 == enf_e);
caminho4 = (st a1 == f_e && st a2 == rp_e && st a3 == pp_e
&& st a4 == t_e && st a5 == enf_e);
caminho5 = (st a1 == f_e && st a2 == rp_e && st a3 == en_e
&& st a4 == t_e && st a5 == enf_e);
caminho6 = (st a1 == f_e && st a2 == rp_e && st a3 == er_e
&& st a4 == t_e && st a5 == enf_e);
caminho7 = (st a1 == i_e && st a2 == rp_e && st a3 == pp_e
&& st a4 == f_e && st a5 == enf_e);
caminho8 = (st a1 == i_e && st a2 == rp_e && st a3 == en_e
&& st a4 == f_e && st a5 == enf_e);
caminho9 = (st a1 == i_e && st a2 == rp_e && st a3 == er_e
&& st a4 == f_e && st a5 == enf_e);
}
```

A. Exemplo 2 (.SAN)

Aqui é apresentado o arquivo .san desenvolvido para o exemplo 2.

A.1 Exemplo 2 (.SAN)

```
{
identifiers
i=1;
f=1;
rs=1;
vl=1;
as=1;
atr=1;
vp=1;
op=1;
atd=1;
vm=1;
cs=1;
cb=1;
br=1;
ab=1;
ctb=1;
ac=1;
vfi=1;
pns=1;
O=1;
t=1;
      f1=(st cond_terreno==t_e)*1;
f2=(st cond_terreno==f_e)*1;
f3=(st cond_musica==t_e && st cond_tempo==f_e )*1;
f4=(st cond_tempo==t_e && cond_musica==f)*1;
f5=(st cond_musica==t_e)*1;
f6=(st cond_resposta==t_e)*1;
f7=(st cond_resposta==f_e)*1;
f8=(st cond_aval_banda==t_e)*1;
f9=(st cond_aval_banda==f_e)*1;
f10=(st cond_permissao==t_e)*1;
f11=(st cond_permissao==f_e)*1;
```

```
events
syn  es1    (i);
  syn  es2    (i);
syn  es3    (f1);
syn  es4    (i);
syn  es5    (f1);
loc  el1    (f1);
loc  el2    (f2);
loc  el3    (i);
loc  el4    (i);
loc  el5    (f4);
loc  el6    (f5);
loc  el7    (f6);
loc  el8    (i);
loc  el9    (f7);
loc  el10   (f6);
loc  el11   (f8);
loc  el12   (i);
loc  el13   (f8);
loc  el14   (f9);
loc  el15   (f10);
loc  el16   (i);
loc  el17   (i);
loc  el18   (i);
loc  el19   (i);
loc  el20   (i);
loc  el21   (i);
loc  el22   (i);
loc  el23   (i);
loc  el24   (i);
loc  el25   (i);
loc  el26   (i);
loc  el27   (f11);
loc  el28   (f12);
loc  el29   (i);
loc  el30   (i);
loc  el31   (i);
loc  el32   (i);
loc  el33   (i);
```

```
loc  el34   (i);

partial reachability  = (st a1==i_e) && (st a2==O_e) && (st a3==O_e)
&& (st a4==O_e) && (st a5==O_e) && (st a6==O_e)
&& (st a7==O_e);

network exemp2 (continuous)

aut a1

    stt i_e
                      to (f_e)  es1

    stt f_e
                      to (i_e)  es5

aut a2

stt O_e
to (rs_e) es1

stt rs_e
to (rs_e) es2

 aut a3

stt O_e
to (vl_e) es2

stt vl_e
to (O_e) es3
to (as_e) el1

stt as_e
to (j_e) el2

stt j_e
to (O_e) es4

aut cond_terreno
```

```
stt t_e
to (f_e) el3

stt f_e
to (t_e) el4

aut a4

stt O_e
to (vm_e) es1

stt vm_e
to (cb_e) el5
to (cs_e) el6
to (cs_e) el7

stt cb_e
to (br_e) el8
to (cs_e) el15

stt cs_e
to (j_e) el16

stt br_e
to (cb_e) el9
to (ab_e) el10
to (ctb_e) el11

stt ctb_e
to (j_e) el12

stt ab_e
to (cb_e) el13
to (ctb_e) el14


stt j_e
to (O_e) es4
```

```
       aut cond_musica

stt t_e
to (f_e) el17

stt f_e
to (t_e) el18

     aut cond_tempo

stt t_e
to (f_e) el19

stt f_e
to (t_e) el20


      aut cond_resposta

stt t_e
to (f_e) el21

stt f_e
to (t_e) el22

      aut cond_aval_banda

stt t_e
to (f_e) el23

stt f_e
to (t_e) el24


aut a5

stt O_e
to (atd_e) es3

stt atd_e
```

```
to (j_e) el25

stt j_e
to (O_e) es4

aut a6
stt O_e
to (atr_e) es3

stt atr_e
to (vp_e) el26

stt vp_e
to (op_e) el27
to (j_e)  el28

       stt op_e

to (j_e)  el29
       stt j_e

to (O_e)  es4

      aut cond_permissao

stt t_e
to (f_e) el30

stt f_e
to (t_e) el31



     aut a7

stt O_e
to (ac_e) es4

stt ac_e
to (vfi_e) el32
```

```
stt vfi_e
to (pns_e) el33

stt pns_e
to (j_e) el34

    stt j_e
      to (O) es5

results
caminho1 = (st a1 == i_e && st a2 == rs_e && st a3 == vp_e && st a4 == vm_e
 && st a5 == ac_e && st a6 == O_e && st a7 == O_e);
caminho2 = (st a1 == i_e && st a2 == vl_e && st a3 == vp_e && st a4 == vm_e
 && st a5 == ac_e && st a6 == O_e && st a7 == O_e);
caminho3 = (st a1 == i_e && st a2 == rs_e && st a3 == op_e && st a4 == vm_e
 && st a5 == ac_e && st a6 == O_e && st a7 == O_e);
caminho4 = (st a1 == f_e && st a2 == rs_e && st a3 == vp_e && st a4 == vm_e
 && st a5 == ac_e && st a6 == O_e && st a7 == O_e);
caminho5 = (st a1 == f_e && st a2 == vl_e && st a3 == vp_e && st a4 == vm_e
   && st a5 == ac_e && st a6 == O_e && st a7 == O_e);
caminho6 = (st a1 == f_e && st a2 == rs_e && st a3 == op_e && st a4 == vm_e
   && st a5 == ac_e && st a6 == O_e && st a7 == O_e);
```