Universidade Estadual de Feira de Santana

Jorge Augusto Vasconcelos Lima

UMA ANÁLISE SOBRE A APLICAÇÃO DE LINGUAGENS E MÉTODOS FORMAIS NO DESENVOLVIMENTO DE SOFTWARE

Feira de Santana – BA

Agosto / 2009

Jorge Augusto Vasconcelos Lima

UMA ANÁLISE SOBRE A APLICAÇÃO DE LINGUAGENS E MÉTODOS FORMAIS NO DESENVOLVIMENTO DE SOFTWARE

Monografia apresentada à Banca de Graduação em Engenharia de Computação da Universidade Estadual de Feira de Santana para a obtenção do título de Engenheiro de Computação.

Orientador:
Prof. Msc. Hugo Saba Pereira Cardoso

CURSO DE ENGENHARIA DE COMPUTAÇÃO
DEPARTAMENTO DE TECNOLOGIA
UNIVERSIDADE ESTADUAL DE FEIRA DE SANTANA

Feira de Santana – BA

Agosto / 2009

Monografia de Projeto Final de Graduação sob o título *"UMA ANÁLISE SOBRE A APLICAÇÃO DE LINGUAGENS E MÉTODOS FORMAIS NO DESENVOLVIMENTO DE SOFTWARE"*, defendida por Jorge Augusto Vasconcelos Lima, em 24 de agosto de 2009, Feira de Santana - Ba, pela banca examinadora constituída pelos professores:

Prof. Msc. Hugo Saba Pereira Cardoso
Departamento de Exatas - UEFS
Orientador

Prof. Msc. Claudia Pinto Pereira Sena
Departamento de Exatas - UEFS
Examinador

Prof. Msc. Eduardo Manuel de Freitas
Departamento de Exatas - UNEB
Examinador

Dedico esta monografia a meus pais,
pela exemplo de vida,
aos meus amigos e irmão,
por tornarem os dias mais leves,
e para minha namorada, que sempre
tem me dado boas recordações.

Agradecimentos

Primeiramente a DEUS por iluminar minha vida.

Aos meus pais pelo constante apoio e carinho que sempre dedicaram em qualquer ocasião, bem como a meus irmãos pelo apoio e ombro amigo.

A minha namorada Josyana pelos momentos de carinho e alegria, e também pelo constante apoio e cobrança.

Aos amigos na UEFS Ângelo, Biscoito, Ademar, Pelezinho, Nara, Davi, Paloma, João, Ive, Eder, Mac, Robério, Talles, Ricardo, Lucas e Gaby, assim como aos amigos/companheiros/irmãos de longa data Vinícus, Dhiego, Estácio, Lucas e Mineiro, por todos momentos de risadas, conversas, revoltas e cumplicidade que passamos juntos.

Aos demais colegas de UEFS, sem discriminar semestre, pelas contribuições na minha vida acadêmica.

Aos meus companheiros diários de trabalho na Assessoria Especial de Informática na UEFS pelos constantes ensinamentos.

Ao Prof. Msc. Hugo Saba pela grande apoio e auxílio prestado na conclusão deste trabalho, além da sua amizade.

Resumo

Empresas de desenvolvimento de software buscam uma maior competitividade através da adoção de padrões, e assim elevam a qualidade de seus produtos. Conjuntamente a este fato, observa-se a demanda crescente por aplicações cada vez mais robustas e com características concorrentes, onde as execuções não são tratadas de forma sistêmica tais como as aplicações web e softwares corporativos para gestão empresarial. Os softwares concorrentes necessitam de uma análise complexa nas partes críticas. Este trabalho insere-se no contexto descrito, discutindo formas de agregar métodos formais no processo de desenvolvimento de software para reduzir o número de erros, e custos de projeto.

Palavras-chave: Engenharia de Software, Métodos Formais, Linguagens Formais, *Process Mining*.

Sumário

Lista de Figuras

Lista de Tabelas

Listagens de Esquemas Utilizados

Lista de Acrônimos

API	Application Programming Interface
CCS	Calculus of Communicating Systems
CSP	Communicating Sequencial Processes
EPC	Event-Driven Process Chain
ITU	International Telecommunication Union
LTL	Linear Temporal Logic
MSC	Message Sequence Charts
UML	Unified Modeling Language
CSP	Communicating Sequencial Processes
XMI	XML Metadata Interchange
XML	Extensible Markup Language
bMSC	Basic Message Sequence Charts
hMSC	High-Level Message Sequence Charts

1 Introdução

Obter êxito em projetos na indústria do software continua sendo um objetivo difícil de ser alcançado. Estudos, como o realizado por Standish Group (2004), mostram a porcentagem de projetos que apresentaram falhas e/ou sofreram mudanças nos requisito estabelecidos, caracterizando aumento de custos no projeto. Para projetos de grande porte, como aplicações web e softwares corporativos para gestão empresarial, os quais apresentam maior grau de complexidade devido a presença de partes críticas faz-se necessário formas de garantir a qualidade e conformidade do projeto. Além de garantir os fatores anteriores, é importante neste tipo de projeto reduzir ao máximo a quantidade de erros nas fases iniciais, contribuindo para redução dos custos (PRESSMAN, 2001).

O acréscimo de formalismo nos projetos, através de métodos formais, possibilita o aumento nas chances de sucesso nos projeto (PRESSMAN, 2001) (SOMMERVILLE, 2003). No desenvolvimento de software , métodos formais atuam em vários momentos do ciclo de vida dprojeto, sendo possível utilizá-los na verificação da conformidade dos requisitos, com o objetivo de evitar contradições, ambigüidades e/ou incompletude (PRESSMAN, 2001). Cada uma dessas características pode ser garantida pelo fato das linguagens formais utilizarem regras de inferência, onde uma proposição inicial pode ser avaliada (i.e. verificação de consistência), bem como permite apenas descrever aspectos do software de uma única maneira (i.e. eliminando as ambigüidades). Embora confira estas características, métodos formais não podem ser encarados como a ”solução perfeita”para corretude de sistemas (HALL, 1990).

Os métodos formais baseiam-se em propriedades matemáticas, como consistência, completude e corretude, para prover maneiras de transpor estes fundamentos para o desenvolvimento de software e deste modo diminuir a sua quantidade de falhas (MARCINIAK, 1994).

O uso de métodos formais dar-se-á através de linguagens formais de especificação, as quais fornecem maneiras de definir precisamente especificações do software. Igualmente a qualquer linguagem possuem um domínio sintático e semântico. O domínio sintático deriva da sintaxe de notação padrão da teoria dos conjuntos e do cálculo de proposições. Embora normalmente

a sintaxe seja descrita com símbolos, pode-se encontrar linguagens que também são descritas através de ícones, como setas e círculos, desde de que não tornem a linguagem ambígua. A semântica é a responsável por definir significado aos termos encontrados no domínio sintático (PRESSMAN, 2001).

Estudos como os desenvolvidos por Hall (1990), Stidolph e Whitehead (2003) e Holloway (1997) vêm desmistificando o uso de métodos formais como forma de eliminar completamente os erros. Mas encontram-se nesses estudos justificativas para a importância da adoção de métodos formais no desenvolvimento de software, de modo a torná-los menos sujeitos a falha . Mesmo havendo opiniões distintas a respeito da adoção de métodos formais, vem-se tornando cada vez mais consenso que sua adoção seja necessária, principalmente em sistemas tidos como críticos (i.e. aplicações web e softwares) corporativos), e/ou complexos. Isto pode ser inferido pelo investimento crescente que está ocorrendo na área formal, no desenvolvimento e integração de ferramentas, muito embora ainda restringindo-se ao âmbito das grandes corporações e empresas que buscam níveis de certificação mais elevados.

Em Stidolph e Whitehead (2003) e em Holloway (1997), encontra-se uma discussão importante de quando deve-se adotar o formalismo e quando pode-se optar por seguir apenas os métodos já em voga no desenvolvimento de software, por exemplo UML (*Unified Modeling Language*).

1.1 Objetivos

Este trabalho analisa linguagens formais ligadas a métodos comportamentais, de modo a identificar uma linguagem que possa propor uma técnica de fácil adoção, sem necessitar de conhecimento prévio aprofundado por parte de desenvolvedores. Deste modo seria aconselhável linguagens que possuíssem características gráficas, que a tornam intuitiva, e que se aproximassem de conceitos já conhecidos e difundidos.

É importante ressaltar que este trabalho não discutirá de forma aprofundada aspectos referentes a formalização das linguagens, assim como métodos formais. Contudo serão citados outros trabalhos que tratam deste assunto.

1.2 Justificativa

Como contribuição, ter-se-á um conjunto de informações de apoio para a adoção de formalismo nos processos de desenvolvimento. Conjuntamente a isso, este trabalho apresentará um

método, baseado em uma técnica apresentada por Lassen, van Dongen e van der Aalst (2007), agregada com a aplicação desenvolvida denominada Ártemis.

1.3 Metodologia

As linguagens formais possibilitam, a seu modo, visualizar e validar comportamentos. Algumas linguagens, como Redes de Petri (PETERSON, 1977), dificultam as descrições dos sistemas, mas podem ser menos complexas na validação, enquanto o oposto ocorre com outras linguagens, a exemplo do MSC (*Message Sequence Charts*) (ITU, 1999).

Por este motivo, a metodologia adotada consiste em selecionar um grupo de linguagens formais com finalidade similar e realizar uma análise sobre estas. A decisão de analisar somente um grupo de linguagens formais se deve a existência de um conjunto amplo de linguagens formais.

Os critérios estabelecidos para a análise das linguagens foram intuitividade, facilidade de manuseio, semelhança com conhecimentos já definidos, maturidade e ferramentas desenvolvidas para sua utilização.

Após a análise, a etapa seguinte consiste em elaborar um método para a adoção da linguagem formal selecionada e uma ferramenta para possibilitar a utilização do mesmo.

1.4 Estrutura do Trabalho

O presente trabalho está estruturado em cinco capítulos assim distribuídos: o primeiro capítulo apresenta esta introdução, que contextualiza a problemática do trabalho e delimita os objetivos, bem como apresenta sua estrutura. O segundo capítulo contempla a fundamentação teórica, na qual são abordados os seguintes tópicos: métodos formais, linguagens formais, *process mining* e Engenharia de Software.

No terceiro capítulo aborda o processo de formalização, com uma proposta de sua aplicação. No capítulo seguinte é apresentado os resultados obtidos e no quinto, e último, capítulo encontram-se as considerações finais do trabalho e as perspectivas futuras.

2 *Referencial Teórico*

2.1 Engenharia de Software

Sistemas computacionais são utilizados em larga escala na sociedade, tornando-se parte intrínseca do cotidiano. A evolução dos sistemas é caracterizada pelo aumento do grau de sofisticação e complexidade no desenvolvimento de softwares.

O aumento na complexidade dos sistemas conduziu a uma necessidade de acompanhamento rigoroso dos problemas apresentados. Não somente problemas relacionados ao processo de desenvolvimento ou funcionamento inadequado dos sistemas, mais também questões referentes às estimativas de custo, tempo e andamento do software (PRESSMAN, 2001).

A Engenharia de Software surgiu neste contexto na busca de soluções para os problemas citados. Segundo Sommerville (2003), engenharia de software se ocupa de todos os aspectos da produção de software, desde os primeiros estágios de especificação do sistema até a manutenção desse sistema, depois que entrou em operação.

Para que seja possível esta produção, a engenharia de software adota métodos e processos. Os processo caracterizam-se como um conjunto de etapas que auxiliam a gerência e sistematizam a adoção de métodos (PRESSMAN, 2001). Os métodos fornecem um conjunto estruturado de tarefas que abrangem todo o ciclo de desenvolvimento de um sistemas, visando a qualidade do software, e a redução de falhas (PRESSMAN, 2001) (SOMMERVILLE, 2003).

A integração de métodos, processos e ferramentas permite a criação de modelos de processos, que corresponde a uma padronização das práticas que estão sendo empregadas (SOMMERVILLE, 2003). Com base nesta definição, é possível perceber que existem várias descrições de modelos, cada qual com uma característica.

Como este trabalho trata de formalismo no desenvolvimento de software, o modelo adotado é o baseado em métodos formais. Neste modelo, as atividades executadas, especificação, desenvolvimento e verificação são alicerçadas por conceitos matemáticos. A adoção visa evitar

a não conformidade e garantir que as especificações de software atendam ao que foi acordado no início do projeto (PRESSMAN, 2001) (SOMMERVILLE, 2003).

2.2 Métodos Formais

O método consiste num conjunto de princípios para selecionar e aplicar técnicas e ferramentas de forma eficaz na tentativa de analisar e sistematizar um determinado artefato. Métodos formais podem ser entendidos como um conjunto de princípios para especificação formal e modelagem de técnicas para realização de inferências (GOGOLLA, 2004).

Uma especificação formal descreve um modo abrangente e consistente de modelagem, que é expresso na forma de uma linguagem formal (MOURA, 1995). Uma especificação deve ser precisa, não podendo dar margem a interpretações errôneas. Em Meyer (1985), encontram-se, como possíveis tipos de erros: redundância, omissão, sobreespecificação, inconsistência, ambigüidades, referência antecipada e descrição subjetiva. Como exemplos de casos que podem ser modelados: levantamento de requisitos, arquiteturas de software e comportamento de entidades.

As técnicas de inferência são definidas como um conjunto de provas e especificações de regras de transformação usados no desenvolvimento formal. No desenvolvimento formal, todas as especificações são formais, todas as provas são obrigatórias e as provas são sistematizadas. O formalismo no desenvolvimento deve ser rigoroso e sistemático (GOGOLLA, 2004).

Em um desenvolvimento considerado rigoroso, todas as especificações devem ser expressas em uma linguagem com características formais, em que os teoremas relevantes usados devem estar expressos. Já um desenvolvimento sistemático é caracterizado pela necessidade de que todos os passos do processo de dedução sejam provados (GOGOLLA, 2004).

2.3 Linguagens Formais

Antes de definir linguagem formal, é preciso conceituar símbolo e cadeia, baseado nas definições encontradas em Ramos, Neto e Vega (2009). Um símbolo é uma representação gráfica, indivisível, utilizada na construção de cadeias. Uma cadeia é formada através da justaposição de um número finito de símbolos, obtidos de algum conjunto finito não-vazio, denominado alfabeto.

Partindo das definições anteriores, ”uma linguagem formal pode ser compreendida como

um conjunto, finito ou infinito, de cadeias de comprimento finito, formada pela concatenação de elementos de um alfabeto finito e não-vazio“ (RAMOS; NETO; VEGA, 2009). O conjunto de símbolos deriva da notação padrão da teoria dos conjuntos e do cálculo de proposições, de modo que não torne esta linguagem ambígua (PRESSMAN, 2001).

Linguagens formais possuem uma semântica para agregar o significado ao conjunto de símbolos definidos na sintaxe, possibilitando a compreensão de uma seqüencia das representações gráficas. Quando os símbolos que constituem a sintaxe são definidos, e uma semântica é associada a estes, tem-se portanto uma gramática formal (PRESSMAN, 2001) (MATEESCU; SALOMAA, 1997).

Nesta seção, serão abordadas, sucintamente, as linguagens formais adotadas no trabalho, descrevendo os elementos que as constituem e as relações entre estes símbolos, isto é, a gramática que norteia essas linguagens.

2.3.1 *Message Sequence Charts*

Message Sequence Charts (MSC) é uma linguagem de especificação gráfica padronizada pelo ITU (1999). O MSC está incluso num grupo de linguagens que proporcionam técnicas gráficas similares, as quais, independentemente, surgiram de diferentes áreas, tais como as de modelagem para orientação a objetos, para tempo real, simulação e metodologia de testes.

O MSC caracteriza-se por ser uma linguagem intuitiva aos desenvolvedores de software por apresentar características do diagrama de seqüencia definido no *Unified Modeling Language* (UML). Basicamente, o MSC retrata o comportamento entre uma quantidade qualquer de entidades, tanto logicamente, quanto fisicamente, demonstrando a ordem em que as mensagens são trocadas (MAUW; RENIERS; WILLEMSE, 2001) (ALUR; ETESSAMI; YANNAKAKIS, 2003). Vale ressaltar que, mesmo sendo uma linguagem gráfica, também é possível utilizá-la de forma textual conforme pode ser encontrado em (ITU, 1999).

Graficamente, uma entidade MSC é representada por uma linha de tempo, composta por uma linha vertical, e por retângulos localizados no inicio e final destas linhas de tempo (Figura 2.1). O retângulo sem preenchimento representa o marco inicial, enquanto o preenchido representa o marco terminal, este último sendo opcional. Outro elemento apresentado são as mensagens, as quais correspondem aos arcos que conectam essas linhas de tempo.

A modelagem no MSC é realizada através de cenários, em que um sistema é descrito em várias partes que são agregadas para constituir o todo.

O MSC é bastante utilizado na área de telecomunicação, sendo usado neste setor antes mesmo de sua formalização oficial pelo ITU em 1992. Desde esse ano, a linguagem é mantida ativamente pela comunidade internacional.

Com relação ao uso do MSC, pode-se classificá-lo em duas categorias quanto ao seus elementos (i.e. constructos): *Basic Message Sequence Charts* (bMSC), ou *High-Level Message Sequence Charts* (hMSC).

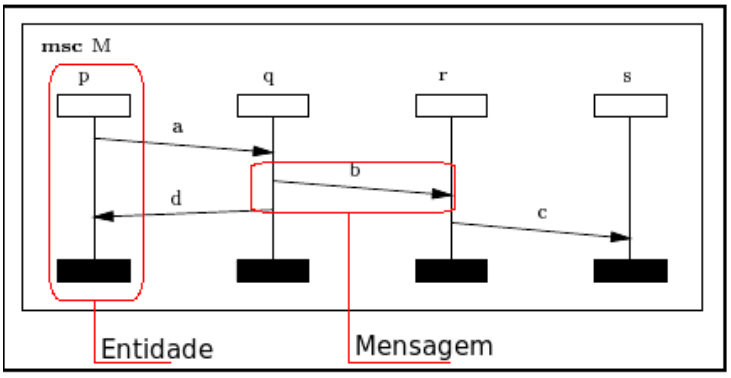

Figura 2.1: Exemplo de um diagrama MSC.
Mauw, Reniers e Willemse (2001)

2.3.1.1 *Basic Message Sequence Charts*

Um bMSC possui como principais constructos as linhas verticais que representam as linhas de tempo das entidades e os arcos que conectam essas linhas, as mensagens. O diagrama MSC, descrito na Figura 2.1, identifica as entidades **p**,**q**,**r** e **s**, onde percebe-se que a instância **p** inicia enviando uma mensagem, denominada **a**, para a instância **q**, a qual recebe essa mensagem. Em Mauw, Reniers e Willemse (2001), pode-se encontrar mais detalhes sobres constructos disponíveis no bMSC.

As mensagens em um MSC são consideradas assíncronas, onde a ordem de ocorrência das mensagens não é observada de forma sequencial, o que significa que as ações de envio de mensagens estão vinculadas à recepção das mesmas. Na definição de mensagem, uma recepção deve ocorrer antes de um envio. Entre os eventos de envio e recepção de uma mensagem, outros eventos como estes podem ocorrer. Isso é perceptível na Figura 2.1, onde observa-se que após a recepção da mensagem **a** pela instância **q** uma outra mensagem, denominada **b**, será enviada. A recepção de **a** e o envio de **b** tem sua ocorrência nesta ordem por estarem na mesma linha de tempo. Posteriormente ao envio da mensagem **b**, dois eventos podem acontecer: a recepção da mensagem **b** ou o envio da mensagem **d**. Interpretações quanto aos detalhes matemáticos podem ser encontradas em (MAUW, 1996).

2.3.1.2 *High-Level Message Sequence Charts*

Um diagrama hMSC pode ser entendido a partir de um MSC-*graphs*. Um MSC-*graphs* é uma estrutura que permite concatenar múltiplos cenários MSC na forma de grafos (Figura 2.2). Neste caso, os nós que constituem o grafo são os diversos cenários MSC: **M1**, **M2** e **M3**. O constructor representado por um hexagonal que conecta os vários cenários é um condicional, sendo usado para indicar escolha ou herança entre os diagramas MSCs (ALUR; YANNAKAKIS, 1999).

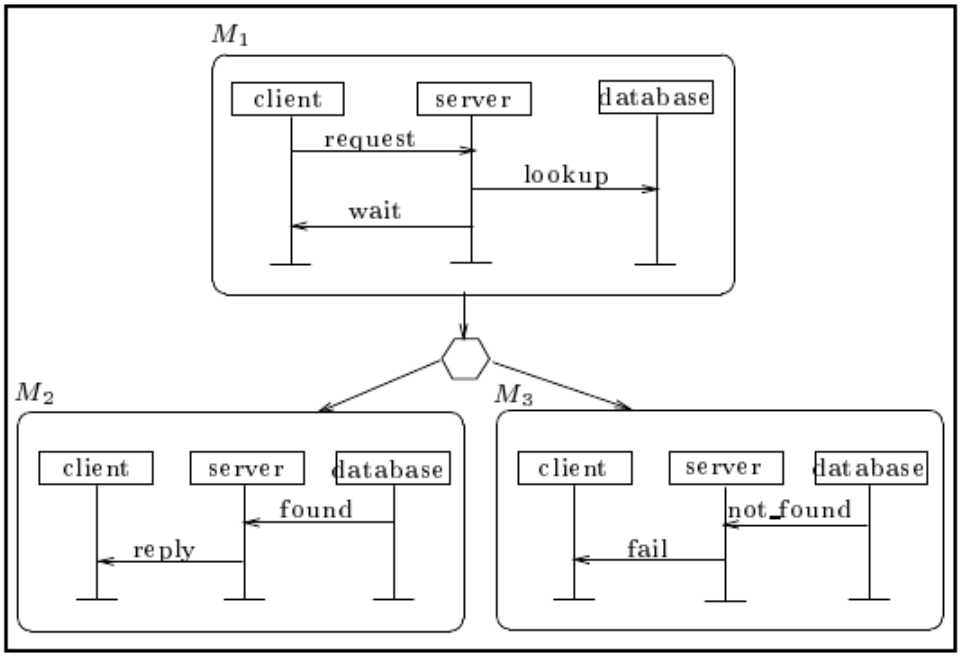

Figura 2.2: Um exemplo de MSC-*graphs*.
Alur, Etessami e Yannakakis (2003)

O hMSC estende o conceito do MSC-*graphs* por oferecer um melhoramento da estruturação dos vários cenários. Continua a existir os condicionais, mas agora um nó pode originar um novo MSC-*graphs* de forma recursiva ou um cenário MSC (ALUR; YANNAKAKIS, 1999).

Na Figura 2.3 observa-se um hMSC, com os três nós **Mi**, **Mb** e **Mf**. O nó **Mb** corresponde ao MSC-*graphs*, enquanto os nós **Mb** e **Mf** são cenários MSC, como observado na Figura 2.3.

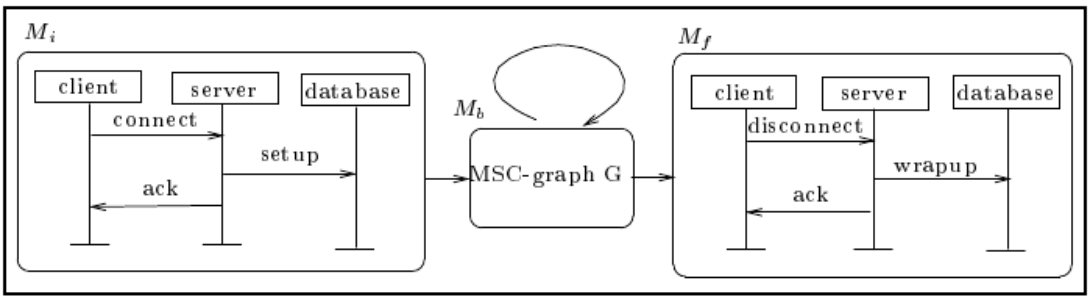

Figura 2.3: Um exemplo de hMSC.
Alur, Etessami e Yannakakis (2003)

2.3.2 Rede de Petri

Redes de Petri é um linguagem formal que descreve fluxo de informações, sendo usada como ferramenta para analisar modelagens. As Redes de Petri são eficientes na análise e simulação de sistemas dinâmicos com comportamento concorrente e não-determinístico.

Contudo, utilizá-lo diretamente na modelagem de sistemas costuma ter como conseqüência a geração de modelos extensos e complexos. A especificação de sistemas através de Redes de Petri é dificultada em razão de não haver mecanismo ou notação que permita especificar a estrutura do sistema, tais como relações de associação, agregação e herança entre seus elementos (DÖLL; STADZISZ, 2002). Em Peterson (1977), encontra-se uma discussão aprofundada sobre Redes de Petri.

Os constructos de uma Rede de Petri são:

- Transição: representa as transformações que ocorrem no sistema. Sua representação gráfica são as barras.
- Lugares: representa os estados que o sistema pode alcançar. Sua representação gráfica são os círculo não preenchidos

As Redes de Petri são interpretadas como grafos, em que Lugares e Transições são vértices interligados por arcos direcionais como apresentado na Figura 2.4. Os arcos representam a relação entre Lugares e Transições, pré-condições ou condições. Através da inferência dessas relações, ações podem acontecer. As execuções numa Rede de Petri são demonstradas com a mudança das marcas, que são retiradas de alguns Lugares e colocadas em outros. Quando a rede é executada, as Transições retiram as marcas de entrada e as colocam na saída. As ações na Rede de Petri são controladas pela quantidade de marcas e a distribuição das mesmas nos Lugares (PÁDUA et al., 2004).

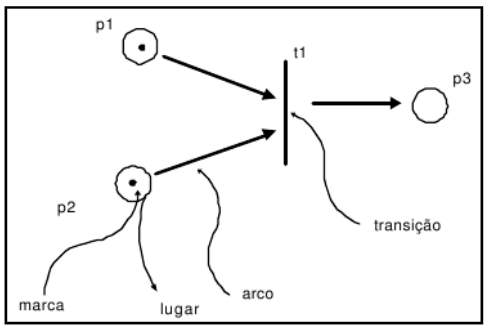

Figura 2.4: Um exemplo de Rede de Petri.
Guimarães (2000)

A execução da Rede de Petri, Figura 2.4, ocorre quando a pré-condição da Transição **t1** é satisfeita. Esta Transição só pode ser acionada quando os dois Lugares, **p1** e **p2**, à esquerda, tiverem pelo menos uma marca. No momento em que a pré-condição é válida a Transição **t1** é acionada e uma marca é passada para o próximo lugar, **p3**. O resultado da ação descrita é apresentado na Figura 2.5. Em Guimarães (2000) encontram-se mais exemplos sobre execução de Redes de Petri.

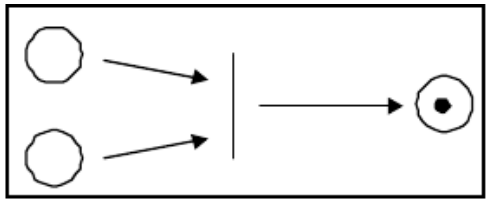

Figura 2.5: Execução de uma Rede de Petri.
Guimarães (2000)

2.3.3 *Calculus of Communicating Systems*

O *Calculus of Communicating Systems* (CSS) consiste numa linguagem formal baseado na álgebra de processos, criada por Robin Milner, por volta 1980. É voltada para descrever e validar a comunicação e a concorrência dentro de sistemas. Esta linguagem utiliza-se dos conceitos das leis algébricas para executar validações (ZHAO et al., 2009) (CARVALHO; FELIX; PETRUCCI, 2006).

O CSS possui processos e eventos como forma de modelar o comportamento do sistema alvo. Os processos representam o comportamento padrão de algum objeto, e eventos ou ações atômicas iniciadas pelo sistema ou por seu ambiente (CARVALHO; FELIX; PETRUCCI, 2006).

Um conceito fundamental em CCS é o processo, descrito por equações, que representam o

comportamento padrão dos objetos do sistema, os quais interagem entre si, criando novos processos. Outro conceito importante é o de eventos, que são ações atômicas, e conceitualmente são responsáveis por criar a comunicação entre os processos (CARVALHO; FELIX; PETRUCCI, 2006). Este conceito é similar ao definido no *Communicating Sequencial Processes* (CSP).

Uma discussão mais detalhada sobre CCS pode ser encontrada em Zhao et al. (2009).

2.3.4 *Communicating Sequencial Processes*

O *Communicating Sequencial Processes* (CSP) foi descrito pela primeira vez por Hoare em 1978. Consiste numa linguagem formal que modela o comportamento em sistemas concorrentes, e desde sua definição vêm sendo agregadas várias características. Foi desenvolvido com base na álgebra de processos, e faz uso de suas leis para realizar verificações de propriedades (FERREIRA, 2006).

A modelagem do CSP tem seu foco na descrição de processos, em que um sistema é representado como um conjunto de processos (FISCHER; OLDEROG; WEHRHEIM, 2001). Segundo Ferreira (2006), um processo é uma equação que pode ser encarada como um evento, um operador da linguagem, ou mesmo outros processos, os quais correspondem a uma parte do sistema. É possível inferir pela definição de processos que estes podem se aglutinar e compor processos maiores e mais complexos. Estes processos podem estar ocorrendo em paralelo e em comunicação.

A comunicação envolve a troca de mensagens entre os processos que constituem o sistema, e esse canal de comunicação se dá através de eventos. Um evento está relacionando processos, ou um processo com o sistema ao qual está inserido. Com relação ao tempo, os eventos são considerados instantâneos, e não há controle de ocorrência.

Um melhor detalhamento sobre a sintática e semântica da linguagem pode ser encontrado em Hoare (1978).

2.4 *Linear Temporal Logic*

Linear Temporal Logic (LTL) consiste numa lógica modal temporal utilizada para exprimir propriedades, e realizar análises. No LTL, o tempo não é mencionado explicitamente, mas entendido como uma seqüência de estados, *traces* ou caminhos. Nesta lógica orientada a estados, pode-se codificar fórmulas para percorrer os caminhos (CUNHA, 2005) (MALACARI, 2004).

O LTL representa as informações na forma de um grafo de acessibilidade denominado estrutura de *Kripke*. Uma estrutura de *Kripke* é definida como sendo uma tupla (S, i, R, L). Nesta tupla, o primeiro item refere-se a um conjunto finito de estados, onde **i** corresponde a seu estado inicial. O próximo elemento descreve uma relação de transição, e o último elemento consiste numa função (CUNHA, 2005).

Um exemplo de estrutura de *Kripke* encontra-se na Figura 2.6. Esta estrutura representa uma variável x, que possui dois estados s_1 e s_2, sendo s_1 o estado inicial. As relações de transição são dada por $(s_1,s_1),(s_1,s_2),(s_2,s_1)$. As funções definidas para este exemplo são $s_1 \mapsto 1$ e $s_2 \mapsto 0$.

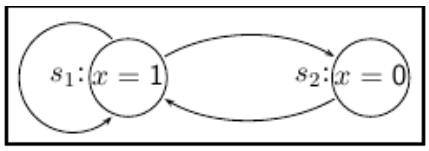

Figura 2.6: Exemplo de Estrutura de *Kripke*.
The University of Manchester (2003)

Na utilização do LTL, é preciso definir proposições atômicas da lógica. O resultado da verificação das proposições no sistema são os *traces* em que a proposição encontra-se válida (CUNHA, 2005). Caso não seja encontrado nenhum *trace*, a proposição não é descrita numa seqüencia válida de estados presentes no sistema.

2.5 *Process Mining*

A técnica do *Process Mining* permite que análises de processos sejam realizadas baseados em registros de eventos deste processo. Esta técnica auxilia a verificação de *workflow*, pelo fato dos *log* de eventos corresponderem ao fluxo de informação do sistemas (van der Aalst; WEIJTERS, 2004).

Os arquivo de *log* gerados baseiam-se em descrições de eventos. Estes arquivos descrevem os eventos seguindo um padrão (van der Aalst; WEIJTERS, 2004) (LASSEN; van Dongen; van der Aalst, 2007), onde:

- cada evento refere-se a uma ação bem-definida;
- cada evento refere-se a uma instância do processo;
- cada evento tem uma referência quanto a ordem de acontecimentos.

Com o uso desta técnica, pode-se realizar inferências baseadas nos arquivos de *log* de eventos, testando a conformidade do modelo, ou seja, verificando se a modelagem é condizente com o arquivo de *log*. E os arquivos de *log* podem ter os eventos ordenados de forma parcial, total, assíncrono ou síncrono.

2.5.1 ProM e ProMImport

O ProM[1] é um *framework* desenvolvido em JAVA® com o intuito de prover o uso do *process mining*. Como o *process mining* necessita do registro de eventos, este software manipula uma estrutura própria de descrição de *log*, conforme encontra-se em van Dongen et al. (2005). Este esquema utiliza o formato estendido do XML desenvolvido especificamente para armazenar este tipo de informação, denominado MXML.

O ProM dá ênfase as funcionalidas de análise e conversão. Estas funcionalidades são adicionadas através de *plugins*, seguindo orientação de sua arquitetura. As funcionalidades providas podem ser classificadas em:

- *Mining*: implementam alguns algoritmos para obtenção de informação por mineração de dados.
- Exportação: implementa a funcionalidade ”salvar como”e deste modo propicia salvar em vários formatos de arquivos.
- Importação: implementa a funcionalidade de abrir os objetos exportados.
- Análise: implementa alguma propriedade de análise. Por exemplo, há um *plugin* de análise, LTL, que compara um *log* de eventos e um modelo, realizando testes de conformidade.
- Conversão: executa as conversões entre diferentes formatos de dados, por exemplo, Redes de Petri para EPCs (*Event-driven Process Chain*).

O ProMImport[2], também desenvolvido em JAVA®, baseia-se na característica do ProM de ser orientado a *plugin*. Esta ferramenta converte diversos tipos de entrada de arquivos em um formato compatível com o ProM, fazendo uso de vários *plugins*.

[1] http://is.tm.tue.nl/~cgunther/dev/prom/
[2] http://is.tm.tue.nl/~cgunther/dev/promimport/

3 *Metodologia Aplicada*

Neste capítulo, será relatado todo o processo executado para alcançar os resultados. Num primeiro momento, se fará um comparativo entre as linguagens abordadas no trabalho e depois será apresentada uma abordagem para uso de metodos formais no desenvolvimento de software.

3.1 Análise sobre Linguagens Formais

Em Pressman (2001) e Sommerville (2003) encontram-se capítulos que tem como objetivo tratar sobre métodos formais e sua inserção na Engenharia de Software. A aplicação de métodos formais é um tema controverso nesta engenharia, contudo sua adoção se torna importante em sistemas que sejam críticos, ou seja possuem características concorrentes (HALL, 1990).

Adotar uma abordagem formal faz necessário o uso de uma linguagem de especificação formal. Segundo MacColl e Carrington (1997), os métodos formais podem ser classificados de acordo com o aspecto em que se propõem: modelo, propriedades ou comportamento. Métodos formais baseados em modelo definem o comportamento do sistema através da construção de modelos usando estruturas matemáticas, como por exemplo os conjuntos, as quais são adequadas à especificação de software que envolva estruturas de dados complexas, uma vez que os tipos de dados são modelados explicitamente. Como exemplo de linguagens para este caso temos Z, VDM.

Métodos baseados em propriedades modelam tipos de dados, de forma implícita, para definir o comportamento do sistema através de um conjunto de propriedades. Nestes casos, pode-se utilizar linguagens como OBJ e Larch. Os métodos baseados em comportamento tem como foco definir sistemas em termos de mudança de estados em vez de tipos de dados. Estes métodos fazem uso de linguagens como Redes de Petri, CCS, CSP e MSC.

Enquanto os métodos baseados em modelos e propriedades tem seu foco centrado nas estruturas de dados, os métodos baseados no comportamento tem seu foco na seqüência de estados que o sistema vai passando ao longo do tempo (MACCOLL; CARRINGTON, 1997).

Como o trabalho visa a adoção do formalismo no desenvolvimento de software, métodos baseados em comportamento se tornam mais atrativos, visto que priorizam a análise das mudanças no sistemas. Com o uso deste tipo de métodos, a preocupação maior é encontrar situação de falha ou problemáticas dentro dos sistemas, não dando ênfase a estrutura de dados.

Com situações de falha ou problemáticas, considera-se os momentos em que o sistema encontra-se num estado de *dealock*, *livelock*, *safety* e *liveness*. Um *deadlock* num software acontece quando um conjunto de processos fica esperando por um evento que somente outro processo pertencente ao conjunto poderá fazer acontecer. O *livelock* é uma particularidade do *deadlock*, em que ocorre o impasse entre um conjunto de processos, embora neste caso seus estados possam ser alterados em resposta a eventos entre os mesmos, persistindo o conflito.

Já *safety* e *liveness* especificam tipos de propriedade que deve ter o sistema. *Safety* impõe que uma propriedade deve ser válida em todas as execuções, e o *liveness*, uma propriedade deve ser válida em pelo menos uma situação eventualmente em cada execução.

Comparativamente, linguagens como CCS e o CSP são fundamentadas na álgebra de processos e, deste modo, o sistema é modelado com um conjunto de processos que se relacionam através de eventos, diferentemente do MSC, em que a modelagem é feita observando a troca de mensagens entre partes do sistema, entidades. Em uma Rede de Petri, entretanto, o sistema é descrito como um conjunto de mudanças de estados em que o sistema, e as entidades que o constituem, podem alcançar.

Linguagens como MSC, Rede de Petri e CSP apresentam formas de visualização gráfica dos modelos gerados, embora o CSP não tenha esta característica bem definida em comparação com as outras duas (FERREIRA, 2006). O MSC e Rede de Petri apresentam características mais intuitivas para o manuseio, devido a sua definição gráfica que se assemelha com diagramas da UML. Contudo, o MSC traz vantagens quando comparado a Redes de Petri no que se refere a modelagem.

No MSC, a modelagem é feita através de cenários de partes específicas do sistema, facilitando este processo. Somente com a compilação destes vários cenários produzidos, é possível obter a completude. A possibilidade de descrever partes do sistema através de cenários garante ao desenvolvedor uma simplificação do processo de modelagem, pois este pode apenas desejar realizar testes em determinadas situações. Entretanto, como desvantagem, tem-se uma quantidade muito pequena de ferramentas que trabalham com esta linguagem, além da análise dos modelos comportamentais elaborados ser complexa devido à necessidade de correlacionar os cenários (HAUGEN, 2004). Por outro lado, as Redes de Petri possuem diversas ferramentas de validação, análise e simulação de sistemas dinâmicos com comportamento concorrente

e não-determinístico (DÖLL; STADZISZ, 2002), mas a modelagem com essa linguagem para sistemas extensos torna-se trabalhosa.

Na Figura 3.1, encontram-se quadros comparativo das vantagens e desvantagens das linguagens elencadas no trabalho, de forma resumida.

Vantagens	Desvantagens
Descreve a troca de mensagens entre entidades	Pequena quantidade de ferramentas
Modelagem facilitada	Difícil realizar análise
Linguagem gráfica	
Similaridade com diagrama UML	
Em constante desenvolvimento	

(a) MSC

Vantagens	Desvantagens
Descreve mudanças de estado	Modelagem dificultada
Análise facilitada	Modelos complexos e extensos
Linguagem gráfica	
Similaridade com diagrama UML	
Muitas ferramentas	

(b) Rede de Petri

Vantagens	Desvantagens
Descreve mudanças através de processos	Representação gráfica insuficientes
Madura	

(c) CCS

Vantagens	Desvantagens
Linguagem gráfica	Representação gráfica insuficientes
Madura	
Descreve mudanças através de processos	

(d) CSP

Figura 3.1: Quadros comparativos das linguagens estudadas no trabalho.

Na revisão bibliográfica observou-se uma exemplo de solução desenvolvida para facilitar o uso de linguagens e métodos formais. A solução encontrada consiste em converter linguagens formais, neste caso a conversão deu-se de CSP para UML-RT (UML for Real Time), pois trabalhar com UML é mais prático que CSP (FISCHER; OLDEROG; WEHRHEIM, 2001) (FERREIRA, 2006). Partindo desta princípio, percebeu-se a possibilidade de utilizar as principais características providas pelo MSC e Redes de Petri desenvolvendo uma idéia similar. Neste caso, modelar utilizando-se do MSC e transcrever um conjunto de diagramas MSC, em uma Rede de Petri e assim fazer uma análise. Desta forma, tem-se a vantagem na descrição de cenários obtida com o uso do MSC, e as facilidades da Rede de Petri com a análise, além do fato de ambas serem mais intuitivas que o CSP e o CCS.

Na etapa de revisão bibliográfica, foi encontrada uma técnica (ver seção 3.2) em concordância com as idéias propostas no trabalho. Esta técnica descrita em Lassen, van Dongen e van der Aalst (2007), baseia-se no uso de *log* de eventos, pois o fluxo do comportamento descrito em MSC, pode ser transcrito em forma de arquivos de *log*. Este arquivo de *log* armazena os eventos de comunicação entre as entidades que constituem o sistema. Neste arquivo de *log*, estariam as descrições dos cenários, os quais seriam compilados de forma a descreverem todo o

sistema. O uso de *log* para realizar análises é denominado de *process mining*. Ainda conforme a técnica, de posse de um conjunto de MSCs haveria uma conversão para outra linguagem e posteriormente análise deste modelo.

3.2 Método Proposto

Nesta seção descreve-se detalhadamente o método proposto, o qual faz uso das três etapas que fazem parte da técnica desenvolvida por Lassen, van Dongen e van der Aalst (2007), agregando-se a isto a ferramenta desenvolvida (ver Figura 3.2). As etapas estão enumeradas a seguir:

1. Modelagem do comportamento do sistema na ferramenta desenvolvida;
2. Conversão de conjunto de diagramas MSC para arquivos de *log* de eventos no ProMImport;
3. Análise e validação do comportamento no ProM;

Na modelagem, os cenários MSC são descritos num esquema XMI. Como em Lassen, van Dongen e van der Aalst (2007), não havia uma descrição de como esses esquemas eram gerados, foi desenvolvida uma ferramenta, editor gráfico MSC, que auxiliasse nesta tarefa. Depois de modelar no editor, este exporta o diagrama elaborado no padrão XMI para que este possa ser utilizado na próxima etapa.

A conversão é uma etapa importante porque o *process mining* é executado na ferramenta ProM, uma ferramenta criada para análise de *workflow*. Contudo o esquema XMI não é compatível com arquivos de entrada do ProM, sendo assim, é preciso realizar a conversão do esquema gerado no editor. Esta etapa objetiva aumentar a compatibilidade do ProM com outras descrições de dados de entrada, e faz uso da ferramenta ProMImport, que tem como funcionalidade fazer a conversão de diversos arquivos de entrada para formatos aceitos pelo ProM, MXML.

A etapa de verificação é executada no ProM, e os cenários MSC podem ser visualizados em Redes de Petri. Verificou-se que com a apliacação ProM, seria possível não só utilizar Redes de Petri para análises, mas que esta ferramenta dispunha de vários *plugins* para esta tarefa, e neste trabalho serão apresentadas duas destas formas.

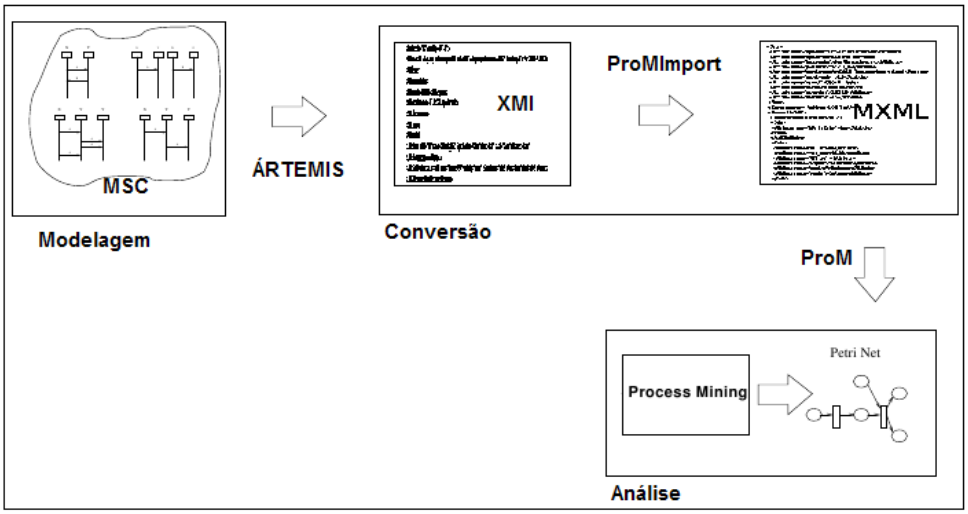

Figura 3.2: Idéia geral do trabalho.

O método desenvolvido pode ser utilizado tanto na etapa de especificação do projeto, em que requisitos possam ser convertidos em cenários MSC e validados, quanto aplicado na verificação com os objetos que compõem o sistema, e a troca de mensagem entre eles. Outra forma é utilizar em vez de objetos, componentes do sistema e a relação entre eles.

3.2.1 Modelagem

Esta etapa é realizada na ferramenta desenvolvida, sendo a confecção dos cenários de comportamento. É importante que o usuário faça uma modelagem coerente para evitar ambigüidades e conseqüentemente interferir na etapa de análise. Um exemplo é visto na Figura 3.3.

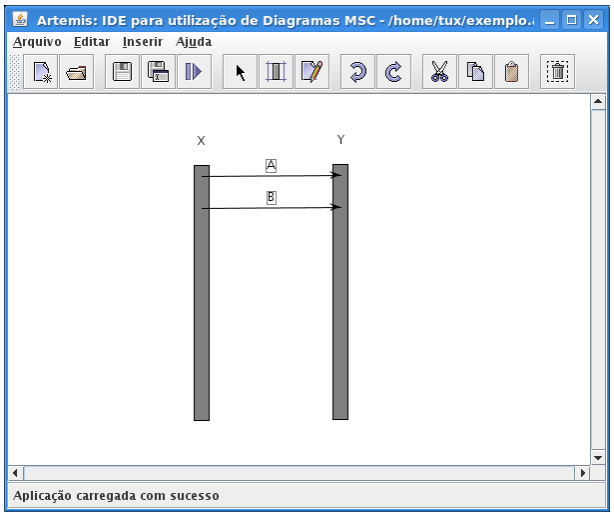

Figura 3.3: Um exemplo de modelagem.

Para realizar a modelagem, é preciso conhecer as funcionalidades presentes no menu. Na Figura 3.4, nota-se a barra de botões, em que encontra-se as funcionalidades principais da ferramenta. Da esquerda para direita, tem-se as seguintes funções: novo arquivo, abrir arquivo, salvar, salvar como, exportar, selecionar, criar linha de tempo, adicionar comentário, desfazer, refazer, cortar, copiar, colar e excluir. Observa-se pelo menu que a ferramenta simplifica o acesso do usuário com os recursos de modelagem.

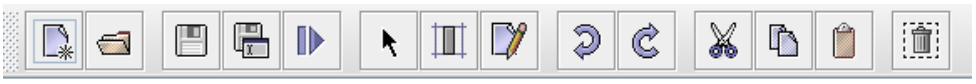

Figura 3.4: Barra de botões da aplicação.

Da modelagem em questão, Figura 3.5, já é possível notar a ocorrência de dois *traces* de execução, ou seja, duas seqüências de comportamento que podem ocorrer. A observação desses *traces* é possível por tratar-se de apenas duas mensagens num único diagrama MSC.

1. send(X,Y,A) . send(X,Y,B) . receive(X,Y,A) . receive(X,Y,B)
2. send(X,Y,A) . receive(X,Y,A) . send(X,Y,B) . receive(X,Y,B)

Figura 3.5: *Traces* de execução observados na modelagem.

3.2.2 Conversão

Feita a modelagem, ainda utiliza-se o Ártemis para gerar uma outra representação desses diagramas, o formato XMI. O processo de conversão ainda não se encontra finalizado, o passo seguinte é traduzir este esquema, num formato de arquivo de log.

Esta etapa é feita utilizando outra ferramenta, o ProMImport versão 3.2, e como já mencionado, através do *plugin* XMI. A entrada do ProMImport é o arquivo XMI exportado pelo Ártemis referente a modelagem e a saída será um arquivo MXML.

Com relação ao uso do ProMImport é importante ressaltar que houveram alguns problemas, quanto ao seu uso, primeiramente foi precisar usar o código-fonte para executar a ferramenta. O plugin XMI não era carregado por necessitar de APIs externas que não se encontravam no pacote com o código-fonte.

O plugin do XMI encontrava-se na versão 4.0, até aquele presente momento, a mais atual do ProMImport. Contudo, esta versão não estava completamente implementada, pois houve uma reestruturação do código. Por este motivo, fez-se uso da versão 3.2, neste caso, foi preciso

apenas baixar as APIS externas necessárias a poder carregar o plugin XMI. As APIs foram: jaxen-core, jaxen-jdom, jdom, saxpath.

3.2.3 Análise

A última etapa é a análise/validação do comportamento modelado com o ProM. De posse do arquivo MXML, contendo os log*s* de eventos, o ProM é capaz de visualizá-lo em diferentes linguagens formais como EPC e Redes de Petri e utilizar diversos plugins de análise.

A análise parte do princípio que a modelagem do cenário está coerente e que nenhuma ambigüidade tenha sido inserida. É necessário também que ela seja completa, com relação a situação que se deseja validar. Além de ter uma modelagem sem ambigüidades e completa, é preciso neste ponto saber o que se deseja analisar. Algumas vezes será preciso refazer a modelagem por esta não possibilitar abranger a análise desejada.

Sabendo-se o que deseja inferir, utiliza-se os recursos providos pelo ProM. Dos vários *plugins* disponíveis pode-se usar o LTL, ou visualizar a Rede de Petri da modelagem a fim de realizar a verificação. Com Rede de Petri, a análise com o ProM é feita sobre a imagem da rede correspondente ao *log* de eventos criado. A verificação é feita através da imagem gerada e da visualização das transições que são iniciadas e por quais lugares, isto é, procura-se observar a execução da rede. O processo para se obter a Rede de Petri é primeiro gerar uma EPC e convertê-la para uma Rede de Petri. Isso se deve ao fato que os *plugins* que utilizam Redes de Petri podem ter outros propósitos, como apenas mostrar onde há conflito e assim não permitindo uma visão geral do sistema.

Com o LTL, é possível utilizar as várias proposições que se encontra no ProM para realizar as verificações, sem a necessidade de criá-las. Apenas torna-se necessário conhecer quais elementos descritos nas proposições podem ser trabalhados. Como o MSC possui dois tipos de elementos principais, seus correspondentes em LTL, segundo o ProM, são: *person* para uma entidade MSC e *activity* para uma mensagem entre entidades. Deste modo, as proposições que fazem uso destes elementos podem ser utilizadas.

Escolhendo a proposições, coloca-se os valores nos parâmetros da fórmula. Na figura 3.6, encontra-se o ProM com o *plugin* LTL. Nota-se uma caixa de seleção para as fórmulas, uma breve explicação a respeito da proposição e por fim os campos para configurar os parâmetros para análise. Nas análises, feitas utilizando LTL, os parâmetros serão as mensagens trocadas entre as entidades.

Após a escolha da proposição e configurados os parâmetros é realizada a inferência da

proposição. Na Figura 3.7, apresenta-se a resposta obtida no ProM. Nesta janela, encontram-se duas abas, *correct process instances* e *incorrect process instances*, respectivamente. Se a troca de mensagens definida for encontrada, a aba *correct process instances* informa quantas vezes foi encontrada esta seqüencia e mostra um diagrama com esta seqüencia ao lado; caso contrário, a aba *incorrect process instances* informará os erros encontrados baseados na proposição declarada inicialmente.

A escolha entre qual dos métodos utilizar fundamentou-se no requisito principal do trabalho, tornar a análise o mais fácil possível. Nos casos de teste que possuem pouca troca de mensagem e não tornam a Rede de Petri muito extensão, é aconselhável seu uso, enquanto nos casos em que se obtém uma Rede de Petri extensa, é preferível adotar o LTL. No caso de o LTL não possuir uma proposição de consulta satisfatória para o usuário, este pode utilizar a Rede de Petri.

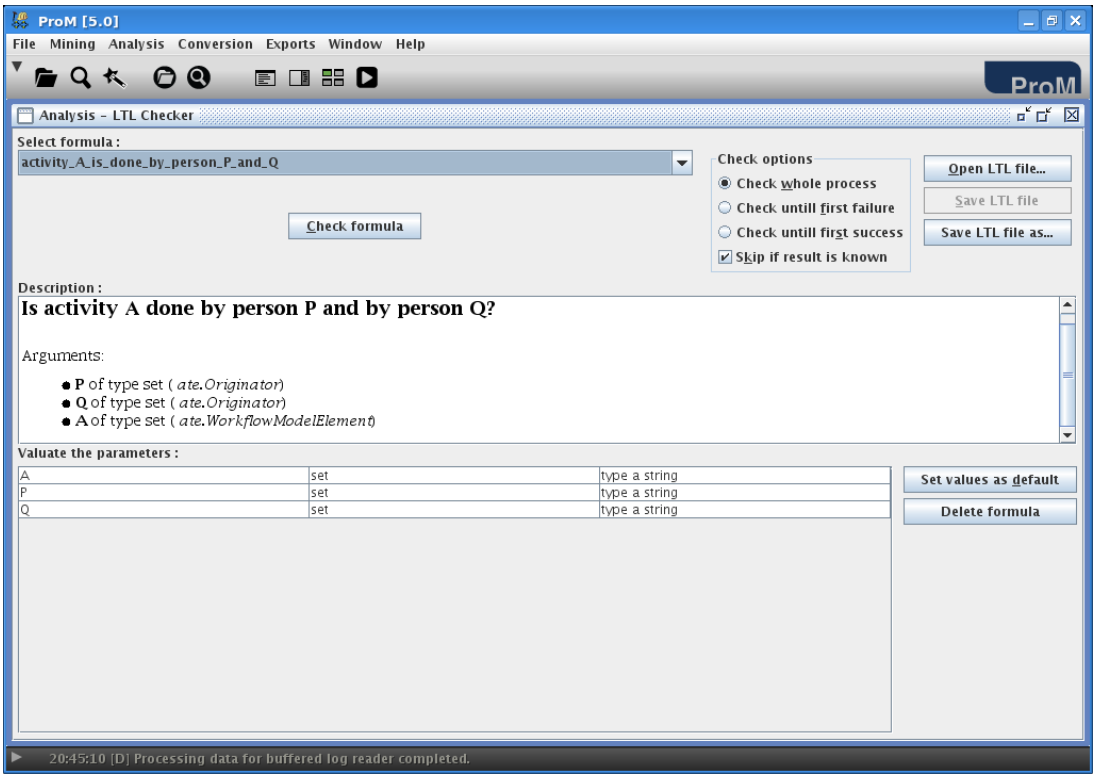

Figura 3.6: Janela do plugin do LTL no ProM.

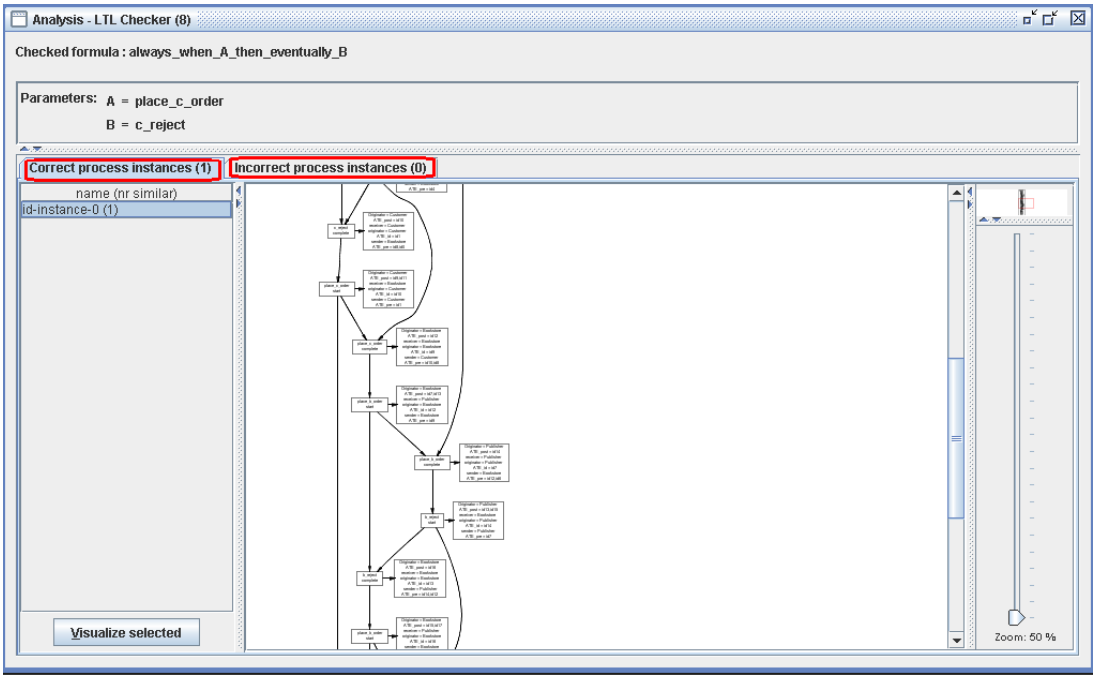

Figura 3.7: Janela do plugin do LTL no ProM após analise.

4 *Discussão dos Resultados*

Neste capítulo serão discutidos aspectos referentes aos produtos regerados: ferramenta e método. Com relação ao método, serão apresentados dois casos de aplicação, um utilizando Rede de Petri na análise e outro com o LTL.

4.1 Ferramenta Desenvolvida

A ferramenta foi desenvolvida na plataforma JAVA®, e denominada Ártemis. Foram utilizadas como APIs complementares o JHotDraw[1] e o JDOM[2]. Enquanto o JHotDraw é usado na confecção da interface gráfica, o JDOM é utilizado na geração de arquivos de saída da ferramenta. A função desta ferramenta é propiciar ao desenvolvedor de software a adoção da forma gráfica do MSC e desta forma aplicar análises formais, com relação ao comportamento do software.

Como editor MSC, a aplicação apenas trata dos constructos básicos desta linguagem, ou seja bMSC. Isto se deve ao fato do *plugin* XMI do ProMImport se restringir a manipular este conjunto.

Na linguagem MSC, não existe limite de mensagens nas entidades, mas na prática é inviável implementar tal requisito. Das observações feitas em alguns artigos que embasarem este trabalho - Dong e Sun (2007), Harel e Kugler (2002), Harel e Thiagarajan (2003), Lassen, van Dongen e van der Aalst (2007), Mauw, Reniers e Willemse (2001), Alur, Etessami e Yannakakis (2003), Mauw (1996) - e dos exemplos que são descritos nestes artigos, verificou-se que em sua grande maioria a quantidade de mensagens variavam entre duas e oito, e com base nestes dados e no fato da API já dar o suporte a seis ligações, definiu-se o limite de seis mensagens para a ferramenta.

[1] www.jhotdraw.org/
[2] www.jdom.org/

4.1.1 APIs Utilizadas

Uma descrição detalhada sobre as APIs utilizadas na confecção da ferramenta e as justificativas de suas escolhas é relatada a seguir.

4.1.1.1 JHotDraw

Na tentativa de descobrir uma API gráfica capaz de prover as funcionalidades desejadas no desenvolvimento da ferramenta em questão, foram selecionadas para análise as seguintes: JHotDraw, Java2D. Como pontos a serem avaliados tem-se: flexibilidade, integração com o toolkit Swing/AWT da plataforma Java e dispor de vários recursos que facilitem o tratamento de objetos gráficos em aplicações.

A API do Java2D, embora esteja presente entre os pacotes básicos da plataforma JAVA®, não garantia recursos para o tratamento de objetos da forma desejada, tornado inviável seu uso. Enquanto o *framework* JHotDraw possibilita inúmeras vantagens no desenvolvimento da aplicação como a criação de janelas, conjuntamente com o toolkit Swing, bem como garante a criação e manipulação de objetos gráficos dinâmicos de forma facilitada. Dentre suas características, propicia a criação de ferramentas que alteram os estados desses objetos gráficos sem muitos problemas. Um ponto forte deste *framework* é o uso de vários padrões em seu desenvolvimento, que forçam o desenvolvedor a adotar padrões quanto do uso da API. Uma descrição completa do *framework* é encontrada em Savolskyte (2004).

Em muitos aspectos o uso do *framework* ajudou no desenvolvimento, como na geração das figuras, criação de arcos pra relacionar figuras, definição de ações como copiar/colar, desfazer/fazer, formato próprio para salvar e abrir arquivos existentes. A integração com o toolkit Swing/AWT também facilitou na construção da interface gráfica da ferramenta. Apresentando apenas um problema: figuras agrupadas perdem a propriedade de realizarem ligação com outros figuras ou agrupamento, deste modo não se poderia criar figuras complexas e modelar a troca de mensagens. Como ocorreu esse problema, a solução foi manipular as figuras simples, quando necessário, agregar somente texto a essas figuras.

4.1.1.2 JDOM

O JDOM oferece maneiras de manipular arquivos XML de forma simplificada, sendo totalmente desenvolvida em JAVA®, e fornece seus próprios meios para leitura e escrita para este tipo de arquivo. O uso desta API ocorreu da necessidade de gerar um arquivo XMI a partir dos diagramas MSC. Foi preciso criar uma estrutura intermediária simples que relacionasse cada

mensagem com a entidade que a origina e que a recebe.

Este esquema guarda os nomes das entidades relacionadas e o nome da mensagem. O esquema XML adotado é descrito na Listagem 4.1, observando que, neste caso, tem-se apenas um nó, ou seja, apenas uma mensagem trocada. Este único nó descreve o nome da mensagem, *translate*, trocada pelas entidades *MSC01* e *MSC02*, sendo respectivamente o originador e o receptor.

Listagem 4.1: Esquema XML desenvolvido

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tmp>
        <message>
                <id>0</id>
                <name>translate</name>
                <start>MSC01</start>
                <end>MSC02</end>
        </message>
</tmp>
```

4.1.2 Arquitetura do Software

A arquitetura da aplicação pode ser compreendida em três pacotes principais ((Figura 4.1), um referente as classes que fazem uso da API do JHotDraw, pacote jhotdraw; uma relacionada a interface gráfica, pacote view, e um outra referente ao tratamento com os arquivos, pacote io.

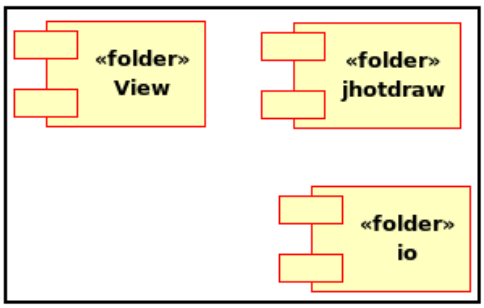

Figura 4.1: Diagrama de componentes do Ártemis.

O pacote view apenas define a estrutura dos botões e menus, e as ações ligadas a estes itens. Este pacote possui uma relação intrínseca com o pacote jhotdraw, pois muitas das ações dos menus e botões, bem como a área de manipulação dos diagramas são descritas por classe que herdam métodos do JHotDraw.

O próximo pacote refere-se às classes que manipulam diretamente instâncias geradas a partir da API do JHotDraw. Realizam a tarefa de criação dos elementos gráficos, como as entidades MSC, comentários, e as suas manipulações. Há outras classes relacionadas a esta API que correspondem a ações do tipo copiar/colar, desfazer/refazer.

O último pacote consiste na criação dos arquivos utilizados pela aplicação. Existem quatro formatos de arquivos que foram adotados PNG, GIF, .art e XMI. Dois formatos para exportação em arquivos de imagem, sendo PNG e GIF. O formato .art, diferentemente dos formatos de imagem que não podem ser reutilizados pela ferramenta, é capaz de ser salvo e reaberto a qualquer momento pelo usuário. Este formato segue o padrão adotado pelo JHotDraw, pois utiliza de alguns métodos desta API.

Este formato, .art, é descrito detalhadamente na Tabela 4.1. Na primeira coluna da tabela tem-se o caminho das classes dos elementos necessários para construir as entidades gráficas, e na segunda coluna encontra-se uma descrição da funcionalidade das classes.

Tabela 4.1: Esquema do arquivo da Ártemis

Classe a ser Instânciada	**Descrição**
uefs.jhotdraw.artemis.core.editor.MSCDiagramImpl	Descrição inicial, obrigatória no inicio de qualquer arquivo.
uefs.jhotdraw.artemis.figures.LifeTimeImpl	Descrição das propriedades do elemento representativo da linha do tempo.
uefs.jhotdraw.artemis.figures.Text	Descrição das propriedades do texto referente ao elemento representativo da linha do tempo, sempre vêm posteriormente a definição do LifeTimeImpl.
uefs.jhotdraw.artemis.figures.MessageImpl	Descrição das propriedades do elemento representativo da mensagem trocada.

O último formato é o mais importante, porque adota o esquema proposto por Lassen, van Dongen e van der Aalst (2007). Somente através deste tipo de arquivo a técnica é realizada, pois assim é possível converter diagramas MSC para um conjunto de *logs*. Este formato consiste

num esquema de descrição XMI, o qual possui três tipos de nós principais:

- *UML2:Interaction.fragment*
- *UML2:Interaction.message*
- *UML2:Interaction.lifeline*

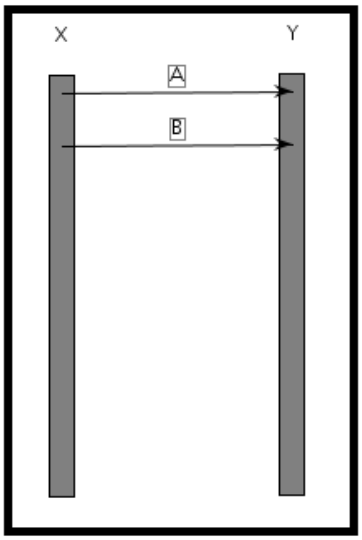

Figura 4.2: Cenário utilizado para descrição de esquema XMI.

Utilizando como base a Figura 4.2, na qual encontra-se descrito um cenário MSC, é apresentado o esquema XMI obtido deste caso (Listagens 4.2, 4.3 e 4.4). Neste cenário, constam apenas duas entidades trocando duas mensagens. O primeiro tipo de nó define a ocorrência dos eventos *send* e *receive* que ocorrem no diagrama MSC. Quando é definido um *send*, é associado a este evento a entidade que o executou, ocorrendo o mesmo com a ação *receive*. É criado um campo *xmi.idref* que é usado para associar esses dois tipos de evento. Na Listagem 4.2, encontra-se a descrição de um *send* e *receive* da modelagem. O nó *send* equivale ao envio da mensagem **A** pela entidade **X**. Já o nó *receive* corresponde a recepção da mensagem **A** pela entidade **Y**.

Listagem 4.2: Descrição de um nó *Interaction.fragment*

```
<UML2:Interaction.fragment>
<UML2:EventOccurence xmi.id="send0" name="" visibility="public"
    isSpecification="false">
<UML2:InteractionFragment.covered>
<UML2:Lifeline xmi.id="X" />
</UML2:InteractionFragment.covered>
<UML2:MessageEnd.sendMessage>
<UML2:Message xmi.idref="message0" />
</UML2:MessageEnd.sendMessage>
</UML2:EventOccurence>

<UML2:EventOccurence xmi.id="receive0" name="" visibility="public"
    isSpecification="false">
<UML2:InteractionFragment.covered>
<UML2:Lifeline xmi.id="Y" />
</UML2:InteractionFragment.covered>
<UML2:MessageEnd.receiveMessage>
<UML2:Message xmi.idref="message0" />
</UML2:MessageEnd.receiveMessage>
</UML2:EventOccurence>
```

O segundo nó descreve a associação dos eventos *send* e *receive*, representando deste modo a troca de mensagem entre as duas entidades MSC. Neste nó descreve-se o nome da mensagem. Na Listagem 4.3, observa-se a definição de uma mensagem trocada entre duas entidades. Esta mensagem corresponde à executada entre as entidades **X** e **Y**, denominada como **A**.

Listagem 4.3: Descrição de um nó *Interaction.message*

```
<UML2:Interaction.message>
<UML2:Message xmi.id="message0" name="A" visibility="public"
    isSpecification="false" messageSort="asynchCall">
<UML2:Message.argument>
<UML2:OpaqueExpression xmi.id="id3" visibility="public" isSpecification="
    false" body="A" language="java" />
</UML2:Message.argument>
<UML2:Message.receiveEvent xmi.idref="receive0">
<UML2:EventOccurrence />
</UML2:Message.receiveEvent>
<UML2:Message.sendEvent xmi.idref="send0">
<UML2:EventOccurrence />
</UML2:Message.sendEvent>
</UML2:Message>
```

No último nó, associa-se todos os eventos presentes nos diagramas, as mensagens *send* e *receive*, as suas respectivas entidades MSC. Observa-se na Listagem 4.4 a descrição dos eventos que foram executas pela entidade **X**. No cenário que se está utilizando, encontra-se a entidade **X**, a qual possui dois eventos *send* associados: *send0* e *send1*.

Listagem 4.4: Descrição de um nó *Interaction.lifeline*

```
<UML2:Interaction.lifeline>
<UML2:Lifeline xmi.id="MSC1" name="X" visibility="public" isSpecification="
    false">
<UML2:Lifeline.coveredBy>
<UML2:EventOccurence xmi.idref="send0" />
<UML2:EventOccurence xmi.idref="send1" />
</UML2:Lifeline.coveredBy>
</UML2:Lifeline>
```

4.2 Caso de Teste 01

Num primeiro momento será apresentado um caso básico (Figura 4.3). Neste teste, foi gerada a Rede de Petri da modelagem do cenário proposto, em que foi possível visualizar os *traces* de eventos que podem ocorrer.

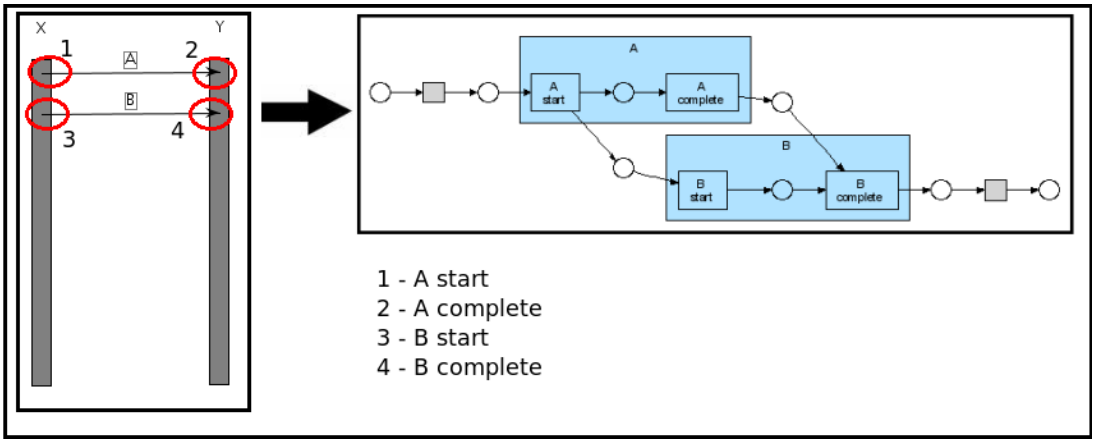

Figura 4.3: Modelagem do caso de teste e Rede de Petri.

Na Figura 4.3, encontra-se a modelagem feita no Ártemis, e a Rede de Petri gerada pela modelagem a partir do arquivo XMI. Pode-se notar os *traces* que são descritos anteriormente na Figura 3.5. A transição A *start*, correspondente ao envio da mensagem A pela entidade X que inicia a troca de mensagens. A partir deste ponto, duas situações podem acontecer, ou a mensagem A é recebida, transição A *complete*, ou a mensagem B *start* é enviada. Essa situação

é observável na Rede de Petri, pois a transição A *start* pode acionar tanto a transição A *complete*, quanto a transição B *start*. Dessa Rede de Petri também é possível inferir que a transição B *complete* só é acionada quando as transições A *complete* e B *start* já tenha acontecido.

Neste caso de teste, o modelo MSC e a Rede de Petri obtido estão em conformidade, pois a conversão foi realizada de forma satisfatória. Observa-se neste exemplo a seqüencia de eventos, mudanças de estado que o sistema possui e as condições para que isto ocorra.

4.3 Caso de Teste 02

Outro caso que está sendo utilizado é uma modelagem proposta no trabalho de Lassen, van Dongen e van der Aalst (2007), sobre uma livraria *online*. O sistema na modelagem contém as seguinte entidades:

- cliente(*Customer*): pessoa que quer comprar um livro.
- livraria(*Bookstore*): usado para o cliente escolher um livro para comprar. A livraria virtual sempre contata uma editora para ver se ele pode lidar com a ordem. Se a livraria não pode encontrar uma editora, a ordem do cliente é rejeitada. Se a livraria encontra uma editora o livro é entregue.
- editora(*Publisher*): pode ou não ter o livro que o cliente quer comprar.

As mensagens trocadas entre as entidades no sistema:

- *place_c_order*: cliente envia solicitação de compra para determinado produto.
- *place_b_order*: livraria verifica produto solicitado pelo cliente com editora.
- *b_reject*: resposta da editora a solicitação da livraria sobre um produto. No caso, o produto não foi encontrado.
- *c_reject*: resposta da livraria ao cliente sobre o produto. Como o produto não pode ser encontrado é retorno esta mensagem.

Com base na descrição da livraria, foram elaborados dois modelos para serem analisados. Na Figura 4.4, observam-se os dois diagramas MSC em que o cliente encomenda um livro a uma livraria. A livraria em seguida, tenta obter o livro de duas editoras até que desiste e informa ao cliente que não pode finalizar transação. Uma característica importante desta modelagem é

que a troca de mensagem para a editora é distinta, pois a editora pode variar, ao contrário das outras entidades encontradas na modelagem.

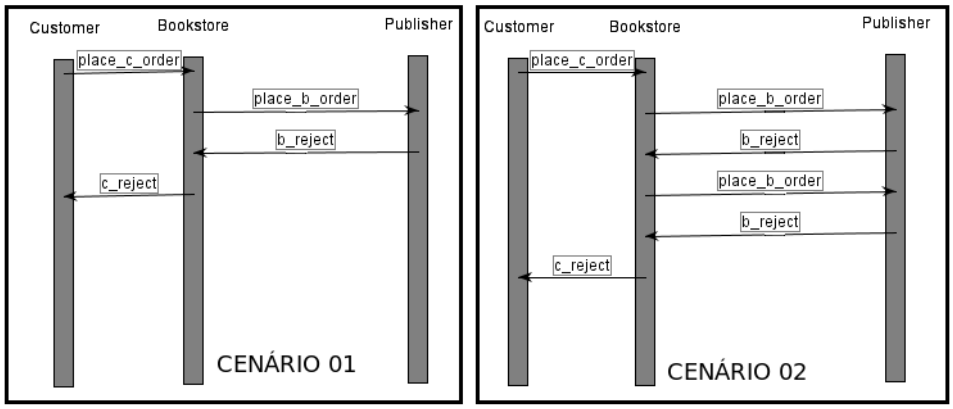

Figura 4.4: Modelagem da *bookstore* no Ártemis.

Como já mencionado, o passo seguinte à modelagem é definir o que deseja ser avaliado. A partir desta modelagem, foram consideradas três inferências a serem realizadas. Ao contrário do primeiro caso de teste, neste utiliza-se também uma verificação através do LTL, pois a Rede de Petri torna-se mais extensa e dificulta uma análise visual.

A primeira avalia se quando é iniciada uma consulta a editora, está pode ser finalizada mesmo com o produto não sendo encontrado. Neste caso, seria observada a seguinte ordem de mensagens: *place_b_order*, seguido *por b_reject* e finalizado por *c_reject*. Como a seqüência de mensagens definida foi encontrada a aba *correct process instances* informa quantas vezes foi encontrado e mostra um diagrama com esta seqüência ao lado (Figura 4.5).

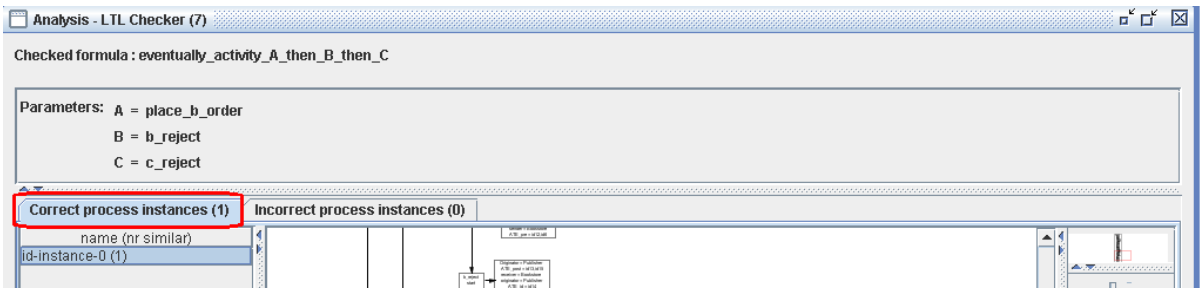

Figura 4.5: Primeira análise da modelagem da *bookstore* com LTL.

Na Figura 4.6, é observado o *trace* gerado pelo ProM para o primeiro teste da modelagem. Percebe-se que a mensagem *place_b_order* é enviada, e embora possa ocorrer dois caminhos, ambos se encontram na recepção da mensagem *por b_reject*. Após a ocorrência da mensagem *por b_reject*, é enviada e recebida a última mensagem *c_reject*. Também nesta figura, ao lado de

cada mensagem encontra-se um quadro com informações sobre a mesma, como entidade que origina a mensagem e entidade que recebe a mensagem.

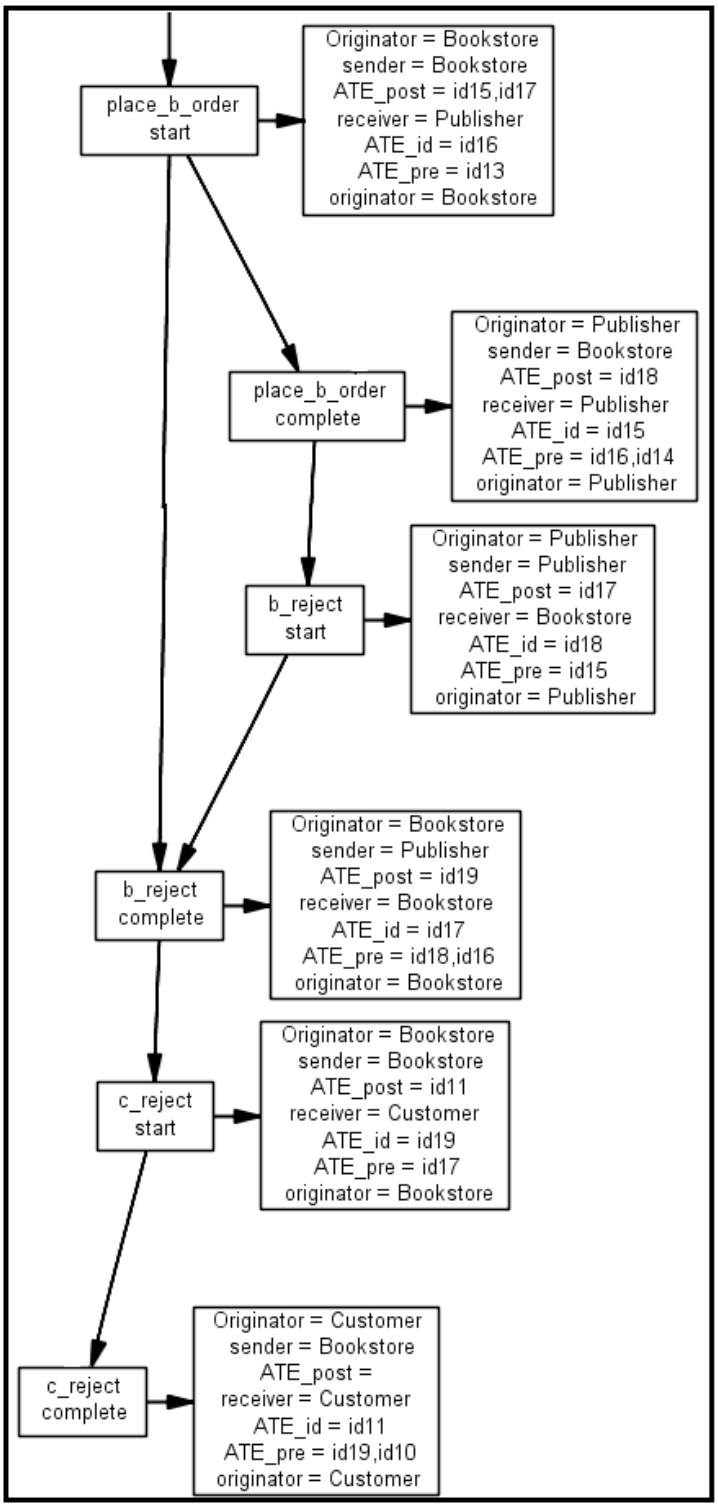

Figura 4.6: *Trace* da primeira análise da modelagem da *bookstore* com LTL.

A segunda inferência verifica se toda solicitação feita por um cliente à livraria é finalizado, no caso informando que o produto não foi encontrado. Dessa vez, procurou-se provar este caso com a troca de mensagem: se *place_c_order* acontecer então a mensagem *c_reject* pode também ocorrer. No ProM é provado que este fato ocorre. Nota-se que, na Figura 4.7, a aba *correct process instances* do ProM informa que houve uma ocorrência.

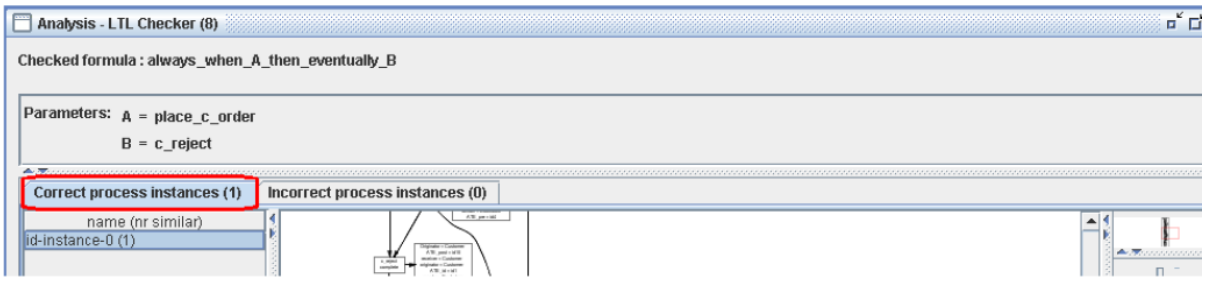

Figura 4.7: Segunda análise da modelagem da *bookstore* com LTL.

O *trace* referente ao segundo teste gerado pelo ProM encontar-se na Figura 4.8. A seqüência de eventos é iniciado com a mensagem *place_c_order* e sempre é finalizada pela mensagem *c_reject*, não importando se existe outras mensagens entre essas duas, as quais são omitidas e representadas na Figura em questão pelo número **1**.

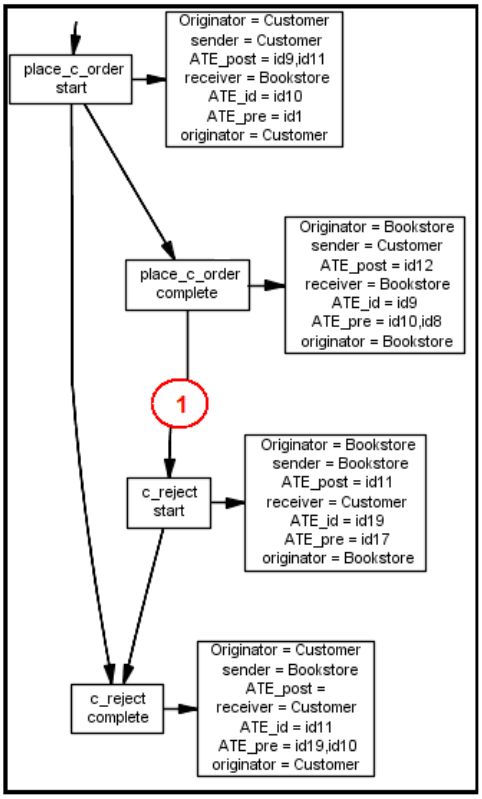

Figura 4.8: *Trace* da segunda análise da modelagem da *bookstore* com LTL.

O terceiro teste tenta avaliar se para qualquer editora a troca de mensagens é sempre a mesma. A resposta esperada neste teste é uma negativa, pois uma das características descritas para o sistema é o tratamento diferenciado às editoras. Deste modo, a propriedade a ser analisada é que se eventualmente acontecer *place_b_order*, ocorrerá em seguida *b_reject*. Neste caso a resposta informada pelo ProM é que esta seqüência de eventos não ocorre (Figura 4.9), pois a aba *incorrect process* instances informa a inexistência desse fato.

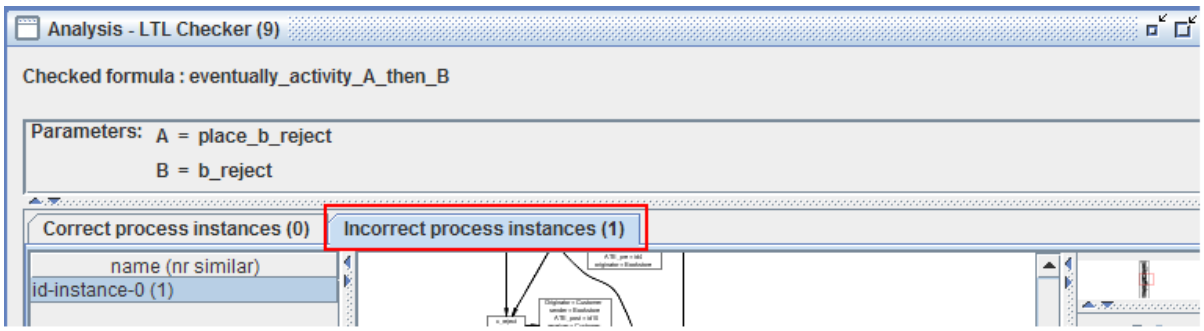

Figura 4.9: Terceiro análise da modelagem da *bookstore* com LTL.

As três análises buscam provar propriedades do sistema e compreender como está o funcionando no sistema. Estas propriedades verificadas são do tipo *safety* e *liveness*, as quais ajudam encontrar situações em que haja problema no sistema. No primeiro teste, a propriedade definida é validada para verificar o fluxo de ações que finalizam o processo de compra, enquanto no segundo avalia-se de forma satisfatória o tratamento do sistema, quando este não encontra o produto solicitado. No último teste verifica-se a comunicação com as editoras, e neste caso a resposta é negativa pelo fato de ser uma característica descrita no sistema. Este sistema trabalha com várias editoras, no caso da modelagem, as editoras presentes são diferentes para cada cenários, por este fato, apresenta uma não padronização com relação ao tratamento desta situação, originando esta falha.

5 *Considerações Finais*

Este trabalho tem como objetivo mostrar a inserção de métodos formais no desenvolvimento de softwares e pontos que apóiam sua adoção simplificada e adaptável. A Engenharia de Software muito embora já possua um descrição de métodos formais em seus processos, encontrada em modelos de desenvolvimento formal, ainda apresenta carência quanto a adoção deste tipo de abordagem.

Na revisão bibliográfica percebeu-se que muitas linguagens formais, apesar de firmadas, estão em constante evolução. Ainda assim,o uso destas linguagens continua insipiente, ou por dificuldade quanto ao modo de uso, ou ainda por terem recursos limitantes. Por isso nesta mesma etapa percebeu-se que as linguagens formais estavam sendo convertidas para outras linguagens de descrição que fossem mais fáceis de manipular e visualizar, mais próxima dos conhecimentos dos desenvolvedores. Este foi um dos fatores que norteou a decisão de utilizar linguagem de modelagem simplificada, normalmente difícil de validar, convertendo-a para uma linguagem que seja fácil de validar, mas eventualmente difícil de modelar.

A modelagem, primeira parte do método, consiste na parte mais importante, pois o resultado final corrresponde a representação do sistema a ser analisado. Somente com uma modelagem coerente do sistema é possível obter resultados satisfatórios.

Sobre a aplicação do método nos testes é importante ressaltar que antes de realizar qualquer tipo de análise é preciso ter uma compreensão da modelagem e a partir disto perceber o que pode-se inferir sobre o sistema. Determinadas análises podem ter resultados errôneas por se tentar inferir proposições que não estão representadas nos modelos.

Contribuições do trabalho são a análise sobre métodos e linguagens formais e o método proposto. Na Engenharia de Software a adoção desta técnica poderia facilitar a análise de partes concorrentes nos códigos, pelo fato da análise ser complexa.

Dentre as perspectivas e desenvolvimentos futuros, podem-se citar:

1. Estender o estudo a um conjunto maior de métodos e linguagens formais;

2. Buscar fazer um estudo de caso em empresas de software sobre a viabilidade da técnica descrita no trabalho;

3. Utilizar mais *plugins* do ProM e realizar mais formas de análise;

4. Utilizar outras linguagens formais que encontram-se no ProM.

Referências Bibliográficas

ALUR, R.; ETESSAMI, K.; YANNAKAKIS, M. Inference of message sequence charts. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 29, n. 7, p. 623–633, 2003. ISSN 0098-5589.

ALUR, R.; YANNAKAKIS, M. Model checking of message sequence charts. In: *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*. London, UK: Springer-Verlag, 1999. p. 114–129. ISBN 3-540-66425-4.

CARVALHO, R. A. de; FELIX, M. F.; PETRUCCI, V. T. Verificação formal de workflow. In: *Revista Eletrônica de Iniciação Científica*. [S.l.: s.n.], 2006. v. 6, n. 2, p. 80 – 91.

CUNHA, M. A. Lógica temporal. 2005. Disponível em: <wiki.di.uminho.pt/twiki/pub/Education/PeC/MaterialApoio/temporal.pdf>. Acesso em: 22 de dezembro de 2008.

DÖLL, L. M.; STADZISZ, P. C. Utilizando redes de petri na modelagem da dinâmica de sistemas orientados a objetos. In: *JIISIC*. [S.l.: s.n.], 2002. p. 44–53.

DONG, J. S.; SUN, J. From live sequence charts to distributed implementation. Jan 2007.

FERREIRA, P. M. *Geração Automática de Diagramas UML-RT a partir de Especificações CSP*. Dissertação (Mestrado) — Universidade Federal de Pernanbuco, 2006.

FISCHER, C.; OLDEROG, E.-R.; WEHRHEIM, H. A csp view on uml-rt structure diagrams. In: *FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*. London, UK: Springer-Verlag, 2001. p. 91–108. ISBN 3-540-41863-6.

GOGOLLA, M. Benefits and problems of formal methods. In: *Ada-Europe*. [S.l.: s.n.], 2004. p. 1–15.

GUIMARÃES, J. de O. *Redes de Petri*. 2000. Apostila. Universidade Federal de São Carlos - Departamento de Computação. Disponível em: <www.dc.ufscar.br/ jose/courses/tg/petri.pdf>. Acesso em: 22 de dezembro de 2008.

HALL, A. Seven myths of formal methods. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 07, n. 5, p. 11–19, 1990. ISSN 0740-7459.

HAREL, D.; KUGLER, H. Synthesizing state-based object systems from lsc specifications. *International Journal of Foundations of Computer Science*, v. 13, n. 1, p. 5–51, Febuary 2002. (Also in *Proc. 5th Int. Conf. on Implementation and Application of Automata* (CIAA 2000), Springer-Verlag, pp. 1–33, Preliminary version appeared as technical report MCS99-20, Weizmann Institute of Science, 1999.).

HAREL, D.; THIAGARAJAN, P. S. Message sequence charts. Kluwer Academic Publishers, Norwell, MA, USA, p. 77–105, 2003.

HAUGEN, Ø. Comparing uml 2.0 interactions and msc-2000. In: *SAM*. [S.l.: s.n.], 2004. p. 65–79.

HOARE, C. A. R. Communicating sequential processes. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 8, p. 666–677, 1978. ISSN 0001-0782.

HOLLOWAY, C. Why engineers shoul consider formal methods. In: *Proceedings of the* 16^{th} *AIAA/IEEE Digital Avionics Systems Conference*. Irvine CA: [s.n.], 1997. v. 1, p. 1.3–16 – 1.3–22.

ITU. *ITU-TS Recommendation Z.120: Message Sequence Chart 1999 (MSC99)*. [S.l.], 1999.

LASSEN, K.; van Dongen, B.; van der Aalst, W. Translating message sequence charts to other process languages using process mining. In: MOLDT, D. et al. (Ed.). *Proceedings of the Workshop on Petri Nets and Software Engineering (PNSE'07)*. [S.l.]: Publishing House of University of Podlasie, Siedlce, Poland, 2007. p. 82–97.

MACCOLL, I.; CARRINGTON, D. User interface correctness. *Crossroads*, ACM, New York, NY, USA, v. 3, n. 3, p. 9–13, 1997. ISSN 1528-4972.

MALACARI, P. Course material. 2004. Disponível em: <www.dcs.qmul.ac.uk/ pm/SaR/2004ltl.pdf>. Acesso em: 22 de dezembro de 2008.

MARCINIAK, J. J. (Ed.). *Encyclopedia of software engineering*. New York, NY, USA: Wiley-Interscience, 1994. ISBN 0-471-54004-8.

MATEESCU, A.; SALOMAA, A. Formal languages: an introduction and a synopsis. Springer-Verlag New York, Inc., New York, NY, USA, p. 1–39, 1997.

MAUW, S. The formalization of message sequence charts. *Comput. Netw. ISDN Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 28, n. 12, p. 1643–1657, 1996. ISSN 0169-7552.

MAUW, S.; RENIERS, M.; WILLEMSE, T. Message sequence charts in the software engineering process. In: CHANG, S. (Ed.). *Handbook of Software Engineering and Knowledge Engineering*. [S.l.]: World Scientific, 2001. p. 437–463.

MEYER, B. On formalism in specifications. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 2, n. 1, p. 6–26, 1985. ISSN 0740-7459.

MOURA, H. P. O uso de especificações formais na construção de software. In: *Caderno de Informática do Jornal de Brasília*. [S.l.: s.n.], 1995.

PÁDUA, S. I. D. et al. O potencial das redes de petri em modelagem e análise de processos de negócio. *Gestão e Produção*, v. 11, n. 1, p. 109–119, april 2004. ISSN 0104-530X.

PETERSON, J. L. Petri nets. *ACM Comput. Surv.*, v. 9, n. 3, p. 223–252, 1977.

PRESSMAN, R. S. *Engenharia de Software*. [S.l.]: McGraw-Hill, 2001.

RAMOS, M. V. M.; NETO, J. J.; VEGA Ítalo S. *Linguagens Formais: teoria, modelagem e implementação*. Porto Alegre, RS: Bookman, 2009. ISBN 978-85-7780-453-5.

SAVOLSKYTE, J. *Review of the JHotDraw framework*. [S.l.], april 2004.

SOMMERVILLE, I. *Software engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 85-88639-07-6.

Standish Group. 2004 third quarter research report. 2004. Disponível em: <http://www.kean.edu/ rmelworm/3040-00/StandishGroup-04-q3-spotlight.pdf>. Acesso em: 14 de julho de 2009.

STIDOLPH, D. C.; WHITEHEAD, E. J. Managerial issues for the consideration and use of formal methods. In: *FME*. [S.l.: s.n.], 2003. p. 170–186.

The University of Manchester. Cs2142 logic in computer science. 2003. Disponível em: <http://www.cs.man.ac.uk/ korovink/cs2142/chapter14.pdf>. Acesso em: 14 de julho de 2009.

van der Aalst, W. M. P.; WEIJTERS, A. J. M. M. Process mining: a research agenda. *Comput. Ind.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 53, n. 3, p. 231–244, 2004. ISSN 0166-3615.

van Dongen, B. F. et al. The prom framework: A new era in process mining tool support. In: *ICATPN*. [S.l.: s.n.], 2005. p. 444–454.

ZHAO, L. et al. A modeling method based on ccs for workflow. In: *ICUIMC '09: Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*. New York, NY, USA: ACM, 2009. p. 376–384. ISBN 978-1-60558-405-8.