FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA

Felipe Cantal de Sousa

UM PROCESSO PARA DETECÇÃO DE CENÁRIOS IMPLÍCITOS EM SISTEMAS CONCORRENTES

Fortaleza

Dezembro / 2007

FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA

Felipe Cantal de Sousa

UM PROCESSO PARA DETECÇÃO DE CENÁRIOS IMPLÍCITOS EM SISTEMAS CONCORRENTES

Dissertação apresentada ao curso de Mestrado em Informática Aplicada da Universidade de Fortaleza como parte dos requisitos necessários para a obtenção do grau de Mestre em Informática Aplicada.

Orientador: Prof. Dr. Nabor das Chagas Mendonça

Fortaleza

Dezembro / 2007

SOUSA, Felipe Cantal de

Um Processo para Detecção de Cenários Implícitos em Sistemas Concorrentes. Fortaleza. Universidade de Fortaleza (UNIFOR), Dissertação de Mestrado, 2007.

77 p.: il. 210 x 297 mm (MIA/UNIFOR, M.Sc. Ciência da Computação)

1. Engenharia de Software
2. Sistemas Distribuídos
3. Engenharia Reversa
3. Cenários Implícitos

I. MIA/UNIFOR II. Título CDU:

AGRADECIMENTOS

A Deus, pela força que sempre esteve comigo neste desafio.

Meus sinceros agradecimentos ao Prof. Nabor das Chagas Mendonça, pela orientação e inestimável ajuda, fundamentais para a conclusão deste trabalho.

Agradeço, em especial, à minha amada esposa Alyne, pelo carinho, inspiração, paciência e apoio incondicional para a concretização desta conquista.

Agradeço a minha querida filha Alice que tanto me trouxe paz e motivação para conclusão desse projeto.

Agradeço a meus pais, Hélio e Teresinha (em memória) pela contínua dedicação e empenho dispensados, me ensinando a sempre almejar e alcançar os mais valiosos objetivos.

Agradeço a todos meus familiares e amigos que me incentivaram durante essa caminhada.

Aos professores Pedro Porfírio Muniz Farias e Paulo Borba, pela presença na banca examinadora.

Aos demais professores do mestrado, pelas contribuições indiretas.

À secretaria do MIA, pela atenção e presteza sempre imediatas.

Aos demais colegas aqui não citados que de alguma forma me ajudaram e incentivaram.

Resumo da Dissertação apresentada ao MIA/UNIFOR como parte dos requisitos necessários para a obtenção do grau de Mestre em Informática Aplicada.

UM PROCESSO PARA DETECÇÃO DE CENÁRIOS IMPLÍCITOS EM SISTEMAS CONCORRENTES

Felipe Cantal de Sousa

Dezembro / 2007

Orientador: Prof. Dr. Nabor das Chagas Mendonça
Programa: Informática Aplicada

Um cenário descreve como um ou mais componentes de um sistema interagem para oferecer um conjunto de funcionalidades. Devido a cada cenário representar apenas uma visão parcial do comportamento global do sistema, especificações baseadas em cenários podem esconder comportamentos inusitados, denominados “cenários implícitos”, não previstos nos cenários originais. A presença de cenários implícitos tanto pode indicar falhas na especificação do sistema, como comportamentos indesejados a serem evitados. Este trabalho propõe um processo de engenharia reversa para apoiar a extração e detecção de cenários implícitos em sistemas concorrentes. A principal contribuição do trabalho é permitir que os desenvolvedores se beneficiem do conceito de cenários implícitos, até então de uso restrito às fases iniciais do ciclo de vida de software, também para apoiar atividades de compreensão e teste de sistemas existentes. A utilização do processo e de suas ferramentas de apoio é ilustrada através de um estudo de caso, onde foram detectados cenários implícitos em uma aplicação web de comércio eletrônico.

Abstract of the Dissertation presented to MIA/UNIFOR as a partial fulfillment of the requirements for the degree of Master of Applied Informatics.

A PROCESS FOR DETECTING IMPLIED SCENARIOS IN CONCURRENT SYSTEMS

Felipe Cantal de Sousa

December / 2007

Adviser: Prof. Dr. Nabor das Chagas Mendonça
Program: Applied Informatics

A scenario describes how one or more system components interact to provide a certain set of functionalities. Because each scenario only represents a partial view of the overall system behavior, scenario-based specifications may hide unexpected interactions (called implied scenarios) which are not part of any scenario individually. Implied scenarios may either indicate gaps in the original scenario specification, or undesired behavior that should be avoided during scenario implementation. This paper presents a reverse engineering process to support extraction and detection of implied scenarios in concurrent systems. The main contribution of the work is to demonstrate how the concept of implied scenarios, which use thus far has been restricted to early phases of the software life-cycle, such as requirements elaboration and reliability prediction, can also be applied to support comprehension and testing of existing systems. The use of the proposed process and its support tools is illustrated through a case study, in which we were able to detect implied scenarios in a existing e-commerce web application.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Introdução

Este capítulo apresenta as principais questões que motivaram a realização deste trabalho, seus objetivos e contribuições, e sua organização.

1.1 Motivação

Especificações baseadas em cenários, geralmente expressas utilizando notações como *Message Sequence Charts* - MSCs (ITU-T, 2004) e Diagramas de Seqüência (OMG, 2002), têm sido cada vez mais empregadas em ambientes corporativos para a descrição de requisitos de software. Um cenário descreve como um ou mais componentes de um sistema, seu ambiente externo, e seus usuários concorrem e interagem para oferecer um conjunto específico de funcionalidades. Dessa forma, cada cenário representa uma visão parcial do comportamento global do sistema, o qual, para ser entendido por completo, depende da combinação dessas diferentes visões (Uchitel *et al.*, 2003).

Cenários são particularmente úteis para descrever o comportamento de sistemas distribuídos e concorrentes (Magee e Kramer, 1999). Modelos de comportamento baseados em cenários têm se mostrado uma forma intuitiva e flexível de representar os diferentes fluxos de mensagens entre componentes que caracterizam a execução de um sistema concorrente, oferecendo, assim, uma especificação mais precisa do comportamento esperado desse tipo de sistema (Uchitel *et al.*, 2003).

Por outro lado, esse tipo de especificação sofre de uma limitação intrínseca, cujos efeitos só começaram a ser adequadamente investigados e entendidos mais recentemente. A limitação está no fato de que um modelo de comportamento de um sistema expresso em cenários pode não corresponder a soma exata do conjunto de comportamentos expressos nas especificações de cada cenário. Combinações inesperadas no modo como os componentes interagem podem levar ao aparecimento de comportamentos inusitados, não presentes nos cenários originais. Tais comportamentos, denominados cenários “decorrentes” ou “implícitos” (do inglês *implied scenarios*) no trabalho pioneiro de Alur *et al.* (2000), surgem principalmente porque os componentes têm em geral uma visão local do que está acontecendo no sistema, e não uma visão do comportamento global esperado. Um processo automatizado para a detecção de cenários implícitos durante a fase de elaboração de requisitos foi

posteriormente proposto por Uchitel *et al*. (2001).

A existência de cenários implícitos tanto pode indicar omissões na especificação do sistema, no caso de cenários implícitos considerados válidos, mas que por algum motivo não foram originalmente especificados; ou simplesmente situações indesejadas que devem ser evitadas, no caso de cenários implícitos considerados inválidos do ponto de vista do comportamento esperado dos componentes (Uchitel *et al*., 2002). Em ambos os casos, detectar e analisar a ocorrência de potenciais cenários implícitos antes da implementação do sistema são atividades que têm se mostrado fundamentais para o processo de especificação de requisitos de software, seja para corrigir eventuais deficiências de especificação, seja para aumentar a compreensão e a confiança dos desenvolvedores nas especificações existentes (Uchitel *et al*., 2004). No âmbito de testes de sistemas, a formulação de especificações de testes mais abrangentes, incluindo testes para os cenários implícitos detectados, favorece o aumento da qualidade do software a ser desenvolvido.

1.2 Objetivos

Este trabalho tem como objetivo principal propor soluções para auxiliar a detecção e a análise de cenários implícitos em sistemas concorrentes já implementados. Desse modo, pretende-se estender os benefícios das pesquisas sobre cenários implícitos realizadas originalmente por Alur *et al*. (2000) e Uchitel *et al*. (2001), na fase de especificação de requisitos, para apoiar atividades relacionadas à compreensão, teste e manutenção de software.

Os resultados específicos a serem alcançados incluem:

- Identificar as técnicas existentes para a engenharia reversa de cenários, escolhendo aquelas que melhor atendam aos requisitos identificados na pesquisa.
- Produzir os insumos necessários para a descoberta de cenários implícitos com a utilização de ferramentas construídas durante a pesquisa ou através do reuso de outras já existentes.
- Conduzir uma avaliação preliminar das soluções propostas e da utilidade de suas ferramentas de suporte na descoberta de cenários implícitos em uma aplicação alvo.

1.3 Contribuições

Para atingir os objetivos descritos acima, este trabalho apresenta um processo de engenharia reversa para apoiar a extração e detecção de cenários implícitos a partir de informações coletadas durante a execução de aplicações concorrentes (Sousa *et al.*, 2007; Sousa e Mendonça, 2007). O processo inclui as seguintes etapas: (i) a execução monitorada da aplicação alvo; (ii) a extração de cenários a partir dos rastros de execução gerados; e (iii) a descoberta e visualização de potenciais cenários implícitos tendo como base os cenários extraídos. Estas três etapas são apoiadas por um conjunto de ferramentas desenvolvidas ou reutilizadas especificamente para este fim.

Embora a possibilidade de detectar cenários implícitos em especificações baseadas em cenários extraídas de sistemas existentes tenha sido sugerida originalmente por Alur *et al*. (2000), as pesquisas nessa área se concentram em atividades em nível de análise e projeto, tais como análise de requisitos (Uchitel *et al*., 2004; Letier *et al*., 2005) e análise de confiabilidade (Rodrigues *et al*., 2005). Dessa forma, a principal contribuição do trabalho é viabilizar a detecção de cenários implícitos, até então restrita às fases iniciais do ciclo de vida de software, como mais um instrumento de apoio às atividades de compreensão, teste e manutenção de sistemas existentes.

1.4 Estrutura da Dissertação

Além desta Introdução, a dissertação está organizada em mais sete capítulos, descritos a seguir.

No Capítulo 2, discorremos sobre o conceito de cenários implícitos, enfocando a sua motivação e o formalismo necessário para sua definição.

No capítulo 3, analisamos os principais trabalhos existentes na linha de extração de cenários e detecção de cenários implícitos.

No Capítulo 4, apresentamos o processo de detecção de cenários implícitos em sistemas concorrentes proposto neste trabalho e revelamos alguns detalhes sobre a implementação do seu conjunto de ferramentas de apoio.

No Capítulo 5, descrevemos alguns exemplos de utilização do processo proposto em um estudo caso, incluindo informações sobre a aplicação alvo escolhida, os cenários extraídos a partir de sua execução, as técnicas de abstração utilizadas, e os cenários implícitos

descobertos.

No Capítulo 6, apresentamos as conclusões do trabalho e oferecemos algumas sugestões para trabalhos futuros.

A dissertação inclui ainda três apêndices, descritos a seguir.

O Apêndice I descreve os marcadores que compõem o documento XML correspondente ao Metamodelo de Rastros de Execução, utilizado pela ferramenta de extração de cenários.

O Apêndice II descreve os marcadores que compõem o documento XML que contém os parâmetros de configuração da ferramenta de detecção de cenários implícitos.

O Apêndice III descreve as funcionalidades da aplicação MyPetStore, desenvolvida e utilizada como exemplo de uso do processo na pesquisa.

Capítulo 2

Cenários Implícitos

Este capítulo apresenta o conceito de cenários implícitos, enfocando a sua motivação e o formalismo necessário para sua definição e classificação.

2.1 Motivação

Atualmente, é notório o sucesso do uso de modelos de comportamento na compreensão de sistemas concorrentes e distribuídos, exatamente pela descoberta de fluxos operacionais não previstos em tempo de projeto. Entretanto, existem problemas intrínsecos decorrentes de seu uso, pois além da dificuldade da construção desses modelos, os benefícios de sua utilização somente aparecem no final do processo de construção.

Em contrapartida, especificações baseadas em cenários, como MSCs (ITU-T, 2004) e Diagramas de Seqüência (OMG, 2002), têm uma larga aceitação na indústria como parte das especificações de requisitos de sistemas, por serem simples e intuitivos. Por outro lado, as técnicas e ferramentas existentes de análise de especificações desta categoria têm poucas aplicabilidades, geralmente limitadas a verificações de consistências sintáticas. Outra grande restrição é que especificações baseadas em cenários geralmente descrevem comportamentos parciais do sistema. Essa característica implica em, mesmo com um grande número de especificações à disposição dos interessados no processo de compreensão e validação, uma visão do comportamento global do sistema ainda é necessária. Ao se combinar os diversos cenários individuais, novos comportamentos podem surgir para complementar ou corrigir as especificações originais, chamados *cenários implícitos* (Uchitel *et al.*, 2001; 2002; 2004) . O conhecimento adicional trazido por esses novos comportamentos, até então ignorado, é fundamental como complemento à documentação existente e fornece uma margem mais abrangente de segurança aos desenvolvedores, necessária a atividades de teste e manutenção de sistemas.

No intuito de apresentar os aspectos técnicos envolvidos na descoberta de cenários implícitos, neste capítulo: (i) descrevemos uma versão da notação MSC proposta por Uchitel *et al.* (2004), própria para cobrir importantes aspectos de sistemas concorrentes; (ii) definimos e discutimos o conceito de cenários positivos no contexto de modelos de arquitetura; e (iii) definimos, classificamos e apresentamos um algoritmo para síntese e

detecção de cenários implícitos utilizando modelos formais de comportamento. Vale salientar que o formalismo, as definições e os exemplos apresentados ao longo deste capítulo foram, em grande parte, retirados dos trabalhos de Uchitel *et al.* (2001; 2002; 2004).

2.2 A Notação MSC

A notação MSC (ITU-T, 2003) é um mecanismo popular de especificação de cenários utilizada para descrever padrões de interações entre processos e objetos. Ela constitui um formalismo visual atrativo também utilizado na captura de requisitos nos estágios iniciais de projetos de desenvolvimento de sistemas.

Devido a sua solidez e praticidade, o MSC serviu de base para o desenvolvimento de outras linguagens, também com foco na descrição de aspectos comportamentais de sistemas, como o Diagrama de Seqüência da UML (OMG, 2002). Segundo Harel *et al.* (2003), a aplicabilidade da notação MSC em metodologias utilizadas para o desenvolvimento de sistemas orientados a objetos é bastante extensa, cobrindo desde a descrição de comportamentos específicos entre instâncias de casos de uso até a captura de requisitos na forma de cenários positivos, que o sistema desenvolvido deve exibir, e de cenários negativos, que não devem ser permitidos no sistema. Artefatos gerados a partir de uma especificação MSC podem ser utilizados como um importante veículo de entendimento entre os usuários de um sistema e a equipe responsável pelo seu desenvolvimento, além de servir de insumo para a elaboração de especificações e execução de testes de alguns aspectos de sua implementação.

No restante dessa seção, nós iremos introduzir uma versão da notação MSC, que é um subconjunto sintático do padrão MSC da *ITU* (ITU-T, 2003) proposta por Uchitel *et al.*(2004)

Diagramas MSC podem ser classificados em duas categorias: MSCs básicos (bMSCs) e MSCs de alto nível (hMSCs). Seguem abaixo as definições de bMSC e hMSC utilizados na notação MSC proposta por Uchitel *et al.* (2004).

Basic Message Sequence Chart – bMSC: é uma estrutura $b = (E, L, I, M, \text{instância}, \text{rótulo}, \text{ordem})$ onde:

- E é um conjunto finito e mensurável de eventos que pode ser particionado em conjuntos de eventos dos tipos $enviar(E)$ e $receber(E)$;
- L é um conjunto de rótulos de mensagens. É utilizada a notação $\alpha(b)$ para representar L;
- I é um conjunto de nomes de instancias;

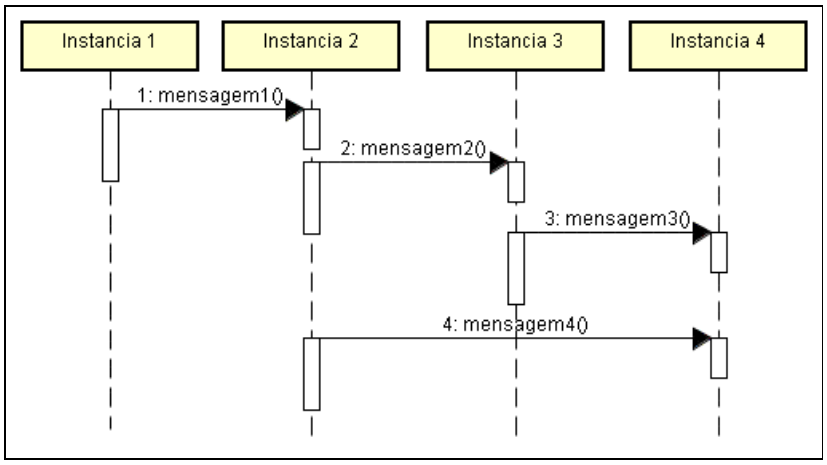

Figura 2.1: Exemplo de um bMSC.

- $M : enviar(E) \rightarrow receber(E)$ é uma função que associa eventos de envio e recebimento, designados como mensagens;
- $instância : E \rightarrow I$ é uma função que mapeia todo evento para a instância na qual o evento ocorre. Dados $i \in I$, designa-se $\{e \in E \mid instância(e) = i\}$ como $i(E)$;
- $rótulo : E \rightarrow L$ é uma função que mapeia eventos para rótulos. Para todo $(e, e') \in M$ tal que $rótulo(e) = rótulo(e')$, e se $(v, v') \in M$ e $rótulo(e) = rótulo(v)$, então $instância(e) = instância(v)$ e $instância(e') = instância(v')$;
- *ordem* é um conjunto de ordens totais: $\leq_i \subseteq i(E) \times i(E)$ com $i \in I$ e $\leq_i$ correspondendo à ordem temporal (de cima para baixo) de eventos ocorridos na instância *i*.

A formalização acima define um bMSC como um descritor de um conjunto finito de interações entre um conjunto de componentes (Figura 2.1). Na figura, cada linha vertical representa uma instância (*Instancia 1, Instancia 2, Instancia 3 e Instancia 4)* pertencente a *I*. As mensagens são associadas a setas horizontais, cujos rótulos (*mensagem1, mensagem2, mensagem3 e mensagem4)* pertencem a *L*, onde sua origem indica a instância que envia a mensagem (saída) e seu final (ponta da seta) indica a instãncia que irá receber a mensagem (entrada). Mensagens de entrada e saída são tratadas como eventos (*E*). A semântica de um bMSC é dada pelo conjunto de seqüências de rótulos de mensagens, também conhecidos como rastros. O conjunto de rastros é determinado pela ordenação dos eventos ocorridos no

cenário.

Segundo Harel *et al.* (2003), são necessárias algumas condições para a existência de MSC válidos:

1. Todas as mensagens que fazem parte de um cenário devem estar linearmente ordenadas;
2. As mensagens devem ser enviadas antes de serem recebidas e não podem sobrepor-se umas as outras;
3. Todas as mensagens enviadas têm que ser recebidas obrigatoriamente, ou seja, não existem mensagens perdidas em diagramas MSCs;
4. A relação de causa entre os eventos em um diagrama MSC é completamente determinada pela ordem na qual os eventos ocorrem em cada cenário.

Um dos aspectos que diferenciam a versão de MSC utilizada no trabalho de Uchitel *et al.* da definida pelo ITU é o tipo de comunicação representado pelas mensagens. O ITU indica que as mensagens trocadas entre componentes são assíncronas, enquanto o trabalho de Uchitel *et al.* assume a existência de operações com bloqueio, ou seja, operações síncronas.

High-Level Message Sequence Chart – hMSC*:* é um grafo no formato (N, E, s_0), onde:

- N é um conjunto de nós;
- $E \subseteq (N \times N)$ é um conjunto de arestas e $s_0 \in N$ é o nó inicial. Dizemos que n é adjacente a n' se $(n, n') \in E$;
- Uma seqüência de nós (possivelmente infinita) $w = n_0, n_1, ...$ é um caminho se $n_0 = s_0$ e n_i é adjacente a n_{i+1} para $0 \leq i < |w|$. Dizemos que um caminho é máximo se ele não é um prefixo de qualquer outro caminho.

A formalização acima define hMSCs como grafos que estabelecem composições de cenários básicos. Eles possuem um nó inicial e representam os bMSCs como nós no diagrama com suas possíveis continuações, mostrando como o sistema pode evoluir de um cenário básico para outro. As continuações são representadas como arestas dirigidas, indicando em seu final (ponta da seta) o cenário considerado uma continuação do de sua origem. Na Figura 2.2, podemos observar um exemplo de um conjunto de cenários básicos e suas possíveis continuações.

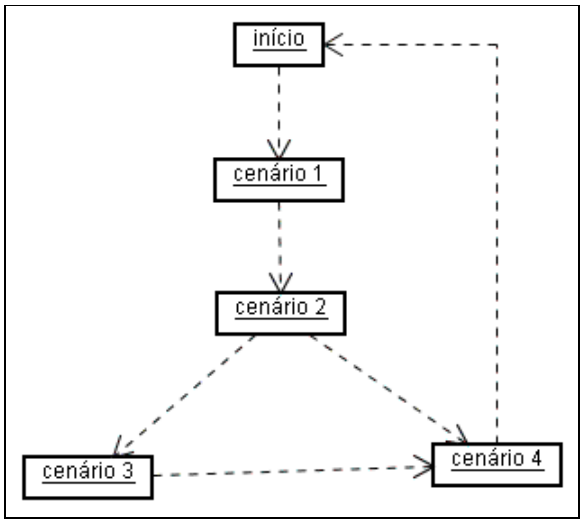

Figura 2.2: Exemplo de um hMSC.

- *cenário 1*: continuação possível - *cenário 2*
- *cenário 2*: continuações possíveis - *cenário 3* e *cenário 4*
- *cenário 3*: continuação possível - *cenário 4*
- *cenário 4*: continuação possível - *nó inicial*

A Figura 2.3 mostra um exemplo de uma especificação de cenários básicos e de alto nível para um sistema de caldeira. No exemplo, existe uma unidade de controle (*Control*) que opera um sensor (*Sensor*) e um componente específico para controle da pressão da caldeira (*Actuator*). Além disso, uma base de dados (*Database*) é utilizada como repositório de informações do sensor, enquanto a unidade de controle realiza cálculos e envia comandos ao componente de controle de pressão. No cenário básico *Initialize*, a mensagem do componente *Control* para o componente *Sensor* representa a ativação do segundo pelo primeiro. O cenário básico *Register* por sua vez representa a ação de registro dos dados de pressão coletados pelo sensor no componente *Database*. O cenário básico *Analysis* representa uma seqüência de mensagens entre os componentes *Control*, *Database* e *Actuator*. Essas mensagens correspondem ao processo de análise do sistema, que pode resultar no aumento ou diminuição da temperatura da caldeira, dependendo dos valores de pressão coletados pelo sensor e armazenados no banco de dados. Por fim, o cenário básico *Terminate* representa a ação de desativação do sensor pelo componente de controle. Note que, do ponto de vista individual de cada cenário, todos os componentes apresentam comportamentos válidos. Uma leitura do diagrama de alto nível revela as possíveis continuações de cada cenário básico representado

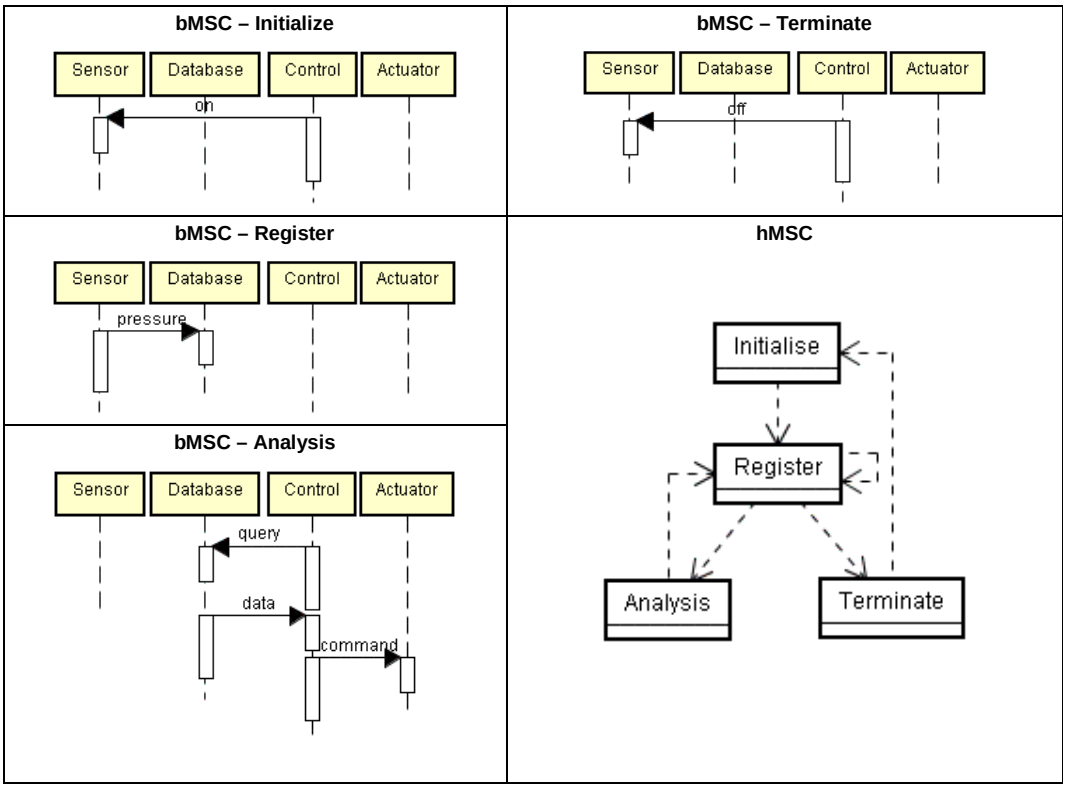

Figura 2.3: bMSCs e hMSC do sistema de caldeira.

neste exemplo: (*Initialise* → *Register*, *Register* → *Register*, *Register* → *Analysis*, *Register* → *Terminate*, *Analysis* → *Register* e *Terminate* → *Initialise*).

2.3 Cenários Positivos

Baseado nas definições de bMSC e hMSC descritas anteriormente, define-se uma especificação MSC positiva da forma abaixo:

Especificação MSC Positiva: é uma estrutura *PSpec* = (*B*, *H*, *F*), onde:

- *B* é um conjunto de bMSCs;
- $H = (N, A, s_o)$ é um hMSC;
- $f : N \rightarrow B$ é uma função que mapeia nós hMSC para bMSCs.

Em resumo, cenários positivos consistem em um conjunto de bMSCs, um hMSC e uma função que mapeia cada nó do hMSC para um bMSC (um para um). Esses tipos de cenários representam comportamentos válidos do sistema que mostram como o sistema pode evoluir de um cenário básico para outro.

2.4 Modelos de Arquitetura

A arquitetura de um sistema é determinada em parte pelo conjunto de especificações MSC definidas para o mesmo. Segundo Uchitel *et al.* (2004), entende-se por arquitetura um conjunto de componentes que implementam um conjunto de funcionalidades descritas em suas interfaces, ou seja, a arquitetura determina o conjunto de rastros aceitáveis no sistema (mensagens que podem ser enviadas ou recebidas entre os componentes). Dentro desse contexto, alguns pontos devem ser levados em consideração. A grande variedade de mensagens e componentes presentes, e a inexistência de um mecanismo que force uma sincronização e ordenação dessas mensagens se traduzem em problemas relevantes que devem ser previstos na modelagem da arquitetura.

Nesse sentido, a construção de um modelo que alinhe a arquitetura exigida àquela definida pelas especificações MSC é um grande desafio. Abaixo descrevemos os mecanismos necessários para que este alinhamento seja realizado.

Interfaces e alfabetos: seja $PSpec = (B, H, f)$ uma especificação MSC positiva com um conjunto de instâncias I e seja $i \in I$ uma instância. A interface de i é o conjunto de rótulo de mensagens $\alpha(i) = \{l \mid \exists b = (E, L, I, M, \textit{instância}, \textit{rótulo}, \textit{ordem}) \in B$ e $\exists e \in E$ tal que $\textit{rótulo}(e) = l$ e $\textit{instância}(e) = i\}$. É também denotada a interface de i como seu alfabeto.

Labeled Transition System (LTS): imaginemos que o símbolo *Estados* seja um conjunto universal de estados onde o estado π é o estado de erro. Imaginemos também que o símbolo *Rótulos* seja um conjunto universal de rótulos e $\textit{Rótulos}_\tau = \textit{Rótulos} \cup \{\tau\}$, onde τ é uma ação interna de componente que não pode ser detectada por outros componentes. Um LTS P é uma estrutura (S, L, Δ, q) onde:

- $S \subseteq \textit{Estados}$ é um conjunto finito de estados;
- $L = \alpha(P) \cup \{\tau\}$, $\alpha(P) \subseteq \textit{Rótulos}$ é um conjunto de rótulos que representa o alfabeto de P;
- $\Delta \subseteq (S \setminus \{\pi\} \times L \times S)$ define as transições rotuladas entre estados.
- $q \in S$ é o estado inicial.

Usamos $s \xrightarrow{l} s'$ para denotar $(s, l, s') \in \Delta$. Adicionalmente, dizemos que um LTS P é determinístico se $s \xrightarrow{l} s_1$ e $s \xrightarrow{l} s_2$ implica em $s_1 = s_2$.

Um LTS (Keller *et al*., 1976) provê um formalismo adaptável para modelar tanto os componentes quanto os sistemas descritos pelas especificações MSC. Especificamente, transições são associadas (rotuladas) às mensagens de cada bMSC, processos LTS modelam componentes de sistemas e uma composição paralela de LTSs é utilizada para definir um resultado sistemático a partir de vários componentes que interagem entre si. Em outras palavras, o sistema alvo é visto como uma composição paralela de todos os seus componentes, de forma que ele execute assincronamente, mas sincronize todos os rótulos de mensagens compartilhadas.

Dado o LTS, pode-se comparar o comportamento modelado com o comportamento total das especificações positivas MSC. Isso significa construir um modelo que leve em conta tanto a arquitetura quanto o comportamento descrito nas especificações MSC e, como comentado anteriormente, represente uma composição paralela de uma coleção de LTSs, onde cada LTS modela um componente na especificação MSC. Modelos LTS que preservam a estrutura e as interfaces ao mesmo tempo que exibem todos os rastros especificados são chamados modelos de arquitetura por Uchitel *et al.* (2004). Note que o conceito citado difere do conceito clássico de engenharia de software citado por Shaw *et al* (1996).

Outro ponto importante a ser comentado é que o alfabeto utilizado nos modelos LTS deve coincidir com o dos componentes modelados por eles, o que pode ser garantido pela existência prévia de interfaces.

Abaixo apresentamos as definições de execuções, rastros, rastros máximos e modelos de arquitetura introduzidos por Uchitel *et al.* (2004).

***Execuções*:** seja $P = (S, L, \Delta, q)$ um LTS. Uma execução de P é uma seqüência $w = q_0 l_0 q_1 l_1 ...$ de estados q_i e rótulos $l_i \in L$ tal que $q_i \xrightarrow{l_i} q_{i+1}$ para todo $0 \leq i < |w/2|$. Uma execução é máxima se ela não pode ser estendida para continuar sendo uma execução do LTS. Define-se também $ex(P) = \{w \mid w$ é uma execução de $P\}$.

***Projeção*:** seja w uma palavra $w_0 w_1 w_2 w_3 ...$ e A um alfabeto. A projeção de w sobre A, a qual denotamos $w|_A$, é o resultado de eliminarmos da palavra w todos os elementos w_i em A.

Rastros e Rastros Máximos: seja P um LTS. A palavra w sobre o alfabeto $\alpha(P)$ é rastro máximo de P se existir uma execução máxima $e \in ex(P)$ tal que $w = e|_{\alpha(P)}$. Usa-se $tr(e)$ para denotar a projeção de uma execução sobre o alfabeto de um LTS. Define-se também $tr(P) = \{w \mid w$ é um rastro de $P\}$ e $L(P) = \{w \mid w$ é um rastro máximo de $P\}$.

Modelo de Arquitetura: seja *PSpec* = (*B*, *H*, *f*) uma especificação MSC positiva com instâncias *I*, e seja A_i com $i \in I$ um conjunto de LTSs. Diz-se que um LTS *A* é um modelo de arquitetura de *PSpec* se e somente se $A = \{A_1 \parallel ... \parallel A_n\}$, $\alpha(A_i) = \alpha(i)$ e $L(PSpec) \subseteq L(A)$.

A seguir é discutido como modelos de arquitetura são construídos a partir de especificações MSC. Para isso, descrevemos o algoritmo utilizado por Uchitel *et. al.* (2004) para construir modelos de componentes, que é um LTS para um componente especificado em uma especificação MSC positiva. Se todos os modelos de componentes forem construídos utilizando esse algoritmo, suas composições paralelas podem ser vistas como um modelo de arquitetura de uma especificação MSC.

O algoritmo consiste em transformar especificações MSC em uma especificação de modelo de comportamento no formato FSP – *Finite Sequential Processes* (Kramer e Magee, 1999), cuja semântica é dada em termos dos LTSs utilizados, e sintetizar um modelo de componente por vez. Abaixo os passos do algoritmo:

1. Primeiramente, o algoritmo constrói um LTS para cada componente em cada bMSC na especificação. Cada um desses LTSs exibe o mesmo comportamento exibido pelo componente no bMSC. Por exemplo, a Figura 2.4 mostra o LTS correspondente ao modelo de comportamento do componente *Control* no bMSC *Analysis* mostrado na Figura 2.3.
2. Criados os LTS, o algoritmo deve combiná-los para modelar o comportamento do componente à medida que ele percorre o hMSC, ou seja, o algoritmo une todos os LTSs de acordo com a especificação MSC de alto nível. Isso é feito adicionando transições rotuladas como *intAction,* ligando os diversos LTSs. Por exemplo, no modelo do sistema de caldeira, o último estágio do LTS correspondente ao bMSC *Analysis* é unido ao primeiro estágio do LTS que modela o bMSC *Register*. O LTS resultante da união dos LTSs de todos os bMSCs onde o componente *Control* tem participação pode ser visto na Figura 2.5.
3. O próximo passo é esconder as transições adicionadas *intAction,* tornando-as não detectáveis.
4. O último passo deve transformar o LTS resultante em um LTS determinístico. Isso é feito minimizando-o, de modo a preservar os seus rastros máximos, facilitando sua manipulação através de ferramentas automatizadas.

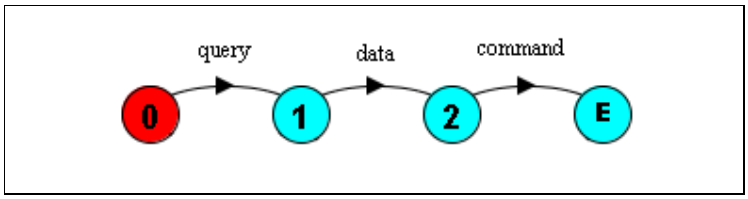

Figura 2.4: Comportamento do componente *Control* no bMSC *Analysis*.

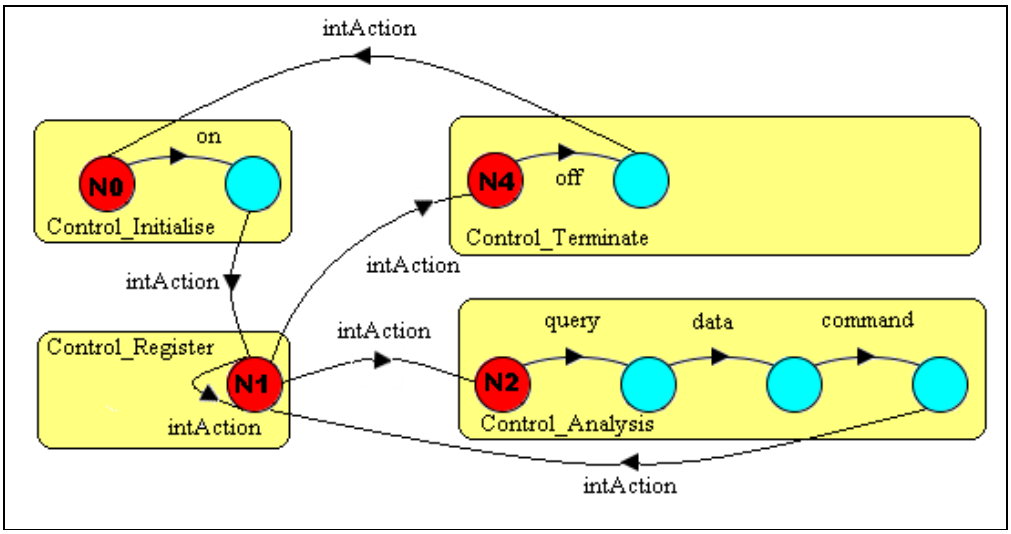

Figura 2.5: Composição do comportamento do componente *Control* em todos os bMSCs.

Após a síntese de cada LTS, o sistema completo corresponde a suas composições paralelas. A Figura 2.6 descreve o modelo de cada componente e o modelo de arquitetura adotado para o exemplo do sistema de caldeira, nos quais são especificados os seguintes alfabetos: para o componente *Sensor* {*pressure, on, off*}, para o componente *Database* {*data, query, pressure*}, para o componente *Control* {*on, off, query, data, command*} e, por fim, para o componente *Actuator* {*command*}. Maiores detalhes do algoritmo podem ser obtidos em (Uchitel *et al*., 2001; Uchitel *et al*., 2004).

2.5 Cenários Implícitos

A Figura 2.7 mostra um rastro que pode ser executado no sistema de caldeira utilizado como exemplo. O rastro mostra como o componente *Control* está acessando o componente *Database* e recebe informações de uma ativação anterior do componente *Sensor*. Esse não é um comportamento esperado porque a especificação MSC indica que depois da inicialização do componente *Sensor* algum dado deve ser registrado no componente *Database* antes que alguma solicitação de informação lhe possa ser feita. O problema aparece porque o

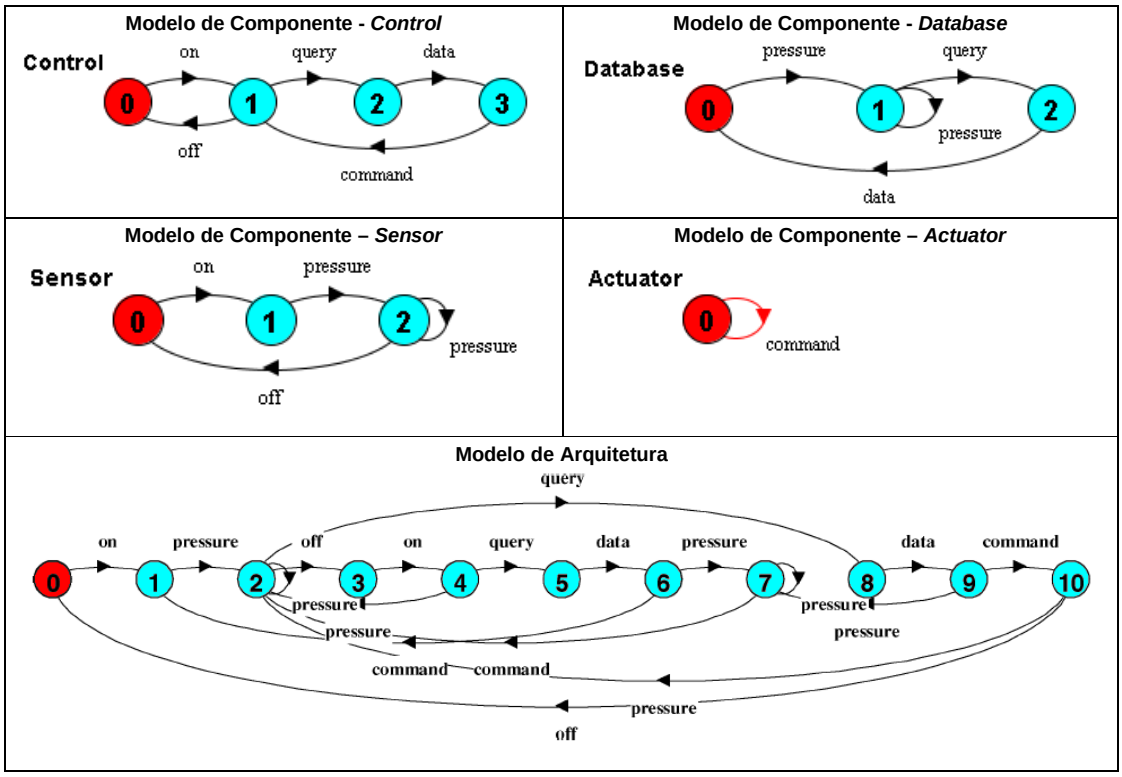

Figura 2.6: Modelo de componentes e de arquitetura do sistema de caldeira.

componente *Control* não pode detectar quando o componente *Sensor* registra dados no componente *Database*; então, se é para ser feita uma consulta no componente *Database* após o dado ter sido inserido pelo menos uma vez, é necessário confiar no componente *Database* para habilitar ou desabilitar consultas quando apropriado. Entretanto, como o componente *Database* não pode dizer quando o componente *Sensor* está no estado *on* (ativado) ou *off* (desativado), também não pode distinguir o primeiro registro de dados dos demais. Desta forma, ele (*Database*) não pode habilitar ou desabilitar consultas apropriadamente.

Em resumo, os componentes envolvidos não têm informação local suficiente para prevenir a execução mostrada na Figura 2.7. Note que cada componente está se comportando corretamente de forma individual, mas em conjunto impõem uma seqüência inválida de bMSCs. Uchitel *et al*. (2001) utilizaram o termo *cenário implícito*, originalmente proposto por Alur *et al.* (2000), para se referir a rastros de sistema dessa natureza, como o mostrado na Figura 2.7. Cenários implícitos são resultados de problemas ocorridos entre a decomposição e o comportamento de sistemas. Eles não são artefatos de um tipo particular de linguagem MSC e são independentes da semântica apresentada em um hMSC.

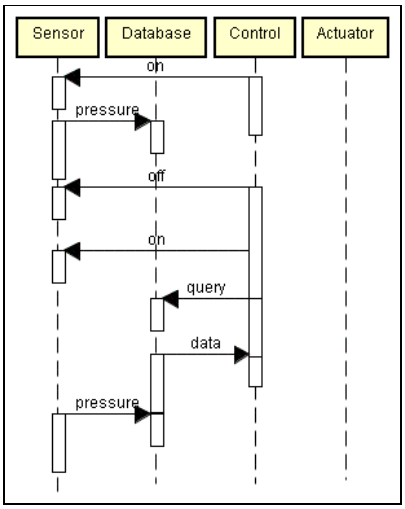

Figura 2.7: Cenário implícito do sistema de caldeira.

Cenários Implícitos (definição 1): dada uma especificação MSC positiva *PSpec* um rastro $w \notin L(PSpec)$ é um cenário implícito de *PSpec* se todos os rastros *y* e para todos os modelos de arquitetura *A* de *PSpec*, $w.y \in L(A)$ implica $w.y \notin L(PSpec)$.

Cenários Implícitos (definição 2): se *PSpec* é uma especificação MSC positiva e *A* é um modelo mínimo de arquitetura de *PSpec,* então *PSpec* tem cenários implícitos se e somente se $L(A) \not\subset L(PSpec)$.

Cenários implícitos não representam necessariamente situações inaceitáveis. Eles podem ser simplesmente cenários aceitáveis que foram desprezados pelos projetistas no processo de especificação do sistema. Uma vez detectados, eles servirão para completar essas especificações. Existe então uma grande vantagem em detectar e validar cenários implícitos de forma automática, pois na medida em que eles são detectados, os projetistas são forçados a levar em conta aspectos cruciais sobre a natureza concorrente dos sistemas que eles estão especificando.

Alur *et al*. (2000) definem que uma especificação MSC é realizável se essa especificação não possui nenhum cenário implícito, ou seja, se existe um modelo de arquitetura que exibe exatamente os mesmos rastros definidos pelo MSC. Segundo o formalismo introduzido por Uchitel *et al*. (2004), cenários realizáveis podem ser definidos da forma abaixo.

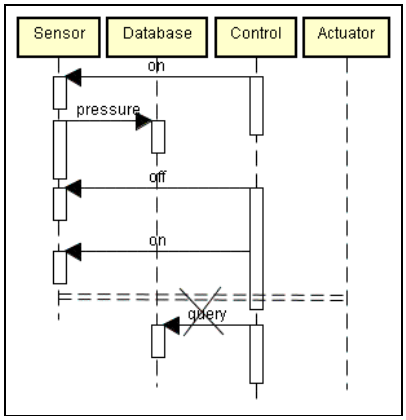

Figura 2.8: Primeiro cenário negativo básico do sistema de caldeira.

Cenário Implícito Realizável: seja *PSpec* uma especificação MSC positiva e seja *A* um modelo de arquitetura de *PSpec*, *A* é uma realização de *PSpec* se $L(PSpec) = L(A)$. Dizemos que um *PSpec* é realizável se existe uma realização de *PSpec*.

2.6 Cenários Negativos

Uchitel *et al.* (2004) definem cenários negativos como situações rejeitadas pelos projetistas do sistema, por se tratarem de comportamentos inaceitáveis, que demandariam desvios do comportamento esperado. Os autores classificam os cenários negativos em três categorias: básicos, abstratos e *after/until*. As próximas seções descrevem cada categoria.

2.6.1 Cenários Negativos Básicos

Para dinamizar o processo de elaboração de requisitos, alguns mecanismos podem ser adotados no intuito de agilizar a documentação das situações indesejadas que podem estar relacionadas à existência de cenários implícitos. O conceito de *cenários implícitos negativos básicos* foi criado como uma notação útil para documentar situações indesejáveis no sistema.

Cenários negativos básicos são MSCs com duas características visuais adicionais. A primeira é que a sua última mensagem é cruzada (marcada com um *X*). A segunda é que a última mensagem é separada das outras por uma linha pontilhada. A Figura 2.8 mostra um exemplo de um cenário negativo básico para o sistema de caldeira estudado. A parte superior à linha pontilhada é chamada *pré-condição* e a mensagem cruzada é chamada *mensagem proscrita*. Intuitivamente, um cenário negativo básico indica que se o comportamento descrito

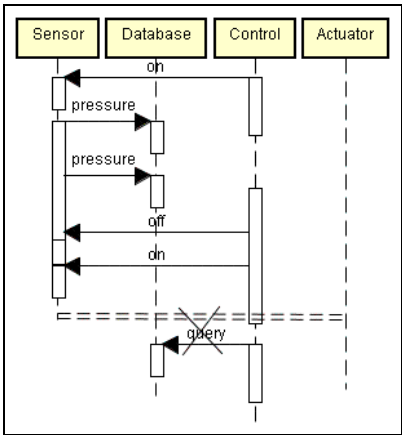

Figura 2.9: Cenário negativo do sistema de caldeira.

na *pré-condição* ocorreu, a próxima mensagem não deve ser a *mensagem proscrita*. A linha pontilhada salienta a necessidade do acontecimento completo da pré-condição, antes do envio da mensagem considerada proscrita.

Cenário Negativo Básico: um cenário negativo básico *n* é um par (*p, l*) onde:

- *p* é um bMSC (representando a *pré-condição*);
- *l* é um rótulo de mensagem (representando a *mensagem proscrita*);

É utilizado $\alpha(n)$ para denotar $\alpha(p) \cup \{l\}$.

Um ponto a ressaltar é que os cenários implícitos negativos ou positivos encontrados também passam a fazer parte da especificação MSC que está sendo alvo do processo de análise. Assim, os cenários adicionais também são submetidos ao algoritmo responsável pela síntese, ou seja, são criados modelos LTS e especificações FSP baseados neles. Dessa forma, a continuidade do processo de descoberta de novos cenários implícitos passa a contar com essas novas informações como insumo.

2.6.2 Cenários Negativos Abstratos

Imaginemos que, a partir do cenário negativo básico descoberto na Figura 2.8 (rastro {*on, pressure, off, on, query*}), continuemos o processo de descoberta de cenários implícitos. Imaginemos, também, que foi descoberto o cenário descrito na Figura 2.9 (rastro {*on, pressure, pressure, off, on, query*}). O novo cenário implícito é nitidamente uma variação do primeiro encontrado. A única diferença é que a instrução *pressure* aparece duas vezes nesse cenário. Claramente, se prosseguirmos com o processo de descoberta certamente será detectado outro cenário, desta vez com três instruções *pressure*. Conseqüentemente, esse

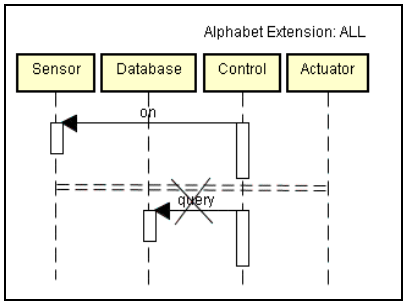

Figura 2.10: Cenário negativo abstrato do sistema de caldeira.

processo não converge para uma especificação onde todos os cenários implícitos tenham sido validados.

Fica claro que o número de instruções *pressure* repetidas é insignificante ao processo, somente acarretando em mais documentação. É preciso, assim, uma notação mais expressiva para esses tipos de cenários negativos, chamada de *cenários negativos abstratos*.

Cenário Negativo Abstrato: um cenário negativo abstrato *n* é um par (*p, E, l*) onde:

- *p* é um bMSC;
- *E* é um conjunto de rótulos de mensagens;
- *l* é um rótulo de mensagem.

É utilizado $\alpha(n)$ para denotar $\alpha(p) \cup E \cup \{l\}$.

Em casos como o citado acima, a necessidade é proibir rastros que possuem sub-rastros particulares, que não necessariamente têm que ocorrer no início do rastro. Essa notação é dada por uma precondição *p* e uma mensagem proscrita *l*. Uma característica adicional é que o alfabeto da precondição pode ser explicitamente estendido com um conjunto de rótulos (“Alphabet Extension: ALL” na parte superior, canto direito do cenário mostrado na Figura 2.10).

Restrições desse tipo podem ser diretamente embutidas no modelo de arquitetura, facilitando a verificação de novos cenários implícitos.

2.6.3 Cenários Negativos After/Until

Consideremos o diagrama exposto na Figura 2.9, onde existe uma chamada duplicada da mensagem *pressure*. O problema é que o componente *Sensor* está sendo desativado exatamente após o cenário *Analysis.* A abordagem correta seria: após (*after*) uma solicitação de informações ao componente *Database*, não (*not*) permitir que o componente *Sensor* seja

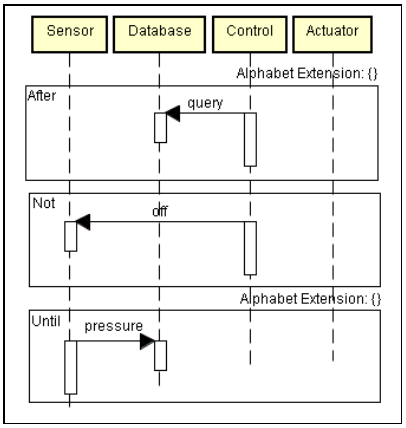

Figura 2.11: Cenário negativo *after/until* do sistema de caldeira.

desativado, até que (*until*) uma mensagem do tipo *pressure* tenha sido enviada para o componente *Database*.

Podemos definir, a partir do exemplo acima, cenários negativos do tipo *after/until* como sendo compostos por três seções distintas chamadas *after*, *not* e *until*, onde as seções *after* e *until* são bMSCs que determinam o escopo na qual a mensagem, presente na seção *not*, deve ocorrer. A Figura 2.11 mostra um exemplo dessa notação para o sistema de caldeira. Nessa figura, o conjunto de diagramas mostrado descreve a seguinte situação: depois da mensagem *query* ter sido disparada, a mensagem *off*, destinada ao componente Sensor, somente pode ser enviada se uma mensagem do tipo *pressure* for enviada anteriormente ao componente *Database*.

Cenário Negativo After/Until: um cenário negativo *after/until* é uma estrutura $n = \{\alpha, E_a, l, u, E_u\}$ onde:

- α e u são bMSCs que modelam as seções *after* e *until* de *n;*
- l é o rótulo da mensagem proscrita;
- E_a e E_u são conjuntos de rótulos de mensagens que estendem o alfabeto das seções *after* e *until*.

Adicionalmente, é requerido que $l \notin \alpha(u) \cup E_u$. O alfabeto de n é denotado por $\alpha(\mathrm{n}) = \alpha(\mathrm{a}) \cup E_a \cup \alpha(u) \cup E_u \cup \{l\}$.

2.7 Sumário

Este capítulo definiu e discutiu o conceito de cenários implícitos utilizando o formalismo proposto por Uchitel *et al.* (2004). O próximo capítulo apresenta o estado da arte em relação a abordagens existentes para extração de cenários e detecção de cenários implícitos.

Capítulo 3

Abordagens para Extração de Cenários e Detecção de Cenários Implícitos

Este capítulo analisa criticamente vários trabalhos existentes para extração de cenários e detecção de cenários implícitos.

Como mencionamos anteriormente no capítulo 1, a principal motivação por trás da descoberta de cenários implícitos em sistemas existentes reside na identificação de possíveis comportamentos indesejados que devem ser evitados, além do enriquecimento da especificação original dos sistemas.. Porém, a forma como os sistemas podem ser monitorados para a extração de seus cenários operacionais, as técnicas necessárias para o processo de abstração de rastros de execução e os métodos e algoritmos utilizados para a detecção dos cenários implícitos, tornam a integração desse conjunto um processo desafiador.

Este capítulo descreve as principais abordagens existentes na linha de engenharia reversa de cenários e que ferramentas comerciais ou acadêmicas estão sendo utilizadas para este fim. Inicialmente, os trabalhos de âmbito acadêmico são classificados quanto ao tipo de análise utilizada no processo de obtenção de rastros de execução de cenários, quais sejam, análise dinâmica e análise estática. Em seguida, são abordados alguns trabalhos que propõem técnicas de diminuição da complexidade de rastros de execução. Na seção seguinte, são relacionados os principais produtos disponíveis comercialmente ou no meio acadêmico para a recuperação de cenários no intuito de facilitar o processo de compreensão de sistemas existentes. Por fim, é apresentada uma ferramenta para a detecção automática de cenários implícitos, utilizada inicialmente para facilitar o processo de especificação de requisitos de sistemas concorrentes.

3.1 Recuperação de Cenários através de Análise Dinâmica

Muitas ferramentas de engenharia reversa empregam a análise dinâmica do comportamento de programas em execução a fim de recuperar cenários representados como diagramas UML ou da família ITU-T. No âmbito acadêmico, três trabalhos merecem destaque. A seguir, apresentamos os trabalhos de Briand *et al.* (2003), Hamou-Lhadj *et al.* (2005) e Richner e Ducassse (2002), representantes dessa categoria.

3.1.1 Briand et al.

Briand *et al.* (2003) propõem a definição de um método de engenharia reversa de cenários no formato de diagramas de seqüência a partir de rastros de execução. A estratégia utilizada prevê a instrumentação do código fonte da aplicação, com posterior análise dos rastros gerados para identificação de repetições de chamadas correspondentes a *loops*. Para isso, foi construído um metamodelo de dados que representa os principais elementos contidos em um diagrama de seqüência. Esse metamodelo foi criado para identificar os requisitos necessários para a instrumentação do código fonte. A ferramenta construída produz representações textuais de diagramas de cenários, e a visualização dos diagramas ocorre através da importação das informações para uma ferramenta de modelagem UML, utilizando, por exemplo, de documentos no formato XMI (OMG, 1999).

O desenvolvimento da ferramenta utilizou a linguagem Perl (Wall *et al.*, 2000) para realizar a instrumentação do código fonte, e a linguagem Java (SUN Microsystems, 2007) para a transformação dos rastros em diagramas de cenários. A linguagem das aplicações alvo da engenharia reversa foi C++ (Holzner *et al.*, 2000).

Como estudo de caso, foi realizada a engenharia reversa de diversos cenários a partir de um sistema *ATM* (*Automated Teller Machine*). O resultado foi a recuperação de diversos diagramas de cenários consistentes com os primeiros diagramas criados pelos projetistas do sistema antes de sua concepção, além da descoberta de diversos detalhes não especificados originalmente pelos projetistas.

3.1.2 Hamou-Lhadj et al.

Hamou-Lhadj *et al.* (2005A) propõem um ambiente para a recuperação de modelos de comportamento a partir de rastros de execução. Um diferencial do ambiente é a remoção de componentes utilitários dos modelos recuperados pela técnica de filtro de rastros de execução conhecida como análise *fan-in* (Hamou-Lhadj e Lethbridge, 2005B). Essa técnica é baseada na exploração do grafo de dependência entre as classes do sistema, obtido através da análise estática do código, utilizada para identificar as classes mais referenciadas.

Para representar os modelos comportamentais de alto nível foram utilizados modelos UCM (ITU-T, 2003), ao contrário de diagramas de seqüência UML, comumente utilizados nesse contexto. Modelos UCM são parte da família ITU-T de linguagens para descrição de

requisitos funcionais e projetos de alto nível. Os autores escolheram essa abordagem por entenderem que esses modelos facilitavam a visualização hierárquica das principais seqüencias de responsabilidades combinada com componentes arquiteturais, abstraindo detalhes da troca de mensagens.

Para avaliar os resultados da pesquisa, foram analisados os rastros de execução de um sistema orientado a objetos chamado *TConfig*. Os modelos de comportamento resultantes foram validados pelo principal projetista do sistema.

3.1.3 Richner e Ducassse

Richner e Ducasse (2002) apresentam uma proposta de recuperação de interações e colaborações utilizando informação dinâmica. Os autores focam no entendimento da interação entre papéis das classes do sistema.

A proposta apresenta as informações de colaboração em termos de classes (que enviam e recebem mensagens) e métodos invocados, representando o comportamento do sistema alvo em termos de padrões de colaboração. Tais padrões permitem aos desenvolvedores solicitar cada um dos itens representados em termos de outros no sistema.

Para validar a pesquisa, foi desenvolvida a ferramenta chamada *Collaboration Browser*. Os autores ilustram, por intermédio de exemplos, como a ferramenta é utilizada na solicitação de informações de execução sobre colaborações e interações em programas desenvolvidos com a linguagem Smalltalk (Lewis *et al*., 1995).

Para a modelagem dos comportamentos foi utilizada a notação dos diagramas de colaboração da UML (OMG, 2002).

3.2 Recuperação de Cenários através de Análise Estática

Outros trabalhos de engenharia reversa empregam a análise estática para a recuperação de diagramas de seqüência da UML ou da família ITU-T a partir de artefatos de código. No âmbito acadêmico, dois trabalhos merecem destaque. Para representar essa categoria, nós apresentamos os trabalhos de Rountev *et al.* (2005) e Mansurov e Campara (2001).

3.2.1 Rountev et al.

Rountev *et al.* (2005) propõem um algoritmo de análise estática que cria estruturas de dados que mapeiam grafos de controle de fluxo de métodos às primitivas de controle da UML

2.0 (OMG, 2003). Em uma segunda etapa, as estruturas de dados resultantes do mapeamento são processadas através de técnicas de simplificação de fragmentos advindos da UML 2.0, e em seguida são utilizadas para construir diagramas de seqüência.

O algoritmo proposto foi implementado como parte da ferramenta *Red*, especializada na engenharia reversa de diagramas de seqüência da UML 2.0 com foco em programas Java.

3.2.2 Mansurov e Campara.

O trabalho de Mansurov e Campara (2001) discute como atividades de manutenção em software podem ser agilizadas com a apresentação dos principais cenários de execução do sistema aos desenvolvedores. A proposta consiste na construção e análise de mapas de cenários (casos de uso do sistema) a partir de sistemas existentes.

É apresentada uma solução, baseada em análise estática, para a extração dos cenários pela navegação e captura de sentenças de código que possam representar eventos de interação do sistema. Após a captura, os rastros detectados são transformados automaticamente em diagramas do tipo MSC.

Os autores desenvolveram o ambiente *PathFinder*, que suporta a captura manual dos cenários principais com um custo-benefício efetivo, pois é utilizada para capturar, gravar, editar, animar e transformar cenários de maneira automática, os quais são representados como rastros de execução no código fonte.

A metodologia consiste inicialmente na escolha dos casos de uso principais da aplicação a ser compreendida, pois segundo os autores, o conhecimento informal dos cenários principais é de grande ajuda no direcionamento do que se quer extrair utilizando a análise estática. Com a seleção dos casos de uso feita, são identificadas as localizações específicas de coleta de informações (âncoras) no código fonte para posterior instrumentação. A partir desse ponto, e para cada caso de uso, é feita a captura dos rastros de execução. Esse passo é responsável pela captura dos rastros contidos no código fonte, possibilitando a navegação e visualização das estruturas capturadas. Após a captura, os rastros de execução se encontram muito detalhados (representam fielmente o próprio código fonte) e precisam ser trabalhados para facilitar a compreensão. Assim, eles são refinados com técnicas de adição e exclusão de métodos, refinamento de eventos, omissão de interfaces e captura de cenários secundários. Depois do refinamento, a ferramenta *PathFinder* pode realizar a transformação dos rastros

para várias representações, dentre elas: MSC (ITU-T, 2004), diagramas de colaboração UML (OMG, 2002) e diagramas de composição de fluxo próprios da ferramenta.

Para validação da proposta, o ambiente *PathFinder* foi utilizado em conjunto com um protótipo de sistema de telecomunicações chamado *ToolExchange*, escrito em C++.

3.3 Estratégias de Diminuição da Complexidade de Rastros de Execução

Um elemento chave para o sucesso da análise dinâmica/estática de rastros de execução consiste na implementação de técnicas eficientes de redução da quantidade de informações presentes nos rastros.

Nessa linha, Hamou-Lhadj e Lethbridge apresentam dois trabalhos que foram utilizados como referência no presente trabalho de pesquisa.

Em (Hamou-Lhadj e Lethbridge, 2005A), os autores têm como objetivo a definição de componentes utilitários em sistemas computacionais, bem como mostrar a diferença desses comportamentos em relação a outros componentes. Além da conceituação, são salientadas as principais vantagens de identificar os componentes utilitários no auxílio a atividades de filtro de grandes rastros de execução para melhor visualização do comportamento do sistema alvo.

Os autores também apresentam as heurísticas utilizadas na detecção de componentes utilitários, as quais são baseadas em técnicas de análise *fan-in* e *fan-out* (Hamou-Lhadj e Lethbridge, 2005B). A freqüência de chamadas (ou grau de dependência) entre componentes do sistema é a base utilizada por essas técnicas para a descoberta de componentes utilitários. Segundo os autores, componentes utilitários podem ser de categorias diferentes e podem possuir escopos também diferentes, dependendo da abrangência e nível de detalhes requeridos no sistema.

Já em (Hamou-Lhadj e Lethbridge, 2003), os autores propõem uma metodologia para facilitar a compreensão de cenários extraídos de rastros de execução, baseada no pré-processamento dos cenários obtidos. Nessa abordagem, os cenários são submetidos de maneira incremental a alterações sucessivas durante todo o processo de compreensão.

A metodologia é composta de três grandes passos que devem ser executados em seqüência. O primeiro passo é baseado na remoção de seqüências de interações repetidas, mais conhecidas como laços. O segundo passo conta com a detecção e exclusão de diferentes tipos de componentes utilitários. Finalmente, o último passo utiliza os conceitos da orientação

a objetos (polimorfismo e herança) para esconder detalhes de baixo nível presentes na implementação. A proposta foi validada em um experimento, onde foram estimados os percentuais de ganho obtidos pela utilização das técnicas apresentadas.

3.4 Ferramentas de Recuperação de Cenários

Essa seção descreve sucintamente algumas das principais ferramentas de engenharia reversa de cenários atualmente disponíveis para apoiar a compreensão de sistemas existentes. Vale salientar que a descrição das ferramentas foi realizada com base apenas nas publicações científicas que as descrevem.

3.4.1 Shimba

Systä *et al*. (2001) apresentam um ambiente de engenharia reversa chamado *Shimba*, que combina análises estática e dinâmica para apoiar a compreensão do comportamento de softwares desenvolvidos utilizando a linguagem Java. A análise estática é utilizada para selecionar um conjunto de componentes que serão analisados posteriormente utilizando a análise dinâmica, restringindo, assim, o escopo da análise.

A ferramenta extrai classes e suas inter-dependências dos arquivos de código fonte do sistema para depois visualizá-las. Tanto a extração quanto a visualização são feitas com auxílio da ferramenta *Rigi* (Müller e Klashinsky, 1988).

A análise dos rastros obtidos é feita com o auxílio de outra ferramenta, *SCED* (Koskimies *et al*., 1996), que permite a representação dos rastros de execução na forma de diagramas de seqüência da UML.

Além disso, para diminuir a complexidade dos rastros de execução obtidos, a ferramenta aplica o algoritmo de *Boyer-Moore* (Boyer e Moore, 1977) para detectar seqüências de eventos repetidos, referenciadas pelos autores como padrões de comportamento.

3.4.2 ISVis

ISVis (Jerding *et al*., 1997; Jerding e Rugaber, 1997) é uma ferramenta de visualização que suporta a análise de rastros de execução extraídos de sistemas orientados a objetos. Utiliza a mesma abordagem da ferramenta *Shimba* para redução de grandes conjuntos de

rastros de execução, pela utilização de um algoritmo que identifica padrões repetidos contidos nos rastros.

Uma alternativa oferecida para a visualização dos rastros de execução é uma variação do diagrama de seqüência da UML, chamado fluxo de mensagens temporal (*temporal message-flow*).

O software emprega outras técnicas que facilitam a compreensão dos cenários extraídos, como: pesquisa por padrões específicos, com a possibilidade de utilização de coringas (*wildcards*) e a possibilidade de alguns grupos de eventos poderem ser abstraídos ou escondidos do cenário geral.

3.4.3 Ovation

A ferramenta *Ovation* (Pauw *et al.*, 1998) utiliza uma estrutura hierárquica em árvore, chamada visão de padrões de execução, para visualizar os rastros de execução de sistemas existentes. Esse tipo de visão difere do formato apresentado pelo diagrama de seqüência da UML, uma vez que permite ao usuário visualizar a execução do programa em diferentes níveis de detalhe.

No intuito de diminuir a complexidade dos rastros de execução, a ferramenta oferece uma visualização mais apurada dos eventos coletados, permitindo que seqüências similares de eventos sejam mostrados como instâncias do mesmo padrão. Na realidade, os padrões passam a ser divididos visualmente pela apresentação de cores diferentes, a fim de permitir um melhor entendimento do comportamento pelos engenheiros de software.

Além disso, a ferramenta oferece ao usuário uma série de critérios que podem ser utilizados para a definição de equivalência entre duas ou mais seqüências de eventos. Os principais critérios são:

1. *Identidade*: duas seqüências de chamadas podem ser consideradas instâncias de um mesmo padrão se elas possuírem a mesma topologia: mesma ordem, objetos, métodos, etc.
2. *Identidade de classe*: se duas seqüências de chamadas envolvem a mesma classe, mas se referem a diferentes objetos, então elas podem ser consideradas similares de acordo com esse critério.

3. *Limitação de profundidade*: esse critério consiste em comparar duas seqüências desde seu início até certa profundidade (cada chamada a um método em uma seqüência de execução acresce a profundidade desta seqüência).
4. *Repetição*: é muito comum ter duas diferentes seqüências de chamadas que diferem entre si somente pelo número de repetições (devido à laços e recursões). Se esse número de repetições é ignorado, essas duas seqüências podem ser consideradas equivalentes.
5. *Polimorfismo*: esse critério sugere que ao se considerar duas subclasses especializadas a partir de uma mesma classe genérica elas podem também ser consideradas equivalentes. Entretanto, esse critério somente é aplicado se elas estejam invocando as mesmas operações polimórficas.

3.4.4 JInsight

JInsight (Pauw *et al*., 1993; Pauw *et al*., 2002) é uma ferramenta especializada em exibir o comportamento e analisar o desempenho da execução de sistemas construídos utilizando a linguagem Java.

Ela permite que o usuário escolha entre diversas visões: Visão de Histograma (permite a detecção de gargalos de desempenho), Visão de Execução (visão da seqüência de passos da execução que permite o entendimento da natureza concorrente de alguns componentes), Visão de Padrão de Referência (mostra a inter-conexão entre objetos utilizada na diminuição da complexidade dos rastros), e Visão de Árvore de Chamada (permite a visão da seqüência, quantidade de invocações e tempo de execução dos métodos chamados).

JInsight usa um modelo para a representação da informação de execução de sistemas orientados a objetos introduzido por Pauw *et al*. (1994). Este modelo utiliza uma estrutura de espaço de eventos de quatro dimensões: classes, instância, métodos e tempo, de onde são retiradas as informações necessárias para a montagem das diferentes visões de usuário.

Para diminuir o tamanho dos rastros de execução, os autores introduzem o conceito de *call frames*. Um *call frame* é uma combinação de eventos que descreve um padrão de comunicação entre um conjunto de objetos no sistema foco da análise. Por exemplo, considere um método *m1* de uma classe *c1* que chama um método *m2* de uma classe *c2*. Essa seqüência tipicamente envolve um objeto *o1* de *c1* e um objeto *o2* de *c2*. Toda a seqüência é salva pela

ferramenta como um único *call frame*, eliminando a necessidade de salvar todo evento simples dessa seqüência. Entretanto, mesmo que essa técnica resulte em uma significante redução do número de eventos, *JInsight* é mais direcionado para análises de performance que propriamente para suporte a compreensão de sistemas legados.

3.4.5 Program Explorer

Program Explorer (Lange e Nakamura, 1997) é uma ferramenta de exploração de código especializada na análise de interações entre objetos e classes em sistemas orientados a objetos desenvolvidos na linguagem C++.

Para superar os problemas decorrentes do tamanho excessivo dos rastros de execução, os autores utilizam diversas técnicas de filtragem. Dentre elas destacam-se as técnicas de união, poda e corte. A primeira prevê a possibilidade do usuário da ferramenta tanto unir chamadas a métodos idênticos entre pares de objetos, quanto unir objetos da mesma classe a fim de reduzir o número de nós do grafo de interações (modelo utilizado pela ferramenta para representar as interações entre elementos). A segunda é utilizada para remover informações sobre objetos, métodos e classes do grafo de interações, bem como os elementos dependentes desses (ex.: chamadas anteriores e posteriores aos elementos). Por fim, a terceira prevê somente a existência dos caminhos de ativação de um dado objeto. Todos os outros são excluídos.

3.4.6 AVID

Walker *et al*. (1998) descrevem a ferramenta *AVID* (*Architecture Visualization of Dynamics in Java Systems*), que permite visualizar o comportamento, obtido através de análise dinâmica, de sistemas orientados a objetos desenvolvidos utilizando a linguagem Java.

A ferramenta permite que o usuário obtenha os rastros de execução em termos de chamadas a métodos e criação e destruição de objetos. O usuário pode controlar a seqüência de eventos que ele quer visualizar, pela quebra do rastro em uma seqüência de visões chamadas de células. O diferencial dessa ferramenta reside no potencial do módulo de animação presente. Esse módulo permite visualizar a execução, célula a célula, dos cenários obtidos, bem como sua parada, o retorno e o avanço de células na animação.

3.4.7 Scene

Scene (Koskimies *et al*. 1995; Koskimies e Mössenböck, 1996) é outra ferramenta utilizada para produzir diagramas de cenário a partir de sistemas existentes.

A ferramenta utiliza algumas técnicas de filtragem para facilitar a visualização dos cenários obtidos. São elas: compressão de chamadas, particionamento, projeção e remoção, e o modo passo simples. A primeira consiste em ocultar as chamadas internas derivadas de uma determinada chamada. A segunda classifica as chamadas internas em partes. Somente serão mostradas as mensagens de uma determinada parte selecionada pelo usuário. A terceira permite que o usuário somente visualize as interações que envolvem um objeto selecionado previamente. Por fim, a quarta técnica permite que sejam mostradas as chamadas internas geradas a partir de uma determinada chamada, um passo por vez.

3.5 Ferramenta para Detecção de Cenários Implícitos

Todos os trabalhos descritos anteriormente focam exclusivamente na tecnologia necessária para extrair diagramas de seqüência (ou similares), que constituem a base para a recuperação de cenários, a partir da análise estática ou dinâmica de aplicações existentes. Trabalhos pioneiros na detecção automática de cenários implícitos foram propostos por Uchitel *et al.* (2003), com o objetivo inicial de facilitar o processo de especificação de requisitos para sistemas concorrentes.

A ferramenta LTSA-MSC (Uchitel *et al.*, 2003) foi a primeira a oferecer suporte automatizado para a detecção de cenários implícitos. Ela foi desenvolvida como uma extensão da ferramenta de verificação de modelos concorrentes LTSA (*Labelled Transition Systems Analyzer*) (Magee e Kramer, 1999). A extensão incorporada pela LTSA-MSC consiste numa linguagem baseada em diagramas MSCs para a descrição de cenários de alto nível (hMSCs) a partir da composição paralela de cenários básicos (bMSCs). Uma especificação formada por um cenário de alto nível e o conjunto de cenários básicos que o compõem constitui o insumo necessário para que a LTSA possa detectar a presença de possíveis cenários implícitos nessa especificação.

O processo de detecção tem seu início com a definição de um diagrama de alto nível (hMSC) pelo projetista. Para tanto, é necessário que o mesmo realize previamente a entrada das informações dos cenários básicos (bMSCs) que se quer estudar. Completada essa etapa, é realizada a síntese automática dos modelos de comportamento pela ferramenta LTSA-MSC,

ou seja, a transformação das especificações MSC em uma especificação de modelo de comportamento equivalente no formato FSP (próprio da ferramenta), ou seja, a transformação preserva os comportamentos descritos nos cenários básicos. Após a síntese, inicia-se a busca por cenários implícitos. Ao se detectar um cenário implícito, o projetista deve categorizá-lo como válido (positivo ou negativo) ou inválido. Se o cenário implícito encontrado for considerado um comportamento válido na aplicação alvo, o mesmo será acrescentado às suas especificações originais, senão ele será descartado. Depois uma nova busca poderá ser realizada. Caso a busca não encontre nenhum cenário implícito, o projetista poderá encerrar o processo indicando a inexistência de comportamentos ainda não especificados, ou alterar a especificação hMSC criada originalmente, refletindo então uma nova visão do relacionamento entre os cenários básicos. Com a alteração do hMSC original, o processo de detecção dos cenários implícitos reinicia, com a realização de outra síntese e novas buscas. A Figura 3.1 mostra um diagrama de atividades que ilustra o processo de detecção de cenários implícitos com a ferramenta LTSA-MSC.

3.6 Discussão

Um grande desafio que permeia o processo de entendimento do comportamento de sistemas existentes, onde a documentação inexiste ou a qualidade deixa a desejar, é extrair visões de alto nível de componentes de baixo nível.

A primeira questão passa pelo tipo de análise que deve ser feita para extrair os dados que irão definir o comportamento dos componentes, ou seja, quando devemos utilizar análise dinâmica de informações de execução ou análise estática do código fonte.

A análise estática tem se mostrado bastante eficiente em sistemas de natureza procedural, enquanto para sistemas orientados a objetos (que utilizam conceitos como polimorfismo e herança), essa mesma análise pode não ser a mais adequada. Para esse tipo de sistema é mais indicada a análise dinâmica. A análise dinâmica consiste basicamente na extração das propriedades de execução de um sistema alvo, oferecendo a vantagem de permitir a utilização de mecanismos de coleta de informações sensíveis aos dados de entrada, diferentemente da coleta realizada pela análise estática, que navega por todos os relacionamentos entre artefatos do sistema.

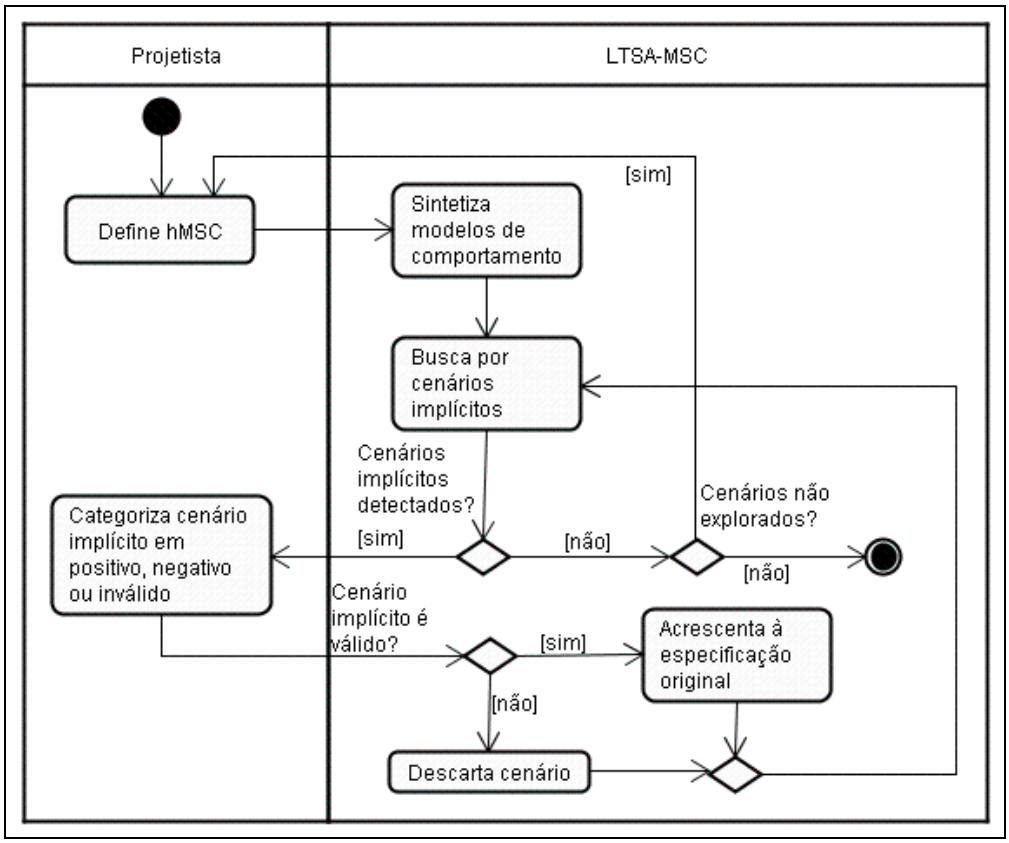

Figura 3.1: Processo de detecção de cenários implícitos com a ferramenta LTSA-MSC.

Outro ponto forte da análise dinâmica é a possibilidade dela ser direcionada para dados de entrada, saída e comportamentais específicos, ajudando os engenheiros de software a entender a aplicação do ponto de vista de suas mudanças de acordo com as diferentes entradas e saídas escolhidas como objeto da análise.

Com relação à extração da informação, outras técnicas além da instrumentação manual do código, podem ser realizadas ao se utilizar a análise dinâmica. A primeira descreve a possibilidade de instrumentação automática dos sistemas envolvidos através da utilização do paradigma da programação orientada a aspectos – AOP (Elrad *et al*., 2001), que não requer a modificação direta do código fonte da aplicação alvo. Também dentro dessa mesma linha, existe a possibilidade de executar o sistema alvo sob o controle de uma ferramenta de depuração. Essa técnica traz a vantagem de não exigir modificações no código fonte nem no seu ambiente de execução. Porém, ferramentas de depuração podem afetar seriamente o desempenho da aplicação durante o processo de coleta.

Quanto à questão do tamanho dos rastros de execução gerados após a coleta, é fundamental a utilização de técnicas de redução da complexidade de rastros a fim de diminuir o seu tamanho e facilitar a sua compreensão. Podemos citar como exemplo as técnicas descritas na seção 3.3.

3.7 Requisitos para um Ambiente de Detecção de Cenários Implícitos em Sistemas Existentes

Em vista das considerações feitas ao longo deste capítulo, podemos concluir que uma abordagem mais efetiva para a detecção de cenários implícitos em sistemas existentes deve atender os seguintes requisitos:

- Utilizar a técnica de análise dinâmica para a coleta de informações sobre a aplicação alvo. Como o foco da pesquisa são sistemas orientados a objetos construídos utilizando a linguagem Java, a análise dinâmica mostra-se mais adequada, uma vez que somente retratará os passos efetivamente utilizados na execução dos cenários operacionais da aplicação alvo. Alternativamente, pode-se utilizar uma abordagem mista, onde tanto a análise dinâmica quanto a estática são utilizadas, em conjunto, no intuito de aumentar a cobertura do comportamento do sistema capturado nos rastros de execução.
- Utilizar análise estática para a descoberta de classes utilitárias, que seriam omitidas dos rastros de execução gerados a partir da análise dinâmica. Essa tática reduziria significamente o tamanho dos cenários extraídos.
- Utilizar uma técnica pouco invasiva para a extração de informações dinâmicas, de modo a minimizar o impacto na alteração da aplicação alvo. Algumas opções viáveis seriam a utilização de AOP ou de uma ferramenta de depuração. A preferência vai para essa última, pois possui a vantagem de não alterar nenhum artefato do sistema alvo.
- Utilizar técnicas efetivas para a diminuição da complexidade dos rastros de execução coletados.
- Permitir representar os rastros de execução no formato de diagramas MSC, fundamentais para a carga dos cenários extraídos da aplicação alvo na ferramenta de detecção de cenários implícitos.

- Ser flexível ao ponto de permitir configurações específicas para atender as necessidades da pesquisa (ex.: geração de documentos em formato XMI para carga em ferramentas de modelagem e de documentos XML para carga na ferramenta LTSA). A maioria das ferramentas de extração de cenários existentes são soluções comerciais que, além de pagas, não permitem esses tipos de configuração.

A busca por uma solução que realize automaticamente a detecção de cenários implícitos e que atenda de forma satisfatória a todos esses requisitos tem sido a principal motivação ao longo deste trabalho de dissertação. A solução proposta como resultado deste estudo será apresentada a partir do próximo capítulo.

Capítulo 4

Um Processo para Detecção de Cenários Implícitos em Sistemas Concorrentes

Este capítulo apresenta o processo proposto para a detecção de cenários implícitos em sistemas concorrentes, bem como dá detalhes da implementação de suas ferramentas de apoio.

4.1 Visão Geral do processo

O processo para detecção de cenários implícitos proposto neste trabalho é ilustrado no formato de um diagrama de atividades presente na Figura 4.1. O processo é composto de quatro etapas principais, cada uma delas executada por uma ferramenta diferente. São elas: (i) seleção de cenários de interesse e geração de seus respectivos rastros de execução (raias 1 e 2 na Figura 4.1); (ii) extração de elementos do cenário a partir de rastros de execução (raia 3); (iii) detecção de cenários implícitos em uma especificação baseada em cenários definida sobre os cenários extraídos (raia 4); e (iv) visualização de cenários no formato de diagramas de seqüência da UML (raia 5). A quarta etapa é opcional, e destina-se a facilitar a compreensão dos cenários extraídos utilizando ferramentas de modelagem UML para visualizá-los.

A próxima seção descreve as três primeiras etapas do processo em maiores detalhes.

4.2 Etapas do processo

4.2.1 Seleção dos Cenários e Geração dos Rastros de Execução

O processo se inicia com a escolha de quais cenários ou conjuntos de funcionalidades da aplicação alvo serão executados para a geração dos rastros via análise dinâmica. Essa escolha normalmente é feita com a ajuda de um usuário ou desenvolvedor da aplicação, ou de algum especialista em seu domínio. Em seguida, a aplicação alvo é configurada para que sua execução seja monitorada por uma ferramenta denominada Monitor de Eventos desenvolvida pelo autor. O Monitor de Eventos ficará responsável por monitorar cada evento ou ação (criação de objetos, chamada de métodos, etc.) executada pela aplicação. As ações classificadas como parte de um ou mais dos cenários selecionados serão incluídas no(s) rastro(s) de execução correspondente(s) a esse(s) cenário(s).

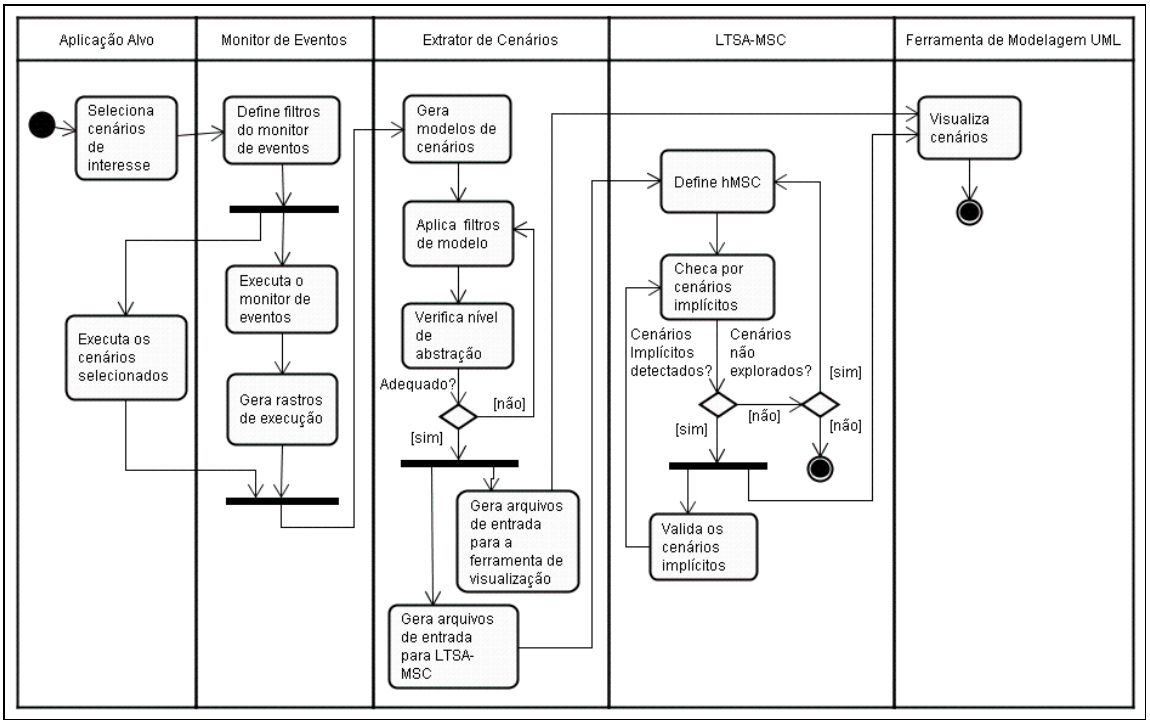

Figura 4.1: Processo de detecção de cenários implícitos em sistemas concorrentes.

Os elementos que compõem um rastro de execução foram definidos a partir de um metamodelo de rastros de execução (Figura 4.2). Este metamodelo foi criado para padronizar o registro e a manipulação das informações contidas em cada rastro, visando facilitar sua abstração na forma de cenários. O metamodelo foi definido com base no metamodelo proposto por Briand *et al.* (2003) para a recuperação de diagramas de seqüência da UML.

O elemento principal do metamodelo é a classe *Cenario*, que representa unicamente um cenário da aplicação (ex.: Incluir aluno, Realizar compras, etc) monitorado pelo Monitor de Eventos. Essa classe possui um identificador único (*id*) para as suas instâncias como também uma breve descrição (*descricao*). Cada cenário monitorado possui associado a ele um conjunto de instâncias (classe *Instancia*) e um conjunto de mensagens (classe *Mensagem*) atribuídas a essas instâncias, sendo ambos associados ao elemento cenário por meio de coleções de itens. A classe *Instancia* possui como único atributo um identificador (*nome*) único dentro do cenário ao qual está associada. Além de um identificador (*id*) único dentro do cenário, a classe Mensagem possui como atributos um índice (*indice*) que representa a sua ordem de execução no cenário, um nome (*nome*) que representa o nome do método invocado entre as instâncias, a identificação da linha de execução (*thread*) que originou a mensagem, e

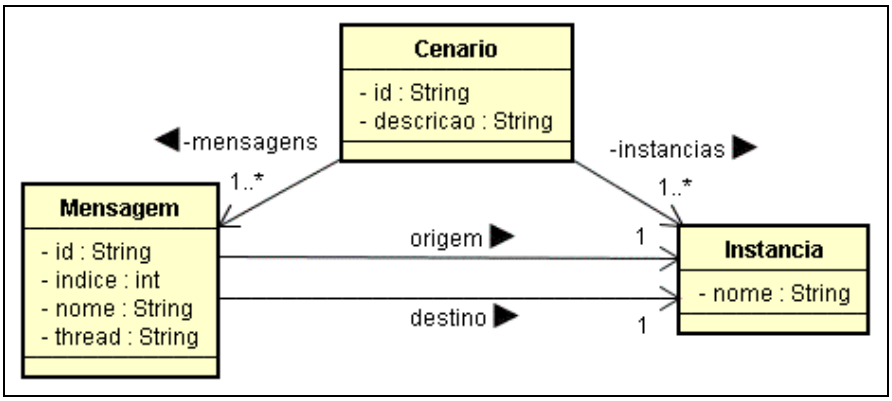

Figura 4.2: Metamodelo de rastros de execução.

as instâncias de origem (*origem*) e destino (*destino*) da mensagem.

Ao monitorar cada um dos cenários selecionados, o Monitor de Eventos registra os elementos que compõem seu rastro de execução em estruturas de dados baseadas no metamodelo descrito acima. Essas estruturas são mantidas em memória e podem facilmente ser traduzidas em documentos XML compatíveis com o metamodelo utilizando ferramentas de transformação de documentos XML, por exemplo, XSLT (W3C, 1999).

Outra grande vantagem de se utilizar um metamodelo para a geração de rastros de execução é que ele permite a coleta e manipulação dos elementos dos rastros de forma independente da ferramenta utilizada como Monitor de Eventos e da linguagem de programação da aplicação alvo. Isto porque a estrutura do metamodelo representa genericamente quaisquer interações entre componentes de código em sistemas orientados a objetos. Assim, ele serve como ponte para a extração de cenários e descoberta de cenários implícitos em sistemas escritos em diferentes linguagens, bastando, para isso, que sejam utilizadas ferramentas de monitoração apropriadas para cada linguagem.

4.2.2 Extração e Abstração de Cenários

A abstração dos elementos que compõem os rastros de execução da aplicação na forma de cenários é um processo subjetivo e complexo, e que pode variar conforme o cenário que está sendo extraído. Na realidade, não existe uma forma única para conseguir êxito nesse tipo de operação, exatamente porque cada cenário possui características próprias. Apenas um estudo minucioso de cada cenário, suas características e o ambiente arquitetural envolvido poderá indicar que passos serão utilizados no processo de abstração e em que ordem. Tal

Tabela 4.1: Elementos filtrados do rastro de execução em cada nível de abstração

Nível de Abstração	Elementos Filtrados
1	Entidades externas ao(s) pacote(s) utilizado(s) pela aplicação.
2	Classes periféricas ao processo de negócio do cenário e classes utilitárias.
3	Chamadas a métodos construtores de classe.
4	Chamadas a métodos de uma mesma classe.
5	Padrões de mensagens repetidos (laços).

estudo deve ter como base um conhecimento detalhado do processo de negócio representado no cenário bem como o seu contexto na aplicação.

Em sistemas concorrentes, caracterizados por um alto grau de compartilhamento de recursos, é comum haver um ou mais componentes de software que atuam em múltiplos cenários. Citemos a existência de componentes de negócio que aparecem em diversos cenários de um mesmo sistema, onde a cada utilização podem desempenhar funcionalidades distintas. Conforme descrito anteriormente, é justamente o fato desses componentes poderem apresentar diferentes comportamentos em cada cenário que participam que propicia o aparecimento de cenários implícitos.

A estratégia de abstração de cenários proposta consiste na configuração manual de filtros e modificadores de conteúdo e posteriores aplicações sucessivas dos mesmos aos elementos originalmente capturados nos rastros, por uma ferramenta denominada Extrator de Cenários desenvolvida pelo autor. Operacionalmente, os cenários têm seus fluxos de mensagens modificados interativamente pelo Extrator de Cenários. Cada saída da ferramenta corresponde a uma nova versão do cenário utilizado como entrada e, caso o nível de abstração desejado ainda não tenha sido alcançado, esta poderá servir de entrada para um novo conjunto de filtros e modificadores de conteúdo. Esse refinamento sucessivo e incremental permite que o especialista de negócio defina o nível de abstração mais adequado para representar o cenário, podendo voltar a submetê-lo à ferramenta ou finalizar o processo quando estiver satisfeito com o resultado.

A atual versão do Extrator de Cenários inclui cinco tipos de filtros. Esses filtros, quando aplicados aos elementos capturados nos rastros de execução, produzem cenários representados em cinco diferentes níveis de abstração. A Tabela 4.1 descreve quais elementos dos rastros são filtrados em cada um desses cinco níveis.

O primeiro nível de abstração tem o objetivo de melhor isolar o que se quer visualizar do sistema. Ele delimita a primeira fronteira do que vai ser extraído e garante que o universo de mensagens coletado deverá pertencer exclusivamente ao(s) pacote(s) utilizado(s) na aplicação alvo, excluindo, assim, mensagens que envolvam classes externas (ex.: classes de pacotes distribuídos na plataforma Java).

O segundo nível de abstração isola o processo de negócio, preservando somente os componentes de software diretamente envolvidos na implementação do(s) cenário(s) de interesse. Também nesse nível são removidas todas as classes consideradas utilitárias, ou seja, classes que oferecem funcionalidades ou serviços comuns a diversas outras classes da aplicação, e que, portanto, pouco contribuem para o entendimento dos objetivos específicos do cenário. O mecanismo de identificação de utilitários implementado é baseado na análise do grau de entrada (*fan-in*) e de saída (*fan-out*) de cada vértice (instância) que compõe o fluxo de mensagens capturado no rastro de execução, utilizando a abordagem proposta por Hamou-Lhadj e Lethbridge (2005).

O terceiro e o quarto níveis de abstração eliminam informações consideradas irrelevantes do ponto de vista da interação entre os componentes, no caso, chamadas a métodos construtores de classe e chamadas a métodos internos (definidos na própria classe), respectivamente.

Por fim, o quinto nível de abstração elimina padrões de mensagens que se repetem entre os componentes do cenário. A recorrência desses padrões geralmente está associada à utilização de instruções de controle de repetição pela aplicação, de modo que sua representação de forma repetida pouco acrescenta ao melhorar o entendimento do objetivo principal do cenário.

É importante enfatizar que o processo não impõe nenhuma estratégia rígida de extração de cenários, e tanto os níveis de abstração quanto os tipos de filtros definidos para cada nível podem ser alterados de acordo com as necessidades dos desenvolvedores e as características da aplicação alvo.

4.2.3 Detecção de Cenários Implícitos

Nesta etapa, os cenários extraídos na etapa anterior são exportados para um formato compatível com os arquivos de entrada esperados pela ferramenta de detecção de cenários implícitos, LTSA-MSC (Uchitel *et al*., 2003). Especificamente, cada cenário recuperado é

representado como um cenário básico (bMSC). Em seguida, um cenário de alto nível (hMSC) é criado manualmente descrevendo os relacionamentos entre os diversos cenários básicos. Conforme descrito anteriormente, esse conjunto de cenários constitui o insumo necessário para a identificação de cenários implícitos pela ferramenta LTSA-MSC.

O processo de detecção de cenários implícitos também ocorre de forma incremental, com o analista de negócios modificando ou refinando sucessivamente o diagrama hMSC, de acordo com os resultados produzidos pela LTSA-MSC.

Ao se submeter uma ação de detecção de cenários implícitos à ferramenta LTSA-MSC, poderão ser mostrados uma série de cenários inexistentes (cenários implícitos) na especificação MSC positiva original. A cada cenário implícito encontrado o projetista deverá categorizá-lo como positivo (situação possível que não incorre em falhas, não documentada inicialmente nas especificações originais do sistema), negativo (indica uma situação possível, não documentada inicialmente, mas que sua execução poderá ocasionar falhas) ou simplesmente ignorá-lo, por não representar uma situação possível na sua visão. Para as primeiras e segundas opções, a ferramenta irá acrescentar o cenário encontrado como um bMSC na especificação, fazendo com que o mesmo também sirva de insumo para a descoberta de mais cenários implícitos.

No caso em que nenhum cenário implícito é encontrado, o analista pode tentar modificar o diagrama hMSC com a remoção de alguns cenários e/ou inclusão de novos cenários. Em último caso, o analista pode decidir por retornar à primeira etapa do processo, e extrair novos cenários da aplicação alvo, ou então descartar o conjunto de cenários investigados como estando livre de cenários implícitos.

4.3 Ambientes de Suporte

Como descrito na seção anterior, o suporte ao processo de detecção de cenários implícitos proposto neste trabalho inclui um Monitor de Eventos, para gerar os rastros de execução da aplicação alvo; um Extrator de Cenários, para abstrair os elementos dos cenários a partir das informações contidas nos rastros de execução; e a ferramenta LTSA-MSC, para a detecção dos cenários implícitos. Uma vez que a ferramenta LTSA-MSC pôde ser reutilizada integralmente, o esforço de implementação do ambiente ficou concentrado no Monitor de Eventos e no Extrator de Cenários, descritos a seguir.

4.3.1 Monitor de Eventos

O Monitor de Eventos foi desenvolvido utilizando a plataforma Java 2, Standard Edition, v.6.0 (Sun Microsystems, 2007). Como mecanismo de captura dinâmica dos eventos gerados pela aplicação alvo, a ferramenta utiliza a arquitetura JPDA (*Java Platform Debugger Architecture*) (Sun Microsystems, 2004), que provê uma infra-estrutura genérica e distribuída para o desenvolvimento de depuradores para a linguagem Java. Dessa forma, nenhuma mudança é necessária no código fonte ou *bytecode* da aplicação alvo para a captura dos eventos, bastando que a mesma seja executada sob controle da plataforma JPDA.

4.3.2 Extrator de Cenários

O extrator de cenários também foi desenvolvido na plataforma Java. Ele utiliza tecnologias existentes de geração e manipulação de documentos XML, no caso, XSLT (W3C, 1999), para aplicar os filtros de abstração aos elementos do rastro de execução, bem como para converter os cenários extraídos nos formatos compatíveis com a ferramenta LTSA-MSC e ferramentas de modelagem. A conversão para ambos os formatos é feita apoiada num catálogo de transformações XSLT, criadas especificamente para este fim. A definição dos filtros de eventos empregados em cada nível de abstração é feita em um arquivo de configuração que serve de insumo para o Extrator de Cenários, utilizando uma sintaxe baseada em XML com elementos marcadores específicos para cada tipo de filtro.

Uma funcionalidades oferecidas pelo Extrator de Cenários para aumentar o nível de abstração dos cenários investigados é a possibilidade de identificação e ocultamento de elementos utilitários no sistema. Segundo a especificação UML (OMG, 2002), uma classe utilitária é definida como um grupo de variáveis globais e procedimentos no formato de uma declaração de classe. Não significando uma construção fundamental, mas uma conveniência na programação.

Essa definição é muito específica para abranger um entendimento completo do que vem a ser um utilitário, pois se restringe somente a classes utilitárias. Ao se estender o conceito a outros tipos de artefatos, nota-se que um utilitário também pode ser um método, pacote ou outro elemento no sistema e, podem ser acessados de diversas formas e pontos diferentes e não de uma única maneira.

Seguindo uma estratégia similar à abordagem proposta por Hamou-Lhadj *et al.* (2005A) para a identificação de componentes utilitários em sistemas computacionais, o

Extrator de Cenários inclui um mecanismo de busca de elementos utilitários no repositório onde residem os arquivos fontes da aplicação alvo.

A busca é realizada de maneira estática nas estruturas de cada arquivo fonte Java e nela são identificadas todas as dependências entre classes na aplicação. O resultado do tratamento é um grafo de dependências entre classes do sistema, base para a identificação dos elementos utilitários.

Para isso foi implementado um algoritmo baseado em análise *fan-in* para detectar somente classes utilitárias dentro do sistema, sendo a restrição ao tipo de utilitário imposta para simplificar o processo de identificação. A heurística por trás dessa análise é identificar um utilitário pela quantidade de acessos feitos a ele por componentes diferentes. Da mesma forma, se um componente faz muitas chamadas a outros componentes, este é muito complexo e possui uma alta taxa de acoplamento, portanto, não pode ser considerado um utilitário. Assim, sua aplicabilidade na solução é baseada na exploração do grafo de dependência de classes montado. Ela é utilizada para extrair classes que possuam um grande número de arestas de entrada (por exemplo, muitas dependências). O mecanismo poderá ser configurado a fim de mudar seu foco de busca possibilitando um ajuste da métrica de acordo com o escopo desejado do utilitário. Dessa forma, diversos níveis de utilitários podem ser identificados, como utilitários por sistema, por componentes de negócio, por pacotes, etc.

Vale salientar que o mecanismo somente identificará e ordenará (por grau de dependências externas) as possíveis classes utilitárias, cabendo ao especialista do sistema decidir o que realmente deverá ser omitido para simplificação dos rastros de execução.

4.3.3 Documentos Resultantes dos Processos de Monitoração e Extração de Cenários

Esta subseção descreve a estrutura dos documentos XML gerados pelo ambiente de suporte durante o processo de monitoração de eventos e extração de cenários, culminando com a carga na ferramenta LTSA-MSC para identificação de cenários implícitos.

São dois os documentos gerados pelo ambiente. Ambos descrevem os rastros de execução extraídos de cenários operacionais da aplicação alvo, embora representados por estruturas diferentes.

O primeiro documento (Figura 4.3) contém a descrição do cenário no formato do metamodelo de rastros de execução criado, cuja estrutura de classes foi descrita na Figura 4.2. Nele, os componentes e as mensagens trocadas entre esses componentes são representados

```
<?xml version='1.0' ?>
 <Cenario>
  <id/>
  <descricao>VerificarProduto</descricao>
  <instancias>
     <Instancia><nome>shopControllerImpl</nome></Instancia>
     <Instancia><nome>stockControllerImpl</nome></Instancia>
     <Instancia><nome>stockDaoImpl</nome></Instancia>
  </instancias>
  <mensagens>
     <Mensagem>
       <nome>findStockItemById</nome>
       <destino>stockControllerImpl</destino>
       <id/>
       <indice>1</indice>
       <origem>shopControllerImpl</origem>
       <thread>http-8080-1</thread>
     </Mensagem>
     <Mensagem>
       <nome>findById</nome>
       <destino>stockDaoImpl</destino>
       <id/>
       <indice>2</indice>
       <origem>stockControllerImpl</origem>
       <thread>http-8080-1</thread>
     </Mensagem>
     <Mensagem>
       <nome>verifyStockItemAvailability</nome>
       <destino>stockControllerImpl</destino>
       <id/>
       <indice>3</indice>
       <origem>shopControllerImpl</origem>
       <thread>http-8080-1</thread>
     </Mensagem>
  </mensagens>
 </Cenario>
```

Figura 4.3: Documento XML representativo do metamodelo de dados de execução.

através de elementos específicos, os quais estão posicionados seqüencialmente de acordo com sua ordem de aparecimento no cenário, ou seja, sua ordem de execução conforme capturada pela ferramenta de monitoração de eventos.

A estrutura do documento se subdivide em uma seção principal e em duas subseções. A seção principal é representada por todas as informações contidas no elemento <*Cenario*>. Nela, é feita a identificação e descrição do cenário capturado. Ela possui duas subseções distintas: a de intâncias e a de mensagens.

A subseção de instâncias identifica todos os componentes presentes em um determinado cenário operacional. Ela é composta pelo elemento <*instancias*>, que possui um conjunto de elementos <*instancia*> que são os descritores dos componentes que podem possuir formatos diferentes de acordo com a configuração adotada.

```
<?xml version="1.0" encoding="UTF-8"?>
<specification>
 <hmsc>
  <bmsc name="init" x="170" y="30" />
  <bmsc name="VerificarProduto" x="170" y="230" />
 </hmsc>
 <bmsc name="init" />
 <bmsc name=" VerificarProduto ">
  <instance name="shopControllerImpl">
   <output timeindex="1">
        <name>shopControllerImpl,stockControllerImpl,findStockItemById</name>
        <to>stockControllerImpl</to>
   </output>
   <output timeindex="3">
        <name>shopControllerImpl,stockControllerImpl,verifyStockItemAvailability</name>
        <to>stockControllerImpl</to>
   </output>
  </instance>
  <instance name="stockControllerImpl">
   <input timeindex="1">
    <name>shopControllerImpl,stockControllerImpl,findStockItemById</name>
    <from>shopControllerImpl</from>
   </input>
   <output timeindex="2">
        <name>stockControllerImpl,stockDaoImpl,findById</name><to>stockDaoImpl</to>
   </output>
   <input timeindex="3">
       <name>shopControllerImpl,stockControllerImpl,verifyStockItemAvailability</name>
       <from>shopControllerImpl</from>
   </input>
  </instance>
  <instance name="stockDaoImpl">
   <input timeindex="2">
       <name>stockControllerImpl,stockDaoImpl,findById</name>
       <from>stockControllerImpl</from>
   </input>
  </instance>
 </bmsc>
</specification>
```

Figura 4.4: Exemplo de documento XML utilizado como entrada para a ferramenta LTSA-MSC.

A subseção de mensagens, por sua vez, descreve como se dá a troca de mensagens entre os componentes. Ela expõe detalhes do fluxo de execução realizado pelos componentes descritos na subseção de instâncias. Todas essas mensagens são descritas como elementos <*Mensagem*> contidas no elemento <*mensagens*>. O Apêndice I apresenta um quadro resumo da estrutura desse documento.

O segundo documento (Figura 4.4) representa o cenário em um formato próprio para a carga dos dados na ferramenta LTSA-MSC. Uma vez carregadas, as informações dos cenários serão visualizadas no formato de diagramas hMSC e bMSC. Para tanto, alguns elementos necessitam de informações extras para o seu posicionamento na tela da ferramenta, como

pode ser observado para os sub-elementos da tag <*hmsc*>. O Apêndice II apresenta um quadro resumo explicando cada elemento desse documento.

4.4 Sumário

Este capítulo apresentou um processo para detecção de cenários implícitos em sistemas existentes, bem com detalhes de implementação de seu ambiente de suporte. O próximo capítulo ilustra a utilização do processo proposto e das ferramentas implementadas em um estudo de caso.

Capítulo 5

Utilização do Processo em um Estudo de Caso

Este capítulo apresenta um estudo realizado para demonstrar a utilização do processo de detecção de cenários implícitos proposto neste trabalho.

Em um esforço inicial de validação, o processo e o ambiente de suporte propostos foram utilizados em um estudo de caso, com o objetivo de detectar possíveis cenários implícitos em uma aplicação existente de comércio eletrônico. Aplicações de comércio eletrônico são caracterizadas por cenários que apresentam um alto grau de concorrência, particularmente em operações que acessam seus componentes de negócios. Isso torna os cenários extraídos a partir da execução dessas aplicações potenciais candidatos para a detecção de cenários implícitos.

5.1 Aplicação Alvo

A aplicação alvo escolhida para o estudo foi a MyPetStore, que é uma versão simplificada desenvolvida pelo autor, da aplicação de comércio eletrônico Pet Store (SUN Microsystems, 2005). Como o nome indica, Pet Store é uma aplicação para comercialização de animais em uma loja virtual de animas de estimação. Ela é uma aplicação de código aberto, desenvolvida e mantida pela SUN com o intuito de ilustrar os principais recursos de programação (interfaces dinâmicas, persistência, segurança, transações, etc.) disponíveis na plataforma *Java 2 EE* (SUN Microsystems, 2005). No caso da MyPetStore, foram implementadas apenas algumas funcionalidades básicas referentes à comercialização de produtos, além de operações para a manutenção de usuários e dos produtos comercializados, compreendendo um total de 21 classes. A Tabela 5.1 apresenta uma relação resumida das funcionalidades e operações implementadas na MyPetStore. Um quadro mais completo pode ser encontrado no Apêndice III.

A razão da opção pela MyPetStore, ao invés da PetStore original, é de ordem estritamente prática, já que com a primeira teríamos total controle sobre como cada cenário de uso da aplicação estaria implementado em nível de código. Como mencionado anteriormente, esse nível de conhecimento é fundamental para guiar o processo de extração e abstração de cenários a partir dos rastros de execução gerados. Uma vez que nosso principal interesse era

Tabela 5.1: Funcionalidades e operações implementadas na aplicação MyPetStore

Funcionalidade	**Operações Implementadas**
Entrar e sair do Sistema	*Login, Logout*
Manter usuários	*Find, Insert, Update, Delete*
Manter produtos	*Find, Insert, Update, Delete*
Realizar compras	*ClearShopCart, ListShopCart, CloseShopCart, InsertItemInShopCart*

validar o processo proposto, ao invés de precisamente re-documentar uma aplicação existente, a utilização da MyPetStore como aplicação alvo provou ser a solução de melhor custo-benefício, antes de avaliarmos o processo proposto e suas ferramentas de suporte em aplicações concorrentes de maior porte. Além disso, o fato da MyPetStore também ter sido desenvolvida seguindo as melhores práticas e padrões de projetos recomendados pela SUN para a plataforma *Java 2 EE* reforça a nossa confiança de que os resultados obtidos nesse estudo também poderão ser reproduzidos em outras aplicações concorrentes com características similares.

5.2 Seleção de Cenários e Geração de Rastros de Execução

Devido à natureza concorrente da aplicação alvo, algumas de suas operações podem ser executadas no contexto de mais de um cenário. Como exemplo, temos o caso da operação de encerramento do carrinho (*CloseShopCart*), que, além de desencadear o cenário de preparação para a compra, ainda dispara os de validação do usuário para a compra, o de verificação de itens em estoque, e o de efetivação da compra propriamente dita. Esse nível de associação ou dependência entre os cenários ilustra bem a dificuldade de se definir exatamente onde começa e onde termina cada cenário, e realça ainda mais a necessidade da presença de especialistas da aplicação durante o processo de extração. A Tabela 5.2 descreve os cenários escolhidos para o estudo, bem como as operações da aplicação alvo que foram monitoradas para gerar os seus respectivos rastros de execução.

5.3 Extração e Abstração dos Cenários

O processo de abstração dos cenários consistiu no processamento das informações contidas nos rastros de execução da aplicação, de acordo com os tipos de filtros definidos para

Tabela 5.2: Cenários selecionados para o exemplo

Cenário	**Operação Monitorada**
Login	*doLogin*
Logout	*doLogout*
Preparar compra	*doShop*
Autenticar usuário	*autenticateUserTask*
Comprar	*saveOrUpdate*
Excluir usuário	*deleteUser*
Verificar itens em estoque	*verifyStockItemAvailability*
Excluir produto	*deleteProduct*

cada nível de abstração.

No primeiro nível de abstração, simplesmente foi definido um filtro estabelecendo o pacote principal da aplicação, *br.unifor.mia.**, como único conjunto de classes a serem analisadas.

Os filtros definidos para o segundo nível de abstração, responsáveis por isolar as classes de negócio e também pela remoção de classes utilitárias, conseguiram obter uma redução significativa (sempre superior a 85%) do conjunto de rastros inicialmente capturado pelo Monitor de Eventos. Em alguns cenários que fazem uso freqüente de componentes visuais e utilitários, como é o caso do cenário de *Exclusão de Usuários*, essa redução foi superior a 97%. A Figuras 5.1 mostra a lista de classes identificadas como candidatas a utilitárias do sistema. A Figura 5.2, por sua vez, mostra as dependências da classe *br.unifor.mia.component.domain.Entity*, identificada como a mais provável utilitária no sistema.

No terceiro nível de abstração (retirada de chamadas a métodos construtores das classes) não existiu regularidade quanto ao percentual de redução conseguido. Em alguns cenários, não houve necessidade de aplicar esse nível de abstração, por se entender que não haveria nenhuma redução. Esse foi o caso dos cenários *Logout*, *Preparar Compra*, *Verificar Itens em Estoque*, *Comprar*, *Excluir Usuários* e *Excluir Produtos.*

O quarto nível de abstração (retirada de chamadas a métodos da mesma classe) segue a mesma linha de raciocínio aplicada ao terceiro nível. Não houve necessidade de sua aplicação para os cenários *Logout, Preparar Compra, Verificar Itens em Estoque e Comprar*.

Dependendo da aplicação alvo, o percentual de redução dos níveis três e quatro pode

```
. . .
IN: 06 U: 0,086 Class: br.unifor.mia.component.dao.HibernateDaoImpl
IN: 06 U: 0,086 Class: br.unifor.mia.component.OperationFacade
IN: 07 U: 0,100 Class: br.unifor.mia.component.to.ShopInfoTO
IN: 08 U: 0,114 Class: br.unifor.mia.commons.exceptions.MyPetStoreException
IN: 08 U: 0,114 Class: br.unifor.mia.component.to.UserInfoTO
IN: 09 U: 0,129 Class: br.unifor.mia.component.domain.UserProfile
IN: 09 U: 0,129 Class: br.unifor.mia.component.MyPetStoreMessageResources
IN: 12 U: 0,171 Class: br.unifor.mia.component.domain.StockItem
IN: 13 U: 0,186 Class: br.unifor.mia.component.to.StockInfoTO
IN: 16 U: 0,229 Class: br.unifor.mia.component.domain.User
IN: 28 U: 0,400 Class: br.unifor.mia.component.domain.Entity
```

Figura 5.1: Lista de classes candidatas a utilitárias da aplicação alvo.

```
Class: br.unifor.mia.component.domain.Entity
    Dependencies:
        None.
    Parents:

                . . .
        br.unifor.mia.component.controller.ShopControllerImpl
        br.unifor.mia.presentation.web.actions.StockOperationAction
        br.unifor.mia.presentation.web.forms.UserOperationForm
        br.unifor.mia.component.OperationFacadeImpl
        br.unifor.mia.component.dao.HibernateDao
        br.unifor.mia.component.controller.UserController
        br.unifor.mia.component.controller.StockControllerImpl
        br.unifor.mia.presentation.web.actions.UserOperationAction
        br.unifor.mia.component.dao.HibernateDaoImpl
        br.unifor.mia.presentation.web.forms.LoginForm
        br.unifor.mia.component.domain.StockItem
        br.unifor.mia.component.controller.UserControllerImpl
        br.unifor.mia.user.component.controller.UserControllerImpl
        br.unifor.mia.component.domain.OrderRequest
        br.unifor.mia.shop.component.controller.ShopControllerImpl
        br.unifor.mia.component.controller.StockController
        br.unifor.mia.component.domain.UserProfile
        br.unifor.mia.component.domain.UserTask
        Quantity: 28
```

Figura 5.2: Dependências da classe utilitária *Entity*.

variar, pois eles estão diretamente ligados a características individuais e arquiteturais de cada solução. A análise da necessidade de sua aplicação deve ser realizada a cada cenário e grau de abstração atingido.

Verificou-se a necessidade de aplicação do quinto nível de abstração (eliminação de padrões de mensagens que se repetem entre os componentes do cenário) em três dos cenários estudados. São eles: *Verificar Itens em Estoque*, *Excluir Usuários* e *Excluir Produtos*. Em todos os cenários citados, esse nível apresentou uma redução de 50% no número de mensagens em relação ao nível anterior. Tal resultado ocorre devido à quantidade de registros

Tabela 5.3: Evolução do número de elementos dos cenários escolhidos durante o processo de extração

Cenário	Nível de Abstração	Quantidade de Elementos Coletados no Cenário		Percentual de Redução em Relação à Coleta Inicial	
		Entidades	Mensagens	Entidades	Mensagens
Login	**1**	8	51	-	-
	2	4	6	50%	88,24%
	3	4	5	50%	90,20%
	4	4	3	50%	**94,12%**
Logout	**1**	5	8	-	-
	2	2	1	60%	**87,50%**
Preparar compra	**1**	6	11	-	-
	2	2	1	67%	**91%**
Autenticar usuário	**1**	13	66	-	-
	2	4	6	69,23%	90,91%
	3	4	5	69,23%	92,42%
	4	4	3	69,23%	**95,45%**
Verificar itens em estoque	**1**	19	200	-	-
	2	3	6	84,21%	97%
	5	3	3	84,21%	**98,50%**
Comprar	**1**	7	28	-	-
	2	2	1	71,43%	**96,23%**
Excluir usuários	**1**	19	455	-	-
	2	4	10	78,95%	97,80%
	4	3	6	84,21%	98,68%
	5	3	3	84,21%	**99,34%**
Excluir produtos	**1**	13	583	-	-
	2	3	10	76,92%	98,28%
	4	3	6	76,92%	98,97%
	5	3	3	76,92%	**99,49%**

manipulados pelas operações. Por exemplo, para os cenários *Excluir Usuários* e *Excluir Produtos*, foi solicitada a exclusão de dois registros em cada, resultando na repetição dessa operação. A aplicação do quinto nível de abstração eliminou uma das operações de exclusão repetidas (padrão de mensagens iguais), alcançando o percentual citado.

A Tabela 5.3 apresenta a evolução dos números de elementos presentes nos oito

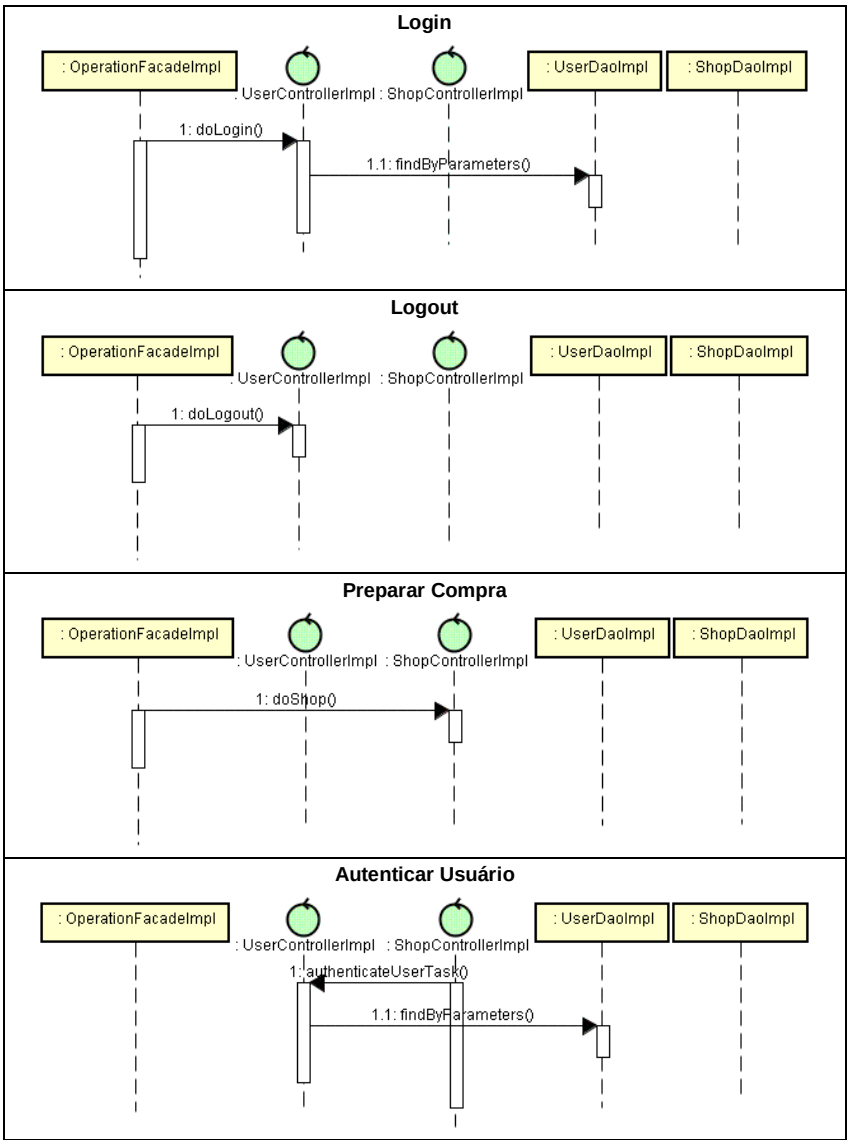

Figura 5.3: Cenários *Login*, *Logout*, *Preparar Compra* e *Autenticar Usuário* extraídos da aplicação MyPetStore

cenários selecionados ao longo do processo de extração de cenários em cada um dos cinco níveis de abstração. As Figuras 5.3 e 5.4 mostram os diagramas de seqüência da UML representando esses oito cenários após a finalização do processo de extração de cenários via abstração dos rastros de execução.

5.4 Detecção de Cenários Implícitos

Cada um dos oito cenários extraídos na etapa anterior foi representado como um cenário básico (bMSC) a ser alimentado como insumo da ferramenta LTSA-MSC. Assim,

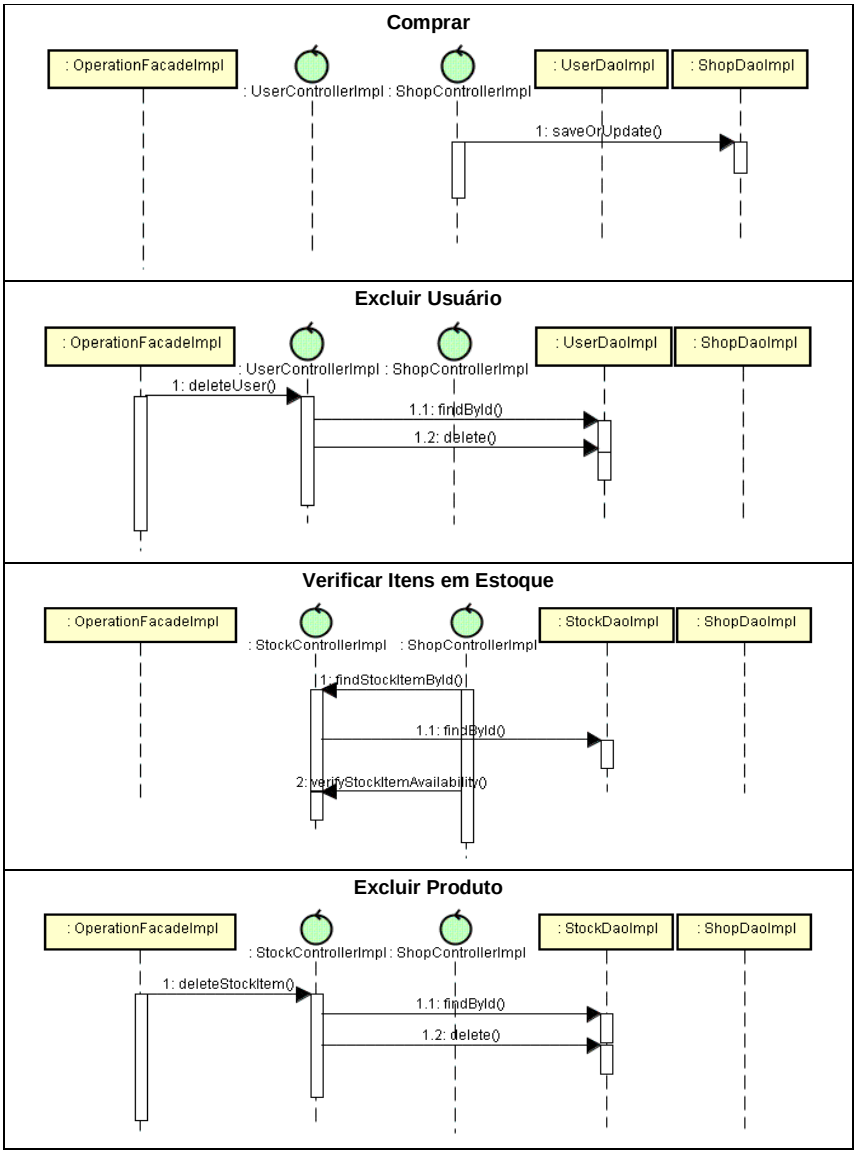

Figura 5.4: Cenários *Comprar*, *Excluir Usuário*, *Verificar Itens em Estoque* e *Excluir Produto* extraídos da aplicação MyPetStore

para viabilizar o processo de detecção de cenários implícitos pela ferramenta, foi necessário ainda definir novos cenários de alto nível (hMSCs) que estabelecessem as possíveis relações de continuidade entre dois ou mais cenários básicos. Essa ação foi necessária visto o conjunto de relações de continuidade entre cenários básicos poder variar de acordo com a visão do projetista do software. Dentre as várias configurações de cenários de alto nível investigadas, três resultaram em cenários implícitos. Esses cenários de alto nível e seus respectivos conjuntos de cenários implícitos serão descritos a seguir.

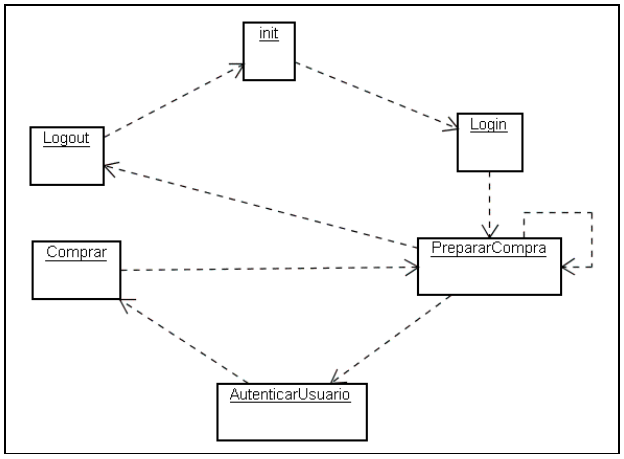

Figura 5.5: Primeiro cenário de alto nível utilizado no estudo.

5.4.1 Primeiro cenário de alto nível

Nessa primeira configuração, foi definido um cenário de alto nível envolvendo cinco dos oito cenários básicos extraídos da aplicação: *Login*, *Logout*, *Preparar Compra*, *Autenticar Usuário* e *Comprar* (Figura 5.5). Esses cenários foram selecionados porque compartilham um ou mais componentes de negócio, possibilitando geração de pontos de conflito quando da execução simultânea dos mesmos (aspecto importante para a escolha dos cenários básicos), como é o caso dos componentes *UserControllerImpl*, utilizado nos cenários *Login*, *Logout* e *Autenticar Usuário*, e *ShopControllerImpl*, utilizado nos cenários *Preparar Compra*, *Autenticar Usuário* e *Comprar*.

O cenário de alto nível inclui um nó inicial (*init*), cuja única transição de saída aponta para o cenário *Login*. A partir desse último, há também uma única transição apontando para o cenário *Preparar Compra*. A partir do cenário *Preparar Compra* há três transições possíveis: para o cenário *Logout*, o que forçaria o sistema a retornar ao estado inicial; para o cenário *Autenticar Usuário*, dando continuidade ao processo de compra de itens; e, por último, para ele mesmo, indicando que uma outra compra poderia ser preparada em substituição à atual. No cenário *Autenticar Usuário* há somente uma transição possível, para o cenário *Comprar*, e deste para o cenário *Preparar Compra*, iniciando um novo ciclo de compras. Esse ciclo (*Preparar Compra* → *Autenticar Usuário* → *Comprar* → *Preparar Compra*) faz parte da operação de encerramento de carrinho, na qual os três cenários devem sempre acontecer nessa ordem.

Tendo como entrada o cenário de alto nível definido com os cinco cenários básicos

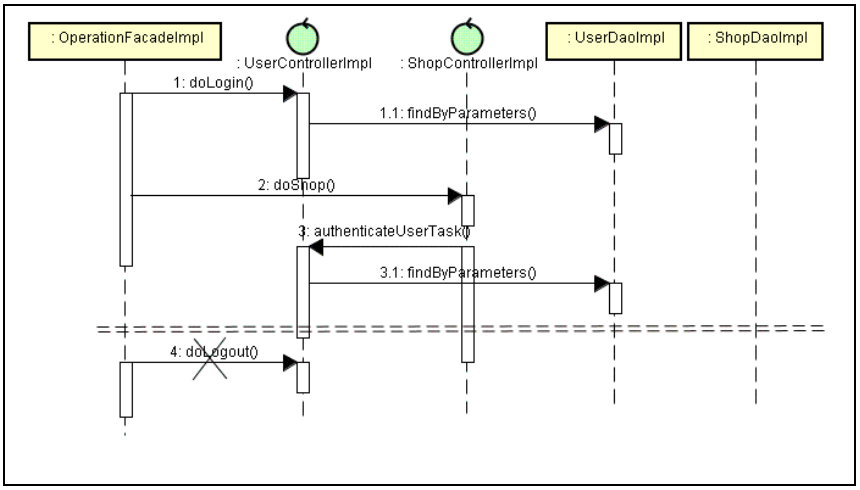

Figura 5.6: Diagrama de seqüência referente ao cenário implícito negativo "Saída do usuário durante compra".

descritos acima, a ferramenta LTSA-MSC detectou vários cenários implícitos. Dentre eles, destacamos três, sendo dois negativos e um positivo.

Cenário Implícito Negativo: Saída do usuário durante compra

A Figura 5.6 ilustra o diagrama de seqüência referente a este cenário. O contexto de execução do cenário implícito é o seguinte:

i. Entrada (*login*) do usuário na aplicação (mensagens 1 e 1.1);
ii. Preparação para a compra (mensagem 2);
iii. Autenticação do usuário para a compra (mensagens 3 e 3.1);
iv. Saída (*logout*) do usuário da aplicação (mensagem 4).

O problema identificado neste cenário reside exatamente na troca da mensagem 4 entre os componentes *OperationFacadeImpl* e *UserControllerImpl*. Onde era esperada a efetivação da compra, através da execução do cenário *Comprar*, o cenário implícito mostra que existe a possibilidade de execução do cenário *Logout*. Isso significa que, caso o cenário implícito venha a acontecer, a aplicação poderia interromper um processo de compra de forma abrupta, contribuindo para o risco de perda de integridade nas informações que mantém. Outro problema, de impacto ainda maior que o anterior, seria a continuação do processo de compra após a operação de *logout*. Esse problema poderia ocorrer em algumas arquiteturas concorrentes onde não haja a implementação da sincronização necessária, nas quais os

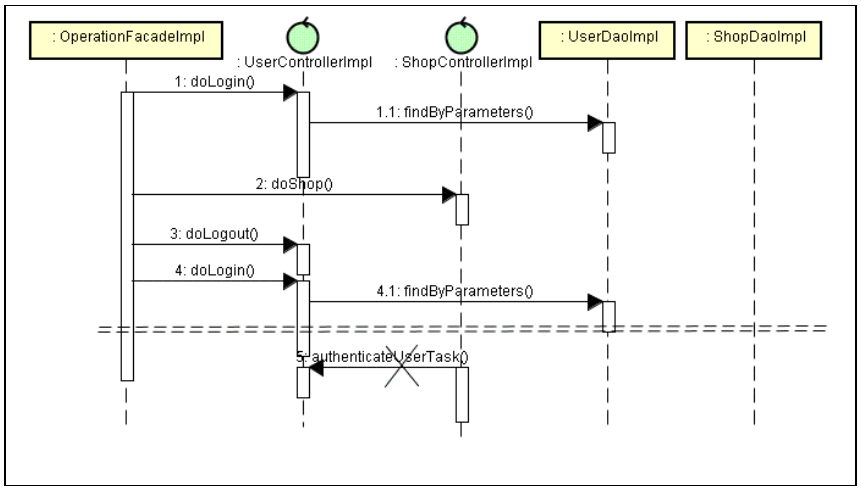

Figura 5.7: Diagrama de seqüência referente ao cenário implícito negativo "Autenticação de usuário para seção já finalizada".

componentes de cada cenário são executados em linhas de execução (*threads*) independentes. Nesse caso, a aplicação permitira a compra de itens mesmo após o usuário ter saído do sistema, representando uma falha de segurança.

Para evitar que os problemas identificados neste cenário implícito aconteçam, uma possível solução seria a aplicação realizar um teste no momento que a operação de *logout* fosse executada. Esse teste verificaria se existe alguma transação de qualquer espécie ainda em aberto na aplicação, e que tenha sido iniciada pelo usuário atual. Caso alguma transação ainda não tenha sido completada, a operação de *logout* poderia mascarar o problema retardando a sua execução até um estágio em que não houvesse mais nenhuma transação em aberto, ou poderia apenas tolerar o erro, informando ao usuário da impossibilidade da saída da aplicação naquele momento.

Cenário Implícito Negativo: Autenticação de usuário para seção já finalizada

Foi identificado um outro cenário implícito negativo para o primeiro cenário de alto nível investigado. O diagrama de sequência referente a esse cenário é mostrado na Figura 5.7. Seu contexto de execução é o seguinte:

i. Entrada (*login*) do usuário na aplicação (mensagens 1 e 1.1);
ii. Preparação para a compra (mensagem 2);
iii. Saída (*logout*) do usuário da aplicação (mensagem 3).
iv. Nova entrada (*login*) do usuário na aplicação (mensagens 4 e 4.1);

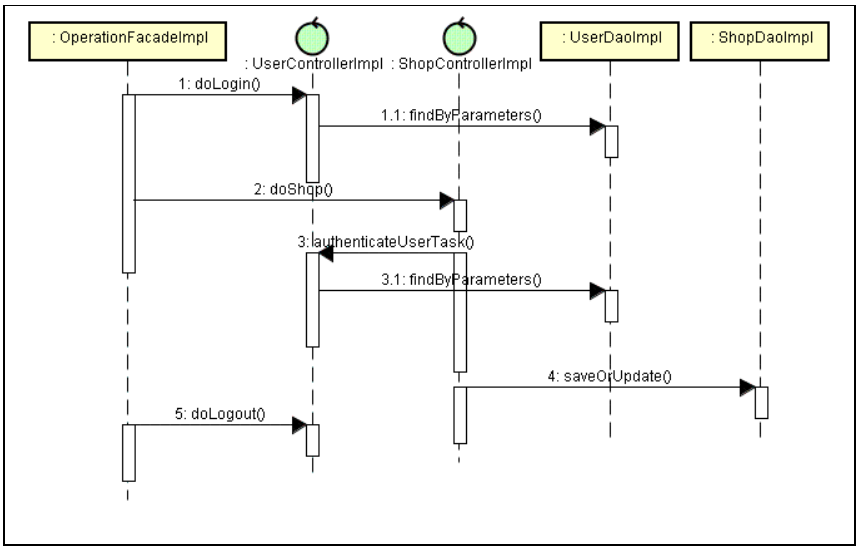

Figura 5.8: Diagrama de seqüência referente ao cenário implícito positivo "Efetivação de compra para usuário logado".

v. Autenticação do usuário para a compra (mensagem 5);

No caso desse cenário implícito, o problema reside na troca da mensagem 5. Nesse ponto, fica clara a confusão entre seções de usuário na aplicação. O que pode ser observado é a possibilidade de execução de uma mensagem inerente a uma determinada seção de usuário no contexto de outra. Mais precisamente, a aplicação permitiria que uma parte do cenário de realização de compra (*autenticateUserTask*) fosse executada em uma seção de um usuário diferente da que a originou, implicando em uma possível perda de integridade. Para evitar situações assim, o sistema deverá finalizar todas as operações pendentes de execução no ato do encerramento da seção de um usuário, garantindo a devida integridade da transação iniciada.

Cenário Implícito Positivo: Efetivação de compra para usuário logado

Nesse caso, foi detectado um cenário implícito qualificado como positivo. A Figura 5.8 mostra o diagrama de sequência deste cenário. Abaixo o contexto de sua execução.:

i. Entrada (*login*) do usuário na aplicação (mensagens 1 e 1.1);
ii. Preparação para a compra (mensagem 2);
iii. Autenticação do usuário para a compra (mensagens 3 e 3.1);
iv. Efetivação da compra (mensagem 4);
v. Saída (*logout*) do usuário da aplicação (mensagem 5).

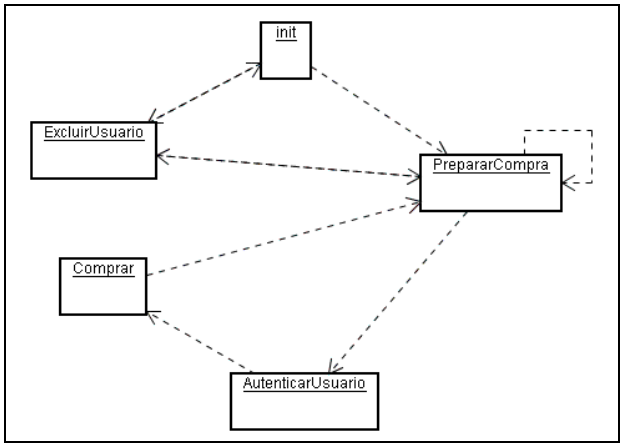

Figura 5.9: Segundo cenário de alto nível utilizado no estudo.

O cenário implícito observado foi considerado plenamente aceitável, uma vez que ele descreve uma situação natural do ponto de vista da realização de uma compra. Nesse caso, ele poderá ser utilizado para agregar informações que facilitarão a compreensão do sistema em questão, caso este comportamento não tenha ainda sido documentado.

5.4.2 Segundo cenário de alto nível

Nessa segunda configuração, foi definido um cenário de alto nível contendo os seguintes cenários básicos: *Preparar Compra*, *Autenticar Usuário*, *Comprar* e *Excluir Usuário* (Figura 5.9). Novamente, esses cenários foram selecionados porque possuem componentes de negócio em comum. Nesse caso, os cenários considerados compartilham os mesmos componentes compartilhados no caso anterior, ou seja, *ShopControllerImpl*, utilizado nos cenários *Preparar Compra*, *Autenticar Usuário* e *Comprar*, e *UserControllerImpl*, utilizado nos cenários *Autenticar Usuário* e *Excluir Usuário.*

O cenário de alto nível da Figura 5.9 mostra novamente um ciclo de transições entre cenários (*Preparar Compra* → *Autenticar Usuário* → *Comprar* → *Preparar Compra*), o qual também compõe a operação de encerramento de carrinho. Porém, dessa vez existe a possibilidade de execução do cenário de exclusão de usuário a partir do cenário de preparação de compra. O cenário de alto nível também estabelece transições de saída do cenário de exclusão de usuário, que tanto pode retornar para o cenário de preparação de compra quanto para o cenário inicial.

A partir desse cenário de alto nível, a ferramenta LTSA-MSC detectou três cenários

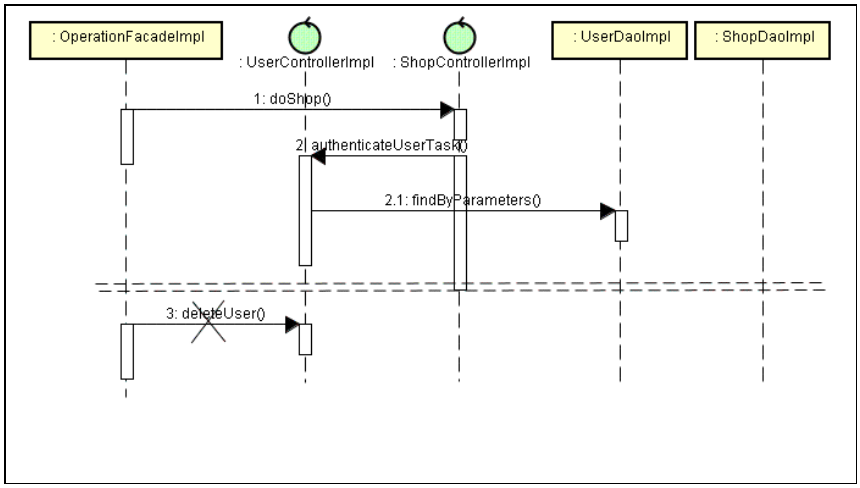

Figura 5.10: Diagrama de seqüência referente ao cenário implícito negativo "Exclusão de usuário durante compra".

implícitos considerados mais relevantes, sendo um negativo e dois positivos.

Cenário Implícito Negativo: Exclusão de usuário durante compra

O diagrama de seqüência referente a este cenário implícito é mostrado na Figura 5.10. O contexto de execução do cenário implícito é o seguinte:

i. Preparação para a compra (mensagem 1);
ii. Autenticação de usuário para a compra (mensagens 2 e 2.1);
iii. Exclusão de usuário (mensagem 3).

A execução do segundo e terceiro passos demonstram uma possível situação de erro. Isto porque essa ordem de execução irá possibilitar a um operador da aplicação a exclusão de um usuário durante o processo de compra de um item. Embora à primeira vista essas duas operações não aparentem estarem relacionadas, a sua execução, nessa ordem específica, pode levar à perda da integridade nos dados da aplicação, bastando, para isso, que o usuário excluído seja o mesmo autenticado durante o processo de compra. Levando-se em conta a concorrência entre as duas operações, a continuidade da execução desse cenário resultará em um pedido de compra autenticado para um usuário que não mais existe na aplicação.

A atual versão da MyPetStore garante a integridade dos dados em operações de exclusão verificando as dependências dos itens a serem excluídos antes da efetivação da operação. Um exemplo seria antes de excluir um usuário verificar se algum pedido de compra já havia sido feito por seu intermédio. Em caso positivo, esse usuário não poderia ser

excluído. Entretanto, no caso do cenário implícito, existe a possibilidade desse usuário estar sendo referenciado pela primeira vez na aplicação, não havendo ainda nenhuma ligação sua com algum pedido previamente realizado. Há duas visões para o tratamento desse tipo de problema. Como as operações de exclusão de usuários e de compra de itens não dependem necessariamente uma da outra, as soluções estão diretamente associadas à ótica de cada cenário. No primeiro caso, a exclusão de um usuário somente poderá ser concretizada quando nenhum pedido de compra estivesse em andamento referenciando o usuário que se deseja excluir. No segundo caso, a compra somente poderá ser efetivada se as dependências do pedido sejam satisfeitas, ou seja, o usuário que fez o pedido deverá necessariamente existir no sistema.

Cenários Implícitos Positivos: Exclusão de usuário durante compra antes do procedimento de autenticação – Variações 1 e 2.

Os contextos de execução destes cenários implícitos positivos (Figuras 5.11 e 5.12) são descritos abaixo.

Variação 1:

i. Exclusão de usuário (mensagens 1, 1.1 e 1.2);
ii. Preparação para a compra (mensagem 2);
iii. Autenticação de usuário para a compra (mensagem 3);

Variação 2:

i. Preparação para a compra (mensagem 1);
ii. Exclusão de usuário (mensagens 2, 2.1 e 2.2);
iii. Autenticação de usuário para a compra (mensagem 3);

Ambos os casos descrevem situações previstas na execução do sistema. A exclusão do usuário acontecendo antes do processo de autenticação permitirá ao sistema interromper o processo de compra de um item (irá ser criada uma mensagem de erro prevista no sistema), sem necessariamente causar inconsistências nos dados. A identificação desses dois cenários implícitos, apenas reforça a necessidade de verificação de permissões e de integridade em pontos estratégicos da aplicação, evitando riscos de inconsistência.

Vale salientar que a exclusão de um usuário somente poderá acontecer caso seja mantida a integridade da informação persistida. Em situações onde a integridade referencial

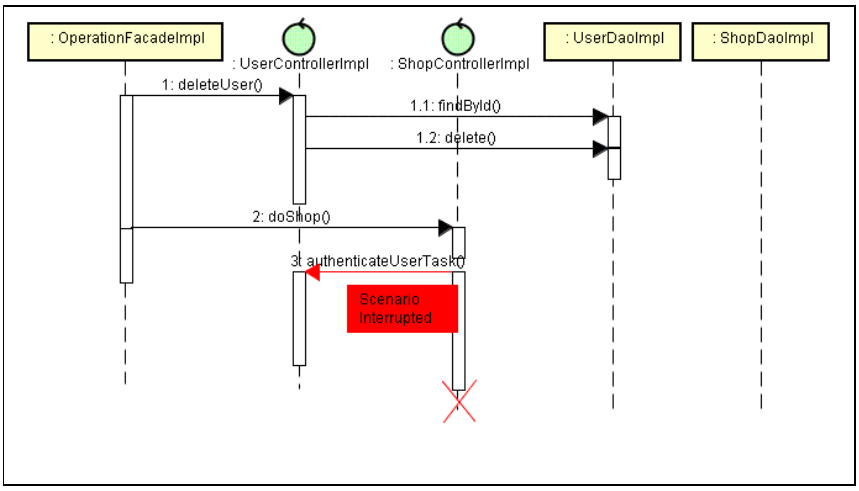

Figura 5.11: Diagrama de seqüência referente ao cenário implícito positivo "Exclusão de usuário durante compra antes do procedimento de autenticação – Variação 1".

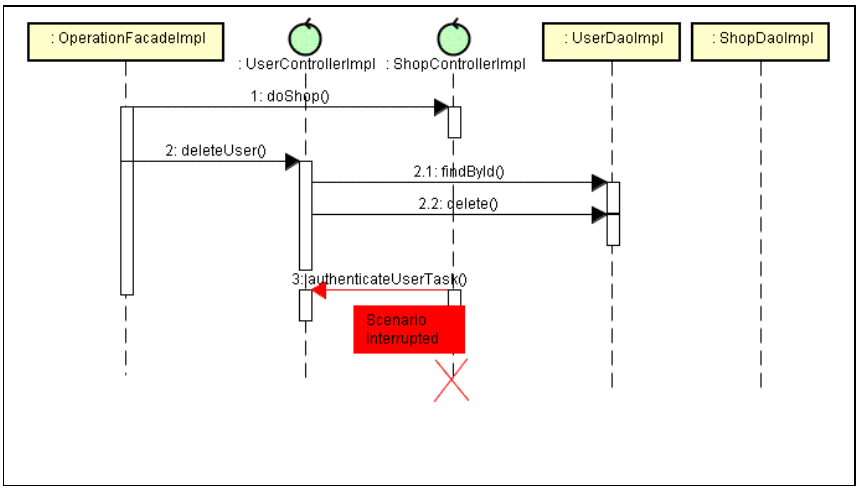

Figura 5.12: Diagrama de seqüência referente ao cenário implícito positivo "Exclusão de usuário durante compra antes do procedimento de autenticação – Variação 2".

do dado é exigida, uma verificação antes da operação de exclusão se faz necessária. Com essa premissa satisfeita, a operação de exclusão poderá ser realizada até mesmo depois da efetivação de uma compra, sem com isso causar problemas à integridade ao sistema.

5.4.3 Terceiro cenário de alto nível

Este terceiro cenário de alto nível utilizou os cenários básicos *Preparar Compra*, *Verificar Itens em Estoque*, *Comprar* e *Excluir Produto* (Figura 5.13).

Dessa vez, o componente compartilhado foi o *StockController*, responsável pelo controle de estoque da aplicação, o qual recebe as mensagens *findStockItemById*,

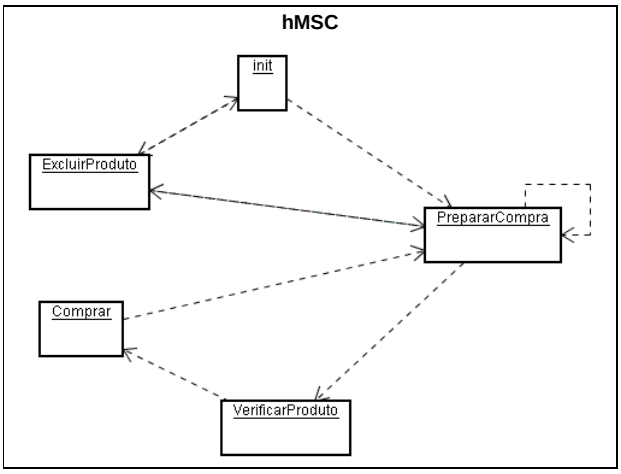

Figura 5.13: Terceiro cenário de alto nível utilizado no estudo.

verifyStockItemAvailability e *deleteStockItem*. As duas primeiras são originadas no mesmo cenário, *Verificar Itens em Estoque*.

Neste cenário de alto nível (Figura 5.13), o cenário básico de verificação de itens em estoque é uma continuação do cenário *Preparar Compra* e figura na mesma posição do cenário *Autenticar Usuario*, presente em outros cenários de alto nível, mas com uma função diferenciada. O cenário de efetivação da compra de itens (*Comprar*) é a continuação natural desse cenário. O cenário *Excluir Produto* não possui uma relação direta com o processo de compra e pode ser executado a partir da preparação de uma compra, como também a partir do estágio inicial.

Para este cenário de alto nível foram selecionados dois cenários implícitos mais representativos, sendo um positivo e um negativo (outros cenários implícitos detectados representam pequenas variações destes). Esses dois cenários implícitos são discutidos a seguir.

Cenário Implícito Negativo: Exclusão de produto após verificação em estoque

Os contexto de execução deste cenário implícito (Figura 5.14) é o seguinte:

i. Preparação para a compra (mensagem 1);
ii. Verificação dos itens em estoque (mensagens 2, 2.1 e 3);
iii. Exclusão do produto (mensagem 4).

Problemáticas similares a do cenário implícito negativo descoberto no segundo cenário

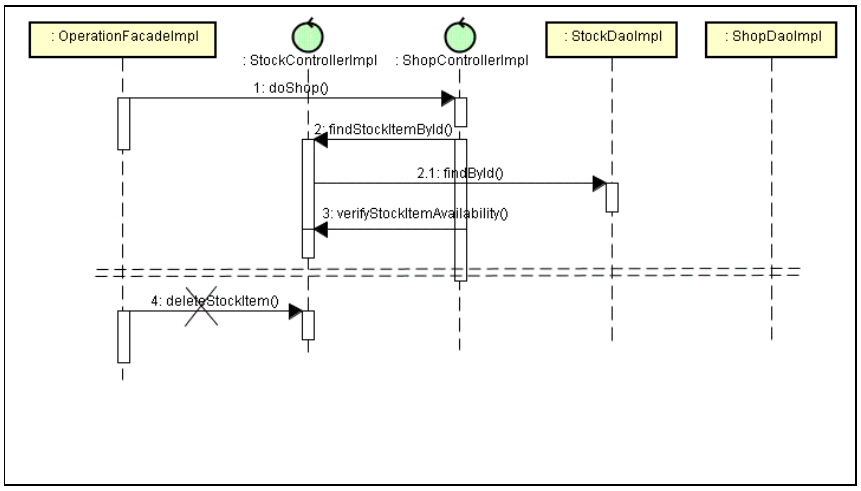

Figura 5.14: Diagrama de seqüência referente ao cenário implícito negativo "Exclusão de produto após verificação em estoque".

de alto nível (Figura 5.10) também são encontradas aqui. Através de uma análise inicial, o que se pode constatar é que o sistema possivelmente permite que um pedido seja realizado para um produto excluído em uma operação concorrente. Novamente, a integridade dos dados parece estar sendo abalada por operações paralelas e não diretamente dependentes.

Situações como essa poderiam ser contornadas, caso algumas verificações à cerca das dependências fossem feitas tanto no cenário de exclusão de pedidos quanto no de efetivação de compras. No primeiro, a exclusão do produto somente teria êxito se nenhuma transação estivesse em curso para o produto que se quer excluir (no mecanismo de persistência dos dados ou ainda para pedidos não concluídos no sistema). No segundo, a compra somente poderia ser concluída para produtos ainda existentes no sistema.

Cenário Implícito Positivo: Exclusão de produto antes da verificação em estoque.

O contexto de execução deste cenário implícito (Figura 5.15) é o seguinte:

i. Preparação para a compra (mensagem 1);
ii. Exclusão de produto (mensagens 2, 2.1 e 2.2);
iii. Verificação dos itens em estoque (mensagem 3).

Este cenário implícito pode ser comparado aos dois cenários implícitos positivos encontrados no segundo cenário de alto nível (Figuras 5.11 e 5.12). O aparecimento de uma exclusão de um produto antes de sua verificação em estoque é naturalmente aceitável, porque resultará em uma falha tolerada pelo usuário do sistema. Da mesma forma que em outros

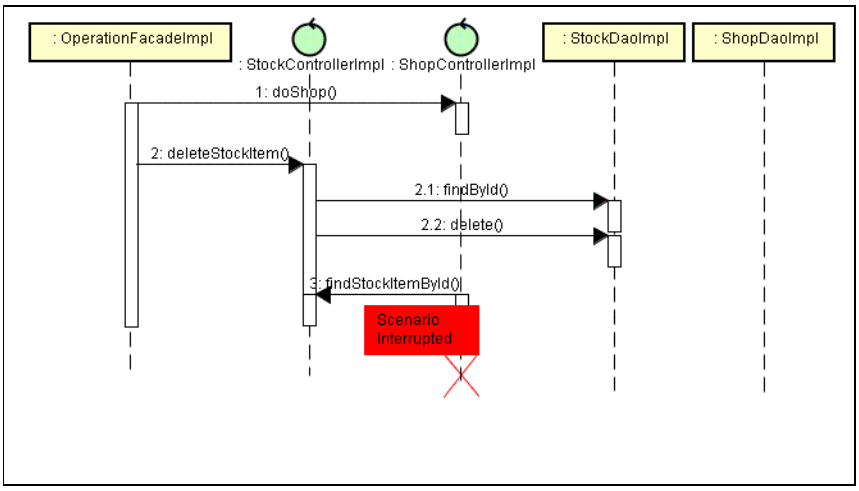

Figura 5.15: Diagrama de seqüência referente ao cenário implícito positivo "Exclusão de produto antes da verificação em estoque".

cenários implícitos que envolvem operações de exclusão, devem ser respeitados os casos onde a integridade referencial é exigida.

5.5 Discussão

Os problemas representados nos cenários implícitos descritos nesse capítulo ilustram bem os benefícios que a detecção de cenários implícitos pode trazer para a atividade de compreensão de um sistema concorrente existente. Muito embora cada cenário executado tenha apresentado um comportamento "correto" do seu ponto de vista local, diferentes combinações desses cenários podem levar a situações indesejadas com respeito ao comportamento global da aplicação, como aquelas discutidas para cada cenário implícito detectado. Portanto, a vantagem que a detecção de cenários implícitos oferece aos desenvolvedores é justamente poder ajudá-los a identificar e se planejar para essas possíveis situações problemáticas, antes que elas venham a acontecer na prática.

Mesmo no caso em que a aplicação já tenha alguma solução implementada para contornar os problemas representados nos cenários implícitos, detectá-los explicitamente ainda seria útil como forma de atrair a atenção para aspectos importantes da aplicação que, de outro modo (ou seja, com base apenas na análise dos cenários realmente executados), poderiam passar despercebidos ao desenvolvedor.

Além do acréscimo notório à documentação de uma aplicação existente, a extração de cenários e a subseqüente identificação de cenários implícitos podem indicar possíveis comportamentos inesperados aos responsáveis pela aplicação. Esse tipo de alerta é de grande

importância para garantir a qualidade do software envolvido, pois permite que situações inusitadas encontradas possam ser tratadas, se assim for necessário, para garantir a coerência do processo. Mesmo no caso de aplicações bem documentadas, a extração de cenários ainda pode ser útil como um instrumento de avaliação do nível de conformidade entre a documentação existente e a realidade do código, permitindo a comparação entre os cenários projetados e os cenários que foram realmente implementados.

A descoberta de cenários implícitos também pode ser útil no suporte a atividades de testes. Por exemplo, dado um conjunto de rastros de execução que foram gerados a partir de um conjunto de casos de teste, um cenário implícito seria uma indicação de um comportamento possível do sistema, mas ainda não coberto pelos casos de teste ou não produzido pela execução deles (por causa de um fluxo alternativo ocorrido devido a um comportamento não determinístico do sistema concorrente). Em essência, o cenário implícito indicará um comportamento do sistema ainda não explorado. Este mecanismo pode ser particularmente útil na seleção de casos de testes dos sistemas, pois cenários implícitos apontariam aspectos do sistema onde haveria uma maior necessidade de testes.

Por fim, deve-se destacar que a quantidade de cenários e de mensagens presentes nos cenários extraídos dos rastros de execução mostrou-se um fator limitante para o desempenho da ferramenta de detecção de cenários implícitos. Isto porque a ferramenta LTSA-MSC foi desenvolvida inicialmente visando cenários tipicamente utilizados na fase de especificação e elaboração de requisitos, cujo tamanho e complexidade tendem a ser bastante reduzidos. Assim, um processo de abstração de cenários, para ser bem sucedido, deve sempre poder garantir uma redução muito significativa do total de elementos presentes nos rastros de execução coletados inicialmente em cada cenário (por exemplo, em alguns dos cenários extraídos o fator de redução foi superior a 97%). Da mesma forma, a presença de um especialista da aplicação durante o processo de abstração dos cenários é imprescindível para não descaracterizar os cenários extraídos e garantir que estes de fato reflitam o comportamento da aplicação alvo.

Capítulo 6

Conclusão

Este capítulo apresenta a conclusão deste trabalho, destacando suas contribuições, principais resultados obtidos e sugestões para trabalhos futuros.

6.1 Contribuições e Resultados

Atualmente podemos observar ambientes de desenvolvimento de software cada vez mais complexos nas corporações. O que antes não passavam de alguns poucos processos e sistemas computacionais isolados, hoje se transformaram em uma estrutura altamente robusta e totalmente interligada. Esse processo evolucional constante é um dos principais fatores integrantes da estratégia das corporações para sobreviver em meio à crescente competitividade.

As diversidades tecnológicas e soluções arquiteturais adotadas são palco cada vez mais comum de discussões entre especialistas e pesquisadores. Essas diversidades estão posicionadas em uma proporção direta aos processos de negócio de cada empresa, onde cada processo possui regras distintas se aplicados a contextos específicos.

A fim de melhor visualizar os diversos cenários operacionais presentes nesses ambientes tornou-se imprescindível aos projetistas e desenvolvedores a utilização de diagramas que pudessem descrever seus comportamentos. Especificações baseadas em cenários, como Message Sequence Charts (MSCs) (ITU-T, 2004) e Diagramas de Seqüência (OMG, 2002), são abstrações essenciais para o entendimento e racionalização de aspectos importantes de seus comportamentos.

A integração entre sistemas, motivada pela necessidade de melhores retornos financeiros, forçou uma maior flexibilidade nos negócios das empresas e em conseqüência um maior ganho. Entretanto, o que parecia ter somente vantagens, trouxe consigo alguns problemas atrelados. Os processos de negócio de tão complexos passaram a influenciar outros processos dentro da mesma cadeia de acontecimentos. O nível de dependência e a concorrência a recursos compartilhados entre eles aumentaram consideravelmente, a ponto de forçar o aparecimento de situações totalmente inesperadas.

Embora os modelos individuais dos diversos cenários operacionais refletissem fielmente comportamentos específicos no sistema, algumas combinações inesperadas de seus componentes poderiam forçar o aparecimento de outros comportamentos não previstos originalmente, denominados *cenários implícitos* nos trabalhos de Alur et al. (2000) e Uchitel et al. (2001; 2003; 2004).

Este trabalho apresentou um processo e um ambiente automatizado para apoiar a descoberta de cenários implícitos a partir de informações coletadas dinamicamente durante a execução de uma aplicação concorrente existente. Essa abordagem, além de enriquecer as especificações de cenários originais, propicia subsídios que podem ser de grande ajuda na melhoria do processo de compreensão da aplicação alvo, trazendo uma maior segurança às atividades de teste, manutenção e evolução. Em particular, a abordagem possibilita que comportamentos inusitados, não previstos nos cenários executados originalmente, possam ser facilmente identificados, rastreados e tratados, contribuindo para a melhoria da qualidade do processo de desenvolvimento.

6.2 Trabalhos Futuros

Vislumbramos várias oportunidades de melhorias e extensões ao trabalho reportado nesta dissertação. Dentre elas, destacamos:

- Realizar novos experimentos de avaliação, preferencialmente com aplicações concorrentes de código aberto de tamanhos e domínios variados, com o intuito de verificar até que ponto os resultados reportados nesse trabalho podem ser generalizados.
- Utilizar outras ferramentas de monitoração para extrair rastros de execução de aplicações escritas em diferentes linguagens de programação. A idéia é poder verificar na prática a característica de independência de linguagem do metamodelo de rastros utilizado no ambiente.
- Investigar novas técnicas de abstração de cenários, particularmente de filtros que possam reduzir ainda mais o tamanho dos rastros de execução gerados pelo monitor de eventos. Essa linha de pesquisa é fundamental para viabilizar a utilização do ambiente proposto em aplicações corporativas de médio e grande porte.

- Avaliar as proporções das quantidades de cenários implícitos válidos ou inválidos encontrados em atuais aplicações corporativas de médio e grande porte.

Referências Bibliográficas

1. ALUR, R., ETESSAMI, K., YANNAKAKIS, M., 2000, *Inference of Message Sequence Charts*. In Proc. of the 22nd International Conference on Software Engineering (ICSE'00), Limerick, Ireland.
2. BECK, K., 2003, *Test-Driven Development by Example*, Addison-Wesley.
3. BOYER, R. S., MOORE, J. S., 1977, *A Fast String Searching Algorithm*, Communications of the ACM, vol. 20, no. 10, pp. 761-772.
4. BRIAND, L.C., LABICHE, Y., MIAO, Y., 2003, *Towards the Reverse Engineering of UML Sequence Diagrams*. In Proc. of the 10th IEEE Working Conference on Reverse Engineering (WCRE'03), Victoria, BC, Canada.
5. ELRAD, T., FILMAN, R. E., BADER, A., 2001, *Aspect Oriented Programming: Introduction*, Communications of the ACM, vol. 44, pp. 29-32.
6. HAMOU-LHADJ, A., BRAUN, E., AMYOT, D., LETHBRIDGE, T., 2005A, *Recovering Behavioral Design Models from Execution Traces*. In Proc. of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05), Manchester, U.K.
7. HAMOU-LHADJ, A., LETHBRIDGE, T. C., 2003, *Techniques for Reducing the Complexity of Object-Oriented Execution Traces*. In Proc. of the 2nd Annual "DESIGNFEST" on Visualizing Software for Understanding and Analysis (VISSOFT'03), Amsterdam, The Netherlands, pp. 35-40.
8. HAMOU-LHADJ, A., LETHBRIDGE, T. C., 2004, *A Survey of Trace Exploration Tools and Techniques*. In Proc. of the 2004 Conference of the Centre for Advanced Studies and Collaborative Research (CASCON'04), Markham, Ontario, Canada, pp. 42-55.
9. HAMOU-LHADJ, A., LETHBRIDGE, T. C., 2005B, *Reasoning about the Concept of Utilities*. In Proc. of the 1st ECOOP Workshop on Practical Problems of Programming in the Large (PPPL'04), Oslo, Norway, LNCS Vol. 3344, Springer.
10. HOLZNER, S., 2000, *C++ Black Book A Comprehensive Guide to C++ Mastery*, O'Reilly & Associates.

11. IBM RESEARCH, 2001, *Jinsight: Visualizing the Execution of Java Programs*. Disponível em http://www.research.ibm.com/jinsight.

12. ITU-T, 2003, *Use Case Maps*, Draft Recommendation Z.152. Disponível em http://www.UseCaseMaps.org/urn

13. ITU-T, 2004, *Message Sequence Chart (MSC)*, Recommendation Z.120.

14. JERDING, D., RUGABER, S., 1997, *Using Visualisation for Architecture Localization and Extraction*. In Proc. of the 4th Working Conference on Reverse Engineering (WCRE'97), Amsterdam, The Netherlands, pp. 56-65.

15. JERDING, D., STASKO, J., BALL, T., 1997, *Visualising Interactions in Program Executions*. In Proc. of the 19th International Conference on Software Engineering (ICSE'97), Boston, USA, pp. 360-370.

16. KELLER, R., 1976, *Formal Verification of Parallel Programs*. Communications of the ACM, vol. 19, no. 7, pp. 371-384.

17. KOSKIMIES, K., MÄNNISTÖ, T., SYSTÄ, T., TUOMI, J., 1996, *SCED: A Tool for Dynamic Modeling of Object Systems*. University of Tampere, Dept. of Computer Science, Technical Report A-1996-4.

18. KOSKIMIES, K., MÖSSENBÖCK, H., 1995, *Scenario Based Browsing of Object-Oriented Systems with Scene*, University of Linz, Department of System Software, Technical Report 4.

19. KOSKIMIES, K., MÖSSENBÖCK, H., 1996, *Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs*. In Proc. of the 18th International Conference on Software Engineering (ICSE'96), Berlin, Germany, pp. 366-375.

20. LANGE, D. B., NAKAMURA, Y., 1997, *Object-Oriented Program Tracing and Visualization*. IEEE Computer, 30(5), pp. 63-70.

21. LEWIS, S., 1995, *The Art and Science of Smalltalk: An Introduction to Object-Oriented Programming using VisualWorks*, Prentice Hall / Hewlett-Packard Professional Books.

22. MAGEE, J., KRAMER, J., 1999, *Concurrency: State Models and Java Programs*, John Wiley & Sons Ltd., New York.

23. MANSUROV, N., CAMPARA, D., 2001, *Using Message Sequence Charts to Accelerate Maintenance of Existing Systems*. In Proc. of the 10th International SDL Forum (SDL'01: Meeting UML), LNCS Vol. 2078, Springer.

24. MÜLLER, H. A., KLASHINSKY, K., 1988, *Rigi – A System for Programming In-the-Large*. In Proc. of the 10th International Conference on Software Engineering (ICSE'88), Singapore, pp. 80-86.

25. OMG, 1999, XML Metadata Interchange Specification V2.1. Disponível em http://www.omg.org/technology/documents/formal/xmi.htm

26. OMG, 2002, Unified Modeling Language Specification V2.1.1. Disponível em http://www.omg.org/technology/documents/formal/uml.htm.

27. OMG, 2003, *OCL - Object Constraint Language*. Disponível em http://www.omg.org/docs/ptc/03-10-14.pdf

28. OMG, 2003, UML 2.0 Infrastructure Specification. Disponível em http://www.omg.org

29. PAUW, W., HELM, R., KIMELMAN, D., VLISSIDES, J., 1993, *Visualizing the Behaviour of Object-Oriented Systems*. In Proc. of the 8th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), Washington, DC., USA, pp. 326-337.

30. PAUW, W., JENSEN, E., MITCHELL, N., SEVITSKY, G., VLISSIDES, J., YANG, J., 2002, *Visualizing the Execution of Java Programs*. In Proc. the International Seminar on Software Visualization, Dagstuhl Castle, Wadern, Germany, pp. 151-162.

31. PAUW, W., KIMELMAN, D., VLISSIDES, J., 1994, *Modelling Object-Oriented Program Execution*. In Proc. of the 8th European Conference on Object-Oriented Programming (ECOOP'94), Bologna, Italy, LNCS vol. 821, Springer, pp. 163-182.

32. PAUW, W., LORENZ, D., VLISSIDES, J., WEGMAN, M., 1998, *Execution Patterns in Object-Oriented Visualization*. In Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98), Santa Fe, NM, USA, pp. 219-234.

33. RICHNER, T., DUCASSE, S., 2002, *Using Dynamic Information for the Iteractive Recovery of Collaborations and Roles*. In Proc. of the International Conference on Software Maintenance (ICSM'02), Montreal, Quebec, Canada.

34. ROUNTEV, A., VOLGIN, O., REDDOCH, M., 2005, *Static control-flow analysis for reverse engineering of UML sequence diagrams*, In Proc. of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, vol. 31, issue 1, pp. 96-102.

35. SHAW, M., GARLAN, D., 1996, *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, nj: Prenttce Hall.

36. SOUSA, F. C., MENDONÇA, N. C., 2007, *Um Ambiente para Detecção de Cenários Implícitos a partir de Rastros de Execução*. Em Anais do XXI Simpósio Brasileiro de Engenharia de Software (SBES'07), João Pessoa, PB, Brasil.

37. SOUSA, F. C., MENDONÇA, N. C., UCHITEL, S., KRAMER, 2007, J. *Detecting Implied Scenarios from Execution Traces*. In Proc. of the 14th Working Conference on Reverse Engineering (WCRE'07), Vancouver, BC, Canada.

38. SUN MICROSYSTEMS, 2004, *Java Platform Debugger Architecture*. Disponível em http://java.sun.com/javase/6/docs/technotes/guides/jpda/index.html.

39. SUN MICROSYSTEMS, 2005, *Java Pet Store Demo 1.4*. Disponível em http://java.sun.com/j2ee/1.4/download.html#samples.

40. SUN MICROSYSTEMS, 2007, *Java 2 SE 6 Platform*, Disponível em http://java.sun.com/javase/6/docs/

41. SYSTÄ, T., 1999, *Dynamic Reverse Engineering of Java Software*. In Proc. of the 13th European Conference on Object-Oriented Programming (ECOOP'99), 3rd Workshop on Experiences in Object-Oriented Reengineering, Lisbon, Portugal.

42. SYSTÄ, T., 2000, *Incremental Construction of Dynamic Models for Object-Oriented Software Systems*, Journal of Object-Oriented Programming, 13(5), pp. 18-27.

43. SYSTÄ, T., 2000, *Understanding the Behaviour of Java Programs*. In Proc. of the 7th Working Conference on Reverse Engineering (WCRE'00), Brisbane, QL, Australia, pp. 214-223.

44. SYSTÄ, T., KOSKIMIES, K., MÜLLER, H. A., 2001, *Shimba – An Environment for Reverse Engineering of Java Software Systems*, Software–Practice and Experience, 31(4), pp. 371-394.

45. UCHITEL, S., CHATLEY, R., KRAMER, J., MAGEE, J., 2003, *LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios*. In Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), LNCS vol. 2619, Springer.

46. UCHITEL, S., KRAMER, J., MAGEE, J., 2001, *Detecting Implied Scenarios in Message Sequence Chart Specifications*. In Proc. of the Joint 8th European Software Engineering Conference (ESEC'01) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'01), Vienna, Austria, pp. 74-82.

47. UCHITEL, S., KRAMER, J., MAGEE, J., 2002, *Negative Scenarios for Implied Scenario Elicitation*. In Proc. of the 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'02), Charleston, SC, USA.

48. UCHITEL, S., KRAMER, J., MAGEE, J., 2003, *Synthesis of Behavioral Models from Scenarios*, IEEE Transactions on Software Engineering, 29(2), pp. 99-115.

49. UCHITEL, S., KRAMER, J., MAGEE, J., 2004, *Incremental Elaboration of Scenario-based Specifications and Behaviour Models Using Implied Scenarios*, ACM Transactions on Software Engineering and Methodology, 13(1).

50. W3C, *XSL Transformations*, 1999, Disponível em http://www.w3.org/TR/xslt

51. WALKER, R., J., MURPHY, G., C., FREEMAN-BENSON B., SWANSON, D., ISAAK, J., 1998, *Visualizing Dynamic Software System Information through High-level Models*. In Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, British Columbia, Canada, pp. 271-283.

52. WALL, L., 2000, *Programming Perl*, O'Reilly & Associates.

53. WARMER, J., KLEPPE, A., 1999, *The Object Constraint Language*, Addison-Wesley.

Apêndice I

Quadro Resumo do Documento XML do Metamodelo de Dados

Elemento Nível 1	Elemento Nível 2	Elemento Nível 3	Elemento Nível 4	Descrição conteúdo
<Cenario>	-	-	-	Elemento Principal do documento, mais conhecido como elemento raíz. Seus subelementos comporão a execução do cenário operacional.
	<id>	-	-	Elemento responsável pela identificação única do cenário monitorado.
	<descricao>	-	-	Elemento que descreve o cenário monitorado.
	<instancias>			Elemento que possuirá o conjunto de elementos descritores dos componentes envolvidos no cenário.
		<Instancia>	-	Elementos que representam os descritores dos componentes envolvidos no cenário.
			<nome>	Elemento que indica os nomes dos componentes envolvidos.
	<mensagens>	-	-	Elemento que possuirá o conjunto de elementos representantes das mensagens trocadas entre componentes.
		<Mensagem>	-	Elementos que representam a mensagem trocada entre componentes.
			<nome>	Elemento que indica o nome da mensagem trocada. É o nome do método chamado no componente destino.
			<destino>	Elemento que representa o componente destino na troca de mensagens.
			<origem>	Elemento que representa o componente origem da troca de mensagens.
			<id>	Elemento responsável pela identificação única da mensagem no cenário monitorado.
			<indice>	Ordem da realização da mensagem no cenário.
			<thread>	Thread onde o processamento da mensagem foi realizado na JVM.

Figura I.1: Descrição dos elementos que compõem o documento do Metamodelo de Dados.

Apêndice II

Quadro Resumo do Documento XML de Carga para o LTSA-MSC

Elemento Nível 1	Elemento Nível 2	Elemento Nível 3	Elemento Nível 4	Elemento Nível 5	Descrição conteúdo
<specification>	-	-	-	-	Elemento Principal do documento, mais conhecido como elemento raíz. Seus subelementos comporão a execução do cenário operacional.
	<hmsc>	-	-	-	Elemento que possuirá o conjunto de elementos descritores dos bmsc envolvidos no cenário. Representa o cenário em formato msc de mais alto nível.
		<bmsc>	-	-	Elementos descritores dos diagramas bmsc no diagrama hmsc cujas propriedades são: name - identificação única de um diagrama bmsc no diagrama hmsc. (obrigatório) x – coordenada X no eixo cartesiado que representa a posição da coluna do componente. (não obrigatório) y - coordenada Y no eixo cartesiado que representa a posição da linha do componente. (não obrigatório)
	<bmsc>	-	-	-	Elemento que possuirá o conjunto de elementos representantes dos diagramas bmsc dispostos no diagrama hmsc. Possui uma única propriedade: name – identificação única de um diagrama bmsc. Deve possuir a mesma identificação colocada no elemento de mesmo nome subelemento de <hmsc>.
		<instance>	-	-	Elemento que representa uma instancia de um diagrama bmsc, cuja propriedade é: name – identificação do componente no diagrama bmsc.
			<output>	-	Elemento que representa as mensagens de saída do componente, cuja propriedade é: timeindex – ordem da mensagem no diagrama bmsc. É uma posição temporal de acordo com sua execução no cenário.
				<name>	Elemento identificador da mensagem. Seu conteúdo se apresenta da seguinte forma: componente_chamador, componente_chamado, nome_mensagem

Elemento Nível 1	**Elemento Nível 2**	**Elemento Nível 3**	**Elemento Nível 4**	**Elemento Nível 5**	**Descrição conteúdo**
<specification> (*continuação)	**<bmsc>**	**<instance>**	**<output>**	**<to>**	Elemento identificador do componente de destino.
			<input>	**-**	Elemento que representa as mensagens de entrada do componente, cuja propriedade é: timeindex – ordem da mensagem no diagrama bmsc. É uma posição temporal de acordo com sua execução no cenário.
				<name>	Elemento identificador da mensagem. Seu conteúdo se apresenta da seguinte forma: componente_chamador, componente_chamado, nome_mensagem
				<from>	Elemento identificador do componente de origem.

Figura II.1: Descrição os elementos que compõem o documento de carga para a ferramenta LTSA-MSC.

Apêndice III

Funcionalidades Implementadas na Aplicação MyPetStore

Funcionalidade	Sub-funcionalidade	Descrição
Entrar no Sistema	-	Processo de autenticação do usuário no sistema.
Sair do Sistema	-	Processo de fechamento de seção de um usuário logado. Saída do sistema.
Manter usuários	-	Processo de manutenção de usuários válidos no sistema. Nesse módulo, detalhes de um usuário do sistema poderão ser criados, alterados, excluídos e consultados.
	Consultar usuário (Find)	Consulta de usuários no sistema. Os usuários poderão ser consultados através de seu nome e de seu perfil.
	Incluir usuário (Insert)	Inclusão de usuários no sistema. Poderão ser incluídas as seguintes informações por usuário: nome completo, login, senha, status da conta (ativo ou não ativo) e perfis de usuário.
	Alterar usuário (Update)	Alteração de usuários no sistema. Poderão ser alteradas todas as informações descritas no item anterior.
	Excluir usuário(s) (Delete)	Exclusão de usuários no sistema. Permite a exclusão de um ou mais usuários.
Manter Produtos	-	Processo de manutenção de produtos no sistema. Nesse módulo, detalhes de um produto do sistema poderão ser criados, alterados, excluídos e consultados.
	Consultar Produto (Find)	Consulta de produtos no sistema. Os produtos poderão ser consultados através de seu código e de sua descrição.
	Incluir Produto (Insert)	Inclusão de produtos no sistema. Poderão ser incluídas as seguintes informações por produto: código, descrição, quantidade atual, quantidade mínima, unidade e preço unitário.
	Alterar Produto (Update)	Alteração de produtos no sistema. Poderão ser alteradas todas as informações descritas no item anterior.
	Exclui Produto(s) (Delete)	Exclusão de produtos no sistema. Permite a exclusão de um ou mais produtos.
Realizar Compras	-	Processo de realização de compras no sistema. Nesse módulo, detalhes necessários na realização de uma compra de animais de estimação são requeridos para sua efetivação.
	Limpa Carrinho (ClearShopCart)	Reinicia carrinho de compras. O carrinho de compras será colocado em seu estado inicial (vazio) e todas as informações sobre itens que se quer adquirir até o momento serão perdidas.
	Lista Carrinho (ListShopCart)	Consulta de itens que se quer adquirir até o momento. Será fornecida uma lista de itens presentes no carrinho.
	Encerra Carrinho (CloseShopCart)	Efetivação da compra. Nessa funcionalidade o usuário poderá confirmar os itens que se quer adquirir e o frete calculado. Ao final será concluída a compra e, um pedido será encaminhado ao setor responsável caso o usuário possua permissão.
	Insere Produtos no Carrinho (InsertItemShopCart)	Itens que se quer adquirir serão colocados no carrinho bem como suas quantidades. Essas quantidades serão validadas com o disponível a cada item.

Figura III.1: Descrição das funcionalidades implementadas na aplicação MyPetStore.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo