UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RAFAEL DE LIMA

CATCHML - UMA LINGUAGEM PARA MODELAGEM E VERIFICAÇÃO DO TRATAMENTO DE EXCEÇÃO SENSÍVEL AO CONTEXTO EM SISTEMAS UBÍQUOS

FORTALEZA, CEARÁ

2013

RAFAEL DE LIMA

CATCHML - UMA LINGUAGEM PARA MODELAGEM E VERIFICAÇÃO DO TRATAMENTO DE EXCEÇÃO SENSÍVEL AO CONTEXTO EM SISTEMAS UBÍQUOS

Dissertação de Mestrado sob o título "CatchML - Uma Linguagem para Modelagem e Verificação do Tratamento de Exceção Sensível ao Contexto em Sistemas Ubíquos" submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Rossana Maria de Castro Andrade, Ph.D.

Co-Orientador: Lincoln Souza Rocha, D.Sc.

FORTALEZA, CEARÁ

2013

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca de Ciências e Tecnologia

L710c Lima, Rafael de.
CATCHML – uma linguagem para modelagem e verificação do tratamento de exceção sensível ao contexto em sistemas ubíquos / Rafael de Lima.- 2013.
110 f. : il. color., enc. ; 30 cm.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Departamento de Computação, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2013.
Área de Concentração: Teoria dos Grafos (Teoria da Computação).
Orientação: Profa. Ph.D. Rossana Maria de Castro Andrade.
Coorientação: Prof. Dr. Lincoln Souza Rocha.

1.Tratamento de Exceção. 2. Sensibilidade ao Contexto.3. Linguagens de Domínio Específico. I. Título.

CDD 005

Rafael de Lima

CatchML - Uma linguagem de domínio específico para modelagem do tratamento de exceção sensível ao contexto.

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação, da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação na Área de Concentração Ciência da Computação

BANCA EXAMINADORA

Profa. Dra. Rossana Maria de Castro Andrade
(Orientadora)
Universidade Federal do Ceará – UFC

Prof. Dr. Lincoln Souza Rocha
Universidade Federal do Ceará – UFC

Prof. Dr. Nabor das Chagas Mendonça
Universidade de Fortaleza – UNIFOR

Prof. Dr. Francisco Heron de Carvalho Júnior
Universidade Federal do Ceará – UFC

Prof. Dr. José Neuman de Souza
Universidade Federal do Ceará – UFC

Fortaleza, 28 de agosto de 2013

À minha Mãe.

AGRADECIMENTOS

Primeiramente, agradeço a Deus por ser o criador do universo e por ter me dado forças para continuar firme na realização deste trabalho.

Agradeço à minha querida mãe Neli, a pessoa a quem mais amo nessa vida, que jamais se curvou às dificuldades e obstáculos que surgiram, e sempre me deu total apoio, educação e carinho em todos os momentos. Agradeço ao meu avô Cazuza, pelos valores e princípios ensinados. Às minhas tias e tios pela disponibilidade, apoio e afeto. Aos meus primos e primas, por me darem momentos de alegria, diversão e fraternidade.

Agradeço à minha orientadora professora Rossana, pela paciência, incentivo, palavras de força, ensinamentos, e, acima de tudo, pela confiança. Agradeço também ao meu coorientador professor Lincoln, por ter colaborado bastante em todas as etapas deste trabalho, pela confiança depositada e pelos ensinamentos.

Agradeço também aos professores membros da banca, pela compreensão e paciência na revisão desta dissertação, e pelos conselhos dados durante a defesa.

Aos meus amigos de pesquisa pelo incentivo, ajuda e companheirismo dedicados durante todo o mestrado. Aos servidores do GREat, por todo o suporte necessário à realização deste trabalho. Aos meus colegas do Projeto *Tools*, pelo apoio, compreensão e exemplo de profissionalismo.

Obrigado ainda ao CNPq, que deu suporte financeiro essencial à realização deste trabalho.

E por fim, a todos os meus amigos e às demais pessoas que confiaram em mim e me deram apoio na realização desta tarefa.

RESUMO

Em sistemas ubíquos, devido à complexidade inserida pela utilização de informações contextuais, a aplicação de técnicas de tratamento de exceção sensível ao contexto (TESC) tem sido objeto de estudo para muitos pesquisadores. Na literatura são encontradas diversas abordagens que definem conceitos e abstrações úteis para modelagem de TESC. Entretanto, apenas uma dessas abordagens propõe um método para especificação e verificação de modelos no domínio de sistemas ubíquos o qual fornece uma ferramenta para especificação do modelo de TESC através de uma API Java, e gera ainda um relatório de erros em um arquivo texto. A desvantagem dessa abordagem é que o projetista deve se esforçar para entender detalhes de programação irrelevantes ao processo de análise do comportamento excepcional do sistema. Esta dissertação tem portanto como objetivo propor uma linguagem de domínio específico para modelagem de TESC, com o intuito de oferecer abstrações e construtores que permitem expressar conceitos pertinentes e tornar a tarefa de projetar modelos de TESC mais simples e intuitiva. Além disso, a linguagem é integrada com a ferramenta citada anteriormente, o que permite realizar a verificação do modelo de forma automática. Os erros gerados pelo verificador são mostrados agora diretamente no código do modelo facilitando a identificação e correção dos mesmos pelo projetista. A fim de avaliar a linguagem, um estudo de caso é realizado para fornecer indícios de sua viabilidade como alternativa para modelagem de TESC.

Palavras-chave: Tratamento de Exceções. Sensibilidade ao Contexto. Linguagens de Domínio Específico.

ABSTRACT

In ubiquitous systems, due to the complexity added by the use of contextual information, the application of context aware exception handling (CAEH) techniques has many challenges and in the literature several approaches have been found to define concepts and abstractions useful for modeling CAEH. However, only one of these approaches proposes a method for specification and verification of models in the field of ubiquitous systems, which provides a tool for specifying the CAEH model using a Java API, and also generates an error report to a text file. The disadvantage of this approach is that the designer should strive to understand programming details that are irrelevant to the analysis process of the exceptional behavior of the system. Then, this work aims to propose a domain specific language for modeling CAEH, which provides abstractions and constructors that allow to express relevant concepts and make the task of designing CAEH models simpler and more intuitive. In addition, the language is integrated with the tool mentioned before that allows automatic model verification. The errors generated by the verifier are now shown directly in the source code making their identification and correction easier for the designer. In order to evaluate the language, a case study is conducted to provide evidence of its viability as an alternative to modeling CAEH.

Keywords: Exception Handling. Context-awareness. Domain-specific Languages.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

TESC	Tratamento de Exceção Sensível ao Contexto
CatchML	*Context Aware excepTion Handling Modeling Language*
MDD	*Model Driven Development*
CAEHV	*Context Aware Exception Handling Verification*
JCAEHV	*Java Context Aware Exception Handling Verification*
API	*Application Programming Interface*
DSL	*Domain Specific Language*
IDE	*Integrated Development Environment*
GPL	*General Purpose Language*
UML	*Unified Modeling Language*
EBNF	*Extended Backus-Naur Form*
EMF	*Eclipse Modeling Framework*

SUMÁRIO

1 INTRODUÇÃO

Esta dissertação tem como objetivo apresentar uma linguagem específica de domínio para modelagem e verificação do tratamento de exceção sensível ao contexto em sistemas ubíquos, denominada CatchML[1]. Na Seção 1.1 é apresentada uma visão geral sobre sistemas sensíveis ao contexto e os desafios relacionados a essa área. Em seguida, na Seção 1.2 é apresentada a motivação que levou ao desenvolvimento dessa linguagem, bem como as possíveis vantagens decorrentes da sua utilização. Já a Seção 1.3 apresenta os objetivos da dissertação e as contribuições esperadas. Por fim, a Seção 1.4 mostra como a dissertação está organizada.

1.1 Contextualização

Weiser introduziu o termo computação ubíqua ao afirmar que as tecnologias mais profundas e duradouras são aquelas que se tornam invisíveis aos usuários (WEISER, 1991). Segundo ele, as tecnologias iriam se misturar com o ambiente, seriam acessadas através de interfaces naturais e trocariam informações entre si para melhorar os serviços. Além de invisível, um sistema ubíquo deve adaptar sua estrutura e comportamento às mudanças no ambiente, o que torna a sensibilidade ao contexto um fator chave no desenvolvimento de aplicações ubíquas (BARDRAM, 2004). De acordo com a literatura, ainda há uma série de desafios e requisitos encontrados no domínio de Sistemas Ubíquos, em especial com respeito as aplicações sensíveis ao contexto e, para superá-los, pesquisadores têm feito esforços para projetar e implementar plataformas de *middleware*, infraestruturas de suporte e mecanismos que proporcionem serviços ao usuário de forma efetiva (ROCHA et al., 2007; MAIA; ROCHA; ANDRADE, 2009; HONG; SUH; KIM, 2009; LIMA et al., 2011).

Nesse sentido, o desenvolvimento de sistemas ubíquos parte do princípio que os mesmos precisam capturar e interpretar informações de contexto, bem como adaptar seu comportamento às mudanças nessas informações. Os principais desafios enfrentados no desenvolvimento de software ubíquo podem ser agrupados em duas categorias (ROCHA et al., 2011): (i) desafios técnicos, tais como, interação desacoplada, coordenação, interoperabilidade, des-

[1] CatchML é um acrônimo para *Context Aware excepTion Handling Modeling Language* que significa **Linguagem de Modelagem do Tratamento de Exceção Sensível ao Contexto**. Mais informações podem ser obtidas na página do projeto: http://www.great.ufc.br/∼rafaellima/catchml/index.php

coberta e composição dinâmica de serviços, mobilidade, sensibilidade ao contexto, adaptação, autonomicidade, dependabilidade e segurança; e (ii) desafios metodológicos, relacionados com os processos, técnicas e ferramentas de engenharia de software apropriadas para suportar o desenvolvimento sistemático e a produção em larga escala de software ubíquo, bem como o uso de metodologias ágeis e de gestão de qualidade.

Diversas abordagens para desenvolvimento desse tipo de aplicação têm surgido, como as Linhas de Produto de Software (LPS) para o domínio de aplicações móveis e sensíveis ao contexto. As LPS focam principalmente no uso de ativos reutilizáveis para agilizar o processo de desenvolvimento (MARINHO; ANDRADE; WERNER, 2011). Outra abordagem é a de desenvolvimento de aplicações móveis com foco em manutenção adaptativa (FERREIRA FILHO et al., 2010). Além dessas, uma abordagem comumente utilizada é o Desenvolvimento de Software Baseado em Modelo, do inglês *Model-driven software development* (MDD) (SERRAL; VALDERAS; PELECHANO, 2010). Muitas iniciativas têm tirado vantagem de técnicas de MDD na construção de sistemas sensíveis ao contexto (HOYOS; GARCÍA-MOLINA; BOTÍA, 2010; SERRAL; VALDERAS; PELECHANO, 2010; CASSOU et al., 2012), porém ainda é preciso realizar um grande esforço de pesquisa com vistas a prover linguagens, métodos e ferramentas apropriados que deem suporte eficiente ao desenvolvimento baseado em modelo nesse domínio (FRANCE; RUMPE, 2007; HOYOS; GARCÍA-MOLINA; BOTÍA, 2010). Com vistas a aumentar a confiabilidade e robustez no desenvolvimento baseado em modelo, algumas abordagens aplicam modelos formais para análise do comportamento de sistemas sensíveis ao contexto (SAMA et al., 2010; SIEWE; ZEDAN; CAU, 2011; ROCHA, 2013).

No projeto de sistemas sensíveis ao contexto, o comportamento adaptativo é geralmente especificado usando regras de contexto que descrevem situações específicas sobre as quais as aplicações ou componentes devem ser notificados e uma computação deve ser realizada (SAMA et al., 2010; ROCHA, 2013). Tais informações possibilitam aplicar modelos formais para análise de comportamento nesse domínio. Por exemplo, em (SAMA et al., 2010), é definido um modelo formal baseado em máquina de estados finita adaptativa. Nessa abordagem, o projetista especifica a máquina de estados que representa o comportamento de uma aplicação sensível ao contexto utilizando uma API Java[2]. A partir daí, o modelo é transformado em árvores de decisão binária e matrizes de estados, que servem de entrada para algoritmos de ve-

[2]https://code.google.com/p/caaaverification/

rificação de propriedades semânticas sobre a máquina de estados inicial. O resultado obtido são relatórios informando se houve erros que provocaram a não satisfação das propriedades. Da mesma forma que Sama et al. (2010), Siewe, Zedan e Cau (2011) fornecem uma linguagem de programação baseada em álgebra de processos para cálculo de ambientes sensível ao contexto[3]. Nesse trabalho, os autores utilizam uma abordagem formal para especificar o comportamento de um sistema sensível ao contexto. A partir de uma especificação fornecida é possível executar o modelo e simular o comportamento das aplicações.

A aplicação de modelos formais (como os trabalhos de Sama et al. (2010) e Siewe, Zedan e Cau (2011)) auxiliam no aumento da confiabilidade e robustez durante o processo de modelagem e desenvolvimento de sistemas sensíveis ao contexto. Entretanto, também é necessário aplicar técnicas que possam aumentar a confiabilidade das aplicações em tempo de execução. Tais técnicas são importantes devido à natureza dos sistemas computacionais de apresentarem falhas, que podem causar desde pequenos inconvenientes aos usuários até colocar a vida das pessoas em risco (CHETAN; RANGANATHAN; CAMPBELL, 2005). Dessa forma, surge a necessidade de se construir sistemas cada vez mais confiáveis ou que tenham níveis desejáveis de dependabilidade[4]. Dependabilidade é a propriedade que define a capacidade de um sistema computacional de prestar um serviço em que se possa justificadamente confiar, oferecendo fortes indícios de que são confiáveis (AVIZIENIS et al., 2004). Além dessa definição, Avizienis et al. (2004) também apresentam dependabilidade como sendo a capacidade que um sistema tem de evitar falhas que sejam mais frequentes e mais graves do que é aceitável pelo usuário.

Dado esse conceito, percebe-se que os sistemas são passíveis de apresentarem falhas, sendo algumas delas toleradas pelos usuários, enquanto que outras têm impacto negativo no grau de confiança dos mesmos em relação ao sistema. A falha (*failure*) é um evento que ocorre quando o serviço entregue por um sistema (ou componente) desvia da sua especificação funcional. A falha de um serviço pode ser vista como uma transição entre o estado no qual um serviço correto está sendo entregue para um estado no qual um serviço incorreto passa a ser entregue (ROCHA, 2013). Portanto, uma falha é um evento passível de observação externa por parte de seus usuários. Um erro (*error*) é definido como parte do estado total do sistema que pode levá-lo a uma posterior falha. A causa física ou algorítmica de um erro é chamada

[3] http://www.tech.dmu.ac.uk/∼fsiewe/fscca.html

[4] termo utilizado nesse trabalho como equivalente ao termo em inglês *dependability*

de falta (*fault*). A falta é qualquer evento, ou sequência de eventos, que pode provocar um erro no estado interno do sistema. Consequentemente, por computação, esse estado errôneo pode propagar-se internamente até afetar o comportamento do sistema, resultando na ocorrência de uma falha. É importante mencionar que alguns erros não afetam o comportamento do sistema, e, portanto, não causam falhas.

Dessa forma, é preciso que haja estratégias para diminuir as causas primárias das falhas nos sistemas, ou seja, reduzir o número de faltas presentes nas aplicações. Na terminologia apresentada em (AVIZIENIS et al., 2004), foram definidos quatro tipos de estratégias, que são: evitar a existência de faltas, eliminar faltas existentes, tolerar a ocorrência de faltas, e prever efeitos de faltas. Devido à natureza complexa dos sistemas sensíveis ao contexto, desenvolver aplicações nesse domínio que sejam livres de faltas é algo bastante incomum. Nesse sentido, torna-se imprescindível a aplicação de técnicas de tolerância a ocorrência de faltas ao desenvolver aplicações sensíveis ao contexto.

Dentre as técnicas aplicadas para aumentar a capacidade do sistema de tolerar faltas, o tratamento de exceções é uma das mais difundidas. As exceções podem ser utilizadas tanto para descrever situações indesejadas como para comunicação de eventos. Ao aplicar a técnica de tratamento de exceções, as exceções que são previamente especificadas podem ser detectadas e tratadas de maneira conveniente, de tal forma que permita ao sistema manter-se funcionando de acordo com sua especificação. Essa técnica não é nova e vem sendo utilizada há bastante tempo no desenvolvimento de sistemas tradicionais (GOODENOUGH, 1975). Na linguagem Java, por exemplo, os termos *try-catch* são abstrações bem difundidas para codificação de blocos de tratamento de exceção. Entretanto, devido à complexidade inserida pela utilização de informações contextuais, a aplicação de técnicas de tratamento de exceção em sistemas sensíveis ao contexto não é trivial e tem sido objeto de estudo para muitos pesquisadores (DAMASCENO et al., 2006; KULKARNI; TRIPATHI, 2010; BEDER; ARAÚJO, 2011; QUEIROZ FILHO, 2012; CHO; HELAL, 2012; ROCHA, 2013).

1.2 Motivação

O tratamento de exceção sensível ao contexto é uma abordagem de tratamento de exceção específica para lidar com situações anormais em ambientes ubíquos, provendo meios para estruturar as atividades de tratamento de erros em sistemas sensíveis ao contexto (DAMAS-

CENO et al., 2006; MERCADAL et al., 2010; KULKARNI; TRIPATHI, 2010; CHO; HELAL, 2012; ROCHA, 2013). Ao aplicar essa técnica, o contexto e a sensibilidade ao contexto são utilizados pelo mecanismo de tratamento de exceção para definir, detectar e tratar condições excepcionais em ambientes ubíquos. Entretanto, o projeto do tratamento de exceção sensível ao contexto é uma atividade complexa e propensa a erros (CHO; HELAL, 2012; ROCHA, 2013). Nesse cenário, é possível que os projetistas insiram faltas de projeto (do inglês *design faults*) durante o processo de especificação do contexto que caracteriza as exceções contextuais (ROCHA, 2013). Dos trabalhos encontrados na literatura que buscam solucionar desafios relacionados ao tratamento de exceção sensível ao contexto, apenas Rocha (2013) propõe uma abordagem para verificação de faltas de projeto.

Em seu trabalho, Rocha (2013) apresenta um método de verificação de modelos do tratamento de exceção sensível ao contexto, o CAEHV[5]. Rocha (2013) define um conjunto de abstrações e conceitos que permite modelar o tratamento de exceção e obter, a partir da especificação, um modelo formal de comportamento baseado em estruturas de kripke. Essas estruturas caracterizam a evolução e relação dos estados de contexto e permitem analisar o comportamento excepcional do sistema. Nesse sentido, o principal objetivo do método é proporcionar um meio de especificar o tratamento de exceção sensível ao contexto e verificar se a especificação satisfaz um conjunto de propriedades que capturam a semântica dos principais tipos de faltas de projeto. Além de propor o método, Rocha (2013) fornece uma ferramenta denominada JCAEHV[6] que automatiza o modelo formal e o processo de verificação de propriedades. O JCAEHV provê uma API Java para modelagem do comportamento excepcional e geração de relatórios de erros referentes às propriedades não satisfeitas.

A ferramenta proposta por Rocha (2013) possui construtores específicos da linguagem Java, o que requer maior conhecimento de programação dos projetistas. Entretanto, é importante que o projetista não tenha que se esforçar para entender detalhes de programação irrelevantes ao processo de análise do comportamento excepcional do sistema. Nesse sentido, é possível que haja um aumento no número de faltas inseridas no modelo devido às dificuldades inseridas pelo uso da interface de programação e não por conta de erros na especificação do modelo propriamente dito. Uma possível solução para esse problema é aumentar o nível de abs-

[5]Sigla do inglês para *Context Aware Exception Handling Verification.*

[6]Sigla do inglês para *Java Context Aware Exception Handling Verification*. Acessível em www.great.ufc.br/∼lincoln/JCAEHV

tração dos conceitos utilizados durante o processo de modelagem, diminuindo sua dependência em relação à linguagem subjacente. Assim, o foco passa a estar nos detalhes relacionados ao domínio do problema, o que pode proporcionar mais facilidade de uso e produtividade aos *experts* da área.

Sendo assim, é importante prover uma alternativa que facilite a modelagem do tratamento de exceção sensível ao contexto. Tal solução deve permitir a especificação de modelos e suporte à análise dos mesmos em relação à faltas de projeto, seja utilizando um verificador próprio ou através de mapeamento para outras abordagens formais de verificação (e.g., (ROCHA, 2013)). Por se tratar de um domínio de aplicações específico, é necessário buscar alternativas de desenvolvimento que mais se adequem ao problema. De acordo com Deursen, Klint e Visser (2000), existem três abordagens de programação para solução de problemas em domínios específicos: (i) *biblioteca de subrotinas*, que contém subrotinas para realização de tarefas em domínios bem definidos (e.g., equações diferenciais, banco de dados e interfaces de usuário); (ii) *framework orientado a objetos* ou APIs, onde as aplicações invocam métodos através de uma biblioteca implementada sob uma linguagem de programação de propósito geral; e (iii) *Linguagem específica de domínio*, do inglês, *Domain Specific Language* (DSL), que é uma linguagem pequena, normalmente declarativa, que oferece poder expressivo focado em um domínio particular. Dentre as alternativas, a DSL é a que pode atingir o maior nível de abstração do domínio do problema. Consequentemente, projetistas podem entender, validar, modificar e, possivelmente, desenvolver o sistema através da DSL (DEURSEN; KLINT; VISSER, 2000), diminuindo consideravelmente a quantidade de faltas de projeto inseridas no processo de especificação.

1.3 Objetivo e Metodologia

O objetivo principal desse trabalho é desenvolver uma linguagem de domínio específico para apoiar o projetista na modelagem do tratamento de exceção sensível ao contexto e verificar, de forma automática, a consistência dos modelos através da aplicação transparente do mecanismo de verificação de modelos provido pela ferramenta JCAEHV (ROCHA, 2013).

No final desta dissertação, espera-se alcançar as seguintes contribuições:

- Fornecer uma DSL para modelagem do tratamento de exceção sensível ao contexto atra-

vés do uso de abstrações de alto nível e de construtores de linguagem específicos desse domínio como variáveis de contexto simbólicas, definição de restrições e regras de contexto.

- Fornecer uma interface de especificação de modelos de tratamento de exceção que possa ser mapeada para a ferramenta JCAEHV que automatiza o modelo formal baseado em *estruturas de kripke* proposto em (ROCHA, 2013) e possibilita a verificação de propriedades semânticas do modelo.

- Fornecer um ambiente de desenvolvimento integrado, do inglês, *Integraded Development Environment* (IDE) que provê ferramentas de apoio ao projeto de TESC.

A metodologia adotada nesse trabalho segue os passos descritos a seguir:

- Revisão bibliográfica sobre os conceitos e desafios em sistemas ubíquos sensíveis ao contexto;

- Análise do domínio de tratamento de exceção sensível ao contexto que envolveu o estudo de modelos e ferramentas desenvolvidos para esse domínio, entre eles o método CAEHV;

- Análise das principais técnicas para construção de linguagens específicas de domínio;

- Projeto e implementação da linguagem específica de domínio com base nas abstrações obtidas na etapa de análise do domínio do tratamento de exceção sensível ao contexto;

- Integração com o JCAEHV de forma iterativa, com o desenvolvimento de protótipos da ferramenta integrados à medida que o verificador evoluir;

- Realização de um estudo de caso para validação da ferramenta.

1.4 Organização da Dissertação

A organização do restante desta dissertação é descrita a seguir. O Capítulo 2 aborda os conceitos de sistemas sensíveis ao contexto, tratamento de exceções e linguagens de domínio específico. O Capítulo 3 elenca trabalhos relacionados e uma visão comparativa é fornecida. Já o Capítulo 4 mostra a proposta desta dissertação, os principais objetivos, a metodologia de desenvolvimento e possíveis contribuições. O Capítulo 5 mostra a metodologia de avaliação e

validação do trabalho proposto. Por fim, o Capítulo 6 encerra a dissertação com as considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo está dividido em quatro seções nas quais são apresentados os fundamentos e definições necessários para esta dissertação de mestrado. O domínio de estudo é o de *Sistemas Ubíquos Sensíveis ao Contexto*, sendo necessário compreender aspectos relacionados à Computação Ubíqua, *sensibilidade ao contexto* e ao processo de captura, gerenciamento e uso de contexto que estão descritos na Seção 2.2. Além disso, na Seção 2.3 são apresentadas diversas abordagens e conceitos relacionados ao *Tratamento de Exceção Sensível ao Contexto* que fornecem meios para aumentar os níveis de robustez de um sistema ubíquo. Já na Seção 2.4 é possível compreender os passos necessários para se desenvolver uma *Linguagem de Domínio Específico*, analisar exemplos e padrões recorrentes, bem como entender vantagens e desvantagens decorrentes da sua adoção. Por fim, a Seção 2.5 sumariza o capítulo.

2.1 Computação Ubíqua

O objetivo da Computação Ubíqua é promover uma nova forma de interação homem-computador marcada pela proatividade dos computadores em auxiliar as pessoas na execução de suas atividades através de uma interação transparente e casual (WEISER, 1991). Nessa visão, a computação passa a estar embarcada nos objetos do cotidiano tornando-se parte integrante do ambiente das pessoas. Esses ambientes, ditos **ambientes ubíquos**, impõem novos requisitos e desafios ao desenvolvimento de software (MAIA; ROCHA; ANDRADE, 2009). Sendo assim, é importante considerar as seguintes características que são intrínsecas ao projeto de sistemas ubíquos:

- **Distribuição**: sistemas ubíquos são sistemas distribuídos onde as responsabilidades são distribuídas entre os vários elementos (hardware e software) presentes no ambiente. Cada elemento exerce um papel específico para que o sistema atinja seu objetivo global (MAIA; ROCHA; ANDRADE, 2009);

- **Diversidade e Heterogeneidade**: sistemas ubíquos apresentam um elevado grau de diversidade e heterogeneidade. Os dispositivos computacionais que compõem o sistema (e.g., *smartphones*, *tablets* e aparelhos de TV), em sua maioria, são de propósitos específicos, diferentes dos computadores pessoais. Além disso, por serem fornecidos por

fabricantes diferentes, apresentam um alto grau de heterogeneidade (COSTA; YAMIN; GEYER, 2008);

- **Conectividade e Interoperabilidade**: na Computação Ubíqua, a conectividade é vista como sem fronteiras (i.e., em qualquer lugar e a todo instante). Dessa forma, tanto os dispositivos quanto o software movem-se com o usuário de forma transparente, através de várias e heterogêneas redes sem fio de longa (e.g., rede celular – GSM) e curta distância (e.g., rede pessoal sem fio – Bluetooth). Entretanto, para que os elementos do sistema possam colaborar, não basta apenas existir a conectividade é preciso que a interoperabilidade entre elementos heterogêneos seja suportada (COSTA; YAMIN; GEYER, 2008);

- **Mobilidade**: A mobilidade é uma característica comum e importante dos sistemas ubíquos (COSTA; YAMIN; GEYER, 2008). Ela pode ser dividida, basicamente, em 3 (três) tipos: (i) mobilidade de usuário, que garante ao usuário mudar de dispositivo de acesso sem perder a sua sessão; (ii) mobilidade de dispositivo, que dá suporte à mobilidade física do dispositivo garantindo sua conectividade e possibilidade de ser alcançado na rede; e (iii) mobilidade de código, que está relacionada com a possibilidade de migração do estado de execução, dos dados e do próprio código executável do software entre os dispositivos computacionais presentes no ambiente.

- **Sensibilidade ao Contexto**: o contexto é considerado por vários autores como um dos aspectos centrais no projeto de sistemas ubíquos (BARDRAM, 2004; DEY, 2001). O contexto pode ser entendido como um conjunto de informações adicionais que são relevantes para caracterizar a interação entre o usuário e o sistema, incluindo o próprio usuário e o próprio sistema. Um sistema sensível ao contexto é aquele capaz de adquirir informações de contexto, das mais diversas fontes, e inferir conhecimento útil sobre si e sobre o seu ambiente. Esse conhecimento permite ao sistema compreender as mudanças ocorridas e reagir adequadamente modificando sua estrutura e comportamento através da execução de estratégias de adaptação apropriadas.

- **Adaptabilidade**: é a característica do sistema que o habilita a reagir de maneira apropriada às mudanças no seu ambiente de execução. Essa reação implica na modificação da estrutura ou do comportamento do sistema com o objetivo de manter a sua execução

de acordo com seus objetivos e requisitos ou para atender algum interesse do usuário. A adaptação é um requisito importante para melhorar a reusabilidade, portabilidade e confiabilidade de sistemas de software como um todo, suportando a redefinição de requisitos, manutenção e mudanças no ambiente de desenvolvimento e execução (MAIA; ROCHA; ANDRADE, 2009). A adaptação pode ser conduzida diretamente pelo próprio sistema ou de maneira transparente por um sistema de suporte.

- **Autonomia**: sistemas autogerenciáveis são sistemas capazes de se autoconfigurar, autoadaptar, autocurar, automonitorar ou autoajustar, sendo geralmente referenciados como sistemas auto-* ou sistemas autônomos. A construção de sistemas ubíquos autogerenciáveis apresentam inúmeros desafios que vão desde a especificação de requisitos, projeto da arquitetura, até a implantação e execução, exigindo uma atividade sistemática denominada de Engenharia de Software de Sistemas Autogerenciáveis (CHENG et al., 2009).

- **Interoperação Espontânea**: sistemas ubíquos são essencialmente voláteis e abertos (COSTA; YAMIN; GEYER, 2008). Consequentemente, os elementos (de hardware e software) que compõem o sistema precisam ser projetados para estarem aptos a coordenar suas atividades com outros elementos previamente desconhecidos. Além disso, essa interação deve ocorrer de maneira automática sem a necessidade da intervenção do usuário. Para isso, técnicas que permitam descrever e localizar funcionalidades disponíveis no ambiente, no formato de serviços, são extremamente necessárias.

- **Invisibilidade**: um dos objetivos principais da Computação Ubíqua é atingir a "invisibilidade" dos recursos computacionais de tal forma que as pessoas possam utilizá-los de maneira transparente e casual. Essa invisibilidade deve ser de natureza física ou mental. A invisibilidade de natureza física está relacionada com o grau de miniaturização e embarcamento dos dispositivos computacionais no ambiente. Por outro lado, a invisibilidade de natureza mental tem a ver com a maneira como os usuários interagem e percebem os computadores no seu cotidiano. Portanto, os dispositivos computacionais devem disponibilizar acesso rápido às suas funções através de interfaces de interação naturais, eliminado a necessidade de treinamento e alto grau de concentração por parte do usuário (COSTA; YAMIN; GEYER, 2008).

- **Segurança e Privacidade**: sistemas ubíquos, por concepção, proveem um ambiente propício para o compartilhamento de recursos e troca de informações entre os usuários. Por

serem sensíveis ao contexto, sistemas ubíquos devem ter acesso a informações de contexto que descrevem o ambiente físico e as pessoas que estão nele. Dessa forma, é possível que esses recursos sejam usados indevidamente e que mensagens sejam capturadas por indivíduos não autorizados. Consequentemente, o compartilhamento de determinadas informações de contexto pode colocar em risco a segurança e a privacidade dos usuários e do sistema como um todo (COSTA; YAMIN; GEYER, 2008).

2.2 Sensibilidade ao Contexto

Dentre as características citadas na seção 2.1, a *Sensibilidade ao Contexto* desempenha um papel central no desenvolvimento de sistemas ubíquos (BARDRAM, 2004). De acordo com Baldauf, Dustdar e Rosenberg (2007): "*Context-aware systems are able to adapt their operations to the current context without explicit user intervention and thus aim at increasing usability and effectiveness [...]* [1]. Esses sistemas de software herdam inúmeros desafios de desenvolvimento, muitos ainda em aberto, que estão relacionados com a engenharia de requisitos, o projeto, a implementação e o suporte adequado à verificação e validação, tanto em tempo de desenvolvimento quanto em tempo de execução (ROCHA, 2013) (ROCHA; ANDRADE; GARCIA, 2013). Uma aplicação é dita **sensível ao contexto** (*context-aware*) se esta usa contexto para prover serviço ou informações relevantes ao usuário de acordo com o domínio a qual a aplicação está inserida (DEY, 2001). O termo sensível ao contexto foi citado pela primeira vez em (SCHILIT; THEIMER, 1994). Muitas aplicações sensíveis ao contexto têm uma arquitetura complexa e componentes que são responsáveis por representar, gerenciar, inferir e analisar informações de contexto (LEE; CHANG; LEE, 2011). Embora existam diferentes tipos de sistemas sensíveis ao contexto, geralmente, esses sistemas seguem um processo geral de quatro etapas como descrito na Figura 2.1.

A primeira etapa consiste na aquisição da informação de contexto através do uso de sensores físicos ou virtuais. Após adquirir a informação de contexto, o sistema armazena os dados de contexto em seu repositório de acordo com o modelo de contexto adotado. O nível de abstração desses dados é controlado pelo sistema à medida que os dados são agregados e interpretados para derivar informações úteis na etapa de pré-processamento (BALDAUF;

[1] Sistemas sensíveis ao contexto adaptam suas características de acordo com o contexto no qual o usuário está inserido e sem a intervenção explícita do mesmo, com o objetivo de melhorar a usabilidade e efetividade [...] (tradução nossa)

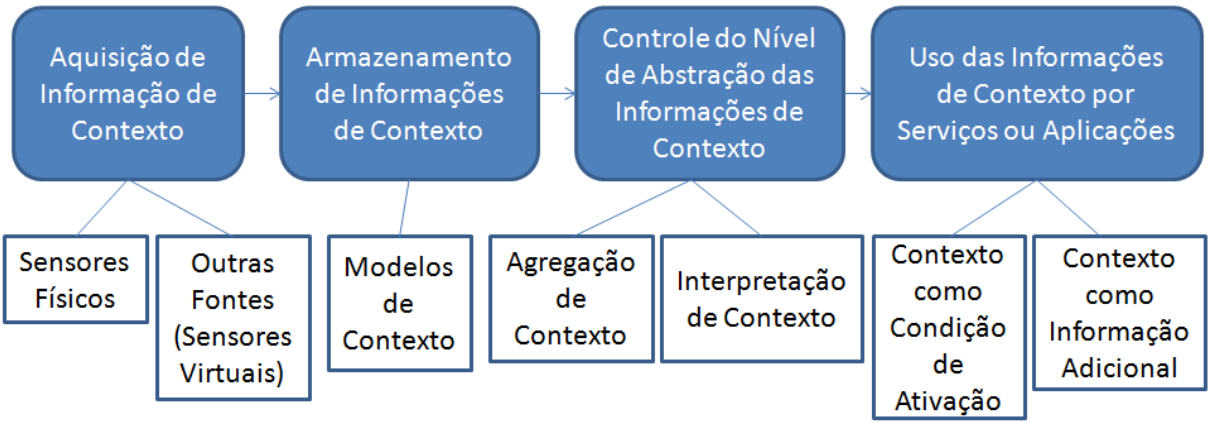


Figura 2.1: Processo geral de um sistema sensível ao contexto. Adaptado de (LEE; CHANG; LEE, 2011)

DUSTDAR; ROSENBERG, 2007). Finalmente, as informações de contexto são utilizadas pelas aplicações como gatilhos para regras de adaptação ou realização de tarefas do usuário (LEE; CHANG; LEE, 2011). As próximas subseções são apresentadas definições, propriedades e questões essenciais sobre esse tipo de sistema.

2.2.1 Definição de Contexto

O termo contexto possui várias definições entre as quais a apresentada em (RYAN; PASCOE; MORSE, 1998) na qual o **contexto** é definido como sendo a localização do usuário, o ambiente, a identidade e o tempo. Há autores que descrevem contexto como sendo localização, identificação de pessoas e objetos próximos, e mudanças naqueles objetos. Dentre as diversas definições, uma das mais referenciadas é a seguinte:

> "Contexto é qualquer informação que possa ser usada para caracterizar a situação de entidades (i.e., uma pessoa, um lugar ou um objeto) que são consideradas relevantes para a interação entre o usuário e uma aplicação, inclusive o próprio usuário e a aplicação." (DEY, 2001)

Já em (VIANA, 2010), a definição de contexto dada por Dey (2001) é estendida e tem foco na aquisição de informações:

> "Toda informação que pode descrever a situação das entidades, e suas relações, envolvidas em uma ação que é considerada importante pelo sistema. Essas entidades são todos os conceitos abstratos e objetos físicos presentes na zona de observação do sistema em um determinado instante t_n de observação".

Essa definição traz consigo os conceitos de zona de observação e zona de interesse. A **zona de observação** abrange todas as entidades e relações que são perceptíveis ao sistema em um dado momento. Restringe-se ao espaço físico coberto pelos sensores e provedores de informações do sistema. Já a **zona de interesse** abrange apenas as entidades e relações às quais o sistema considera serem importantes para seu funcionamento. Portanto, o contexto de um sistema é formado pela interseção entre a zona de observação (suas entidades e relações que podem ser observadas) e a zona de interesse (suas entidades e relações que realmente interessam ao sistema) (VIANA et al., 2009) (VIANA, 2010), conforme ilustrado na Figura 2.2. A vantagem na definição de Viana (2010) é que ela depende da capacidade de captura de contexto em cada instante, caracterizando a evolução dos estados de contexto do sistema. Tal definição será utilizada no decorrer desta dissertação.

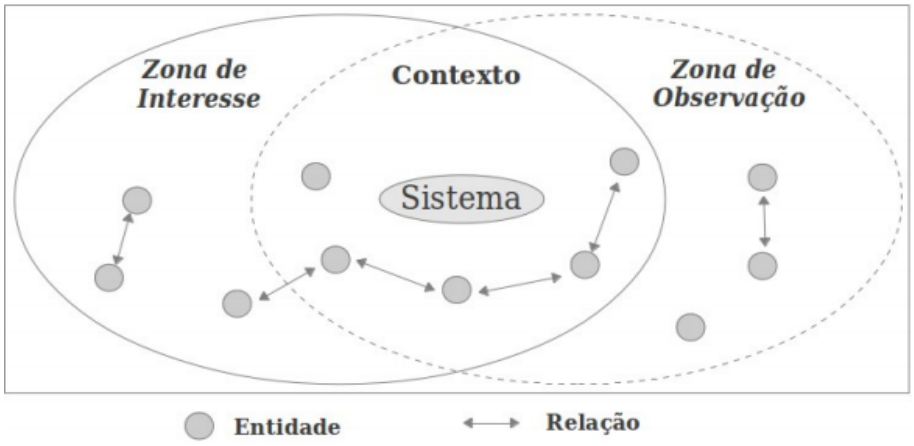


Figura 2.2: Relação entre contexto, zona de observação e zona de interesse. Adaptado de (VIANA, 2010)

Quando se lida com contexto, três entidades podem ser distinguidas (DEY; ABOWD; SALBER, 2001): **lugares** (e.g., salas, edifícios), **pessoas** (e.g., indivíduos, grupos) e **coisas** (e.g., objetos físicos, componentes de computadores). Cada uma dessas entidades pode ser descrita por vários atributos e são classificadas em quatro categorias: **identidade** (cada entidade tem um único identificador), **localização** (a posição de uma entidade, proximidade), **status** (ou atividade, significa as propriedades intrínsecas de um entidade, por exemplo, temperatura e luminosidade para uma sala, processos executando em um dispositivo) e **tempo** (usado para definir uma situação, ordenação de eventos).

2.3 Tratamento de Exceção

Em seu relatório técnico de pesquisa intitulado Conceitos Básicos e Taxonomia de Computação Segura e Confiável (do inglês, *Basic Concepts and Taxonomy of Dependable and Secure Computing*), Avizienis et al. (2004) apresentam uma taxonomia de dependabilidade que abrange os principais conceitos relacionados a essa característica inerente aos sistemas computacionais (AVIZIENIS et al., 2004). De acordo com essa taxonomia, são empregadas técnicas para tratamento e correção de erros em sistemas computacionais. Dentre elas tem-se o tratamento de exceções que é a capacidade que um *software* possui de reagir diante da ocorrência de exceções, continuando ou interrompendo sua execução (DAMASCENO et al., 2006).

Desenvolvedores de sistemas confiáveis frequentemente se referem a erros como exceções porque erros raramente se manifestam durante a atividade normal do sistema. Em situações de erro, um componente gera exceções que modelam a condição de erro e o sistema deve realizar o tratamento daquelas exceções. O tratamento de exceções é uma técnica de recuperação de erros por avanço (*forward error recovery*) tipicamente empregada para melhoria da robustez de sistemas de software (ROCHA, 2013). Nas próximas subseções são apresentados conceitos básicos de exceções contextuais, mecanismo de tratamento de exceções e modelos de tratamento. Para entender melhor o que é o tratamento de exceções, é necessário compreender a motivação para o uso dessa técnica.

O conceito de tratamento de exceções foi apresentado originalmente por Goodenough (1975), que definiu inicialmente o **lançamento de exceções** (do inglês, *raising exception*) como sendo um meio de parametrizar uma resposta a certa condição detectada por uma operação. Essa resposta ou reação a uma exceção levantada recebe o nome de **tratamento de exceção** (do inglês, *exception handling*).

2.3.1 Falha, Erro e Falta

Uma **falha** (*failure*) é um evento que ocorre quando o serviço entregue por um sistema (ou componente) desvia da sua especificação funcional. A falha de um serviço pode ser vista como uma transição entre o estado no qual um serviço correto está sendo entregue para um estado no qual um serviço incorreto passa a ser entregue (ROCHA, 2013). Portanto, uma falha é

um evento passível de observação externa por parte de seus usuários. Um **erro** (*error*) é definido como parte do estado total do sistema que pode levá-lo a uma posterior falha. A causa física ou algorítmica de um erro é chamada de **falta** (*fault*). A falta é qualquer evento, ou sequência de eventos, que pode provocar um erro no estado interno do sistema. Consequentemente, por computação, esse estado errôneo pode propagar-se internamente até afetar o comportamento do sistema, resultando na ocorrência de uma falha. É importante mencionar que alguns erros não afetam o comportamento do sistema, e, portanto, não causam falhas.

2.3.2 Exceção

Uma **exceção** (*exception*) é um evento que modela uma situação em que o fluxo normal de execução do sistema não pode continuar (KNUDSEN, 1987). Para que o sistema continue executando corretamente, o fluxo de execução deve ser desviado e uma computação adicional deve ser empregada para tratar aquela situação (KNUDSEN, 1987). Em sistemas confiáveis, um erro por ser considerado um evento que raramente ocorre durante a execução do sistema e pode ser modelado como uma exceção. O tratamento de exceções provê meios para estruturar as atividades de tolerância a faltas (*fault tolerance*) através da recuperação de erros.

Lee e Anderson (1990) definiram exceções como abstrações que modelam condições de erro em um componente. Essas abstrações permitem aos componentes realizarem o tratamento adequado às condições de erro detectadas. Nesse sentido, o tratamento de exceções pode ser visto como a capacidade que um *software* possui de reagir de maneira adequada a ocorrência de exceções, continuando ou interrompendo a sua execução, com o intuito de preservar a integridade do sistema (GARCIA et al., 2001).

2.3.3 Tipos de Exceções

No modelo de componente Tolerante a Faltas Ideal descrito por Lee e Anderson (1990) (figura 2.3), cada componente de *software* pode receber requisições de serviço de outros componentes. Quando as respostas àquelas requisições estão em conformidade com a especificação, diz-se que são respostas **normais**. Por outro lado, se o componente não é capaz de atender, por algum motivo, àquela requisição de serviço, ele retorna respostas **anormais** ou **exceções** (ROCHA, 2013; LEE; ANDERSON, 1990). Nesse modelo, as exceções podem ser classificadas em três categorias (GARCIA et al., 2001):

- **Exceções de interface**: são sinalizadas quando a requisição não está em conformidade com a interface de serviço do componente;
- **Exceções de falhas**: são sinalizadas para indicar que, por algum motivo, o componente não pôde atender a requisição de serviço; e
- **Exceções internas**: são levantadas internamente ao componente para que este provenha suas próprias medidas de tratamento.

É importante perceber que a partir dessa categorização pode-se distinguir dois tipos de exceção: as **exceções internas**, que são **levantadas** internamente ao componente, e as **exceções externas** (de interface e serviço), que são **sinalizadas** externamente entre componentes.

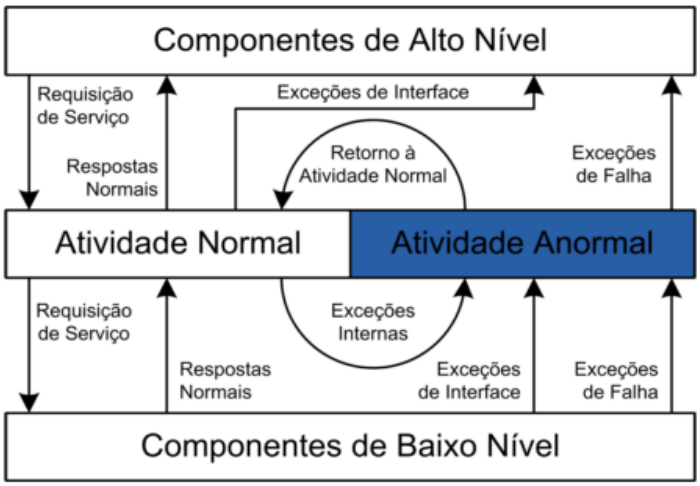


Figura 2.3: Componente Tolerante a Faltas Ideal. Adaptado de (GARCIA et al., 2001)

A atividade de cada componente do modelo pode ser dividida em duas partes: atividade normal e atividade anormal (ou excepcional). Na **atividade normal**, o componente processa as requisições de serviço de acordo com a sua especificação. Por outro lado, na **atividade excepcional**, o componente aplica medidas para tratar as condições de erro detectadas. Dessa forma, componentes podem tratar exceções levantadas durante sua atividade normal ou exceções sinalizadas por componentes de mais baixo nível (que receberam requisição de serviço). Caso o componente não consiga tratar a exceção levantada, esta é propagada para os componentes de mais alto nível (requisitantes de serviço). Depois que a exceção é tratada, o sistema retorna à sua atividade normal (ROCHA, 2013).

2.3.4 Tratadores de Exceção

O **mecanismo de tratamento de exceções** permite aos desenvolvedores definirem exceções e estruturarem o comportamento da atividade excepcional dos componentes através dos **tratadores de exceção**. Os mesmos representam a parte do código do componente que provê medidas específicas para o tratamento da exceção levantada. Dessa forma, quando uma exceção é levantada durante a atividade normal do sistema (ou componente), o mecanismo de tratamento de exceções desvia o **fluxo de controle normal** para o **fluxo de controle excepcional**.

Em tempo de execução, quando uma exceção é levantada, diz-se que houve uma ocorrência de exceção. A instrução que estava sendo executada quando uma ocorrência de exceção foi detectada é chamada de **instrução sinalizadora**. O trecho de código cuja instrução sinalizadora faz parte é chamado de **sinalizador**. Quando uma ocorrência de exceção é detectada, o mecanismo de tratamento de exceções procura o tratador adequado para o tratamento daquela exceção e o invoca. Os tratadores estão associados a uma parte particular do código do componente chamada de *região protegida* ou contexto de tratamento, delimitando o seu *escopo* de atuação. Idealmente, as instruções sinalizadoras devem ficar dentro da região protegida. Por fim, o mecanismo de tratamento de exceções pode ser parte integrante da linguagem de programação utilizada para construir o sistema, possuindo construtores bem definidos, ou ser tratado como uma funcionalidade inserida através de chamadas a bibliotecas externas (GARCIA et al., 2001; ROCHA, 2013).

2.3.5 Tratamento de Exceção Sensível ao Contexto

O tratamento de exceção é bastante utilizado no desenvolvimento de sistemas tradicionais, porém ainda não é muito empregado no desenvolvimento de sistemas ubíquos devido aos requisitos desafiadores desse tipo de sistema. Em sistemas ubíquos, devido à característica de sensibilidade ao contexto, tanto o comportamento normal quanto excepcional são influenciados pelo contexto, tornando mais complexa a tarefa de projetar o tratamento de exceções. Alguns trabalhos têm buscado soluções para o tratamento de exceções sensível ao contexto através da aplicação de abstrações, *middlewares* e mecanismos (BEDER; ARAÚJO, 2011; DAMASCENO et al., 2006; KULKARNI; TRIPATHI, 2010; ROCHA, 2013) que possam tornar menos complexa a tarefa de especificar, projetar e desenvolver aplicações ubíquas com maior

nível de robustez e corretude.

Uma taxonomia com dez aspectos importantes para o tratamento de exceções em sistemas construídos utilizando um paradigma orientado a objetos foi elaborada em (GARCIA et al., 2001). A partir desses aspectos, Damasceno et al. (2006) destacou cinco que se relacionam com o tratamento de exceções em aplicações móveis sensíveis ao contexto, descritos a seguir:

- **Representação da exceção**: que permite que exceções sejam utilizadas e processadas internamente aos sistemas, podendo estarem representadas sob uma forma simples de tipos de dados (e.g., um número inteiro ou uma *string*) ou até na forma de objetos complexos;
- **Separação entre exceção interna e externa**: essa característica permite a diferenciação entre exceções levantadas dentro de um componente das que foram sinalizadas entre componentes;
- **Localização de tratadores**: que possibilita a busca por tratadores em diferentes regiões protegidas, que vão desde a porções de código dentro de um objeto de uma aplicação orientada a objetos até o escopo da própria aplicação onde encontram-se tratadores globais (que são válidos em qualquer parte da aplicação);
- **Associação de tratadores**: que realiza a ligação entre os tratadores e as regiões protegidas onde alguma exceção deve ser tratada. Essa ligação pode ser realizada de forma estática, dinâmica ou semi-dinâmica; e
- **Propagação de exceção**: que é responsável por propagar a exceção dado que um tratador local não foi encontrado. Nesse caso, a exceção deve ser propagada ou sinalizada para outro componente.

Além dos aspectos elencados em (DAMASCENO et al., 2006), outros conceitos importantes para o tratamento de exceção sensível ao contexto têm sido trabalhados, entre os quais o conceito de **exceção baseada em situação** proposto por Cho e Helal (2011). Para eles, é interessante prover uma forma de capturar e representar as situações contextuais que são sequências de estados contextuais em conjunto. Dessa forma, pode-se detectar situações excepcionais que antes não era possível. Um exemplo simples é o caso em que um usuário

entra duas vezes numa sala sem que o sistema registre uma saída. Logo, tem-se uma situação excepcional que só pode ser detectada analisando a sequência de estados e não apenas um único estado de contexto. A partir dessa observação de vários estados sequenciais, o sistema pode caracterizar o erro e realizar algum tratamento excepcional.

Tipos de Exceções Contextuais

As exceções contextuais representam situações anormais que requerem que um desvio no fluxo de execução seja feito para que o tratamento da excepcionalidade seja conduzido. A detecção da ocorrência de uma exceção contextual pode indicar uma eventual falha em algum dos elementos (hardware ou software) que compõem o sistema ou que alguma invariante de contexto, necessária a execução de alguma atividade do sistema, tenha sido violada. Segundo Rocha (2013), as exceções contextuais estão agrupadas em 3 (três) categorias:

- **Exceções Contextuais de Infraestrutura**: Esse tipo de exceção contextual está relacionada com a detecção de situações de contexto que indicam que alguma falha de hardware ou software ocorreu em algum dos elementos que constituem o sistema ubíquo. Um exemplo desse tipo de exceção contextual é descrito em (DAMASCENO et al., 2006) no escopo de um sistema de aquecimento "inteligente". A função principal daquele sistema é ajustar a temperatura do ambiente às preferências dos usuários. Naquele sistema, uma situação de excepcionalidade é caracterizada quando a temperatura do ambiente atinge um valor acima do limite estabelecido pelas preferencias do usuário. Esse tipo de exceção contextual ajuda a identificar, de forma indireta, a ocorrência de falhas cuja origem pode ser o sistema que controla o equipamento de aquecimento (falha de software) ou o próprio equipamento (falha de hardware). O mau funcionamento do sistema de aquecimento é considerado uma situação anormal, pois pode colocar em risco a saúde dos usuários. Observe que para detectar essa exceção contextual é necessário ter acesso à informações de contexto sobre a temperatura do ambiente e as preferências dos usuários.

- **Exceções Contextuais de Invalidação de Contexto**: Esse tipo de exceção contextual está relacionado com a violação de determinadas condições de contexto durante a execução de alguma tarefa do sistema. Essas condições de contexto funcionam como invariantes da tarefa e, quando violadas, caracterizam uma situação de anormalidade. Por exemplo, os autores de (KULKARNI; TRIPATHI, 2010) descrevem esse tipo de exceção em uma

aplicação de leitor de música sensível ao contexto. O leitor de música executa no dispositivo móvel do usuário, enviando um fluxo continuo de som para a saída de áudio do dispositivo. Entretanto, quando o usuário entra em uma sala vazia, o aplicativo busca por algum dispositivo de áudio disponível no ambiente e transfere o fluxo de som para aquele dispositivo. Nessa aplicação, é estabelecido como contexto invariante a necessidade do usuário estar sozinho dentro da sala. Para os autores de (KULKARNI; TRIPATHI, 2010), a violação desse invariante é considerado uma situação excepcional, pois o seu não cumprimento pode trazer desconforto ou aborrecimento para as demais pessoas presentes na sala. Note que a detecção dessa exceção depende de informações de contexto sobre a localização do usuário e o número de pessoas que estão na mesma sala que ele.

- **Exceções Contextuais de Segurança**: Esse tipo de exceção está relacionada com situações de contexto que ajudam a identificar a violação de políticas de segurança (e.g., autenticação, autorização e privacidade) e demais situações que podem colocar em risco a integridade física ou financeira dos usuários do sistema. Por exemplo, o sistema de registro médico sensível ao contexto apresentado em (KULKARNI; TRIPATHI, 2010) descreve esse tipo de exceção. Nesse sistema, existem três usuários envolvidos: os pacientes, os enfermeiros e os médicos. Os médicos podem fazer registros sobre seus pacientes e os enfermeiros podem ler e atualizar esses registros ao tempo em que assistem aos pacientes. Entretanto, os enfermeiros só podem ter acesso aos registros se estiverem dentro da enfermaria em que o paciente se encontra e se o médico responsável estiver presente. Naquele sistema, quando um enfermeiro tenta acessar os registros do paciente, porém não se encontra na mesma enfermaria que este paciente ou encontra-se na enfermaria, mas o médico responsável não está presente, caracteriza-se uma situação excepcional. Perceba que a detecção desse tipo de exceção depende das informações de contexto sobre a localização e o perfil do paciente, do enfermeiro e do médico.

2.4 Linguagens Específicas de Domínio

Uma **linguagem específica de domínio**, ou do inglês *domain-specific language* (DSL) é uma linguagem de especificação ou programação dedicada a um domínio de problema específico, uma solução de um problema particular ou uma técnica de representação (VOELTER et al., 2013). As abstrações e notações utilizadas são naturais (ou adequadas) para os

especialistas daquele domínio específico. Exemplos comuns de DSLs são a linguagem HTML, projetada para representar o *layout* de páginas da Web, e a linguagem SQL, destinada a consultar e atualizar bancos de dados. As DSLs são geralmente declarativas, podendo serem vistas, consequentemente, como linguagens de especificação (DEURSEN; KLINT; VISSER, 2000). As DSLs também são conhecidas como linguagens de quarta geração ou "Pequenas linguagens", caraterizando linguagens específicas que não incluem muitas características encontradas em **linguagens de propósito geral**, do inglês *General-purpose language* (GPL) (e.g., Java, C e C++) (MERNIK; HEERING; SLOANE, 2005; VOELTER et al., 2013).

2.4.1 Conceitos

Além dos conceitos já discutidos de *linguagens específicas de domínio* (DSL) e *linguagens de propósito geral* (GPL), outros termos relacionados ao tema foram explicitados em (VOELTER et al., 2013) e serão utilizados no decorrer dessa dissertação. Os termos **modelo**, **código** e **programa** são utilizados como sinônimos para o código que é escrito em uma GPL ou DSL. Se modelo e programa estiverem juntos, modelo se refere à representação mais abstrata (e.g., o programa foi gerado a partir do modelo). Além disso, é possível diferenciar DSLs **internas**, que são criadas tendo como base outras linguagens, das DSLs **externas**, que são criadas desde o início sem reutilizar nenhuma linguagem.

Também é importante diferenciar o *motor de execução* e a *plataforma alvo*. Esses termos se aplicam tanto para programas escritos em DSLs quanto em GPLs. A **plataforma alvo** é o sistema operacional ou ambiente no qual o programa precisa ser executado e assume-se que é algo que não pode ser alterado durante o processo de desenvolvimento. Já o **motor de execução** (*execution engine*) pode ser alterado e preenche a lacuna entre a linguagem e a plataforma alvo. Ele pode ser um interpretador ou compilador. Um **interpretador** é um programa que executa na plataforma alvo cuja função é carregar um programa e agir sobre ele. Já um **compilador** transforma um programa em um artefato (frequentemente um código fonte de uma GPL) que pode executar diretamente na plataforma alvo.

Um **metamodelo** de um modelo (ou programa) é um modelo que define uma linguagem usada para descrever um modelo. Por exemplo, o metamodelo da UML é um modelo que define todas os conceitos da linguagem que define a UML como classes, associação e propriedades. Dessa forma o prefixo *meta* pode ser entendido como *a definição do* (VOELTER

et al., 2013). Já a noção de **abstração** é diferente, embora esta caracterize o relacionamento entre dois artefatos (programas ou modelos). Um artefato a_1 é mais abstrato que um artefato a_2 se deixa de fora alguns dos detalhes de a_2, enquanto preserva as características de a_2 que são importantes para o propósito de a_1.

Segundo Voelter et al. (2013), toda linguagem, seja ela de domínio específico ou não, é composta dos seguintes ingredientes:

- **Sintaxe Concreta**: define a notação com a qual os usuários podem expressar programas. Ela pode ser textual, gráfica, tabular ou uma combinação dessas;
- **Sintaxe Abstrata**: é a estrutura de dados que guarda a informação semântica expressa pelo programa. É tipicamente uma árvore ou um grafo. Ela não contém alguns detalhes sobre a notação como palavras-chave, símbolos ou espaços em branco;
- **Semântica Estática**: é o conjunto de restrições ou regras de tipagem que devem estar em conformidade nos programas (que também devem estar estruturalmente corretos); e
- **Semântica de Execução**: refere-se ao comportamento do programa, uma vez que o mesmo é executado. É realizada pelo motor de execução.

2.4.2 Projeto

As principais etapas no desenvolvimento de uma DSL são discutidos em (MERNIK; HEERING; SLOANE, 2005) e em (VOELTER et al., 2013). A primeira fase consiste na análise do domínio que objetiva identificar e descrever os conceitos do domínio e suas propriedades. Para tanto, é preciso identificar o domínio do problema e coletar todo o conhecimento relevante no domínio escolhido. Após isso, é necessário mostrar as conclusões obtidas da análise que levam aos requisitos da DSL a ser criada. Finalmente, construir uma biblioteca que implementa os requisitos e projetar um compilador que possa traduzir programas escritos na DSL para a biblioteca (DEURSEN; KLINT; VISSER, 2000). Para Voelter et al. (2013), o processo de desenvolvimento se resume em duas etapas distintas: o projeto da linguagem e a implementação da linguagem.

A seguir, têm-se os requisitos comuns levados em consideração no projeto de uma DSL (DEURSEN; KLINT; VISSER, 2000; MERNIK; HEERING; SLOANE, 2005; HOYOS;

GARCÍA-MOLINA; BOTÍA, 2010):

- **Alto nível de abstração**: a linguagem deve prover alto nível de abstração através de construtores que sejam pertinentes ao domínio, de forma a motivar usuários que não são desenvolvedores a utilizarem a DSL;
- **Independência de plataforma**: a linguagem deve permitir a criação de modelos independente da plataforma e que abstenha o usuário de ter que prover detalhes de implementação;
- **Reusabilidade**: a linguagem deve promover o reuso de modelos, significando que a especificação fornecida pode ser utilizada em outras aplicações usando o mesmo contexto; e
- **Usabilidade**: a linguagem dever ser fácil e intuitiva para o usuário.

Voelter et al. (2013) elenca sete dimensões a serem consideradas no projeto de uma DSL a saber:

- **Expressividade**: uma linguagem *A* é *mais expressiva* do que uma linguagem *B* em um domínio *D* se o tamanho de qualquer programa que resolva um problema em *D* escrito na linguagem *A* seja menor do que o seu correspondente escrito na linguagem *B*. Dessa forma, quanto menor o domínio, mais especializada é a linguagem, consequentemente têm-se programas menores e mais expressivos;
- **Cobertura**: é o domínio que pode ser expresso por uma DSL. A cobertura é dada pela razão entre a quantidade de programas expressáveis pela linguagem pelo total de programas existentes no domínio. Há casos em que a DSL pode ser projetada para cobrir apenas um subconjunto de um domínio;
- **Semântica**: divide-se em semântica estática e semântica de execução. A **semântica estática** diz respeito a restrições e sistemas de tipos utilizados para garantir a consistência da linguagem. **Semântica de execução** diz respeito a como o programa será executado (qual o comportamento observado). Pode ser através de transformação (mapeamento de um programa escrito na DSL em uma outra linguagem) ou interpretação (um programa avalia a estrutura do programa escrito na DSL à medida que é executado);

- **Separação de interesses**: um domínio pode ser composto de diferentes interesses (*concerns*). Cada interesse cobre diferentes aspectos do domínio. A separação em interesses é definida por aspectos do sistema especificado pelos projetistas ou em momentos distintos no processo de desenvolvimento;

- **Completude**: refere-se ao grau a qual uma linguagem pode expressar programas que contém toda a informação necessária para executá-los. Um programa expresso em uma DSL incompleta necessita de especificações adicionais para torná-lo executável, tais como arquivos de configuração ou código escrito em uma linguagem de baixo nível;

- **Modularidade**: o reuso de partes modulares de código torna o desenvolvimento mais eficiente, dado que funcionalidades semelhantes não precisam serem implementadas repetidamente. O mesmo conceito pode ser aplicado para o reuso de linguagens ou de partes da linguagem, o que torna as DSLs mais eficientes; e

- **Sintaxe concreta**: uma linguagem deve utilizar notações que representam o domínio de forma direta e simples. Para tanto, têm-se algumas características importantes que devem ser pensadas ao se projetar a sintaxe concreta. A sintaxe deve poder ser escrita de forma eficiente e fácil de ser lida. O aprendizado deve ser intuitivo para novos usuários e eficiente para especialistas, e, por fim, deve possibilitar aos usuários expressar de forma efetiva problemas típicos do domínio.

Para Mernik, Heering e Sloane (2005) a forma mais fácil de projetar uma DSL é baseá-la em uma linguagem ou API já existente. Benefícios possíveis dessa abordagem são uma implementação mais fácil e maior familiaridade para os usuários. Porém, a última só se aplica se os usuários também forem programadores da linguagem existente o que pode não ser o caso. Outra abordagem é estender uma linguagem existente com novas funcionalidades com foco em questões do domínio específico. Na maioria dos casos, as funcionalidades da linguagem se mantém disponíveis. O desafio é integrar as funcionalidades específicas do domínio com a linguagem anterior de uma maneira transparente (MERNIK; HEERING; SLOANE, 2005).

2.4.3 Implementação

Existem diversas abordagens para se implementar uma DSL. Cada abordagem tem suas vantagens e desvantagens. É possível criar uma linguagem através da construção de compi-

ladores, interpretadores, preprocessadores ou até embarcando ou estendendo a nova linguagem em um linguagem base.

Abordagens

A abordagem clássica para se desenvolver uma nova linguagem é criando um **compilador** ou **interpretador**. É possível utilizar ferramentas padrão para construção de compiladores ou ferramentas dedicadas à implementação de DSLs. A principal vantagem de construir um compilador ou interpretador é que a implementação é completamente adaptada para a DSL com respeito à notações e primitivas. A sintaxe da linguagem pode ser mais próxima das notações utilizadas pelos especialistas do domínio. Além disso, detecção de erros, análise estática e otimizações podem ser feitas no nível do domínio (DEURSEN; KLINT; VISSER, 2000). Um dos principais problemas é o custo de construir tal compilador ou interpretador desde o início, ou seja, não reutilizar outras implementações, embora algumas ferramentas voltadas para DSLs serem particularmente desenhadas para superarem esse problema como os *frameworks* de construção de linguagens.

Para Mernik, Heering e Sloane (2005) é possível diferenciar as vantagens e desvantagens ao se optar por construir compilador ou interpretador. Segundo ele, o interpretador oferece maior simplicidade, maior controle sobre o ambiente de execução e extensão mais fácil. Já o compilador traria ganhos em relação à análise estática, permitindo que a mesma seja feita de forma completa em um programa/modelo da DSL. Como alternativa à implementação de uma DSL do início através de compiladores ou interpretadores, é possível *estender* uma linguagem base. A principal vantagem dessa abordagem é que todas as funcionalidades da linguagem base permanecem disponíveis na DSL não necessitando serem re-implementadas, o que diminui razoavelmente o custo em relação à construção de compiladores ou interpretadores. Ao implementar extensões de domínio específico de uma linguagem base, a implementação da linguagem base pode ser reutilizada de três formas diferentes:

- **Bibliotecas específicas de domínio embarcadas**: Nessa abordagem, mecanismos existentes, tais como definições para funções ou operadores com sintaxe definida pelo usuário, são utilizadas para construir uma biblioteca de operações do domínio específico. Os mecanismos de sintaxe da linguagem base são utilizados para expressar o idioma do domínio. Uma vantagem dessa abordagem é que o compilador ou interpretador da linguagem base

é reutilizado integralmente para a DSL. Além disso, o custo para treinamento pode ser menor, visto que muitos usuários já podem conhecer a linguagem base. Porém, existem desvantagens que envolvem o uso de uma linguagem base, tais como a expressividade limitada dos mecanismos sintáticos e a má-qualidade dos relatórios de erros, pois não utilizam conceitos do domínio (MERNIK; HEERING; SLOANE, 2005);

- **Preprocessamento ou macro processamento**: Nessa abordagem, os novos construtores são traduzidos em expressões da linguagem base por um preprocessador. A principal vantagem dessa abordagem é a simplicidade. A principal desvantagem é que análise estática e otimização não podem ser feitas a nível de domínio. Consequentemente, o código gerado é propenso a erros que são reportados ao usuário apenas no nível da linguagem base ou em tempo de execução, o que pode causar perda de produtividade. Segundo Mernik, Heering e Sloane (2005), uma vantagem dessa abordagem é que apenas uma varredura léxica simples é suficiente, sem a necessidade de análises sintáticas complexas baseadas em árvore; e

- **Compilador ou interpretador extensível**: Essa abordagem é similar a anterior com a diferença de que a fase de preprocessamento é agora integrada ao compilador. A principal vantagem é que uma melhor checagem de tipos e melhor otimização é possível.

Para Mernik, Heering e Sloane (2005), existem diversos fatores que determinam a escolha da melhor abordagem para o desenvolvimento de uma DSL. Segundo ele, se a DSL é projetada do início não tendo nenhuma característica comum com linguagens existentes, a melhor abordagem é implementar como uma linguagem embarcada. Porém, se houver a necessidade de se realizar análise, verificação, otimização, paralelização ou transformação no nível do domínio, ou que a notação do domínio específico deva ser estritamente obedecida, ou ainda que a comunidade de usuários esperada seja muito grande, então é mais adequado implementar um compilador ou interpretador. É possível que a DSL tenha elementos em comum, restrinja características ou que estenda tipos de dados ou sintaxe de linguagens existentes. Nesses casos, é preciso analisar todas as abordagens e escolher a que tenha o maior custo-benefício. Compiladores ou interpretadores têm o pior custo de implementação, porém trazem mais benefícios para os usuários. Na prática, tal análise de custo-benefício é dificilmente praticada sendo a decisão guiada somente pela experiência do desenvolvedor (MERNIK; HEERING; SLOANE, 2005).

Suporte

Para desenvolver uma DSL, é preciso conhecimento no domínio e competência técnica no desenvolvimento de linguagens. O processo de desenvolvimento pode ser facilitado através do uso de um sistema de desenvolvimento de linguagens (*Language Developing System*) ou kit de ferramentas (*toolkit*). O princípio geral é que eles geram ferramentas a partir da descrição de linguagens (MERNIK; HEERING; SLOANE, 2005). As ferramentas geradas podem variar de um checador de consistência e interpretador para um ambiente integrado de desenvolvimento (IDE), que consiste de editor com marcador para sintaxe, formatador, checador de consistência, ferramentas de análise, interpretador ou compilador/gerador de aplicações e depuradores de código se a DSL for executável (os outros benefícios também se aplicam se a DSL for não-executável). Mernik, Heering e Sloane (2005), Deursen, Klint e Visser (2000) enumeram dezenas de sistemas de desenvolvimento de linguagens, cada um com suas vantagens e desvantagens.

Apesar da grande quantidade de sistemas para desenvolvimento de linguagens, o assunto ainda tem sido abordado por diversos pesquisadores, entre os quais Martin Fowler que introduz em seu artigo intitulado *Language Workbenches: The Killer-App for Domain Specific Languages*[2] o termo ***language workbench*** para denominar os sistemas para desenvolvimento de linguagens mais recentes. Segundo ele, seria um nome genérico para essa nova classe de ferramentas que buscam simplificar o processo de desenvolvimento de DSLs através de facilidades como a não necessidade de escrever um *parser* e a integração simbólica que permite que mudanças nos símbolos da linguagem sejam propagados por todo o sistema. Voelter et al. (2013) menciona em seu livro que a utilização de ferramentas de suporte ao desenvolvimento de DSLs não é algo novo. Há algum tempo atrás, já era possível construir linguagens customizadas através do uso de geradores de *parsers* como lex/yacc[3], ANTLR[4] or JavaCC[5]. Entretanto, ele afirma que sente que as *language workbenchs* fazem uma diferença qualitativa em relação a seus predecessores. Voelter et al. (2013) destaca a utilização de três *frameworks* para construção de linguagens que representam o atual estado da arte: Spoofax, Xtext e MPS. Outros exemplos de ferramentas podem ser vistas no site *Language Workbench Competition*[6].

[2] http://www.martinfowler.com/articles/languageWorkbench.html
[3] http://dinosaur.compilertools.net/yacc/
[4] http://www.antlr.org/
[5] https://javacc.java.net/
[6] http://www.languageworkbenches.net

- **Eclipse Modeling + Xtext**: O projeto *Eclipse Modeling* é um ecossistema - *frameworks* e ferramentas - para modelagem ou criação de DSLs e tudo que é preciso para tal. O Xtext[7] é um *framework* para construção de linguagens textuais que faz parte do projeto Eclipse Modeling e é bastante maduro com uma grande comunidade de usuários. Além disso, o projeto Eclipse Modeling provê um grande número de complementos que dão suporte à construção de ambientes de desenvolvimento integrado para DSLs bastante sofisticados;

- **SDF/Stratego/Spoofax**: SDF é um formalismo para definir *parsers* para linguagens livre de contexto. Stratego é um sistema de reescrita de termos usado para transformações em árvores de sintaxe abstrata e para geração de código. Spoofax é uma IDE baseada no Eclipse que provê um ambiente robusto para trabalhar com SDF e Stratego. O mesmo não é largamente utilizado como Xtext, mas tem uma boa quantidade de características avançadas para composição e modularização de linguagens; e

- **JetBrains MPS**: O *Meta Programming System* (MPS) é um *framework* de construção de linguagens que não necessita de gramática ou *parser*. Ao invés disso, edições no programa mudam a árvore de sintaxe abstrata diretamente, que é então projetada em forma de texto. Como consequência, MPS suporta notações mistas (textual, simbólica, tabular e gráfica) e uma grande quantidade de características composicionais. Não é tão usada quanto Xtext, mas provê inúmeras características avançadas.

Voelter et al. (2013) aborda a implementação de uma DSL de forma prática, enumerando as diferentes preocupações que um desenvolvedor deve ter durante o processo de implementação ao utilizar uma *language workbench*. É preciso definir as sintaxes concreta e abstrata, bem como o mapeamento entre as duas. O mapeamento pode ser feito tanto através de *parser* como de forma projecional. Há duas maneiras de definir o relacionamento entre a sintaxe concreta e a sintaxe abstrata. A primeira é partir da definição da sintaxe concreta e derivar a sintaxe abstrata. Essa abordagem é utilizada pelo Xtext, o qual deriva o metamodelo Ecore de uma gramática Xtext. A outra forma é definindo primeiro a sintaxe abstrata.

[7] http://www.eclipse.org/xtext

2.4.4 Vantagens e desvantagens

Em combinação com uma biblioteca de programação, uma GPL pode agir como uma DSL. Porém, uma DSL oferece especificidade de domínio com mais vantagens, através do fornecimento de notações, conceitos e abstrações específicos, bem como de possibilidades para análise, verificação, otimização, paralelização e transformação, que são operações essenciais para um sistema e que, normalmente, não são viáveis quando o programa é escrito através de uma GPL. Por exemplo, em (SOUZA; CASTRO FILHO; ANDRADE, 2010) se utiliza uma DSL com abordagem orientada a modelo que permite a criação de objetos de aprendizagem customizados para serem usados posteriormente como componentes de outros objetos de aprendizagem pelos professores. Essa abordagem poderia ter sido feita através de uma API integrada em uma GPL, porém preferiu-se criar uma DSL. Isso quer dizer que a escolha de qual abordagem utilizar na criação da DSL depende também do conhecimento de quem irá desenvolver a linguagem.

As DSLs trocam generalidade por expressividade em um domínio limitado, ao prover notações e conceitos específicos, oferecendo ganhos substanciais em expressividade e facilidade de uso quando comparadas com GPLs (MERNIK; HEERING; SLOANE, 2005). Porém adotar uma abordagem de domínio específico para engenharia de software envolve ambos riscos e oportunidades. Os benefícios incluem (DEURSEN; KLINT; VISSER, 2000):

- DSLs permitem que as soluções sejam expressadas no idioma e nível de abstração do domínio do problema. Consequentemente, *experts* no domínio podem entender, validar, modificar e, inclusive, desenvolver programas através da DSL;
- DSLs melhoram a produtividade, confiabilidade, manutenibilidade e portabilidade;
- Os modelos oferecidos por DSLs são concisos, auto-documentados e podem ser reutilizados de diferentes formas; e
- DSLs permitem validação e otimização no nível do domínio.

Apesar das vantagens na aplicação de DSLs, seu uso elenca novos desafios (FRANCE; RUMPE, 2007). Um deles é o de melhoria (evolução) contínua da ferramenta, visto que cada DSL tem seu conjunto de ferramentas e que as mesmas precisam evoluir à medida que o domínio evolui, o que pode ter um alto custo de manutenção. Outro desafio é o aumento significativo

de problemas de interoperabilidade, versionamento da linguagem e migração da linguagem, decorrente do uso de muitas DSLs. Dessa forma, é preciso que o desenvolvedor tenha profundo conhecimento do domínio e de questões relacionadas à metodologia e processo ao qual a DSL será submetida. Além das questões citadas, Deursen, Klint e Visser (2000) elencam as seguintes desvantagens no uso de DSLs:

- O custo de projetar, implementar e manter uma DSL;
- Os custos para treinar os usuários da DSL;
- A disponibilidade limitada das DSLs; e
- A dificuldade de encontrar um escopo apropriado para DSLs.

2.5 Sumário

Este capítulo introduziu os conceitos básicos sobre sistemas sensíveis ao contexto, tratamento de exceções e linguagens específicas de domínio.

Na seção sobre sistemas sensíveis ao contexto foi apresentada uma arquitetura recorrente de uma aplicação sensível ao contexto. Também foram apresentadas algumas definições de contexto e, dentre elas, a que é utilizada nessa dissertação. Além disso, foram mostradas diferentes abordagens para modelagem de contexto, bem como técnicas utilizadas para controlar o nível de abstração das informações contextuais. Essas técnicas são importantes pois nas fases iniciais do desenvolvimento de um sistema sensível ao contexto é comum os projetistas trabalharem com as informações contextuais em um nível de abstração mais alto do que em fases posteriores. Ao final da seção, foram apresentadas características relacionadas à modelagem do comportamento desse tipo de aplicação e quais modelos formais podem ser utilizados para representá-lo. O levantamento dessas características também é importante, pois permite entender melhor o projeto de um sistema sensível ao contexto.

Na seção sobre tratamento de exceções, foi apresentada uma taxonomia que elenca os principais atributos, ameaças e meios para alcançar a dependabilidade em sistemas de software. Esses conceitos auxiliam a entender as maneiras possíveis de se prover níveis desejados de robustez a um sistema. Logo após, foram apresentados os conceitos de exceção, tipos de exceções e tratadores que constituem os fundamentos básicos relacionados à técnica de tratamento

de exceções. Esses fundamentos permitem compreender melhor essa técnica e vislumbrar meios para aplicá-la em sistemas sensíveis ao contexto.

Por fim, a seção sobre linguagens específicas de domínio, foram mostrados os conceitos básicos necessários para seu desenvolvimento. Além disso, foram mostrados detalhamentos sobre o projeto e as principais abordagens para se implementar uma DSL. O uso de ambientes de suporte ao desenvolvimento (como o Xtext) também foi apresentado e mostrou o estado da arte em se tratando de desenvolvimento de linguagens de domínio específico. Também foram mostradas as vantagens e desvantagens decorrentes da utilização dessas linguagens reforçando a importância de estar ciente dos riscos no desenvolvimento e dos benefícios de sua adoção.

3 TRABALHOS RELACIONADOS

Nesta seção são elencados alguns trabalhos propostos na literatura que tem relação com esta dissertação. Inicialmente, são discutidos trabalhos que fornecem abordagens para tratamento de exceções sensível ao contexto. Em seguida, são discutidos trabalhos que fornecem abordagens específicas de domínio para solução de problemas em sistemas sensíveis ao contexto. Nas pesquisas realizadas nesta dissertação não foi encontrado nenhum trabalho que fornecesse uma abordagem de domínio específico com o foco na modelagem do processo de tratamento de exceção sensível ao contexto. No entanto, foi possível obter informações relevantes do domínio que possibilitou a conclusão do presente trabalho. No final do capítulo são apresentadas as conclusões.

3.1 Abordagens para Tratamento de Exceção Sensível ao Contexto

Na literatura há diversos trabalhos relacionados ao tratamento de exceção sensível ao contexto que definem aspectos importantes desse domínio. Porém, apenas o trabalho de Rocha (2013) aborda a modelagem de TESC que é um dos objetivos dessa dissertação. A seguir, são apresentados alguns desses trabalhos.

3.1.1 Damasceno et al. (2006)

Damasceno et al. (2006) elencam um conjunto de abstrações essenciais para o tratamento de exceção sensível ao contexto. Nesse trabalho, os autores propõem e implementam um mecanismo de tratamento de exceção sensível ao contexto para aplicações móveis. Para a implementação da solução, os autores utilizaram o *middleware* ***MoCA*** (SACRAMENTO et al., 2004). O ***MoCA*** permite a construção de aplicações móveis e colaborativas sob um modelo baseado em *publish-subscribe* para coordenação dos agentes de software. Além disso, este *middleware* possui serviços que possibilitam aos agentes perceberem o contexto (DAMASCENO et al., 2006).

Em seu trabalho, Damasceno et al. (2006) definem dois conceitos importantes que são utilizados nesta dissertação, os conceitos de **contexto excepcional** e **escopo**. Segundo eles, o contexto excepcional é um conjunto formado por uma ou mais condições contextuais

indesejadas ou perigosas que representem uma falta no ambiente. Esse contexto excepcional pode estar associado a um usuário específico, a um agente da aplicação ou a um dispositivo móvel (DAMASCENO et al., 2006). Além da definição de contexto excepcional, os autores estabeleceram a definição de escopo com a delimitação de níveis de captura e propagação de exceções, sendo eles: um dispositivo, um grupo de dispositivos, um servidor e uma região. Esse conceito de escopo é importante para o gerenciamento dos fluxos excepcionais e de retorno à atividade normal da aplicação.

O gerenciamento dos fluxos excepcionais se dá através de quatro funcionalidades: (i) detecção de exceções contextuais; (ii) busca sensível ao contexto de tratadores; (iii) tratamento sensível ao contexto; e (iv) propagação sensível ao contexto de exceções. Essas funcionalidades abrangem os principais requisitos de tratamento de exceções em aplicações móveis e sensíveis ao contexto. Tais funcionalidades do mecanismo são providas através de componentes definidos em uma arquitetura de alto nível. A Figura 3.1 mostra os componentes dessa arquitetura e como eles estão organizados. A seguir, são mostradas as funções de cada componente:

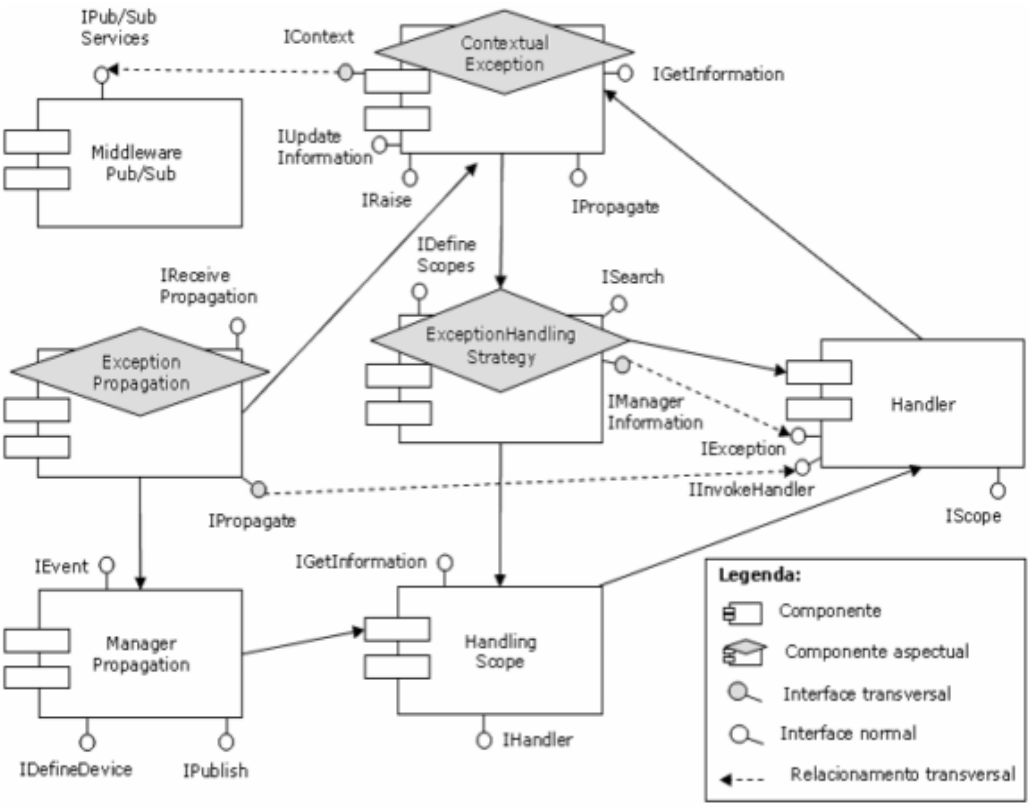


Figura 3.1: Arquitetura para tratamento de exceção sensível ao contexto (DAMASCENO et al., 2006)

- ***Middleware Pub/Sub***: é um componente externo que representa a infraestrutura *publish-subscribe* utilizada para coordenação dos agentes e recuperação das informações de contexto. Na implementação específica de Damasceno et al. (2006), é utilizado o *MoCA*;

- ***Contextual Exception***: é o componente responsável pela especificação de exceções contextuais e por subscrever eventos de interesse associados a cada informação contextual. É esse componente que levanta as exceções contextuais na ocorrência de alguma condição de contexto excepcional ou se houver solicitação pelo componente de propagação;

- ***Exception Handling Strategy***: deve gerenciar as exceções e seus respectivos tratadores, permitindo a especificação de estratégias de buscas de tratadores e de ordem de prioridade entre os escopos;

- ***Handling Scope***: gerencia os escopos de tratamento das exceções contextuais e lida com quais tratadores e dispositivos estão associados a cada escopo;

- ***Handler***: é o componente responsável pela especificação dos tratadores e pela verificação das condições de seleção de contexto. Além disso, ele tem a tarefa de chamar os tratadores quando suas condições de contexto são satisfeitas;

- ***Manager Propagation***: tem a função de manter uma infra-estrutura *publish-subscribe* para permitir a propagação de exceções. Esse componente permite informar se a aplicação executará o papel consumidor ou publicador de eventos, de acordo com a API do *middleware publish-subscribe* adotado; e

- ***Exception Propagation***: Esse componente tem a função de enviar solicitações para realizar a propagação de exceções, seguindo a especificação do componente gerenciador. Também é responsável pelo recebimento de exceções propagadas por outros dispositivos e recuperação local dos tratadores associados com a ocorrência da exceção.

O trabalho de Damasceno et al. (2006) traz grandes contribuições com relação ao tratamento de exceção sensível ao contexto. As abstrações e conceitos citados compõem uma importante base de conhecimento para esse domínio. Além disso, o mecanismo proposto atinge seus objetivos no que diz respeito à reusabilidade, flexibilidade e separação de interesses. Apesar das vantagens, o trabalho citado não provê suporte à modelagem e verificação do comportamento excepcional sensível ao contexto como no trabalho de Rocha (2013). Outro ponto

em que há diferença em relação a esta dissertação diz respeito ao nível de abstração das interfaces de programação propostos, visto que a abordagem de Damasceno et al. (2006) busca prover suporte ao tratamento de exceção sensível ao contexto em fases avançadas do processo de desenvolvimento, o que requer o conhecimento de detalhes de implementação da linguagem adjacente e do *middleware* adotado por parte do desenvolvedor. Além disso, a solução proposta não possui suporte à verificação da consistência do comportamento excepcional, estando sujeita à faltas de projeto que só serão capturados em tempo de execução.

3.1.2 Cho e Helal (2011, 2012)

Cho e Helal (2011) propõem uma nova estratégia de definição de exceções com base no conceito de **situações contextuais**. Uma situação contextual é representada por uma sequência de estados formados por um conjunto de informações contextuais que juntas caracterizam a situação. Essa solução possibilita a detecção de exceções contextuais mais complexas que não poderiam ser detectadas com a observação de cada estado de contexto de forma separada e sim através da análise de uma sequência de estados. Um exemplo discutido pelos autores é o de que a captura de contexto comum não consegue detectar o mal funcionamento de um sensor de presença caso algum usuário entre duas vezes no mesmo ambiente sem que a saída tenho sido efetuada. Isso porque o sistema só verifica um estado de cada vez. No exemplo citado, há a necessidade de se observar vários estados em sequência para constatar a falha. Dessa forma, a utilização dessa abordagem traz ganhos na expressividade de eventos contextuais em sistemas sensíveis ao contexto.

Já em (CHO; HELAL, 2012), os autores apresentam um novo método para detecção de exceções semânticas através de uma linguagem de alto nível. Exceções semânticas são exceções que representam padrões lógicos de erros no contexto de natureza temporal, espacial, entre outros tipos de contexto. As exceções semânticas têm características semelhantes às situações contextuais apresentadas pelos autores em seu trabalho anterior (CHO; HELAL, 2011). Cho e Helal (2012) afirmam que utilizar apenas lógica de primeira ordem não garante expressividade suficiente para capturar alguns cenários de exceções específicos. Dessa forma, os mesmos propõem uma linguagem de alto nível com uma série de construtores definidos para solucionar esse problema de expressividade, que são: *duration*, *flap*, *repeat*, *toonear*, *toofar*, *approach*, *toofast*, *stop*. Tais construtores possibilitam a definição de situações excepcionais contextuais de forma

simplificada. Por exemplo, quando uma porta fecha e abre de forma intermitente pode-se usar a expressão (1). Já a expressão em (2) representa a situação na qual o usuário entra em uma sala mais que duas vezes sem sair nenhuma vez (sensor com defeito).

(1) $repeat([portaAberta(d); portaFechada(d)], 10seg])$

(2) $[usuarioEntrou(u); !usuarioSaiu(u); usuarioEntrou(u)]$

Além de proporem a linguagem de alto nível, Cho e Helal (2012) propõem uma linguagem intermediaria que provê um conjunto de formas normais a partir da linguagem de alto nível. Essa linguagem intermediária possibilita otimização e implementação mais eficientes. No trabalho são mostradas as regras de transição (mapeamento) entre as linguagens. A Figura 3.2 mostra o processo de como as descrições de exceções são usadas e traduzidas.

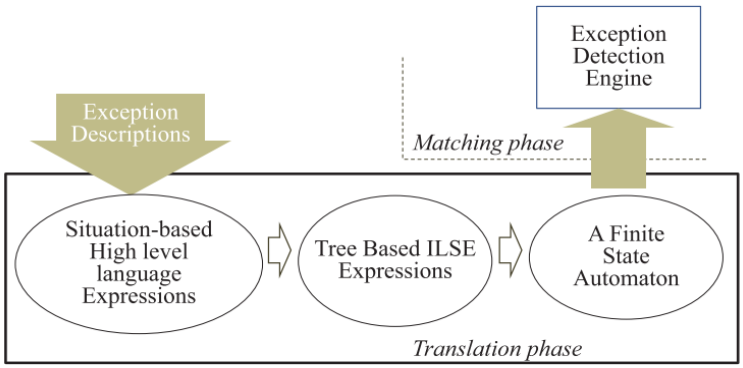


Figura 3.2: Processo de tradução e uso de descrições de exceções situacionais (CHO; HELAL, 2012)

Na fase de tradução, as exceções são descritas na linguagem de alto nível baseada em situações, depois é traduzida para a **Linguagem Intermediária para Exceções Semânticas** (do inglês, *Intermediate Language for Semantic Exceptions - ILSE*). Finalmente, as expressões obtidas na linguagem intermediária são convertidas em um autômato finito, que é usado pelo motor de detecção de exceções na fase de *matching* em tempo de execução.

Apesar da solução proposta por Cho e Helal (2012) ser importante em relação a detecção de exceções contextuais, outros aspectos do tratamento de exceções sensível ao contexto, como a seleção de tratadores, o tratamento de exceções e a retomada do fluxo de controle não foram considerados. Entretanto, o trabalho é interessante por abordar questões importantes relacionadas ao tema desta dissertação como ganhos de expressividade e representação de exceções.

3.1.3 Beder e Araújo (2011)

Nesse trabalho, o autor apresenta um mecanismo a ser incorporado em um *Middleware* (MidSensorNet) baseado no mecanismo *publish/subscribe* para prover interoperabilidade entre serviços em Redes de Sensores sem Fio (RSSF). Inicialmente o autor define o que é contexto e aplicações sensíveis ao contexto. Logo após, busca exemplificar aplicações desse domínio junto às RSSFs. Beder e Araújo (2011) afirmam que a chave para engenheiros de *software* desse domínio é saber como lidar e tratar erros na presença de mudanças contextuais e comunicação assíncrona. Em seguida, diferenciam tratamento de exceção, que é uma técnica de correção de erro por avanço, da replicação, que é uma técnica por retrocesso.

Beder e Araújo (2011) enfatizam que sistemas, abstrações e mecanismos convencionais não são adequados devido à complexidade dos sistemas sensíveis ao contexto. Nesse sentido, são necessárias novas abstrações que possibilitem a delimitação de escopos, a caracterização de contextos excepcionais e o tratamento de exceções concorrentes. Também são necessárias abstrações que permitam aos engenheiros definirem o escopo colaborativo quando a detecção é feita por um membro de serviço composto. Toda a problemática de tratamento de exceções é levantada em conjunto com o domínio de RSSFs.

Segundo os autores, os nós sensores coletam informação contextual e enviam para o nó sumidouro. Essas informações coletadas auxiliam o especialista no processo de tomada de decisão de emergência. Uma extensão possível seria a incorporação de tratamento de exceção sensível ao contexto nos sensores e nós atuadores. Dessa forma, eles poderiam realizar tarefas de tratamentos de exceção em resposta à situações anormais (e.g., uma situação de emergência). Como exemplo, os nós atuadores poderiam realizar ações apropriadas em dispositivos atuadores ao receberem os valores coletados dos nós sensores (BEDER; ARAÚJO, 2011).

Beder e Araújo (2011) sugerem o uso de ontologias e regras de inferência como forma de caracterizar e definir situações excepcionais tais como $temperatura > 70$ °C. Além disso, eles sugerem uma extensão dos mecanismos de escopos baseados em *publish-subscribe* que possibilitem a definição de escopos de tratamento de exceção sensíveis ao contexto (por exemplo, mesma localização geográfica). Já para o tratamento de exceções concorrentes, os autores propõem o uso de algoritmo de agregação de dados (eventos de interesse) que torne mais fácil a implementação de mecanismos de resolução concorrente. Tal algoritmo deve agregar

todos os eventos excepcionais e inferir uma única exceção a partir deles.

O trabalho de Beder e Araújo (2011) é interessante pois aponta a necessidade de se prover suporte à abordagens genéricas relacionadas ao tratamento de exceção sensível ao contexto que possam ser aplicadas em outras áreas como, por exemplo, Redes de Sensores sem Fio. O foco dos autores é apontar questões e sugestões relevantes que devem ser consideradas ao definir um projeto de tratamento de exceção sensível ao contexto. Entretanto, os autores não disponibilizam soluções para as questões levantadas e se limitam a sugerir possíveis soluções para o tratamento de exceção no domínio de redes de sensores sem fio.

3.1.4 Rocha (2013)

Rocha (2013) propõe um método de verificação de modelos do tratamento de exceção sensível ao contexto, o CAEHV, que permite responder questões sobre o comportamento excepcional de um sistema sensível ao contexto através da aplicação da técnica de verificação de modelos. Em seu trabalho, Rocha (2013) define um modelo formal que busca oferecer meios para que os projetistas responsáveis pela especificação e implementação do tratamento de exceções sejam capazes de responder alguns tipos de questionamentos como por exemplo: (i) Existe algum estado possível para o sistema onde um determinado tipo de exceção contextual seja lançada? (ii) Dado um estado do sistema, quais exceções podem ser lançadas? (iii) Considerando a ocorrência de uma exceção contextual em um determinado estado do sistema, é possível que o mesmo consiga tratar a exceção e retornar para um estado normal?

O modelo é baseado no formalismo de *estruturas de kripke* para representar o comportamento evolutivo do sistema e em lógicas temporais para especificação de propriedades a serem verificadas. Essas estruturas são construídas a partir de uma especificação provida pelo projetista. Rocha (2013) realizou um estudo dos principais conceitos relacionados ao tratamento de exceção sensível ao contexto e reuniu um conjunto mínimo necessário para sua especificação que são as exceções contextuais, tratadores e escopos de tratamento. Dada uma especificação no modelo CAEHV, uma estrutura pode ser obtida sob a forma de um grafo orientado, onde os nós são rotulados e representam os possíveis estados do sistema enquanto que as arestas não são rotuladas e indicam simplesmente as transições entre os estados. Dessa forma, é possível representar o comportamento excepcional do sistema, ou fluxo de controle excepcional (ou anormal), como uma sequência de estados e transições.

Além de definir o modelo formal, Rocha (2013) define em seu trabalho um conjunto de propriedades comportamentais que garantem a consistência do modelo e que, se violadas, ajudam a detectar determinados tipos de faltas de projeto. Em sua tese, foram catalogadas cinco propriedades comportamentais, a saber: **progresso de detecção** (determina que para cada estado da estrutura de kripke do contexto deve existir pelo menos um estado onde cada exceção contextual é detectada), **progresso de captura** (estabelece que para cada exceção de contexto levantada deve existir pelo menos um caso de tratamento habilitado a capturar aquela exceção), **progresso de tratador** (determina que para cada estado do contexto onde uma exceção contextual é levantada deve existir pelo menos um destes estados onde cada caso de tratamento é selecionado para tratar aquela exceção), **estabilidade de tratamento** (determina que para cada exceção tratada, o estado de retomada do fluxo de controle não deve ser um estado onde a mesma exceção é levantada) e **alcançabilidade** (estabelece que para cada exceção contextual levantada e capturada sempre é possível executar todas as medidas de tratamento e fazer com que o fluxo de controle normal seja retomado).

Além de fornecer o método, Rocha (2013) também propõe em sua tese uma ferramenta para automatização do modelo, a JCAEHV (figura 3.3). A ferramenta utiliza como entrada uma especificação composta por proposições contextuais, restrições sobre as proposições, exceções contextuais, tratadores e escopos, que são fornecidos através de uma API Java e mapeadas para um solucionador de problemas baseado no paradigma de programação por restrições[1], o ChocoSolver, [2](*framework*) para gerar a *estrutura de kripke* do sistema. Na etapa seguinte, as *estruturas de kripke* são utilizadas em conjunto com as propriedades comportamentais especificadas em lógica temporal como entrada para uma biblioteca de verificação de modelos, que responde se as propriedades são satisfeitas ou não para aquela *estrutura de kripke*.

O trabalho de Rocha (2013) é interessante pois, ao prover o método CAEHV, abriu-se caminho para a modelagem do comportamento excepcional sensível ao contexto em sistemas ubíquos. Além disso, ao fornecer uma ferramenta para modelagem, o autor facilita a utilização e verificação do método, tornando sua aplicação viável. Apesar das vantagens, o trabalho apresenta algumas limitações. Segundo o autor, duas estratégias existentes na literatura para tratamento de exceção concorrente foram implementadas no JCAEHV, porém nenhuma análise

[1]Nesse paradigma é possível definir um problema de forma declarativa fornecendo variáveis, o domínio dessas variáveis e restrições sobre as mesmas, e é resolvido como um problema de satisfação booleana. Mais detalhes podem ser encontrados em (ROCHA, 2013)

[2]www.emn.fr/z-info/choco-solver/

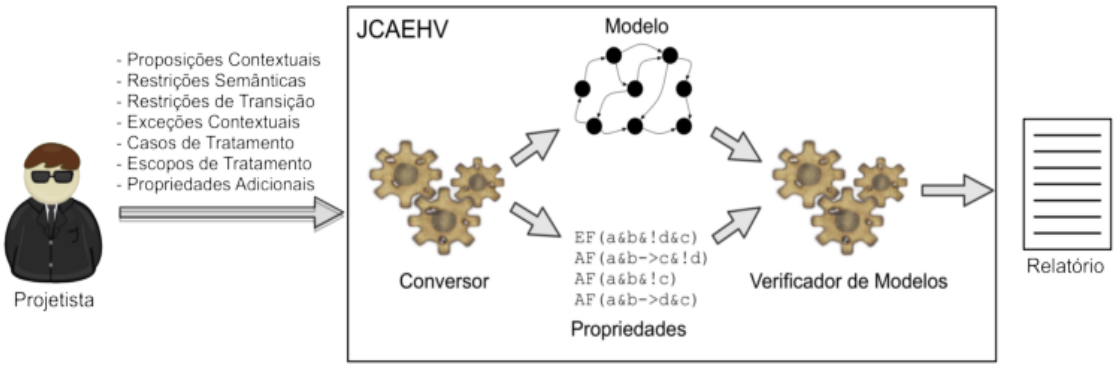


Figura 3.3: Visão geral da ferramenta JCAEHV (ROCHA, 2013)

mais aprofundada foi conduzida. Além disso, não há suporte à propagação de exceções que é um conceito pertinente nesse domínio. Por fim, não há suporte ou integração com nenhum mecanismo de tratamento de exceções que permita aos desenvolvedores construir aplicações mais robustas.

Além disso, Rocha (2013) estabelece, ao final de sua tese, que, embora o JCAEHV provenha uma API para que os projetistas construam seus modelos, lidar com as abstrações do CAEHV nesse nível de granularidade pode fazer com que os projetistas percam o interesse em utilizar o método. Segundo ele, um possível trabalho futuro consiste no desenvolvimento de uma DSL que permita aumentar o nível de abstração na atividade de modelagem. Um outro ponto de melhoria seria a criação de alguma metáfora visual para exibir o modelo de comportamento e informar ao projetista a violação de propriedades. Essas duas questões são objeto de estudo desta dissertação.

3.2 Linguagens Específicas de Domínio para Aplicações Sensíveis ao Contexto

Não foram encontradas na literatura trabalhos que envolvessem os temas de modelagem do tratamento de exceção sensível ao contexto e linguagens específicas de domínio em conjunto. Porém, foram encontrados diversos trabalhos que apresentam abordagens específicas de domínio para modelagem de sistemas sensíveis ao contexto e que de alguma forma incorporam alguns detalhes de tratamento de exceção. Tais abordagens são interessantes pois possibilitam compreender o estado da arte em se tratando de DSLs aplicadas a esse domínio. A seguir, são apresentados alguns desses trabalhos.

3.2.1 MLContext

Em (HOYOS; GARCÍA-MOLINA; BOTÍA, 2010), é proposta uma DSL textual para modelagem de contexto, a MLContext[3]. O alto nível de abstração da linguagem permite especificar modelos de contexto de forma mais clara e mais simplificada do que usando outras notações. Como prova de conceito, foram geradas ontologias e código Java para um *middleware* a partir das especificações de modelos MLContext. De acordo com (HOYOS; GARCÍA-MOLINA; BOTÍA, 2010), após extensa análise do domínio de sistemas sensíveis ao contexto, foram identificados os principais requisitos para a linguagem MLContext. Os requisitos genéricos para linguagens de domínio específico são *alto nível de abstração*, *independência de plataforma*, *independência do domínio da aplicação* e *usabilidade*. Já os requisitos específicos do domínio são *tipos de contexto*, *mecanismos temporais, de qualidade e de rastreabilidade*, e *reuso de modelo*. Tais requisitos são descritos a seguir:

- **Alto nível de abstração**: A linguagem deve prover alto nível de abstração por meios de construtores que são próximos dos conceitos do domínio (e.g., entidade, contexto ou tipo de contexto). Isso deve encorajar usuários que não são desenvolvedores a participarem da modelagem de contexto;

- **Independência de plataforma**: A linguagem deve permitir a criação de modelos independentes da plataforma e sem a necessidade de que os usuários forneçam detalhes de implementação;

- **Independência do domínio da aplicação**: Como o contexto pode ter muitas similaridades em diferentes aplicações sensíveis ao contexto, MLContext deve permitir que o contexto seja modelado independentemente do domínio sensível ao contexto;

- **Tipos de contexto**: A linguagem deve dar suporte à habilidade de modelar tipos de contexto, tais como contexto físico, social ou computacional, porque são mais próximos da realidade. Mais ainda, separar a informação em múltiplos contextos deve ser explorado de modo a melhorar a eficiência de armazenamento e dos métodos de busca;

- **Mecanismos temporais, de qualidade e de rastreabiliadade**: A linguagem deve prover um mecanismo de rastreabilidade com suporte a histórico de informações, um mecanismo

[3]http://www.modelum.es/MLContext/features.htm

para medir a qualidade da informação utilizada e deve dar suporte à informação temporal estática e dinâmica;

- **Reuso de modelo**: A linguagem deve promover o reuso de modelos em diferentes aplicações utilizando o mesmo contexto; e
- **Usabilidade**: Finalmente, a linguagem deve ser fácil e intuitiva para o usuário.

A modelagem utilizando MLContext é centralizada no conceito de *entidades*. Na especificação do contexto, são modeladas instâncias específicas de cada entidade. Também é possível definir os tipos de contexto mais comuns: *físico*, *ambiente*, *computacional*, *pessoal*, *social* e *tarefa*. Além disso, a linguagem permite expressar as fontes de contexto e se uma determinada informação é estática ou dinâmica.

A definição da linguagem textual envolve a criação de ferramentas necessárias para suporte aos usuários da linguagem. As ferramentas básicas são um editor para criar especificações da linguagem, um parser para extrair modelos da especificação e um gerador de código que transforma os modelos em artefatos de software. Para tanto, foi utilizada a ferramenta TCS[4] que é um componente do projeto GMT da plataforma Eclipse que permite a especificação de sintaxes concretas textuais para DSLs ao anexar informação sintática (e.g., palavras-chaves) ao metamodelo das sintaxes abstratas. Dessa forma, o TCS usa especificação textual para gerar: (i) um editor Eclipse, com marcação de sintaxe, *outline* e *hiperlinks*; (ii) um *parser* em Java que utiliza o código fonte do modelo expresso em um sintaxe concreta textual e gera um modelo de acordo com o metamodelo da DSL; e (iii) um extrator que faz a operação inversa e gera especificações textuais a partir dos modelos. Além de permitir a especificação textual, a MLContext também permite a geração de ontologias baseadas em OWL-DL para a base de conhecimento do sistema, de classes Java dos produtores de informações contextuais e um esqueleto java do programa principal que inicializa todas as instâncias de contexto e os produtores.

O trabalho citado tem relação direta com esta proposta, pois provê abstrações de alto nível no domínio de sistemas sensíveis ao contexto. O conhecimento relacionado ao projeto e implementação de linguagens específicas de domínio foi muito importante para o desenvolvimento desta dissertação. De toda forma, o foco da linguagem MLContext é a modelagem estrutural do contexto e não a modelagem do tratamento de exceção sensível ao contexto. Além

[4]http://www.eclipse.org/gmt/tcs/

disso, o foco da DSL é na geração de ontologias e código fonte, e não em validar a especificação textual com relação a sua consistência, seja ela de propriedades ou boa formação (a menos da própria gramática da linguagem);

3.2.2 PervML

A Linguagem de Modelagem Pervasiva (do inglês, *Pervasive Modeling Language - PervML*) é uma abordagem específica de domínio com foco na modelagem de sistemas pervasivos (SERRAL; VALDERAS; PELECHANO, 2010). Sua sintaxe é baseada na notação UML 2.0[5]. A linguagem fornece um conjunto de primitivas conceituais que permitem a descrição do sistema, independentemente da tecnologia. Além disso, PervML define um método de desenvolvimento, fornece suporte de ferramentas e utiliza uma *engine* de transformação para traduzir as especificações para uma ontologia OWL. A linguagem consiste de diferentes visões e modelos que compõem um Modelo Independente de Plataforma (do inglês *Platform Independent Model - PIM* como pode ser visto na figura 3.4.

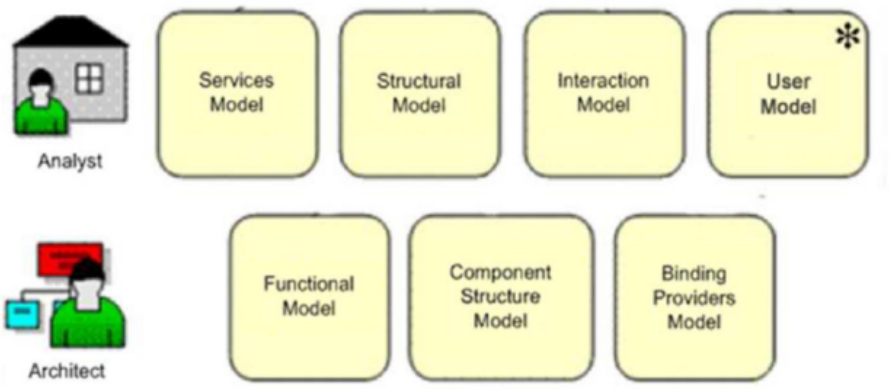


Figura 3.4: Modelos fornecidos na linguagem PervML (SERRAL; VALDERAS; PELECHANO, 2010)

Por ser baseado na UML 2.0, PervML fornece vários tipos de modelos baseados em diagramas de classes, diagramas de atividades e diagramas de componentes. As funcionalidades abstratas e os requisitos do sistema são especificados na **visão do analista** (*Analyst*) que é composto pelos modelos de serviço, estrutural, de interação e de usuário. Já a **visão do arquiteto** (*Architect*) recebe como entrada a especificação escrita na visão do analista e enriquece o mesmo ao descrever como as funcionalidades serão implementadas tanto a nível de *software*

[5]http://www.omg.org/spec/UML/2.0/

quanto de *hardware*. Essa visão é composta pelos modelos funcional, de estrutura dos componentes e de ligação dos provedores. Diagramas de transição de estados da UML são usados para especificar o comportamento dos diferentes componentes. No final do processo, o PIM é transformado em código Java que pode rodar em um *middleware* SOA.

O método proposto provê três elementos: uma linguagem específica de domínio para modelagem de sistemas pervasivos, um motor de transformação que traduz modelos PervML para código Java e, uma ontologia que traduz os modelos PervML em especificações OWL. O processo é dividido em duas etapas: a fase de desenvolvimento e fase de implantação. A Figura 3.5 ilustra de forma geral os passos da fase de desenvolvimento de um sistema pervasivo utilizando o método proposto. A partir das especificações, são gerados código Java e especificações OWL. Além disso, é preciso implantar os os *drivers* que gerenciam os dispositivos e sistemas que devem ser implementados. A fase de implantação é responsável por prover todo o código do sistema pervasivo sensível ao contexto: o código Java que implementa as funcionalidades do sistema e a especificação OWL para adaptação.

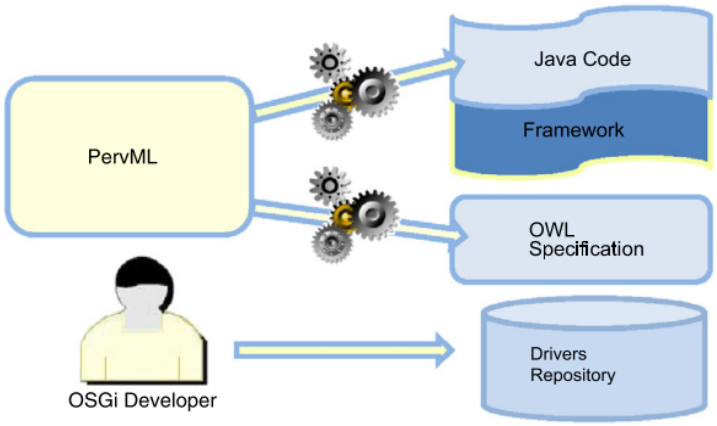


Figura 3.5: Fase de desenvolvimento (SERRAL; VALDERAS; PELECHANO, 2010)

O trabalho citado é importante pois provê um método praticamente completo para modelagem e implementação de um sistema pervasivo sensível ao contexto. Uma linguagem de modelagem é fornecida juntamente com vários diagramas que facilitam o trabalho do projetista desse tipo de sistema. Apesar das vantagens, o método deixa a desejar por não fornecer suporte a modelos formais que possam garantir maior robustez na especificação do sistema. Além disso, não há o enfoque devido ao tratamento de exceção nesse tipo de sistema, o que pode acarretar uma menor capacidade de tolerância a faltas.

3.2.3 DiaSpec

DiaSpec é uma abordagem específica de domínio para dar suporte ao desenvolvimento de sistemas pervasivos (CASSOU et al., 2009, 2012). Diaspec fornece uma linguagem declarativa que define uma taxonomia dos tipos de módulos relevantes para uma dada área no domínio de sistemas pervasivos. Além disso, provê uma linguagem para descrição de arquiteturas com suporte a todo o ciclo de desenvolvimento: programação, simulação e execução. A Figura 3.6 apresenta o estilo arquitetural adotado pelo DiaSpec. Nele, é possível destacar o papel central das entidades de contexto que são utilizadas para representar o contexto que deve ser capturado do ambiente de computação pervasiva. A partir dessa captura, os dados sensoriais são fornecidos diretamente para os controladores que, por sua vez, definem ações sobre essas entidades que irão refletir no ambiente de computação pervasiva.

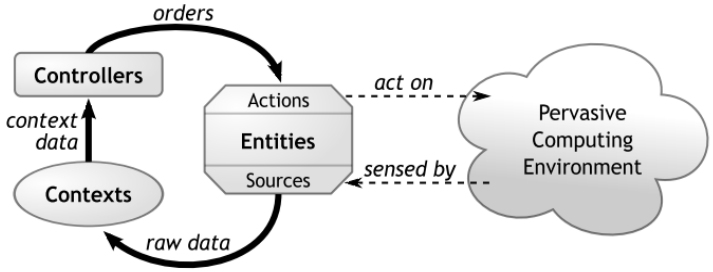


Figura 3.6: Arquitetura do DiaSpec (CASSOU et al., 2009)

Uma outra importante contribuição desse trabalho é a definição de um modelo de tratamento de exceção. Nesse modelo, os erros são caracterizados utilizando uma taxonomia definida para ambientes pervasivos. Além disso, é possível definir exceções tanto a nível de aplicação quanto a nível de sistema. Tais exceções são transformadas posteriormente em exceções na linguagem Java.

O trabalho citado é importante, pois fornece uma infraestrutura para suporte ao projeto e implementação de sistemas pervasivos através de uma abordagem específica de domínio. Os autores obtiveram uma recente publicação em uma conferência importante(CASSOU et al., 2012), o que pode representar indícios da relevância do tema para a comunidade acadêmica.

O foco principal do trabalho é prover uma arquitetura específica de domínio para modelagem de sistemas pervasivos, incluindo suporte à tratamento de exceção, geração de código fonte em Java e simulação de ambientes pervasivos. Apesar das vantagens, o trabalho não

fornece suporte à análise formal do comportamento do sistema nem do modelo de tratamento de exceção, entretanto fornece subsídios importantes para construção de linguagens de domínio específico.

3.3 Análise Comparativa

Dos trabalhos citados anteriormente, três abordagens trazem as contribuições mais recentes em relação ao tratamento de exceção sensível ao contexto (DAMASCENO et al., 2006; BEDER; ARAÚJO, 2011; CHO; HELAL, 2012). Elas fornecem conceitos e abstrações importantes para esse domínio, como a definição de mecanismos para tratamento de exceção sensível ao contexto (DAMASCENO et al., 2006), a detecção de exceções semânticas (CHO; HELAL, 2012) e a aplicação do tratamento de exceção sensível ao contexto em outros domínios (BEDER; ARAÚJO, 2011). O trabalho de Cho e Helal (2012) tem um papel importante pois fornece uma alternativa de mais alto nível para detecção de exceções complexas baseadas em situações. Entretanto, esses trabalhos não buscam fornecer técnicas específicas para modelagem do tratamento de exceção nem suporte à verificação da consistência dos modelos.

Já o trabalho de Rocha (2013) é o único que fornece um método pra verificação de modelos do tratamento de exceção sensível ao contexto, utilizando abstrações e conceitos pertinentes a esse domínio. Porém, apesar de fornecer um método para modelagem e análise, a abordagem de Rocha (2013) deixa a desejar ao não prover uma interface de alto nível para especificação do modelo e por consequência, pode dificultar a aceitação do método por parte dos projetistas que são especialistas no domínio.

Os demais trabalhos relacionados neste capítulo são abordagens específicas de domínio para sistemas sensíveis ao contexto e fornecem alternativas para modelagem de sistemas, bem como suporte de ferramentas (SERRAL; VALDERAS; PELECHANO, 2010; CASSOU et al., 2009; HOYOS; GARCÍA-MOLINA; BOTÍA, 2010). Tais abordagens fornecem alto nível de abstração no domínio de sistemas sensíveis ao contexto e suporte de ferramentas bastante adequado, o que facilita o trabalho dos projetistas e garante uma maior produtividade nas tarefas de modelagem. Entretanto, esses trabalhos não fornecem alternativas robustas para modelagem do tratamento de exceção. De qualquer forma, tais trabalhos são importantes pois provêm contribuições valiosas para o desenvolvimento de uma alternativa de alto nível para modelagem do tratamento de exceção sensível ao contexto.

3.4 Sumário

Neste capítulo foram apresentados os trabalhos relacionados a esta pesquisa. Para identificar tais trabalhos, foram executadas buscas nas principais fontes de pesquisa por abordagens que tratassem do tratamento de exceção sensível ao contexto ou de abordagens de domínio específico relacionadas ao tratamento de exceção ou sistemas sensíveis ao contexto. Inicialmente foram mostrados alguns trabalhos relacionados ao tratamento de exceção sensível ao contexto. Em seguida foram elencadas abordagens de domínio específico para sistemas sensíveis ao contexto. Por fim, foi apresentada uma análise comparativa desses trabalhos.

4 A LINGUAGEM CATCHML

Neste capítulo é apresentada a proposta desta dissertação ficando na metodologia utilizada na definição e desenvolvimento da linguagem CatchML. Essa linguagem é proposta nesta dissertação como alternativa para modelagem do tratamento de exceção sensível ao contexto, com o objetivo de prover um ambiente de suporte para projetistas através de uma interface de alto nível. Tal ambiente é integrado com a ferramenta JCAEHV, possibilitando a verificação automática dos modelos do tratamento de exceção sensível ao contexto.

Inicialmente, na Seção 4.1 são apresentados uma visão geral sobre a linguagem CatchML e o ambiente de desenvolvimento proposto. Já na Seção 4.2 são mostradas as características principais do tratamento de exceção sensível ao contexto que serviram de base para a definição da sintddaxe abstrata da linguagem. Na Seção 4.3 são mostradas as sintaxes abstrata e concreta da linguagem. Na Seção 4.4 são mostrados aspectos referentes à implementação do ambiente de desenvolvimento integrado proposto em forma de *plugin* do Eclipse. Além disso, são apresentados detalhes da integração com a ferramenta JCAEHV, que é o componente responsável por verificar a consistência do modelo em relação à determinadas propriedades. Por fim, a Seção 5.5 encerra o capítulo com as considerações finais sobre a linguagem.

4.1 Visão geral da proposta

Conforme mencionado no Capítulo 1, o principal objetivo desta dissertação é prover uma linguagem específica de domínio para modelagem e verificação do tratamento de exceção sensível ao contexto. Essa linguagem deve prover uma interface de especificação com abstrações de alto nível e que seja viável em termos do esforço do projetista. Neste trabalho foram investigadas as abstrações apropriadas para modelar o tratamento de exceção sensível ao contexto. Tais abstrações foram obtidas através de uma análise das abordagens relacionadas a esse domínio. Além disso, é preciso fornecer os módulos que representam as entidades do sistema, como esses módulos se relacionam e qual o modelo de interação desses módulos. A partir dessas características é que se pode extrair a sintaxe adequada para a DSL.

Na ferramenta JCAEHV, os principais conceitos utilizados para modelar o tratamento de exceção são exceções contextuais, casos de tratamento e escopos de tratamento. Além

das abstrações relativas ao tratamento de exceção, há também abstrações referentes às informações de contexto que são as variáveis (ou proposições) de contexto, as expressões de contexto e as restrições (semânticas ou de transição) sobre as variáveis. Nesse sentido, para que se possa fazer uma conversão automática do modelo especificado através da DSL, é preciso fornecer abstrações que possam ser mapeadas diretamente para esses conceitos. O modelo obtido a partir da especificação serve como entrada para o mecanismo de verificação de propriedades presente na ferramenta JCAEHV. Em tempo de execução, é possível executar o verificador de forma automática em busca de faltas de projeto presentes na especificação.

De forma a expressar o que foi dito anteriormente, uma visão geral da proposta é fornecida na Figura 4.1. Inicialmente, é necessário que o projetista especifique o modelo de tratamento de exceção sensível ao contexto do sistema utilizando a notação fornecida pela linguagem CatchML. Durante esse processo, o projetista pode fazer chamadas ao verificador de modelos utilizando a interface de desenvolvimento integrado. A cada nova chamada, a especificação do modelo é interpretada para a ferramenta JCAEHV e uma verificação é realizada. Ao final de cada análise, o relatório reportado pelo verificador é interpretado e as faltas são apresentadas diretamente no código-fonte, facilitando o entendimento e correção do modelo.

Na seções seguintes são apresentadas as etapas de desenvolvimento da linguagem a saber: análise do domínio, projeto e implementação da linguagem.

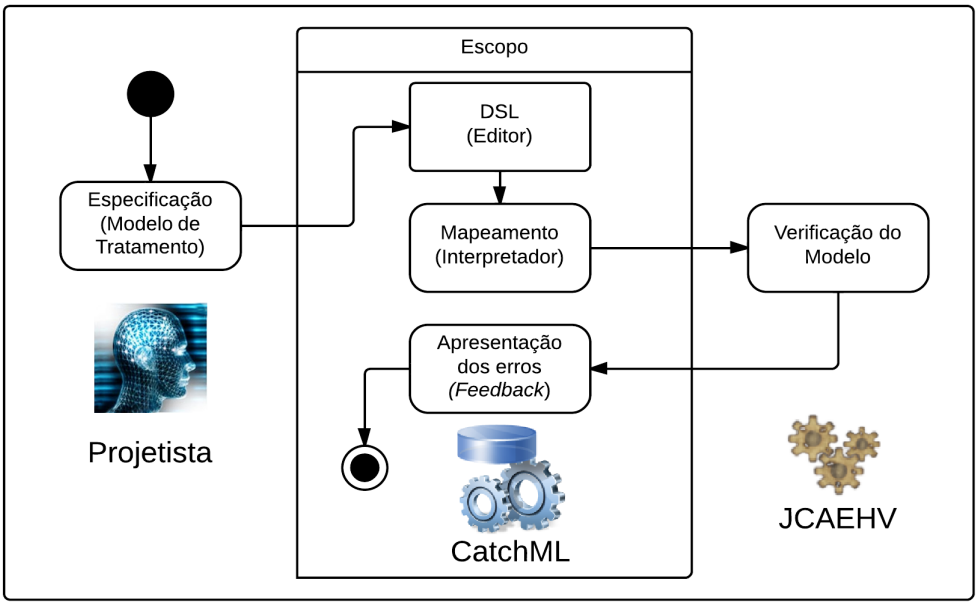


Figura 4.1: Visão Geral da Proposta

4.2 Análise do domínio

As fases principais no desenvolvimento de uma linguagem específica de domínio são discutidas em (VOELTER et al., 2013; MERNIK; HEERING; SLOANE, 2005; DEURSEN; KLINT; VISSER, 2000). No processo geral, a primeira fase é a análise do domínio cujo objetivo é identificar e descrever os conceitos do domínio e suas propriedades.

Neste trabalho, para capturar o conhecimento do domínio, foram analisadas as abordagens mais relevantes no que diz respeito ao tratamento de exceção sensível ao contexto e linguagens específicas no domínio de sistemas sensíveis ao contexto. Ao construir uma DSL para criação de modelos de tratamento de exceção sensível ao contexto, uma das principais dificuldades é a falta de trabalhos que definam explicitamente o processo de modelagem. Foram encontrados diversos trabalhos que definem abstrações para o domínio de tratamento de exceção sensível ao contexto como exceções contextuais, escopos e tratadores, exceções semânticas e outros conceitos. Porém, na maioria dos casos, esses conceitos são aplicados diretamente no código fonte ou APIs, onde o nível de abstração é baixo em relação ao uso em fases anteriores.

Dessa forma, foi necessário relacionar os principais conceitos e abstrações relevantes para o domínio de tratamento de exceção sensível ao contexto e buscar compreender as relações entre os mesmos.

Por exemplo, no trabalho de Damasceno et al. (2006) são elencados diversos conceitos referentes a esse domínio. Damasceno et al. (2006) explorou o conceito chave que é o de **contextos excepcionais**. Segundo eles, um contexto excepcional pode ser visto como um contexto que caracteriza uma situação excepcional de uma entidade, onde *excepcional* pode variar de acordo com os requisitos da aplicação. Um contexto excepcional corresponde então a um conjunto indesejável de condições que podem estar relacionadas a diferentes tipos de informação tais como uma região geográfica específica, a temperatura de uma sala e batimentos cardíacos de um paciente (DAMASCENO et al., 2006).

Essa definição de Damasceno et al. (2006) é semelhante a citada em (KULKARNI; TRIPATHI, 2010), que utiliza o conceito de condição de guarda para definir a exceção. Em ambos os casos, a exceção é definida por uma expressão lógica que relaciona informações de contexto.

Os trabalhos de Beder e Araújo (2011), Cho e Helal (2012) também foram importantes para elencar os conceitos principais relacionados ao domínio de tratamento de exceção sensível ao contexto.

Rocha (2013) relaciona em sua tese alguns conceitos fundamentais relacionados ao tratamento de exceção sensível ao contexto, os quais serviram de base para a definição de um método de verificação de modelos nesse domínio, o CAEHV. Tais conceitos em conjunto com os fundamentos abordados nos trabalhos de Damasceno et al. (2006), Kulkarni e Tripathi (2010), Beder e Araújo (2011), Cho e Helal (2012) serviram de base para a definição da gramática da linguagem CatchML.

No método CAEHV, durante o processo de modelagem do comportamento nesse domínio, existem, basicamente, quatro grandes questões de projeto a serem pensadas: (i) a definição e a detecção de exceções contextuais; (ii) a seleção das medidas de tratamento; (iii) a execução das medidas de tratamento; e (iv) a retomada do fluxo de controle. A primeira questão diz respeito a forma como o contexto pode ser empregado para definir e detectar uma exceção contextual. Já a segunda está relacionada com a escolha da estratégia de tratamento mais adequada para uma exceção contextual levantada em um determinado estado do sistema. Por outro lado, a terceira está relacionada com a sequência de ações de tratamento que provocam alterações no estado do sistema e no contexto. Por último, a quarta questão está relacionada com o estado do sistema após o término do tratamento (ROCHA, 2013).

Essas questões são relacionadas ao processo em si de modelagem do tratamento de exceção contextual. Entretanto, embora a modelagem e verificação do comportamento adaptativo não seja alvo do método, o comportamento excepcional sensível ao contexto é combinado, em tempo de execução, com o comportamento adaptativo para formar o comportamento total do sistema. Por esse motivo, Rocha (2013) estabelece que é importante observar algumas decisões de projeto tomadas para a modelagem do comportamento adaptativo durante a atividade de modelagem do comportamento excepcional sensível ao contexto. Tais decisões incluem definir a forma de abstrair as informações de contexto, eliminar inconsistências semânticas entre as informações de contexto, estabelecer um espaço de estados válidos a serem explorados e evitar que transições inconsistentes entre estados ocorram. Dessa forma, quatro abstrações são adotadas no método CAEHV para lidar com essas questões (ROCHA, 2013):

- **Proposições contextuais**: são proposições contextuais que representam a base de conhecimento sobre a qual são realizadas inferências para derivar informações de contexto de mais alto nível ou caracterizar situações contextuais excepcionais ou não. Em seu trabalho, Rocha (2013) afirma que é pertinente abstrair detalhes sobre a forma como as informações de contexto são obtidas ou a estrutura utilizada para representá-las. Dessa forma, o contexto é representado de forma simbólica na qual cada proposição contextual responde à uma determinada pergunta sobre o contexto podendo ter valores verdadeiro ou falso. Essa representação de contexto também é utilizada no trabalho de Sama et al. (2010) que utiliza fórmulas lógicas compostas por proposições simbólicas e uma máquina de estados finita para representar o comportamento adaptativo de aplicações sensíveis ao contexto. Por exemplo, em um sistema de estacionamento podemos ter a pergunta "O veículo está em movimento?" que pode ser representada pela proposição contextual *inMovement* (que significa "em movimento"). Outro exemplo de pergunta seria "O veículo está no pátio de vagas do estacionamento?" cuja proposição contextual pode ser *atParkPlace* (que significa "no pátio de vagas"). Além das proposições atômicas, é possível que haja questões de mais alto nível que podem ser expressas através de uma fórmula lógica composta por proposições contextuais como, por exemplo, "O veículo está estacionado?" $(\neg inMovement \wedge atParkPlace)$ ou "O veículo está saindo do estacionamento?" $(inMovement \wedge atParkExit)$;

- **Restrições semânticas**: são fórmulas lógicas que representam restrições globais cujo objetivo é reduzir inconsistências semânticas entre proposições contextuais e reduzir o espaço de estados a ser explorado. Por exemplo, se tomarmos um veículo no sistema de estacionamento, pode-se afirmar que o veículo só pode estar em um dos seguintes lugares: na entrada ($atParkEntrance$), no pátio de vagas ($atParkPlace$) ou na saída ($atParkExit$) do estacionamento, não podendo estar em mais de um local simultaneamente. Nessa situação, pode-se inferir uma restrição semântica relacionada à localização do veículo. Desse modo, para o exemplo dado, é possível derivar uma fórmula lógica que garanta a restrição semântica utilizando os operadores lógicos e as proposições contextuais relacionadas;

- **Estados de contexto**: é uma tupla valorada composta por todas as proposições contextuais que satisfaz as fórmulas lógicas definidas nas restrições semânticas. Cada estado de contexto representa um *snapshot* do sistema em um dado momento, ou seja, define os

valores das proposições contextuais em um dado instante; e

- **Restrições de transição**: são restrições definidas entre estados de contexto sequenciais que buscam garantir que sejam atendidas propriedades evolutivas nessas sequências. Para tanto, uma restrição de transição caracteriza a situação de contexto do estado de origem e a restrição que deve ser satisfeita pelo estado de destino da transição. Por exemplo, no sistema de estacionamento, se o veículo está fora do estacionamento em um estado, no próximo estado o veículo só poderá assumir duas possibilidades: permanecer fora ou ir para a entrada. Dessa forma, as restrições garantem que transições que não são possíveis do ponto de vista lógico seja levadas em conta ao avaliar os estados de contexto.

Além dessas quatro abstrações, que estão mais relacionadas com a definição de contexto e de restrições sobre ele, Rocha (2013) apresenta mais três abstrações, que tem maior relação com o tratamento de exceção em si. Tais abstrações estabelecem uma forma padrão para representar os aspectos comportamentais relacionados com a definição e a detecção de contextos excepcionais, o agrupamento, seleção e execução das medidas de tratamento e a retomada do fluxo de controle. Esses conceitos também foram citados em outros trabalhos que abordam o tratamento de exceção sensível ao contexto e são os seguintes:

- **Exceções contextuais**: uma exceção contextual é definida por um nome e uma fórmula lógica utilizada para caracterizar o seu contexto excepcional. A fórmula lógica é composta por proposições contextuais ligadas por conectivos lógicos de primeira ordem. Por exemplo, considerando as proposições contextuais *hasSmoke* (que significa "presença de fumaça") e $isHot$ (que significa "presença de calor"), é possível definir uma exceção contextual de nome $fireException$ cuja fórmula lógica é $hasSmoke \wedge isHot$ para definir uma situação excepcional de incêndio (ROCHA, 2013). Essa definição é semelhante à aplicada por Damasceno et al. (2006) e Kulkarni e Tripathi (2010) com a diferença de que nesses trabalhos não há uma preocupação em se trabalhar exclusivamente com proposições contextuais simbólicas.

- **Casos de tratamento**: definem as diferentes estratégias que podem ser empregadas para tratar uma exceção contextual em função do contexto do sistema (ROCHA, 2013; DAMASCENO et al., 2006). Um caso de tratamento é composto por uma **condição de seleção** e um conjunto de **medidas de tratamento** que são fórmulas lógicas utilizadas

para descrever a situação de contexto esperada após a execução de cada ação (ou bloco de ações) de tratamento de forma sequencial. Por exemplo, no sistema de estacionamento, considere a exceção $fireException$ definida no item anterior e as seguintes proposições contextuais $inMovement$, $isSprinklerOn$, $atParkPlace$, $atParkEntrance$ e $atParkExit$. É possível que haja diferentes casos de tratamento dependendo da situação de contexto do veículo. Em um dos casos, suponha que o veículo está entrando no estacionamento. Dessa forma, a condição de seleção seria $inMovement \wedge atParkEntrance$ e possíveis medidas de tratamento seriam $isSprinklerOn \wedge (\neg atParkEntrance \wedge \neg atParkPlace \wedge \neg atParkExit)$. Isso quer dizer que dado que há uma exceção de incêndio e que o veículo está em uma situação entrando no estacionamento, para que haja o tratamento da exceção, é preciso que o sistema chegue a um estado em que os aspersores estão ligados ($isSprinklerOn$) e que o veículo esteja fora do estacionamento ($\neg atParkEntrance \wedge \neg atParkPlace \wedge \neg atParkExit$); e

- **Escopos de tratamento**: delimitam a atuação dos casos de tratamento e estabelece uma relação de precedência entre eles. Essa relação de precedência é essencial para resolver situações de sobreposição entre condições de seleção de casos de tratamento (i.e., situações em que mais de um caso de tratamento pode ser selecionado num mesmo estado de contexto). Dessa forma, o CAEHV avalia primeiro o caso de tratamento de maior precedência, se este não tiver a sua condição de seleção satisfeita, o próximo caso de tratamento com maior precedência é avaliado, e assim por diante (ROCHA, 2013). Damasceno et al. (2006) apresenta a necessidade de se especificar níveis de escopo de tratamento mais específicos e flexíveis (QUEIROZ FILHO, 2012). Eles identificaram algumas circunstâncias em que o tratamento de exceções poderia envolver diversos dispositivos, dependendo de sua localização ou outras informações de contexto (e.g., servidor ao qual ele está conectado ou grupo de dispositivos ao qual ele pertence). Assim, foi proposto pelos autores quatro níveis de escopo, que são: dispositivo, servidor, localização ou grupos de dispositivos.

O trabalho de Rocha (2013) é importante para o entendimento do domínio de tratamento de exceção sensível ao contexto porque reúne as abstrações elementares que formam uma base concisa para sua especificação. As abstrações relacionadas com a definição de contexto e restrições sobre ele são semelhantes às utilizadas no trabalho de Sama et al. (2010) que

lida com especificação e análise do comportamento adaptativo. Já as abstrações relativas aos aspectos do tratamento de exceção em si são semelhantes às citadas no trabalho de Damasceno et al. (2006) e Kulkarni e Tripathi (2010). Além dessas abstrações elementares, é possível identificar as seguintes abstrações que compõem um conjunto maior de conceitos necessários para uma especificação mais completa do tratamento de exceção sensível ao contexto:

- **Propagação de exceções**: Damasceno et al. (2006) apresentam em seu trabalho a necessidade de se definir um fluxo de propagação de exceções sensível ao contexto. Isso acontece porque as informações de contexto e de escopo são utilizadas para que o mecanismo de tratamento decida quando deve ocorrer a propagação das exceções. Quando uma exceção contextual é levantada no sistema, a mesma deve ser capturada em algum caso de tratamento e, dependendo da especificação, pode ser propagada para um outro escopo ou componente, de acordo com as condições contextuais. A propagação de exceções também tem relação com o conceito de escopos de tratamento em níveis citado anteriormente, pois dependendo da severidade especificada para uma exceção ela deve ser capaz de ser propagada entre os diversos níveis de escopo;

- **Exceções semânticas**: também chamadas de exceções baseadas em situações, essa abordagem é proposta no trabalho de Cho e Helal (2012) e tem como objetivo definir uma linguagem de alto nível para capturar situações contextuais. Devido à sua natureza complexa, essas situações necessitam da observação de eventos em uma sequência de estados de contexto para serem expressas (ao invés de uma simples visão instantânea do contexto). Nesse sentido, são definidas funções que relacionam proposições e variáveis de contexto que caracterizam a ocorrência dessas situações; e

- **Exceções concorrentes**: o termo exceções concorrentes é utilizado para designar a ocorrência simultânea de mais de uma exceção. Tipicamente, quando isso ocorre, existe um mecanismo de resolução que permite selecionar as medidas de tratamento mais adequada face ao conjunto de exceções levantadas (ROCHA, 2013). Na ferramenta JCAEHV, proposta por Rocha (2013), foram implementadas duas estratégias existentes na literatura: uma delas é baseada em árvore de exceções que permite ao projetista especificar situações de contexto em determinados nós da árvore como critério de decisão, e a outra baseada em prioridades pré-definidas pelo projetista de tal forma que a exceção com maior prioridade é a selecionada.

De acordo com Voelter et al. (2013), ao finalizar a análise do domínio é importante saber quais abstrações e notações são mais relevantes para a linguagem. Segundo ele, esse é um dos fatores chave para desenvolver uma DSL. Para esta dissertação, além do conhecimento do domínio, também há a disponibilidade da ferramenta JCAEHV, proposta por Rocha (2013), que provê uma API para especificação do tratamento de exceção sensível ao contexto. Dessa forma, decidiu-se apresentar uma notação tendo como base os conceitos expressos na ferramenta e em seguida definir uma linguagem formal baseada nesses conceitos. O objetivo dessa atividade é retirar o foco da linguagem base de forma a aumentar o nível de abstração e, por conseguinte, tornar a tarefa de modelagem mais simples e produtiva para o projetista.

4.3 Projeto

Um dos objetivos deste trabalho é poder transformar um modelo especificado na DSL em um modelo de especificação do JCAEHV para que seja possível utilizar o mecanismo de verificação de propriedades. Nessa seção, são apresentadas as decisões de projeto em relação à linguagem CatchML tomando como base o conhecimento sobre o domínio de tratamento de exceção sensível ao contexto analisado anteriormente.

4.3.1 Sintaxe Abstrata

A sintaxe abstrata de uma linguagem é uma estrutura de dados que guarda a informação principal em um programa, mas sem nenhum detalhe de notação contidos na sintaxe concreta como palavras-chave e símbolos, comentários e informações de *layout*. Inicialmente foi definida a gramática da linguagem CatchML no formato EBNF utilizando o Xtext[1]. A partir dessa gramática, o Xtext deriva a sintaxe abstrata como uma instância da linguagem de metamodelos *Ecore* que faz parte do *Eclipse Modeling Framework* (EMF)[2]. Como apresentado por Voelter et al. (2013), sintaxe abstrata e metamodelo são tipicamente considerados sinônimos, mesmo que esses termos tenham surgido historicamente de forma diferente. Consequentemente, os formalismos para definir árvores de sintaxe abstrata são conceitualmente similares aos meta metamodelos.

O metamodelo da linguagem CatchML pode ser visto na Figura 4.2. Esse metamo-

[1] A versão completa da gramática gerada pela ferramenta ANTLRworks pode ser vista no Apêndice A
[2] http://www.eclipse.org/modeling/emf

delo descreve as entidades que compõem um conjunto de abstrações suficiente para expressar um modelo de tratamento de exceção sensível ao contexto que possa ser verificado pela ferramenta JCAEHV (alguns componentes relacionados à definição de expressões lógicas foram omitidos).

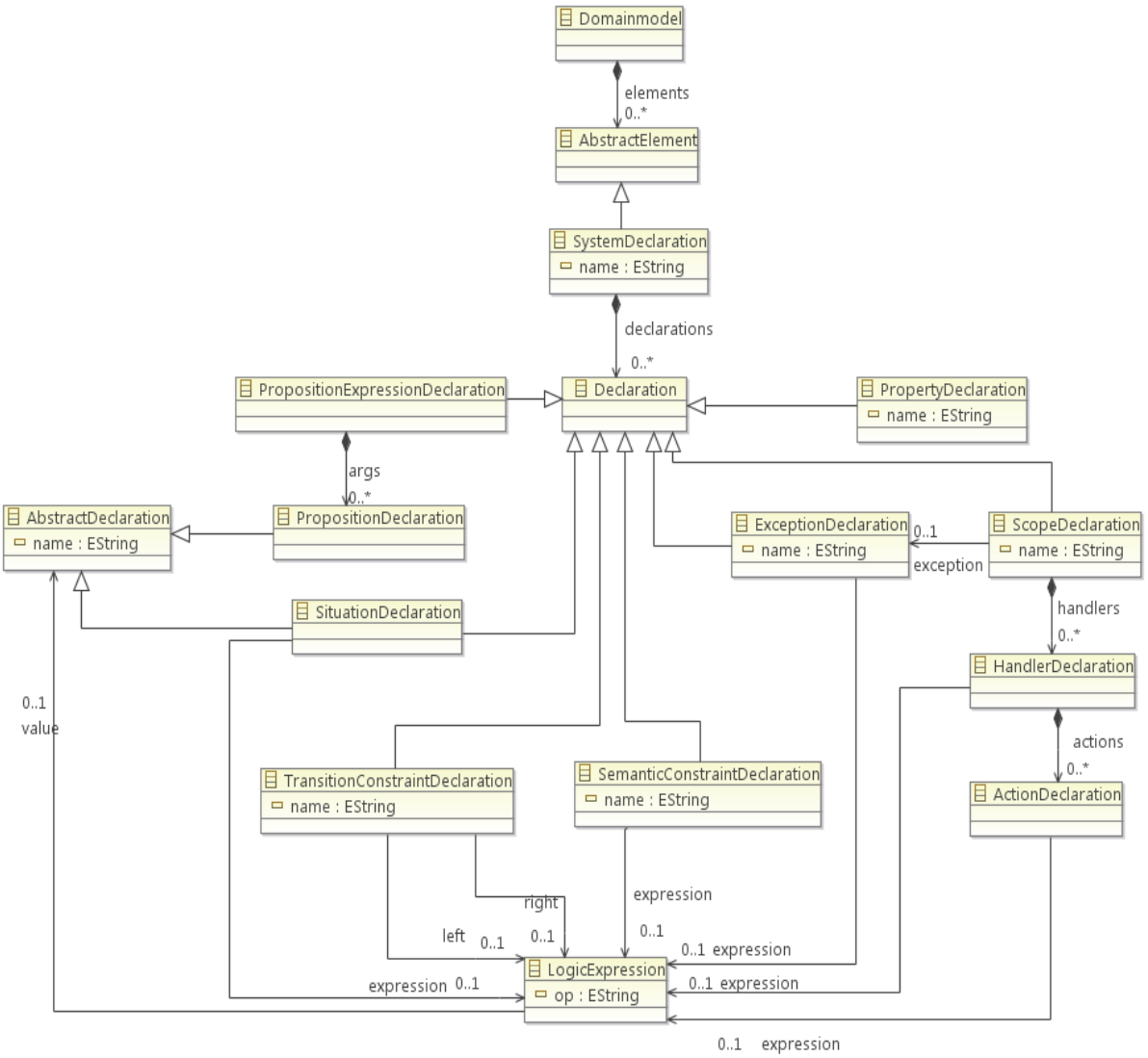


Figura 4.2: Metamodelo da linguagem CatchML

O modelo de domínio na linguagem é composto de **elementos abstratos** (metaclasse *AbstractElement*) que nesta especificação podem ser apenas **sistemas** (metaclasse *SystemDeclaration*). Um sistema define uma aplicação sensível ao contexto e nele deve ser especificado o modelo de tratamento de exceção. Dessa forma, o sistema é composto por **declarações** (metaclasse *Declaration*) que expressam as abstrações necessárias para a modelagem. Inicialmente podem ser definidas as **proposições contextuais** (metaclasse *PropositionDeclaration*)

que compõem a base de conhecimento de contexto. Também é possível definir as **situações contextuais** (metaclasse *SituationDeclaration*) que são expressões de mais alto nível compostas por fórmulas lógicas sobre as proposições contextuais. Existem dois tipos de restrições sobre o contexto com o intuito de reduzir o espaço de estados a ser verificado que são: (i) as **restrições semânticas** (metaclasse *SemanticConstraintDeclaration*) que contém fórmulas lógicas que devem ser satisfeitas durante toda a evolução do sistema; e (ii) as **restrições de transição** (metaclasse *TransitionConstraintDeclaration*) que expressam uma restrição lógica de contexto que deve ser obedecida entre uma sequência de dois estados.

Além das abstrações citadas, que estão mais relacionadas com o contexto, também é possível declarar os conceitos relacionados ao tratamento de exceção propriamente dito. As **exceções contextuais** (metaclasse *ExceptionDeclaration*) definem uma situação contextual que deve ser considerada como excepcional pelo sistema. Já os **escopos de tratamento** (metaclasse *ScopeDeclaration*) definem uma região de tratamento para uma exceção específica e são agregações de **tratadores de exceção** (metaclasse *HandlerDeclaration*) definidos para esta exceção. Os tratadores por sua vez possuem uma lista de **ações de tratamento** (metaclasse *ActionDeclaration*) que devem ser obedecidas a fim de trazer o sistema para o fluxo de controle normal. Por fim, também é possível especificar **propriedades temporais** (metaclasse *PropertyDeclaration*) que definem expressões lógico-temporais que devem ser satisfeitas no modelo de tratamento de exceção.

4.3.2 Sintaxe Concreta

A sintaxe concreta de uma linguagem é exatamente a parte em que o usuário interage para criar os modelos. Na linguagem CatchML, a sintaxe concreta é textual e será apresentada em detalhes a seguir. Essa sintaxe é influenciada pela estrutura de classes, atributos e métodos de linguagens orientadas a objetos como Java. Além disso, os trabalhos de Serral, Valderas e Pelechano (2010), Hoyos, García-Molina e Botía (2010) e Cassou et al. (2009) serviram como base para a definição da sintaxe, pois apresentam abordagens para representação de abstrações no domínio de sistemas sensíveis ao contexto.

Para tornar a apresentação dos conceitos mais simples, decidiu-se utilizar o estacionamento ubíquo (*UbiParking*), que é um exemplo de sistema sensível ao contexto abordado nos trabalhos de Rocha (2013) e Queiroz Filho (2012). O *UbiParking* utiliza informações coletadas

por sensores e informa aos usuários sobre a disponibilidade de vagas livres no estacionamento. Dessa forma, ao utilizar a aplicação em seus dispositivos móveis, os cidadãos podem obter informações sobre a distribuição de vagas livres, podendo fazer reservas e solicitação de rotas apropriadas. Além disso, o estacionamento possui sensores que monitoram a temperatura do estacionamento para controle de sistemas de refrigeração e ventilação. Por fim, ainda há a presença de detectores de fumaça e aspersores controlados automaticamente, para o caso de incêndio (ROCHA, 2013).

Para criar um modelo de tratamento de exceção sensível ao contexto para o *UbiParking* é preciso inicialmente definir o sistema ao qual estamos interessados em modelar. A Listagem 4.1 mostra como isso pode ser feito na linguagem CatchML (linha 1).

Listagem 4.1: Declaração do sistema a ser modelado

```
1 system UbiParking {
```

Em seguida, é necessário especificar as proposições contextuais que formam a base de informações de contexto do sistema. Lidar com essas informações é bastante intuitivo para o projetista e, portanto, é interessante que essas informações possam ser especificadas de maneira simples e direta. Nesse sentido, foi introduzida a palavra-chave **prop** que caracteriza a declaração de uma proposição contextual. Na Listagem 4.2, é mostrado como declarar uma lista de proposições contextuais para o modelo do *UbiParking* (linha 2):

Listagem 4.2: Declaração das proposições contextuais

```
1 system UbiParking {
2         prop atParkPlace, inMovement, atParkEntrance, atParkExit, hasSmoke, hasSpace, isSprinklerOn, isHot;
```

Além das proposições contextuais é possível declarar situações contextuais de mais alto nível através de fórmulas lógicas sobre as proposições contextuais. Uma situação é declarada utilizando a palavra-chave **situation** junto com seu nome. Em seguida, é atribuída uma fórmula lógica que caracteriza essa situação. No caso do *UbiParking* foi introduzida uma situação "Fora do Estacionamento" ($outOfPark$) que pode ser expressa como uma conjunção de negações das proposições que representam posições dentro do estacionamento, a saber $atParkEntrance$, $atParkPlace$ e $atParkExit$. Essa situação será importante para facilitar a especificação de restrições e exceções mais adiante. Sua declaração pode ser vista na Listagem 4.3.

Listagem 4.3: Declaração de situações contextuais

```
3    situation outOfPark := !atParkEntrance && !atParkPlace && !atParkExit;
```

Continuando a especificação do modelo, é necessário definir algumas restrições com vistas a diminuir o espaço de estados a ser avaliado. Para declará-las é preciso utilizar a palavra-chave **sconstraint** (do inglês *semantic constraint*) e atribuir uma fórmula lógica que expressa a restrição. No exemplo do *UbiParking*, é necessário definir uma restrição semântica de natureza espacial que restringe a possibilidade de um veículo estar em dois lugares distintos ao mesmo tempo (no mesmo estado de contexto). Na Listagem 4.4, tal restrição foi especificada com o nome *allPlacesDisjoined* que significa "todos os lugares disjuntos" e uma fórmula lógica lhe foi atribuída. Dessa forma, em todos os estados de contexto do sistema é obrigatório que essa restrição seja satisfeita.

Listagem 4.4: Declaração de restrições semânticas

```
4    sconstraint allPlacesDisjoined := xor(atParkEntrance, atParkPlace, atParkExit, outOfPark);
```

Um outro tipo de restrição, com o objetivo de diminuir mais ainda o espaço de estados, é a restrição de transição. Tal restrição pode ser entendida da seguinte forma: se uma certa condição é satisfeita em um estado de contexto, então uma nova condição contextual também deve ser satisfeita no estado seguinte. Uma restrição desse tipo pode ser declarada utilizando a palavra-chave **tconstraint** (do inglês *transition constraint*). Além disso, para tornar mais intuitiva a declaração dessa restrição, foram introduzidas as palavras-chave **pre**, que representa a condição contextual a ser satisfeita no estado de saída, e **post**, que representa a pós-condição contextual que deve ser satisfeita a fim de restringir de forma semântica os contextos presentes no estado de destino. Na Listagem 4.5, para o exemplo do *UbiParking*, foram declaradas três restrições de transição (linhas 5 a 7), que representam relações espaciais dentro do estacionamento. A restrição $t1$ garante que dado que o veículo está fora do estacionamento em um estado de contexto, ele não poderá estar, no próximo estado de contexto, no pátio de vagas ou na saída, visto que ele precisa antes disso passar pela entrada do estacionamento. As outras restrições $t2$ e $t3$ têm características semelhantes.

Listagem 4.5: Declaração de restrições de transição

```
5    tconstraint t1 := pre outOfPark post !atParkPlace && !atParkExit;
6    tconstraint t2 := pre atParkEntrance post !atParkExit;
7    tconstraint t3 := pre atParkPlace post !atParkEntrance && !outOfPark;
```

Após concluída a etapa mais voltada para definição de características relacionadas ao contexto é necessário especificar as exceções contextuais que compõem o modelo de tratamento de exceção. Uma exceção contextual pode ser definida de forma bastante simples e intuitiva. Basta declarar a palavra-chave **exception** junto com o nome da exceção e atribuir uma condição excepcional de contexto expressa por uma fórmula lógica. Na Listagem 4.6, para o caso do *UbiParking*, foi definida uma exceção de incêndio (*fireException*) que representa um estado excepcional em que se detecta a presença de fumaça (*hasSmoke*) e calor fora do comum (*isHot*) com os aspersores desligados (!*isSprinklerOn*).

Listagem 4.6: Declaração de exceções contextuais

```
8   exception fireException := isHot && hasSmoke && !isSprinklerOn;
```

Além das exceções contextuais, é preciso definir os escopos de tratamento que tratam tais exceções. A definição de escopo é feita utilizando a palavra-chave **scope** junto com o nome do escopo e atribuindo a esse escopo uma exceção a ser tratada através da palavra-chave **handle** junto com o nome da exceção contextual. Na Listagem 4.7, para o exemplo do *UbiParking*, temos a definição do escopo para tratamento da exceção de incêndio (*fireScope*).

Listagem 4.7: Declaração de escopos

```
9   scope fireScope handle (fireException) {
10      case (inMovement && atParkPlace) {
11          do isSprinklerOn && (inMovement && atParkExit);
12          do isSprinklerOn && (inMovement && outOfPark);
13      }
14      case (inMovement && atParkExit) {
15          do isSprinklerOn && (inMovement && outOfPark);
16      }
17  }
```

Dentro de um escopo de tratamento, é possível definir vários casos de tratamento para uma determinada exceção contextual de acordo com uma condição de seleção. Para tanto, foi introduzida a palavra-chave **case** seguida dessa condição tornando simples a definição de novos casos de tratamento. Entretanto, ao declarar um caso de tratamento também é necessário definir as ações de tratamento que devem trazer o sistema para o fluxo de controle normal. Na linguagem CatchML, foi utilizada a palavra-chave **do** junto com a condição de contexto que deve ser satisfeita no próximo estado como uma metáfora para representar o resultado de uma ação de tratamento. No *UbiParking*, para a exceção de incêndio, foram modelados dois casos de tratamento. Na Listagem 4.7, é mostrada a declaração do primeiro caso (linha 10). O mesmo

representa o estado de contexto em que, dado que o incêndio foi detectado, o veículo está em movimento (*inMovement*) no pátio de vagas (*atParkPlace*). As ações de tratamento (linhas 11 e 12) definem estados sequenciais em que os aspersores são ligados (*isSprinklerOn*) e o veículo se dirige para a saída (*atParkExit*) e, em seguida, para fora do estacionamento (*outOfPark*).

Ainda na Listagem 4.7, é mostrado o segundo caso de tratamento (linha 14) que define o tratamento a ser realizado para o caso de o veículo estar na saída do estacionamento (*atParkExit*) no estado de contexto em que o incêndio é detectado. Nesse caso, é necessário realizar apenas uma ação de tratamento (linha 15): ligar os aspersores e levar o veículo para fora do estacionamento em segurança.

No exemplo do *UbiParking* apresentado por Rocha (2013) também foi definida uma exceção de vaga ocupada (*noSpaceException*). Na Listagem 4.8, é mostrada a declaração dessa exceção (linha 18) juntamente com seu escopo de tratamento (linhas 19 a 23), mas a explicação dos detalhes são omitidos para que não fique cansativo e repetitivo.

Listagem 4.8: Declaração da exceção de falta de vaga e seu escopo de tratamento

```
18  exception noSpaceException := inMovement && atParkPlace && !hasSpace;
19  scope noSpaceScope handle (noSpaceException) {
20      case (inMovement && atParkPlace) {
21          do inMovement && atParkExit;
22      }
23  }
```

A última abstração fornecida na linguagem é a que permite a definição de propriedades em lógica temporal. Para tanto, foi introduzida a palavra-chave **spec** que deve ser fornecida junto com o nome da propriedade e uma fórmula em lógica temporal sobre as proposições e situações contextuais. Na Listagem 4.9, para o exemplo do *UbiParking*, foi especificada a propriedade que verifica a existência de um evento de exceção de incêndio (*hasFireException*) na árvore de comportamento do sistema.

Listagem 4.9: Declaração de propriedades temporais

```
24  spec hasFireException := EX(((isHot && hasSmoke) && (!isSprinklerOn)));
```

4.4 Implementação

A definição de uma DSL textual envolve a criação de ferramentas necessárias para dar suporte aos usuários da linguagem (HOYOS; GARCÍA-MOLINA; BOTÍA, 2010). As ferramentas básicas são um editor de texto para criação das especificações da DSL, um *parser* para extrair modelos da especificação da DSL e um gerador de código ou transformador que é apto a transformar modelos da DSL em artefatos de *software*. Atualmente, existem diversas ferramentas e frameworks que geram automaticamente grande parte da infraestrutura de *software* necessária para se ter um ambiente integrado de desenvolvimento que possa dar suporte à especificação. Hoyos, García-Molina e Botía (2010) cita em seu trabalho ferramentas que geram essa infraestrutura a partir de metamodelos (e.g. TCS[3]) ou a partir de gramáticas (e.g. Xtext). Voelter et al. (2013) cita em seu livro que o Xtext, juntamente com Spoofax e MPS, compõem o que há de mais avançado em se tratando de *frameworks* para construção de linguagens. Optou-se por utilizar o Xtext por ter uma boa integração com o ambiente Eclipse, boa documentação e uma grande comunidade de usuários e, além disso, ser escrito em Java que é uma linguagem bastante madura e utilizada em alguns trabalhos relacionados como (SAMA et al., 2010; QUEIROZ FILHO, 2012; ROCHA, 2013).

4.4.1 Desenvolvimento da Linguagem

No Xtext, a sintaxe é especificada utilizando a notação EBNF, uma coleção de produções que são tipicamente chamadas de regras de interpretação (do inglês *parser rules*). Essas regras especificam a sintaxe concreta de um elemento do programa, bem como o seu mapeamento para a sintaxe abstrata. O Xtext gera a sintaxe abstrata representada em um metamodelo *Ecore* a partir da gramática. Na Listagem 4.10, temos a definição da regra *SituationDeclaration*:

Listagem 4.10: Definição da regra *SituationDeclaration* no Xtext

```
1 SituationDeclaration:
2         'situation' name = ID ':=' expression = LogicExpression ';'
3 ;
```

A regra é definida pelo nome (*SituationDeclaration*, no exemplo acima), em seguida dois pontos e, após isso, o corpo da regra. O corpo define a estrutura sintática do conceito da linguagem definido por uma regra. Nesse exemplo, é esperada a palavra-chave **situation** se-

[3] http://www.eclipse.org/gmt/tcs/

guida de um **ID** (identificador). O ID é uma regra terminal que deve receber uma string em um formato adequado. Essa regra é herdada de uma gramática genérica estendida pelas gramáticas especificadas no Xtext. Após esse ID, é esperado o símbolo de atribuição ":=" seguido por uma fórmula lógica (*expression*) que deve ser definida na regra *LogicExpression*. Ao final, terminamos a regra com um ponto e vírgula. Além de expressar a sintaxe concreta, a regra determina o mapeamento para a sintaxe abstrata. O nome da regra é o nome da metaclasse derivada (como observamos no metamodelo apresentado no início desse capítulo), a atribuição $name = ID$ define uma propriedade *name* que guarda o valor do ID e, por fim, um relacionamento de nome *expression* é definido com a metaclasse *LogicExpression*.

As demais regras são definidas de forma semelhante, com exceção das expressões lógicas, onde houve a necessidade de se definir toda a sintaxe de expressões de lógica proposicional e de lógica temporal, seus operadores e regras de precedência. As expressões lógicas "comuns" dão suporte a operações de conjunção, disjunção, negação e consequência lógica. Além dessas operações básicas, foi adicionado suporte à funções derivadas como *xor*, *nand* e *nor*. Já nas expressões em lógica temporal foram adicionados os operadores temporais específicos dessa lógica.

No Xtext, ao compilar a gramática, é gerada a infraestrutura com *parser* e editor de texto, na qual é possível iniciar uma nova instância da IDE Eclipse disponibilizando um editor da linguagem com (Figura 4.3). Dessa forma, foi possível construir a gramática da linguagem de forma iterativa e prototipada, sempre adicionando novos construtores e abstrações e verificando sua viabilidade em tempo de execução. Além da verificação *default* em relação a sintaxe concreta da linguagem, o Xtext permite a criação de métodos de validação para checar alguma inconsistência semântica nas especificações. Nesse sentido, foram adicionados métodos de validação para verificar a duplicidade de identificadores. Por exemplo, um dos métodos verifica o modelo de forma automática se há a presença de duas proposições contextuais com o mesmo nome. Em caso positivo, uma mensagem de erro é gerada e sinalizada para o usuário na tela de edição da linguagem.

4.4.2 Integração com o JCAEHV

O objetivo principal desta dissertação é prover uma alternativa de alto nível para modelagem do tratamento de exceção sensível ao contexto. Além disso, também é um dos obje-

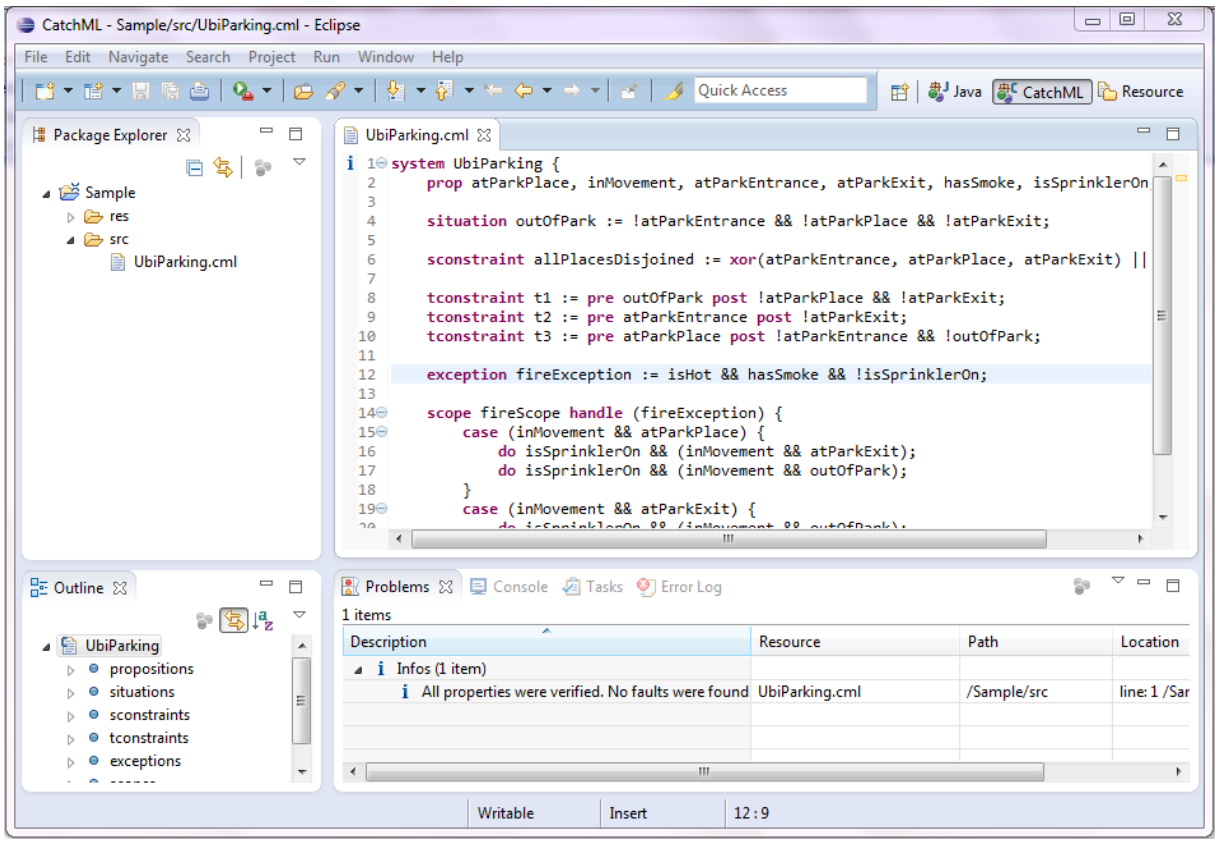


Figura 4.3: Ambiente de desenvolvimento integrado com editor e *parser*

tivos do trabalho possibilitar, através do ambiente de desenvolvimento, a verificação automática do modelo em busca de faltas de projeto. Nesse sentido, o trabalho de Rocha (2013) já fornece uma ferramenta, a JCAEHV, que permite a modelagem do tratamento de exceção e provê um mecanismo de verificação que analisa a ocorrência de faltas na especificação. Entretanto, essa ferramenta é fornecida como uma API Java e, dessa forma, traz consigo a necessidade de se conhecer detalhes de implementação da linguagem subjacente que poderiam ser abstraídos pelo projetista. Com o intuito de facilitar essa tarefa e manter o mesmo nível de funcionalidade (manter intacto o mecanismo de verificação de propriedades), foi realizado um mapeamento direto do modelo da linguagem CatchML para a API Java fornecida pelo JCAEHV.

Inicialmente foi necessário definir uma estrutura genérica contendo as informações do modelo necessárias para a especificação no JCAEHV. Tal estrutura é denominada *EModel* e contém as listas dos objetos definidos na especificação da linguagem CatchML (proposições, situações, restrições, escopos, casos, ações e propriedades). Essa estrutura é utilizada para instanciar um modelo JCAEHV (*CAEHModel*) através da interpretação de seus atributos em instâncias de seus equivalentes no método CAEHV. Na Listagem 4.11, é apresentada uma versão

do modelo do *UbiParking* especificado na ferramenta JCAEHV.

Listagem 4.11: Especificação do *UbiParking* na ferramenta JCAEHV

```
1  CAEHModel caeh = new CAEHModel("UbiParkingCorrect");
2  ContextProposition inMovement = new ContextProposition("inMovement");
3  caeh.addContextProposition(inMovement);
4  ContextProposition atParkEntrance = new ContextProposition("atParkEntrance");
5  caeh.addContextProposition(atParkEntrance);
6  ContextProposition atParkPlace = new ContextProposition("atParkPlace");
7  caeh.addContextProposition(atParkPlace);
8  ContextProposition atParkExit = new ContextProposition("atParkExit");
9  caeh.addContextProposition(atParkExit);
10 ContextProposition hasSpace = new ContextProposition("hasSpace");
11 caeh.addContextProposition(hasSpace);
12 ContextProposition isHot = new ContextProposition("isHot");
13 caeh.addContextProposition(isHot);
14 ContextProposition hasSmoke = new ContextProposition("hasSmoke");
15 caeh.addContextProposition(hasSmoke);
16 ContextProposition isSprinklerOn = new ContextProposition("isSprinklerOn");
17 caeh.addContextProposition(isSprinklerOn);
18 ContextExpression outOfPark = and(not(atParkEntrance), not(atParkPlace), not(atParkExit));
19 caeh.addSemanticConstraint(new SemanticConstraint("AllPlacesDisjoined", or(xor(atParkEntrance, atParkPlace,
       atParkExit), outOfPark)));
20
21 caeh.addTransitionConstraint(new TransitionConstraint(outOfPark, and(not(atParkPlace), not(atParkExit))));
22 caeh.addTransitionConstraint(new TransitionConstraint(atParkEntrance, not(atParkExit)));
23 caeh.addTransitionConstraint(new TransitionConstraint(atParkPlace, and(not(atParkEntrance), not(outOfPark))))
       ;
24
25 ContextualException fire = new ContextualException("Fire", and(isHot, hasSmoke, not(isSprinklerOn)));
26 HandlingScope fireScope = new HandlingScope(fire);
27 HandlingCase handlingCase = new HandlingCase();
28 handlingCase.setCatchConstraint(and(inMovement, atParkPlace));
29 HandlingBehavior handlingBehavior = new HandlingBehavior();
30 handlingBehavior.addHandlingStep(and(isSprinklerOn, and(inMovement, atParkExit)));
31 handlingBehavior.addHandlingStep(and(isSprinklerOn, and(inMovement, outOfPark)));
32 handlingCase.setHandlingBehavior(handlingBehavior);
33 fireScope.addHandlingCase(handlingCase);
34
35 handlingCase = new HandlingCase();
36 handlingCase.setCatchConstraint(and(inMovement, atParkExit));
37 handlingBehavior = new HandlingBehavior();
38 handlingBehavior.addHandlingStep(and(isSprinklerOn, and(inMovement, outOfPark)));
39 handlingCase.setHandlingBehavior(handlingBehavior);
40 fireScope.addHandlingCase(handlingCase);
41 caeh.addHandlingScope(fireScope);
42
43 ContextualException noSpace = new ContextualException("NoSpace", and(inMovement, atParkPlace, not(
```

```
    hasSpace)));
44  HandlingScope noSpaceScope = new HandlingScope(noSpace);
45  handlingCase = new HandlingCase();
46  handlingCase.setCatchConstraint(and(inMovement, atParkPlace));
47  handlingBehavior = new HandlingBehavior();
48  handlingBehavior.addHandlingStep(and(inMovement, atParkExit));
49  handlingCase.setHandlingBehavior(handlingBehavior);
50  noSpaceScope.addHandlingCase(handlingCase);
51  caeh.addHandlingScope(noSpaceScope);
```

Já na Listagem 4.12 é mostrado o mesmo modelo especificado na linguagem CatchML. Nesse exemplo, é possível notar uma diferença no tamanho dos dois modelos. Isso se deve, principalmente, à retirada dos detalhes de implementação do Java e da aplicação de construtores específicos do domínio de tratamento de exceção sensível ao contexto.

Listagem 4.12: Especificação do *UbiParking* na linguagem CatchML

```
1  system UbiParking {
2          prop atParkPlace, inMovement, atParkEntrance, atParkExit, hasSmoke, hasSpace, isSprinklerOn, isHot;
3          situation outOfPark := !atParkEntrance && !atParkPlace && !atParkExit;
4          sconstraint allPlacesDisjoined := xor(atParkEntrance, atParkPlace, atParkExit, outOfPark);
5          tconstraint t1 := pre outOfPark post !atParkPlace && !atParkExit;
6          tconstraint t2 := pre atParkEntrance post !atParkExit;
7          tconstraint t3 := pre atParkPlace post !atParkEntrance && !outOfPark;
8          exception fireException := isHot && hasSmoke && !isSprinklerOn;
9          scope fireScope handle (fireException) {
10                 case (inMovement && atParkPlace) {
11                         do isSprinklerOn && (inMovement && atParkExit);
12                         do isSprinklerOn && (inMovement && outOfPark);
13                 }
14                 case (inMovement && atParkExit) {
15                         do isSprinklerOn && (inMovement && outOfPark);
16                 }
17         }
18         exception noSpaceException := inMovement && atParkPlace && !hasSpace;
19         scope noSpaceScope handle (noSpaceException) {
20                 case (inMovement && atParkPlace) {
21                         do inMovement && atParkExit;
22                 }
23         }
24         spec hasFireException := EX(((isHot && hasSmoke) && (!isSprinklerOn)));
25 }
```

Na Listagem 4.13, é mostrado o relatório de verificação gerado contendo informações sobre modelo, propriedades violadas e as faltas de projeto.

Listagem 4.13: Relatório de análise do *UbiParking* gerado pelo JCAEHV

```
1                                The JCAEHV Report
2
3  + System name: UbiParkingCorrect
4  + State number: 65
5  + Initial state number: 1
6  + Reachable state number: 17
7  + Unreachable state number: 48
8  + Exceptional state number: 8
9  + Duration Time: 129ms
10
11  Properties Verification
12
13 [Accepted][Fire](Detection Liveness)
14         Property Formula: EX(((isHot & hasSmoke) & ( ! isSprinklerOn)))
15 [Accepted][Fire](Catch Liveness)
16         Property Formula: EX(((isHot & hasSmoke) & ( ! isSprinklerOn)) & ((inMovement & atParkPlace) | (
               inMovement & atParkExit)))
17 [Accepted][Fire](Handler Liveness)
18         Property Formula: EX(((isHot & hasSmoke) & ( ! isSprinklerOn)) & (inMovement & atParkPlace)) &
               EX(((isHot & hasSmoke) & ( ! isSprinklerOn)) & (inMovement & atParkExit))
19 [Accepted][Fire](Handling Stability)
20         Property Formula: AX(!((((isHot & hasSmoke) & ( ! isSprinklerOn)) & (inMovement & atParkPlace))
               & EF((isSprinklerOn & (inMovement & ((( ! atParkEntrance) & ( ! atParkPlace)) (...)
21 [Accepted][Fire](Reachability)
22         Property Formula: AX((((isHot & hasSmoke) & ( ! isSprinklerOn)) & (inMovement & atParkPlace)) ->
               EX((isSprinklerOn & (inMovement & atParkExit)) (...)
```

No ambiente de desenvolvimento integrado, foi adicionado um menu de contexto *CatchML* que pode ser executado a partir do editor de texto da linguagem. Nesse menu, há duas opções de execução do verificador. Ao utilizar a primeira opção, uma verificação é realizada e, caso erros sejam detectados, os mesmos são mostrados diretamente nos trechos de código do modelo. Essa funcionalidade foi criada com o objetivo de tornar a análise das faltas mais direta e intuitiva para o projetista. Por exemplo, suponha que o projetista do *UbiParking* tenha inserido uma falta na especificação da exceção de incêndio (linha 12) de acordo com a Listagem 4.14. Nesse exemplo, foi adicionada uma contradição $(isHot \wedge \neg isHot)$ na fórmula lógica que representa a exceção.

Listagem 4.14: Exceção de incêndio especificada de forma inconsistente

```
8          exception fireException := isHot && hasSmoke && !isSprinklerOn && !isHot;
```

Ao executar o verificador de modelos, caso uma falta de projeto seja detectada,

então uma mensagem de erro é mostrada diretamente no trecho de código responsável pela inconsistência semântica. No exemplo do *UbiParking*, onde uma falta foi inserida na exceção de incêndio, a ferramenta JCAEHV detectou uma falta de projeto do tipo Exceção Morta (Dead Exception Fault) e a mesma foi exibida diretamente no código do modelo (Figura 4.4). Dessa forma, espera-se que o projetista tenha um rápido *feedback* das regiões do modelo que contém erros semânticos (propriedades temporais não satisfeitas) e que assim possa fazer os ajustes necessários.

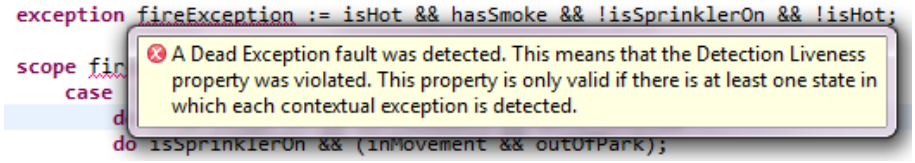


Figura 4.4: Faltas de projeto mostradas diretamente na especificação

A segunda opção do menu de contexto é uma funcionalidade opcional que executa o verificador e mostra o relatório de erros padrão do JCAEHV no console da ferramenta.

4.5 Sumário

Este capítulo apresentou a linguagem CatchML, uma DSL para modelagem e verificação do tratamento de exceção sensível ao contexto em sistemas ubíquos. Inicialmente, foi apresentada uma visão geral da metodologia adotada neste trabalho. Foram apresentados os detalhes necessários para se definir a gramática base para a linguagem, bem como os requisitos para se realizar a integração da linguagem com a ferramenta JCAEHV, com vistas a fornecer um mecanismo de verificação do modelo. Em seguida, as principais etapas na construção de uma DSL foram apresentadas, incluindo a análise do domínio, a definição das sintaxes abstrata e concreta, bem como as fases de implementação e transformação utilizando um *framework* de suporte ao desenvolvimento de linguagens específicas de domínio, o Xtext.

Na etapa de análise do domínio foram elencados os principais conceitos e abstrações relacionados ao tratamento de exceção sensível ao contexto citados em trabalhos recentes na área. Os conceitos foram comparados e listados gerando uma base de conhecimento a ser utilizada para determinar a sintaxe abstrata da linguagem. Considera-se essa etapa de análise como a mais exaustiva dessa dissertação, entretanto, foi muito importante por fornecer os fundamentos necessários para definir os construtores da linguagem.

A etapa de projeto da linguagem foi a mais complexa devido à necessidade de se raciocinar sobre o conhecimento do domínio de forma a aumentar o nível de abstração dos principais conceitos relacionados ao tratamento de exceção sensível ao contexto. Uma das principais dificuldades dessa etapa foi prover uma alternativa que fosse simples e intuitiva para o projetista, sem perda de expressividade do modelo. Após a criação de vários protótipos, foi obtida uma gramática consistente da qual foi derivada a sintaxe abstrata da linguagem.

A implementação da linguagem foi feita utilizando o *framework* Xtext. Esse *framework* foi escolhido por possuir boa documentação e por já ter sido utilizado em outros trabalhos, como (CASSOU et al., 2012). A utilização do Xtext possibilitou a implementação de um ambiente integrado de desenvolvimento em forma de *plugin* para a plataforma Eclipse. Esse ambiente provê suporte de ferramentas tais como editor de texto com *parser* e assistente para completar código, marcação, formatação, *outline* e perspectiva. Além das funcionalidades padrão, foram implementados métodos de verificação de inconsistências sintáticas com o intuito de tornar mais simples a tarefa de especificar o modelo.

Por fim, foi possível integrar a linguagem com a ferramenta JCAEHV, dando a possibilidade de verificar automaticamente os modelos de tratamento de exceção especificados na DSL. Como melhoria proveniente da integração, foi adicionado um método de validação da especificação que exibe os erros gerados pelo verificador diretamente no código fonte da especificação.

5 ESTUDO DE CASO

Neste capítulo é apresentado um estudo de caso realizado com o intuito de investigar se os resultados obtidos neste trabalho atingem os objetivos esperados. O capítulo está organizado da seguinte forma: a Seção 5.1 apresenta o protocolo definido para o estudo de caso; na Seção 5.2 é definido o *design* do estudo conduzido; na Seção 5.3 são apresentados os passos para se realizar o estudo de caso; já na Seção 5.4, os dados coletados são discutidos e conclusões são obtidas; e, por fim, na Seção 5.5 são feitas as considerações finais sobre o estudo realizado.

5.1 Protocolo do Estudo de Caso

Segundo Yin (2003), o estudo de caso é "uma forma de se fazer pesquisa social empírica ao investigar um fenômeno dentro do seu contexto real, onde as fronteiras entre o fenômeno e o contexto não são claramente definidas e na situação em que múltiplas fontes de evidência são usadas". Dessa forma, sendo o principal objetivo desta dissertação propor uma linguagem específica de domínio para modelagem do tratamento de exceções sensível ao contexto de aplicações ubíquas para avaliar o impacto de sua aplicação, decidiu-se realizar um estudo de caso com alunos do curso de ciência da computação para coletar indícios de que a ferramenta atinge seus objetivos.

5.1.1 Objetivo

O objetivo do estudo de caso é definir se a linguagem CatchML oferece suporte para modelagem e verificação do tratamento de exceção sensível ao contexto, e suporte à apresentação eficiente de faltas de projeto presentes no modelo do ponto de vista do projetista. Espera-se alcançar indícios de que o ambiente integrado possa auxiliar o projetista, tornando sua tarefa de modelagem simples e intuitiva.

5.1.2 Questões de pesquisa

As questões de pesquisa avaliam o caso na perspectiva do projetista utilizando a linguagem e o ambiente para modelar o tratamento de exceção sensível ao contexto de aplica-

ções ubíquas. Para cada questão de pesquisa, são definidas métricas a serem aplicadas na etapa de análise dos dados resultantes do estudo de caso. Tais métricas devem prover subsídios para responder às questões de pesquisa. Para tanto, as seguintes questões foram definidas:

Q1: Quão fácil para o projetista é modelar o tratamento de exceção sensível ao contexto utilizando a linguagem CatchML?

Análise: Esta pergunta busca investigar a viabilidade da linguagem como alternativa simples e intuitiva para o projetista que é *expert* ou não no domínio de tratamento de exceção sensível ao contexto.

Métrica: O tempo gasto para especificar o modelo de tratamento de exceção sensível ao contexto e o número de não conformidades apresentado após a modelagem.

Q2: Quão fácil é para o projetista coletar *feedback* sobre faltas de projeto existentes no modelo?

Análise: Esta pergunta busca investigar o impacto para o projetista na visualização das faltas de projeto reportadas pelo JCAEHV que são mostradas diretamente no código do modelo.

Métrica: Tempo gasto para analisar e remover faltas de projeto, contagem da quantidade de vezes que o projetista executou a verificação do modelo, questionário contendo perguntas sobre como as faltas foram reportadas pela ferramenta e questões sobre a interface da ferramenta.

5.2 Planejamento

O propósito deste estudo de caso é avaliar, com respeito ao suporte à modelagem, à verificação de faltas, e à apresentação de *feedback* sobre as faltas reportadas, do ponto de vista do projetista, no contexto de estudantes de graduação e pós-graduação do curso de ciência da computação da UFC, o impacto da utilização da linguagem CatchML como alternativa para o projeto do tratamento de exceção sensível ao contexto em sistemas ubíquos.

Ao analisar as questões de pesquisa, foi necessário definir as variáveis independentes e dependentes relacionadas ao estudo. As variáveis independentes são aquelas que são manipuladas, e nesse sentido independentes dos padrões de reação inicial, intenções e caracte-

rísticas dos sujeitos da pesquisa. A seguir tem-se as variáveis independentes:

- A ferramenta a ser analisada, que neste caso é a linguagem CatchML;
- A descrição do modelo de tratamento de exceção para especificação; e
- Os cenários de modelagem para verificação e análise de *feedback*.

Já as variáveis dependentes são as que devem ser medidas e dependem das ações dos sujeitos que participam do estudo de caso. A seguir tem-se as variáveis dependentes:

- O tempo gasto para especificar o modelo de acordo com a descrição fornecida. Será calculado o tempo em minutos gasto desde o início da leitura da descrição até o término da especificação por parte do projetista;
- O número de não conformidades observado após o término da modelagem. Será calculado pelo número de faltas observadas no modelo;
- O grau de entendimento do projetista em relação aos faltas verificados na ferramenta. Será analisado o questionário sobre as faltas reportadas e contabilizado o número de não conformidades em relação ao *feedback* fornecido pela ferramenta; e
- O grau de satisfação do projetista em relação à interface da ferramenta. Serão analisadas as respostas ao questionário qualitativo sobre facilidade de uso da ferramenta.

5.3 Avaliação da Linguagem

O estudo de caso contou com a participação de nove voluntários, sendo três alunos de graduação e seis alunos de pós-graduação. A Figura 5.1 apresenta as atividades executadas no estudo conduzido.

A primeira atividade correspondeu à execução de um questionário pré-experimento (Apêndice B), aplicado para caracterizar o perfil dos participantes. Os resultados do questionário pré-experimento revelaram que todos os seis alunos de pós-graduação que participaram do estudo tinham um nível alto de conhecimento de lógica proposicional, da linguagem Java e da plataforma Eclipse. Entretanto, três voluntários conheciam o tratamento de exceção sensível

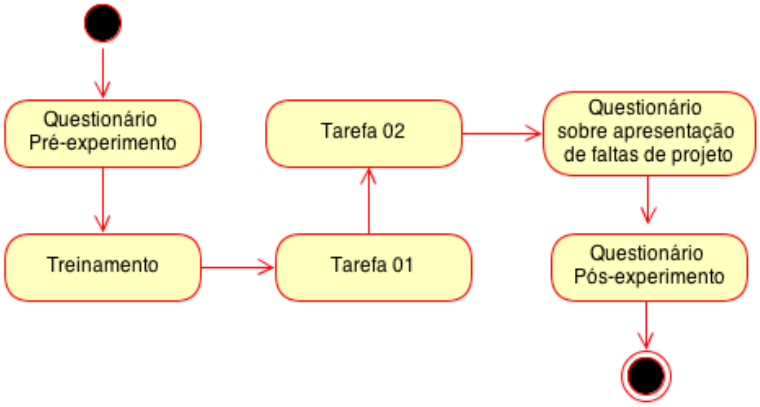


Figura 5.1: Atividades realizadas no estudo de caso

ao contexto sem nunca ter aplicado a técnica, e os outros três nem sequer conheciam a técnica. Porém, três dos voluntários conheciam o tratamento de exceção de sistemas tradicionais, sendo que os demais conheciam mas não haviam aplicado. Logo, considera-se que a experiência em tratamento de exceção em sistemas tradicionais pode ser um fator diferencial no processo de modelagem.

Com relação aos alunos de graduação, dos três que participaram do estudo, dois conheciam bastante lógica proposicional e possuíam um conhecimento médio em Java e Eclipse, enquanto que o outro voluntário tinha um conhecimento médio em lógica proposicional e conhecimento básico em Java e Eclipse. Além disso, todos os alunos de graduação não conheciam o tratamento de exceção sensível ao contexto. Entretanto, dois dos alunos já haviam desenvolvido aplicações utilizando o tratamento de exceções de sistemas tradicionais enquanto que um deles só conhecia a técnica, sem nunca tê-la utilizado. Do mesmo modo que ocorreu para os alunos de pós-graduação, o fato de conhecer e ter aplicado tratamento de exceção em sistemas tradicionais pode representar um diferencial no processo de modelagem para os alunos de graduação.

Nesse sentido, com o objetivo de minimizar os efeitos de falta de conhecimento no domínio de estudo, na sessão de treinamento[1] foram explorados conceitos sobre tratamento de exceção sensível ao contexto, modelagem e verificação do tratamento de exceção e detalhes da API da linguagem CatchML. Ainda nessa sessão, os voluntários aprenderam sobre as cinco propriedades descritas no método CAEHV as quais um modelo de tratamento de exceção deve satisfazer, bem como sobre as faltas geradas se tais propriedades forem violadas. Ao final

[1] http://www.great.ufc.br/∼rafaellima/catchml/treinamento.zip

do treinamento, os voluntários fizeram um exercício de modelagem e verificação para fixar os conceitos aprendidos.

Após a fase de treinamento, os participantes realizaram uma tarefa[2] de modelagem (Tarefa 01) para verificar sua desenvoltura ao utilizar a linguagem CatchML. Nessa tarefa, os participantes receberam uma descrição textual de um modelo de tratamento de exceção para que pudesse ser especificado utilizando a ferramenta. Após a modelagem, os participantes deveriam analisar a especificação em busca de faltas de projeto com o apoio do verificador. A tarefa era dada por encerrada quando o participante conseguisse executar o verificador e o mesmo não reportasse nenhum erro ou quando o participante alegasse não saber como remover as faltas de projeto. Nessa tarefa, foi utilizado o modelo do *UbiParking* contendo a exceção de incêndio e o tratamento para esta exceção.

Em seguida, os participantes deveriam realizar uma tarefa[3] voltada à análise das faltas de projeto apresentadas pela ferramenta (Tarefa 02). Nessa tarefa, os participantes foram convidados a escolher entre duas especificações do tratamento de exceção sensível ao contexto do *UbiParking*, o *UbiParking1* e o *UbiParking2*, ambos contendo duas exceções: a exceção de incêndio e a exceção de falta de vaga no estacionamento. Tais especificações receberam propositalmente faltas de projeto através da injeção de contradições nas fórmulas lógicas. O *UbiParking1* recebeu duas injeções de faltas que violavam três propriedades e o *UbiParking2* recebeu duas injeções de faltas que violavam quatro propriedades.

Após escolher um dos modelos, os participantes deveriam analisá-los manualmente em busca de faltas de projeto. Após essa atividade, os voluntários deveriam, quando julgassem oportuno, utilizar o verificador para ajudá-los na tarefa de remoção das faltas. A tarefa era dada por encerrada quando o participante conseguisse realizar uma verificação livre de faltas ou se alegasse não conseguir remover as faltas. Ao final dessa atividade, os participantes foram convidados a responder dois questionários.

O primeiro questionário tinha como objetivo avaliar o nível de compreensão dos usuários em relação às faltas. As questões estavam relacionadas aos tipos de faltas de projeto encontrados antes e depois de utilizar o verificador, bem como em relação às dificuldades para entender as faltas e removê-las. Já o segundo questionário tinha o objetivo de avaliar de forma

[2]http://www.great.ufc.br/∼rafaellima/catchml/task1.zip
[3]http://www.great.ufc.br/∼rafaellima/catchml/task2.zip

geral aspectos da metodologia adotada no estudo de caso, bem como coletar opiniões dos voluntários sobre facilidade de uso e sobre a forma como as faltas de projeto foram apresentadas. As respostas obtidas nesse questionário forneceram dados importantes para análise de problemas e limitações no ambiente de desenvolvimento proposto.

5.4 Análise dos Resultados

Para avaliar a facilidade de uso da linguagem e adequação dos seus construtores e abstrações, foram utilizados os seguintes dados: (i) tempo para especificar o modelo; (ii) número de não conformidades; e (iii) número de verificações realizadas pelos participantes na Tarefa 01. Os dados coletados podem ser vistos na Tabela 5.1.

Tabela 5.1: Dados obtidos na Tarefa 01

Voluntário	Tempo(min)	Erros	Execuções
Voluntário 1	16	0	3
Voluntário 2	50	4	16
Voluntário 3	12	0	2
Voluntário 4	18	0	7
Voluntário 5	50	1	6
Voluntário 6	50	3	8
Voluntário 7	41	2	9
Voluntário 8	36	0	7
Voluntário 9	25	1	6

Conforme observado na Tabela 5.1, podemos realizar a seguinte análise:

- (i) Quatro voluntários (1, 3, 4 e 8) tiveram seus modelos livres de não conformidades com uma média de aproximadamente quatro execuções do verificador e um tempo médio de 15 minutos, com exceção do voluntário 8 que demorou 36 minutos para concluir o modelo. Esse resultado era esperado visto que os quatro voluntários são alunos de pós-graduação com boa experiência em tratamento de exceção de sistemas tradicionais;

- (ii) Dois voluntários (5 e 9) tiveram uma não conformidade com média de seis execuções e um tempo médio de 38 minutos. A presença de uma não-conformidade fez com que os voluntários demorassem mais tempo analisando o modelo em busca de faltas e com a ajuda do verificador. Os dois voluntários são alunos de graduação com experiência em tratamento de exceção de sistemas tradicionais;

- (iii) Dois voluntários (7 e 6) tiveram duas e três não conformidades respectivamente, com média de oito execuções e tempo médio de 45 minutos. Esses voluntários podem ter demorado mais tempo por conta da dificuldade em encontrar as faltas de projeto inseridas por eles. Ambos são alunos de pós-graduação sem experiência em tratamento de exceção em sistemas tradicionais; e

- (iv) Um voluntário (2) teve quatro não conformidades e realizou 16 execuções com tempo de 50 minutos. Esse resultado pode ser justificado devido ao voluntário ser aluno de graduação sem nenhuma experiência em tratamento de exceção em sistemas tradicionais.

A Tarefa 02 executada no estudo de caso tinha como objetivo investigar o nível de compreensão dos voluntários com respeito à análise e correção de faltas de projeto utilizando o verificador de modelos. Ao final da Tarefa 02, os voluntários responderam o questionário de teste de compreensão das faltas de projeto reportadas e da interface de visualização das faltas adotada (Apêndice C). Um sumário dos dados obtidos pode ser encontrado na Tabela 5.2:

Tabela 5.2: Dados obtidos na Tarefa 02

Voluntário	Quantos erros injetados?	Quantos tipos de propriedades violadas reportadas?	Quais faltas de projeto reportadas?	Corrigiu os erros?	Tempo (min)
Voluntário 1	1	0	1	1	13
Voluntário 2	0	1	0	0	22
Voluntário 3	1	1	1	1	7
Voluntário 4	1	0	0	1	4
Voluntário 5	0	1	0	0	30
Voluntário 6	1	0	0	1	25
Voluntário 7	0	1	1	1	6
Voluntário 8	1	1	1	1	29
Voluntário 9	0	1	1	1	5

Legenda

1	Acertou a questão
0	Errou a questão

A pergunta *Quantos erros injetados?* tinha por objetivo verificar a capacidade dos usuários em identificar as faltas sem apoio do verificador. A resposta à essa questão era considerada correta se o participante acertasse a quantidade de faltas injetadas. Já a pergunta *Quantos tipos de propriedades violadas reportadas?* tinha o objetivo de verificar a compreensão dos participantes com relação às propriedades violadas. O participante deveria responder a questão com a quantidade de propriedades apresentadas pelo verificador. A pergunta *Quais faltas de projeto reportadas?* tinha o objetivo de verificar a compreensão do *feedback* apresentado pelo verificador em relação às faltas de projeto. Para responder corretamente, o participante deveria

listar as faltas de projeto apresentadas. Por fim, a última questão tinha o objetivo de verificar se o participante conseguiu remover as faltas de projeto com o suporte da ferramenta.

Ao analisar os resultados, percebe-se que no geral os participantes fizeram bom uso do verificador como suporte à identificação e remoção das faltas de projeto que foram injetadas. No total, sete voluntários analisaram e removeram as faltas com tempo médio de 13 minutos, o que representa 78% do total. Apesar disso, houve certa dificuldade dos participantes descobrirem as faltas injetadas sem o uso da ferramenta, visto que apenas cinco voluntários (1, 3, 4, 6 e 8) acertaram essa questão (55%).

Outra constatação que pode ser tirada dos resultados é em relação ao *feedback* da ferramenta em relação às faltas. Apenas quatro voluntários (3, 7, 8 e 9) responderam corretamente os tipos de propriedades violadas bem como as faltas geradas pelo verificador. Já dois deles (4 e 6) responderam incorretamente as duas questões. Os outros dois responderam uma correta e outra errada. Nesse sentido, tem-se que apenas 44% dos participantes conseguiram compreender bem as faltas de projeto e propriedades apresentados diretamente no código do modelo. Durante a execução do caso, foi observado que boa parte dos participantes relatou problemas com relação ao tamanho da mensagem de descrição do erro e à quantidade de repetições do mesmo erro[4], o que pode ter impactado em sua compreensão.

Além das questões direcionadas às propriedades e faltas de projeto, também estava presente nesse questionário uma questão pedindo a opinião dos voluntários sobre a forma como as faltas foram apresentadas na ferramenta. Cinco voluntários responderam positivamente a essa questão, enquanto que os outros quatro afirmaram ter algum tipo de crítica, seja com relação ao conteúdo da mensagem de erro, seja com relação à quantidade de mensagens mostradas. Por fim, houve uma outra questão solicitando aos usuários sugestões ou criticas à ferramenta. A Tabela 5.3 apresenta os resultados obtidos desse questionamento e outras melhorias possíveis obtidas pela observação direta dos voluntários utilizando a ferramenta.

Conforme pode ser observado na Tabela 5.3, ainda existem problemas que precisam ser resolvidos na linguagem e no ambiente integrado. É importante mencionar que esses defeitos podem aumentar a incidência de faltas de projeto, tornando a ferramenta não tão vantajosa quanto deveria ser. Dessa forma, a aplicação dos questionários em um estudo prático foi útil

[4]Por exemplo, no falta de Retomada impossível, por ser relacionada às ações de tratamento, as mensagens de erro eram repetidas para cada ação declarada, tornando sua apresentação confusa.

Tabela 5.3: Problemas reportados pelos participantes do estudo

Comentários dos voluntários	Problemas observados
• Descrição dos erros muito vaga e difícil de compreender • Não verificar duplicidade de nomes • Sublinhar as linhas pode causar confusão • Mostrar apenas o erro inicial do encadeamento • Separar erros de arquivos diferentes • Menu de contexto no *package explorer* não funciona • O *plugin* não funcionou em notebook Mac • As abstrações *tconstraint* e *sconstraint* não ficaram muito claras • Novo arquivo poderia vir com um *template* • Abstração de escopo não ficou clara	• Acrescentar mais verificações sintáticas • Colocar o evento de execução do verificador no botão RUN do Eclipse • Adicionar ajuda com descrição e exemplos das propriedades • Adicionar *sample projects*

por ajudar a identificar problemas que precisam ser corrigidos a fim de obter maior facilidade de uso.

Além dos comentários, os voluntários foram questionados sobre a adequação dos construtores e abstrações adotados na linguagem, e a forma como as faltas de projeto foram apresentadas em uma escala de um a cinco: 1 - Péssimo; 2 - Regular; 3 - Bom; 4 - Ótimo; e 5 - Excelente. Quanto à adequação dos construtores, cinco participantes afirmaram que os construtores são ótimos e os outros quatro afirmaram que são excelentes. Já com relação à apresentação das faltas, quatro afirmaram ser regular, três afirmaram ser boa, e somente dois afirmaram ser excelente.

Dessa forma, pelos resultados apresentados nesta seção, percebe-se que a ferramenta atingiu seu objetivo principal que era fornecer uma interface intuitiva para modelagem do tratamento de exceção sensível ao contexto com suporte à verificação do modelo. Entretanto, ela ainda precisa de melhorias em termos de interface e apresentação das faltas antes que possa ser realizado um estudo quantitativo mais rigoroso.

É importante mencionar que os resultados obtidos no estudo de caso representam indícios iniciais e não podem ser generalizados. Isso porque somente um experimento controlado mais rigoroso, com um número maior de voluntários e maior diversidade de cenários de modelagem e verificação é que permitirá uma análise mais robusta sobre a adequação das contribuições propostas neste trabalho. Além disso, apesar dos resultados iniciais positivos, constatou-se com o estudo de caso que a ferramenta ainda precisa de melhorias para ser uma alternativa útil para modelagem e verificação do tratamento de exceção sensível ao contexto.

Tais resultados podem não ser estatisticamente significantes, porém indicam a viabilidade de se utilizar a linguagem CatchML para suporte à modelagem do tratamento de exceção sensível ao contexto.

5.5 Sumário

Neste capítulo foi apresentado um estudo de caso realizado para verificar a viabilidade da linguagem CatchML como alternativa para modelagem do tratamento de exceção sensível ao contexto. Além disso, era objetivo desse estudo investigar o impacto de apresentar as faltas de projeto diretamente no código da ferramenta no processo de verificação e remoção de faltas de projeto no modelo.

Com relação ao estudo em si, participaram nove estudantes, entre eles três alunos de graduação e seis alunos de pós-graduação do curso de ciência da computação da UFC. Ambos receberam um treinamento sobre os conceitos de modelagem do tratamento de exceção sensível ao contexto e das abstrações utilizadas na linguagem CatchML. Após isso os mesmos realizaram tarefas de modelagem e verificação, e, em seguida, preencheram questionários para coleta de dados sobre o processo. Os resultados obtidos indicaram que a linguagem CatchML facilita a especificação de modelos de tratamento de exceção sensível ao contexto , pois permite definir de forma intuitiva os conceitos relacionados a esse domínio. Além disso, foi constatado que a interface de apresentação de faltas de projeto é adequada, porém precisa de melhorias para se tornar realmente efetiva.

6 CONCLUSÃO

Esta dissertação propôs uma linguagem específica de domínio para modelagem e verificação do tratamento de exceção sensível ao contexto. Para facilitar a utilização da linguagem foi proposto um ambiente de desenvolvimento com parser para a linguagem CatchML e integração com a ferramenta JCAEHV.

Este capítulo é dedicado às considerações finais. Na Seção 6.1 é descrita a visão geral da dissertação e os principais resultados alcançados. A seção 6.2 enumera algumas restrições dessa pesquisa. Por fim, a Seção 6.3 apresenta os possíveis trabalhos futuros.

6.1 Resultados Alcançados

Nesta dissertação foi proposta uma linguagem de domínio específico para modelagem do tratamento de exceção sensível ao contexto. Para reduzir a quantidade de faltas de projeto presentes na especificação, a linguagem foi integrada com a ferramenta JCAEHV, possibilitando a verificação de faltas em tempo de projeto. Além disso, uma alternativa para apresentação das faltas de projeto reportadas pelo JCAEHV é fornecida com o objetivo de facilitar a identificação e correção do modelo. Ao final, um estudo de caso foi realizado para verificar a aceitação da linguagem. Tal estudo apresenta indícios iniciais de que a linguagem em conjunto com o ambiente de desenvolvimento são viáveis para auxiliar os projetistas no processo de modelagem do tratamento de exceção sensível ao contexto.

Em resumo, os principais resultados alcançados desta dissertação foram:

- Uma linguagem para modelagem do tratamento de exceção sensível ao contexto denominada CatchML que provê construtores para os aspectos centrais desse domínio;
- Um ambiente de desenvolvimento integrado com a ferramenta JCAEHV o que possibilita a verificação do modelo de tratamento de exceção;
- Uma melhoria na visualização e identificação de erros gerados pelo verificador de modelos, através de uma abordagem para apresentação das faltas de projeto diretamente no código do modelo; e

- Publicação no Workshop de Teses e Dissertações do WTDSoft (LIMA; ROCHA; ANDRADE, 2012).

6.2 Limitações

No decorrer desta dissertação foram apresentados benefícios e vantagens no desenvolvimento e utilização da linguagem CatchML. Porém, também é importante ressaltar limitações na pesquisa que foram citadas anteriormente de forma implícita e que são enumeradas a seguir:

- A ferramenta JCAEHV fornece suporte à exceções concorrentes, porém essa funcionalidade ainda não esta disponível na linguagem CatchML. Dessa forma, fica restrita a composição e modelagem de situações onde há ocorrência simultânea de exceções;
- A análise do estudo de caso mostrou alguns problemas relacionados ao ambiente de desenvolvimento integrado que são aceitáveis dado que a implementação é apenas um protótipo. No Capítulo 5 foi apresentada a Tabela 5.3 com sugestões de melhoria relacionadas ao ambiente que devem ser implementadas como forma de mitigar essas restrições; e
- A disponibilização de uma nova linguagem exige uma certa curva de aprendizado antes da utilização. De qualquer forma, essa restrição pode ser reduzida através de uma boa documentação e o provimento de exemplos de uso da linguagem.

6.3 Trabalhos Futuros

Os trabalhos propostos como atividades de pesquisa a serem desenvolvidas posteriormente para dar continuidade a este trabalho são listadas a seguir:

- **Composição dos modelos adaptativo e excepcional**: assim como no método CAEHV, este trabalho está focado em modelar o comportamento excepcional sensível ao contexto. Um possível direcionamento de pesquisa consiste em investigar uma maneira de compor os dois modelos de comportamento e entender a forma como eles interagem entre si. Dessa forma, pode-se definir abstrações para modelagem do comportamento normal do sistema de forma conjunta com o comportamento excepcional. Para que o verificador

de modelos possa ser utilizado, é preciso desenvolver essa funcionalidade na ferramenta JCAEHV;

- **Modelagem de exceções concorrentes e propagação de exceções**: de acordo com o estudo do domínio de tratamento de exceção sensível ao contexto, uma das características que um projetista deve ser capaz de modelar é a forma como o sistema irá se comportar no caso da ocorrência simultânea de mais de uma exceção. Nesse sentido, é necessário que seja possível especificar na linguagem CatchML, uma função de resolução que permita resolver exceções concorrentes. Na ferramenta JCAEHV, já são implementadas duas estratégias existentes na literatura: uma baseada em árvores de exceções e a outra em prioridades, porém não foi feita uma análise mais aprofundada. Além da modelagem de exceções concorrentes, outro conceito importante é o de propagação de exceções. Essa abstração não é implementada na ferramenta JCAEHV, portanto, representa um desafio tanto na parte do verificador de modelos quanto em sua especificação na linguagem de domínio específico;

- **Geração de artefatos a partir do modelo CatchML**: neste trabalho, o modelo CatchML é transformado para o modelo CAEHV, tornando possível sua verificação em relação às propriedades comportamentais definidas no método. Entretanto, é possível utilizar o modelo CatchML para geração de outros artefatos que podem ser utilizados nas demais etapas do processo de desenvolvimento. Uma possível aplicação seria a geração de código para *frameworks* ou mecanismos de suporte ao tratamento de exceção (e.g., o *framework* FRonTES (QUEIROZ FILHO, 2012) ou o mecanismo proposto em (DAMASCENO et al., 2006)); e

- **Suporte ao uso de variáveis de contexto inteiras e reais**: o contexto que é modelado na linguagem CatchML é composto por proposições lógicas ou fórmulas lógicas sobre essas proposições. Essa solução é útil, pois possibilita reduzir o espaço de estados a ser analisado pelo verificador de modelos. Porém, é possível que o projetista do tratamento de exceção sensível ao contexto tenha interesse em trabalhar com informações contextuais inteiras ou reais, e que possa fazer comparações e relações com esses valores. Dessa forma, seria necessário prover abstrações e construtores que pudessem representar essas expressões, tornado a tarefa do projetista mais intuitiva. Além disso, seria necessário derivar essas expressões em expressões lógicas, para que o verificador pudesse tratá-las.

REFERÊNCIAS BIBLIOGRÁFICAS

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, IEEE, v. 1, n. 1, p. 11–33, 2004.

BALDAUF, M.; DUSTDAR, S.; ROSENBERG, F. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, Inderscience, v. 2, n. 4, p. 263–277, 2007.

BARDRAM, J. E. Applications of context-aware computing in hospital work: examples and design principles. In: *Proceedings of the 2004 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2004. (SAC '04), p. 1574–1579. ISBN 1-58113-812-1. Disponível em: <http://doi.acm.org/10.1145/967900.968215>.

BEDER, D. M.; ARAÚJO, R. B. de. Towards the definition of a context-aware exception handling mechanism. In: IEEE. *Dependable Computing Workshops (LADCW), 2011 Fifth Latin-American Symposium on*. [S.l.], 2011. p. 25–28.

CASSOU, D.; BERTRAN, B.; LORIANT, N.; CONSEL, C. A generative programming approach to developing pervasive computing systems. In: *Proceedings of the eighth international conference on Generative programming and component engineering*. New York, NY, USA: ACM, 2009. (GPCE '09), p. 137–146. ISBN 978-1-60558-494-2. Disponível em: <http://doi.acm.org/10.1145/1621607.1621629>.

CASSOU, D.; BRUNEAU, J.; CONSEL, C.; BALLAND, E. Toward a tool-based development methodology for pervasive computing applications. *Software Engineering, IEEE Transactions on*, v. 38, n. 6, p. 1445–1463, 2012. ISSN 0098-5589.

CHENG, B. H. C.; LEMOS, R. de; GIESE, H.; INVERARDI, P.; MAGEE, J. Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments. In: *Software Engineering for Self-Adaptive Systems*. [S.l.]: Springer, 2009. (Lecture Notes in Computer Science, v. 5525), p. 146–163. ISBN 978-3-642-02160-2.

CHETAN, S.; RANGANATHAN, A.; CAMPBELL, R. Towards fault tolerance pervasive computing. *Technology and Society Magazine, IEEE*, IEEE, v. 24, n. 1, p. 38–44, 2005.

CHO, E.-S.; HELAL, S. A situation-based exception detection mechanism for safety in pervasive systems. In: IEEE. *Applications and the Internet (SAINT), 2011 IEEE/IPSJ 11th International Symposium on*. [S.l.], 2011. p. 196–201.

____. Toward efficient detection of semantic exceptions in context-aware systems. In: IEEE. *Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC), 2012 9th International Conference on*. [S.l.], 2012. p. 826–831.

COSTA, C. A. da; YAMIN, A. C.; GEYER, C. F. R. Toward a general software infrastructure for ubiquitous computing. *IEEE Pervasive Computing*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 7, n. 1, p. 64–73, jan. 2008. ISSN 1536-1268. Disponível em: <http://dx.doi.org/10.1109/MPRV.2008.21>.

DAMASCENO, K.; CACHO, N.; GARCIA, A.; ROMANOVSKY, A.; LUCENA, C. Context-aware exception handling in mobile agent systems: the moca case. In: ACM. *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*. [S.l.], 2006. p. 37–44.

DEURSEN, A. van; KLINT, P.; VISSER, J. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, n. 6, p. 26–36, jun. 2000. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/352029.352035>.

DEY, A. K. Understanding and using context. *Personal Ubiquitous Comput.*, Springer-Verlag, London, UK, UK, v. 5, n. 1, p. 4–7, jan. 2001. ISSN 1617-4909. Disponível em: <http://dx.doi.org/10.1007/s007790170019>.

DEY, A. K.; ABOWD, G. D.; SALBER, D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, L. Erlbaum Associates Inc., Hillsdale, NJ, USA, v. 16, n. 2, p. 97–166, dez. 2001. ISSN 0737-0024.

FERREIRA FILHO, J. B.; LIMA, C. R. F.; LEITE, S. J. C.; VIANA, W.; DANTAS, L. L.; ANDRADE, R. M. C. Manutenção Adaptativa de Software Embarcado para Telefones Celulares Apoiado por Ferramentas de Automação. *Simposio Brasileiro de Qualidade de Software SBQS, 2010, Belem*, p. 327–334, 2010.

FRANCE, R.; RUMPE, B. Model-driven development of complex software: A research roadmap. In: *2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007. (FOSE '07), p. 37–54. ISBN 0-7695-2829-5. Disponível em: <http://dx.doi.org/10.1109/FOSE.2007.14>.

GARCIA, A. F.; RUBIRA, C. M.; ROMANOVSKY, A.; XU, J. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of systems and software*, Elsevier, v. 59, n. 2, p. 197–222, 2001.

GOODENOUGH, J. B. Exception handling: issues and a proposed notation. *Communications of the ACM*, ACM, v. 18, n. 12, p. 683–696, 1975.

HONG, J.; SUH, E.; KIM, S. J. Context-aware systems: A literature review and classification. *Expert Systems with Applications*, Elsevier, v. 36, n. 4, p. 8509–8522, 2009.

HOYOS, J. R.; GARCÍA-MOLINA, J.; BOTÍA, J. A. MLContext: A context-modeling language for context-aware systems. *Electronic Communications of the EASST*, v. 28, n. 0, 2010.

KNUDSEN, J. L. Better exception-handling in block-structured systems. *IEEE Software*, v. 4, n. 3, p. 40–49, 1987.

KULKARNI, D.; TRIPATHI, A. A framework for programming robust context-aware applications. *Software Engineering, IEEE Transactions on*, IEEE, v. 36, n. 2, p. 184–197, 2010.

LEE, P. A.; ANDERSON, T. *Fault tolerance*. [S.l.]: Springer, 1990.

LEE, S.; CHANG, J.; LEE, S.-g. Survey and trend analysis of context-aware systems. *Information-An International Interdisciplinary Journal*, v. 14, n. 2, p. 527–548, 2011.

LIMA, F. F. P.; ROCHA, L. S.; MAIA, P. H. M.; ANDRADE, R. M. C. A Decoupled and Interoperable Architecture for Coordination in Ubiquitous Systems. *2011 Fifth Brazilian Symposium on Software Components, Architectures and Reuse*, Ieee, v. 3, p. 31–40, set. 2011.

LIMA, R. de; ROCHA, L. S.; ANDRADE, R. M. C. Uma dsl para modelagem de comportamento de sistemas ubìquos sensìveis ao contexto. *II WorkShop de Teses e Dissertações do CBSoft. III Congresso Brasileiro de Software de 2012: Teoria e Prática*, 2012.

MAIA, M. E. F.; ROCHA, L. S.; ANDRADE, R. M. C. Requirements and challenges for building service-oriented pervasive middleware. In: *Proceedings of the 2009 international conference on Pervasive services*. New York, NY, USA: ACM, 2009. (ICPS '09), p. 93–102. ISBN 978-1-60558-644-1. Disponível em: <http://doi.acm.org/10.1145/1568199.1568214>.

MARINHO, F. G.; ANDRADE, R. M. C.; WERNER, C. A verification mechanism of feature models for mobile and context-aware software product lines. In: *Software Components, Architectures and Reuse (SBCARS), 2011 Fifth Brazilian Symposium on*. [S.l.: s.n.], 2011. p. 1 –10.

MERCADAL, J.; ENARD, Q.; CONSEL, C.; LORIANT, N. A domain-specific approach to architecturing error handling in pervasive computing. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. New York, NY, USA: ACM, 2010. (OOPSLA '10), p. 47–61. ISBN 978-1-4503-0203-6. Disponível em: <http://doi.acm.org/10.1145/1869459.1869465>.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM Computing Surveys*, v. 37, n. 4, p. 316–344, dez. 2005. ISSN 03600300.

QUEIROZ FILHO, C. A. B. d. *Um Mecanismo de Tratamento de Exceções Sensível ao Contexto para Sistemas Ubíquos Orientados a Tarefas*. 2012. Dissertação de Mestrado, Universidade Federal do Ceará.

ROCHA, L. S. *CAEHV: Um Método para Verificação de Modelos do Tratamento de Exceção Sensível ao Contexto em Sistemas Ubíquos*. 2013. Tese de Doutorado, Universidade Federal do Ceará.

ROCHA, L. S.; ANDRADE, R. M. C.; GARCIA, A. F. A method for model checking context-aware exception handling. *XXVII Brazilian Symposium on Software Engineering (SBES'13)*, Proceedings of the XXVII Brazilian Symposium on Software Engineering (SBES'13), 2013.

ROCHA, L. S.; CASTRO, C. E. P. de L.; MACHADO, J. C.; ANDRADE, R. M. C. Utilizando reconfiguração dinâmica e notificação de contextos para o desenvolvimento de software ubìquo. In: *XXI Simpòsio Brasileiro de Engenharia de Software (SBES'2007)*. [S.l.: s.n.], 2007. p. 219–235.

ROCHA, L. S.; FERREIRA FILHO, J. B.; LIMA, F. F. P.; MAIA, M. E. F.; VIANA, W.; CASTRO, M. F. D.; ANDRADE, R. M. C. Ubiquitous Software Engineering: Achievements, Challenges and Beyond. *XXV Simpósio Brasileiro de Engenharia de Software (SBES 2011), Trilha Especial SBES, II Congresso Brasileiro de Software: Teoria e Prática (CBSoft)*, Ieee, p. 132–137, set. 2011.

RYAN, N. S.; PASCOE, J.; MORSE, D. R. Enhanced reality fieldwork: the context-aware archaeological assistant. In: GAFFNEY, V.; LEUSEN, M. van; EXXON, S. (Ed.). *Computer Applications in Archaeology 1997*. Oxford: Tempus Reparatum, 1998. (British Archaeological Reports). Disponível em: <http://www.cs.kent.ac.uk/pubs/1998/616>.

SACRAMENTO, V.; ENDLER, M.; RUBINSZTEJN, H. K.; LIMA, L. S.; GONCALVES, K.; NASCIMENTO, F. N.; BUENO, G. A. Moca: A middleware for developing collaborative applications for mobile users. *Distributed Systems Online, IEEE*, IEEE, v. 5, n. 10, p. 2–2, 2004.

SAMA, M.; ROSENBLUM, D. S.; WANG, Z.; ELBAUM, S. Multi-layer faults in the architectures of mobile, context-aware adaptive applications. *Journal of Systems and Software*, Elsevier Inc., v. 83, n. 6, p. 906–914, jun. 2010. ISSN 01641212.

SCHILIT, B.; THEIMER, M. Disseminating active map information to mobile hosts. *Network, IEEE*, v. 8, n. 5, p. 22 –32, sept.-oct. 1994. ISSN 0890-8044.

SERRAL, E.; VALDERAS, P.; PELECHANO, V. Towards the model driven development of context-aware pervasive systems. *Pervasive and Mobile Computing*, Elsevier, v. 6, n. 2, p. 254–280, 2010.

SIEWE, F.; ZEDAN, H.; CAU, A. The calculus of context-aware ambients. *J. Comput. Syst. Sci.*, Academic Press, Inc., Orlando, FL, USA, v. 77, n. 4, p. 597–620, jul. 2011. ISSN 0022-0000. Disponível em: <http://dx.doi.org/10.1016/j.jcss.2010.02.003>.

SOUZA, M. d. F. C. d.; CASTRO FILHO, J. A. d.; ANDRADE, R. M. C. Model-Driven Development in the Production of Customizable Learning Objects. *2010 10th IEEE International Conference on Advanced Learning Technologies*, Ieee, p. 701–702, jul. 2010.

VIANA, W. *Mobilité et sensibilité au contexte pour la gestion de documents multimédias personnels: CoMMediA*. Tese (Doutorado) — University of Grenoble, 2010.

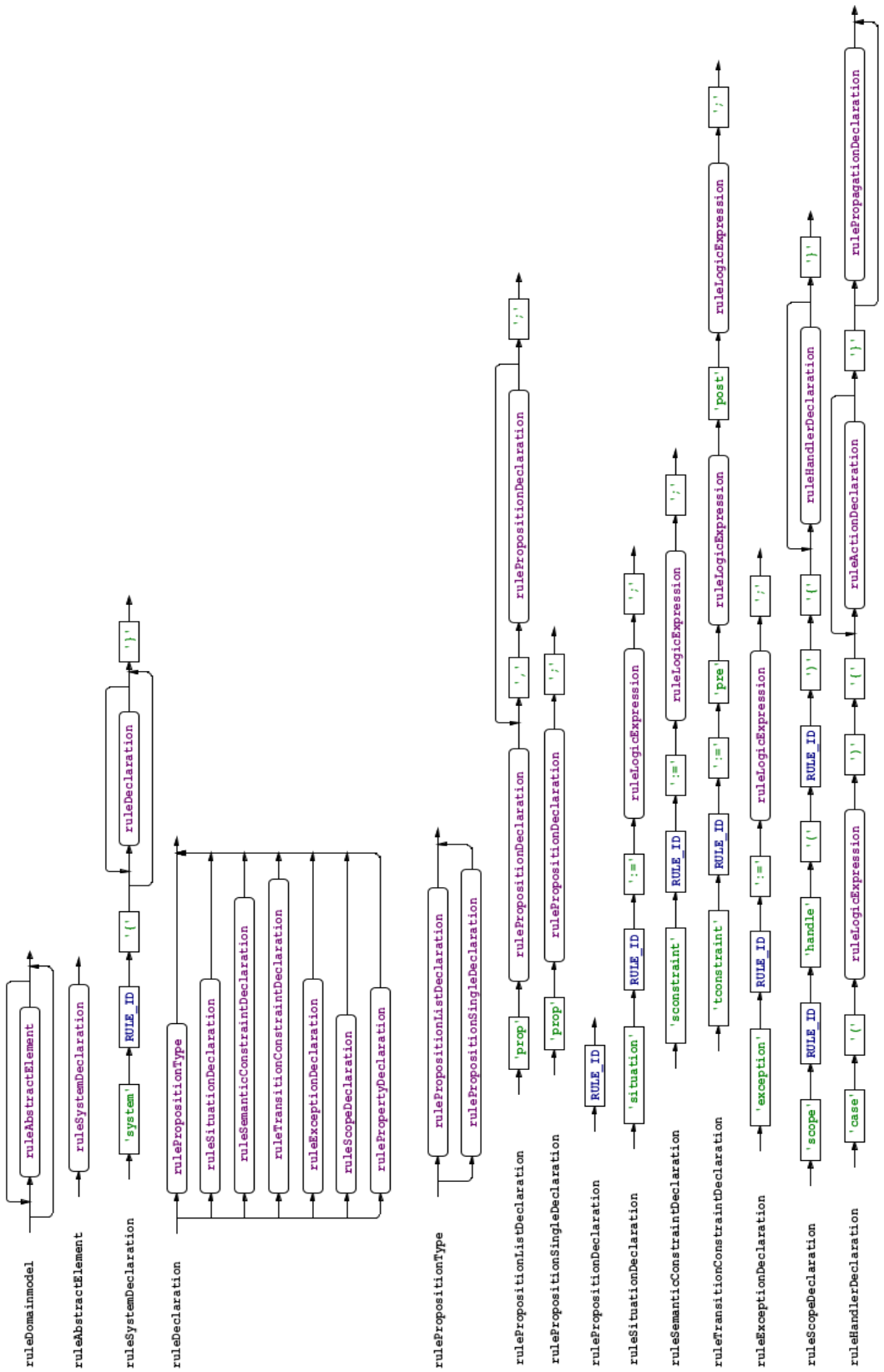
VIANA, W.; FILHO, J. B.; GENSEL, J.; VILLANOVA-OLIVER, M.; MARTIN, H. Aide au développement et au déploiement d applications mobiles et sensibles au contexte : l architecture commedia. *Actes de l'atelier ERTSI*, INFORSID 2009, v. 1, p. 150–166, 2009.

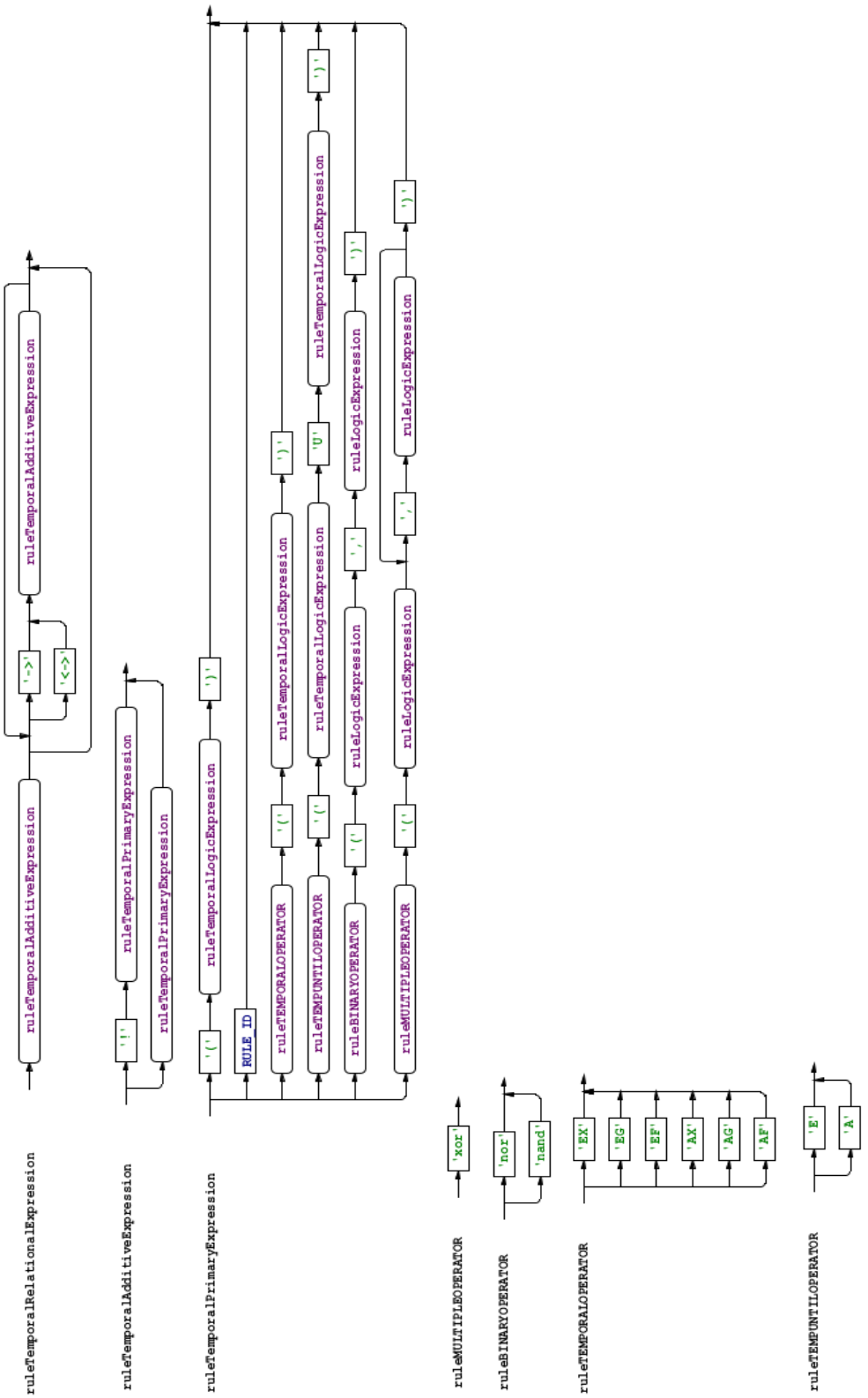
VOELTER, M.; BENZ, S.; DIETRICH, C.; ENGELMANN, B.; KATS, L.; HELANDER, M.; VISSER, E.; WACHSMUTH, G. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. [S.l.: s.n.], 2013.

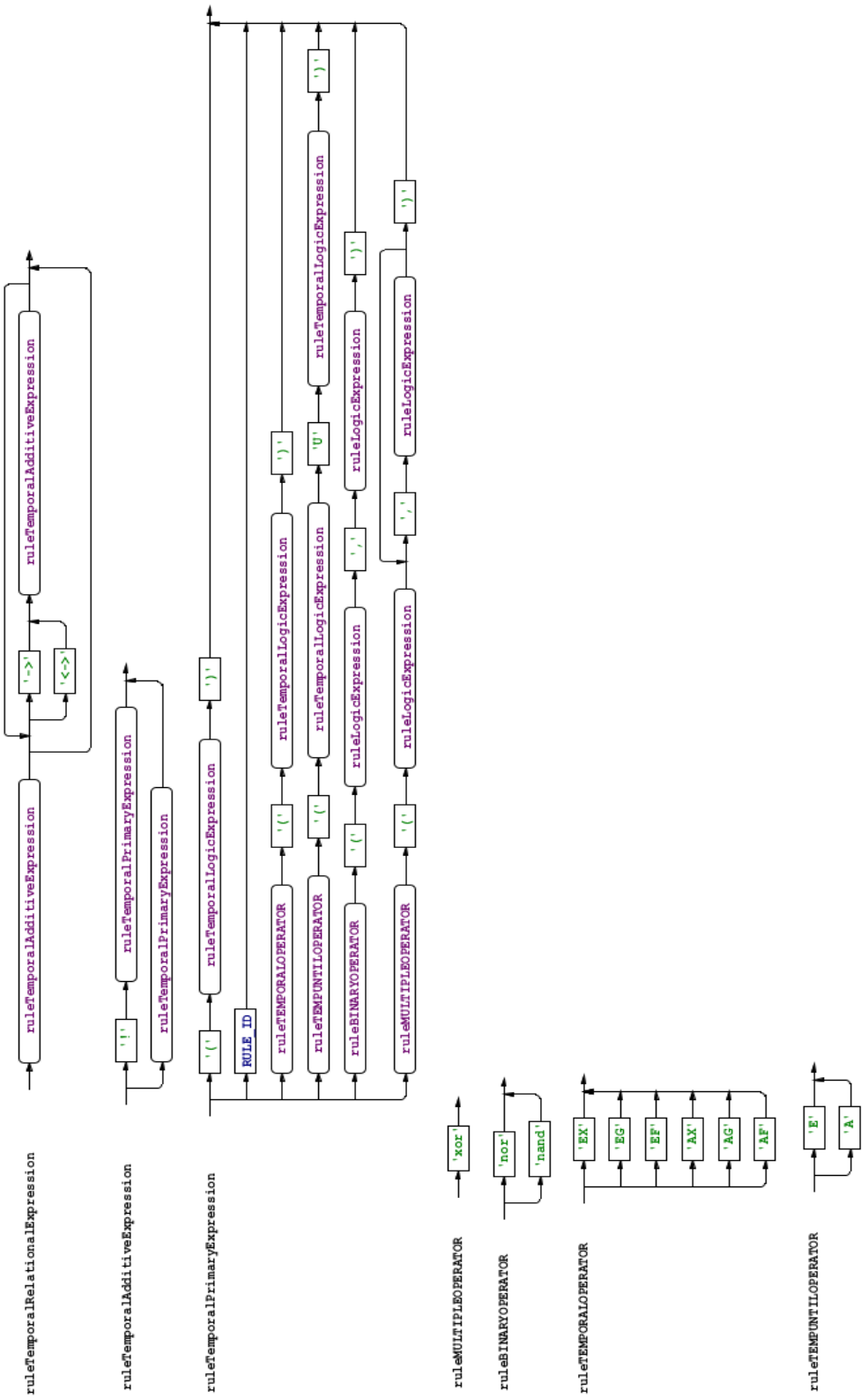
WEISER, M. The computer for the 21st century. *Scientific American*, v. 265, n. 3, p. 94–104, September 1991.

YIN, R. K. *Case study research: Design and methods*. [S.l.]: sage, 2003.

APÊNDICE A – GRAMÁTICA DA LINGUAGEM CATCHML




ruleTemporalRelationalExpression
ruleTemporalAdditiveExpression
'->'
'<->'
ruleTemporalAdditiveExpression
ruleTemporalAdditiveExpression
'!'
ruleTemporalPrimaryExpression
ruleTemporalPrimaryExpression
ruleTemporalPrimaryExpression
'('
ruleTemporalLogicExpression
')'
RULE ID
ruleTEMPORALOPERATOR
'('
ruleTemporalLogicExpression
')'
ruleTEMPUNTILOPERATOR
'('
ruleTemporalLogicExpression
'U'
ruleTemporalLogicExpression
')'
ruleBINARYOPERATOR
'('
ruleLogicExpression
','
ruleLogicExpression
')'
ruleMULTIPLEOPERATOR
'('
ruleLogicExpression
','
ruleLogicExpression
')'
ruleMULTIPLEOPERATOR
'xor'
ruleBINARYOPERATOR
'nor'
'nand'
ruleTEMPORALOPERATOR
'EX'
'EG'
'EF'
'AX'
'AG'
'AF'
ruleTEMPUNTILOPERATOR
'E'
'A'


ruleTemporalRelationalExpression
ruleTemporalAdditiveExpression
'->'
'<->'
ruleTemporalAdditiveExpression
ruleTemporalAdditiveExpression
'!'
ruleTemporalPrimaryExpression
ruleTemporalPrimaryExpression
ruleTemporalPrimaryExpression
'('
ruleTemporalLogicExpression
')'
RULE ID
ruleTEMPORALOPERATOR
'('
ruleTemporalLogicExpression
')'
ruleTEMPUNTILOPERATOR
'('
ruleTemporalLogicExpression
'U'
ruleTemporalLogicExpression
')'
ruleBINARYOPERATOR
'('
ruleLogicExpression
','
ruleLogicExpression
')'
ruleMULTIPLEOPERATOR
'('
ruleLogicExpression
','
ruleLogicExpression
')'
ruleMULTIPLEOPERATOR
'xor'
ruleBINARYOPERATOR
'nor'
'nand'
ruleTEMPORALOPERATOR
'EX'
'EG'
'EF'
'AX'
'AG'
'AF'
ruleTEMPUNTILOPERATOR
'E'
'A'

APÊNDICE B – QUESTIONÁRIO PRÉ-EXPERIMENTO DO ESTUDO DE CASO PARA AVALIAÇÃO DA LINGUAGEM CATCHML

Nome:

Q1. Qual a sua experiência em tratamento de exceção de sistemas tradicionais?

a) Não conheço esta técnica.

b) Conheço esta técnica, porém nunca projetei ou desenvolvi sistemas aplicando-a.

c) Já projetei ou desenvolvi sistemas aplicando esta técnica por até 1 ano.

d) Já projetei ou desenvolvi sistemas aplicando esta técnica no período entre 1 e 3 anos.

e) Já projetei ou desenvolvi sistemas aplicando esta técnica por mais de 3 anos.

Q2. Qual a sua experiência em tratamento de exceção sensível ao contexto?

a) Não conheço esta técnica.

b) Conheço esta técnica, porém nunca projetei ou desenvolvi sistemas aplicando-a.

c) Já projetei ou desenvolvi sistemas aplicando esta técnica por até 1 ano.

d) Já projetei ou desenvolvi sistemas aplicando esta técnica no período entre 1 e 3 anos.

e) Já projetei ou desenvolvi sistemas aplicando esta técnica por mais de 3 anos.

Q3. Qual a sua experiência em desenvolvimento de programas utilizando a linguagem de programação Java?

a) Não conheço esta linguagem.

b) Conheço esta linguagem, porém nunca projetei ou desenvolvi sistemas utilizando-a.

c) Já projetei ou desenvolvi sistemas utilizando esta linguagem por até 1 ano.

d) Já projetei ou desenvolvi sistemas utilizando esta linguagem no período entre 1 e 3 anos.

e) Já projetei ou desenvolvi sistemas utilizando esta linguagem por mais de 3 anos.

Q4. Qual a sua experiência com relação à lógica proposicional?

a) Não conheço lógica proposicional.

b) Conheço **o básico** de lógica proposicional (operadores AND, OR e NOT)

c) Conheço **mais do que o básico** de lógica proposicional (operadores "se e somente se", "se-então")

d) Conheço **bastante** de lógica proposicional (operadores NAND, NOR, XOR)

e) Conheço **além** da lógica proposicional (operadores "para todo", "existe")

Q5. Qual a sua experiência em desenvolvimento de programas utilizando a plataforma Eclipse?

a) Não conheço esta IDE.

b) Conheço esta IDE, porém nunca projetei ou desenvolvi sistemas utilizando-a.

c) Já projetei ou desenvolvi sistemas utilizando esta IDE por até 1 ano.

d) Já projetei ou desenvolvi sistemas utilizando esta IDE no período entre 1 e 3 anos.

e) Já projetei ou desenvolvi sistemas utilizando esta IDE por mais de 3 anos.

Q6. Grau de escolaridade:

a) Graduando

b) Graduado

c) Cursando especialização

d) Especialista

e) Mestrando

f) Mestre

g) Doutorando

h) Doutor

Q6. Qual a sua profissão atual?

APÊNDICE C – TESTE DE COMPREENSÃO DA ANÁLISE DE *FEEDBACK* DA LINGUAGEM CATCHML

Nome:

Modelo:

Q1. Quantos tipos de erros você encontrou antes de executar o verificador?

a) Nenhum.

b) Um(1).

c) Dois(2).

d) Três(3).

e) Quatro(4).

Q2. Quantos tips de erros você encontrou após executar o verificador?

a) Nenhum.

b) Um(1).

c) Dois(2).

d) Três(3).

e) Quatro(4).

Q3. Qual a dificuldade para entender e localizar os erros após executar o verificador?

a) Alta

b) Média/Alta

c) Média

d) Média/Baixa

e) Baixa

Q4. Qual modelo você verificou?

a) *UbiParking* 1

b) *UbiParking* 2

Q5. Selecione os tipos de erros de projeto que foram reportados pela ferramenta: *

a) Exceção morta (*Dead exception*)

b) Tratador nulo (*Null handler*)

c) Tratador morto (*Dead handler*)

d) Tratamento cíclico (*Cyclic handling*)

e) Retomada impossível (*Resume impossible*)

Q6. Você conseguiu corrigir os erros reportados e obteve uma verificação livre de erros?

a) Sim, corrigi todos os erros.

b) Não, mas corrigi alguns erros.

c) Não consegui corrigir os erros.

Q7. Qual a sua opinião sobre a forma como os erros são apresentados na ferramenta?

Q8. Você tem alguma sugestão ou crítica em relação à verificação de erros?

APÊNDICE D – QUESTIONÁRIO PÓS-EXPERIMENTO DO ESTUDO DE CASO PARA AVALIAÇÃO DA LINGUAGEM CATCHML

Nome:

Q1. Eu tive tempo suficiente para executar as tarefas.

a) Concordo Fortemente

b) Concordo

c) Neutro

d) Discordo

e) Discordo Fortemente

Q2. Os objetivos das tarefas foram perfeitamente claros para mim.

a) Concordo Fortemente

b) Concordo

c) Neutro

d) Discordo

e) Discordo Fortemente

Q3. As tarefas que eu executei foram perfeitamente claras para mim?

a) Concordo Fortemente

b) Concordo

c) Neutro

d) Discordo

e) Discordo Fortemente

04. Como você avalia as abstrações utilizadas na linguagem para representar os conceitos de tratamento de exceção sensível ao contexto?

a) Excelente

b) Bom

c) Regular

d) Ruim

e) Péssimo

05. Como você avalia o *feedback* em relação aos erros fornecido pela ferramenta?

a) Excelente

b) Bom

c) Regular

d) Ruim

e) Péssimo

06. Você tem alguma sugestão, elogio ou crítica em relação à ferramenta?