Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Usando MDA e MDT para Modelagem e Geração Automática de Arquiteturas de Teste para Sistemas de Tempo Real

Everton Leandro Galdino Alves

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Patrícia D. L. Machado
Franklin S. Ramalho
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

A474u Alves, Everton Leandro Galdino.
Usando MDA e MDT para modelagem e geração automática de arquitetura de teste para sistemas de tempo real / Everton Leandro Galdino Alves. — Campina Grande, 2011.
173 f.: il. col.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientadores: Profª. Drª. Patrícia D. L. Machado, Prof. Dr. Franklin S. Ramalho.

1. Testes. 2. Modelagem. 3. Tempo Real. 4. MDA. 5. MDT. I. Título.

CDU - 004.415.53(043)

"USANDO MDA E MDT PARA MODELAGEM E GERAÇÃO AUTOMÁTICA DE ARQUITETURAS DE TESTE PARA SISTEMAS DE TEMPO REAL"

EVERTON LEANDRO GALDINO ALVES

DISSERTAÇÃO APROVADA EM 30.06.2011

PATRICIA DUARTE DE LIMA MACHADO, Ph.D
Orientador(a)

FRANKLIN DE SOUZA RAMALHO, Dr.
Orientador(a)

HYGGO OLIVEIRA DE ALMEIDA, D.Sc
Examinador(a)

ROBERTA DE SOUZA COELHO, Drª
Examinador(a)

CAMPINA GRANDE - PB

Resumo

As atividades de projeto e teste de sistemas de tempo real (STR) na atualidade ainda são extremamente desafiadoras, sendo a qualidade das mesmas diretamente proporcionais ao quão experientes são seus respectivos projetistas e testadores. Isto de deve ao fato desta classe de sistemas computacionais possuir um conjunto de características diferenciadas (*e.g.* restrições de tempo e manipulação de interrupções) que aumentam a complexidade em desenvolvê-los. Abordagens como *Model-Driven Architecture* (MDA) e *Model-Driven Testing* (MDT) têm procurado trazer os modelos ao centro dos processos de desenvolvimento e de teste, respectivamente, com o intuito de gerar automaticamente artefatos de software em diferentes níveis de abstração (*e.g.* código e casos de teste). Porém, diversas lacunas ainda impedem que estas promissoras abordagens também possam ser usadas em sua plenitude no contexto dos sistemas de tempo real. Desta forma, este trabalho objetiva fornecer o embasamento necessário para permitir o uso destas abordagens neste contexto. Para tal, este trabalho apresenta os seguintes resultados: i) um conjunto de diretrizes para modelagem de STR usando UML, construído a fim de ajudar projetistas inexperientes a desenvolver mais facilmente modelos de STRs mais expressivos; ii) um conjunto de extensões ao perfil de testes da UML (UTP), com o intuito de dotá-lo com os mecanismos necessários para a construção de arquiteturas de teste para STRs no nível independente de plataforma; iii) um mapeamento informal dos elementos do perfil de testes estendido para uma plataforma específica para STRs (linguagem C para o FreeRTOS); iv) um auxílio ferramental para proporcionar a geração automática das arquiteturas de teste nos níveis independente e específico de plataforma. Para avaliar os resultados obtidos, estudos de caso foram aplicados, tanto para o conjunto de diretrizes de modelagem, quanto para as extensões/mapeamento.

Abstract

Nowadays, the activities of project and test of real-time systems (RTS) are still considered extremely challenging. The quality of this activities are direct proporcional to how experientes are the designers and testers who participates of the process. This fact is mainly because this special computacional systems have a set of differentiated features (*e.g.* time restrictions and interruptions manipulation) which increase the complexity needed to develop them. Approaches such *Model-Driven Architecture* (MDA) and *Model-Driven Testing* (MDT) are trying to increase the importance of putting models into the center of the development and test process (*e.g.* test case). However, many gaps still impede that this approaches became fully used in the context of real-time systems. Therefore, this work has the goal of providing the basement to allow the use of those approaches in this context. As results, we propose: i) a set of guidelines to modeling RTS using UML; ii) a set of extensions to the UML testing profile (UTP) extended, with the intention of including mechanisms to build test real-time architectures at independent platform level; iii) an informal mapping from UTP elements to a specific RTS platform (C language and FreeRTOS OS); and iv) an auxiliary tool to automatically generate test architectures at different abstractions levels. To evaluate the results, a set of case studies were realized.

Agradecimentos

Agradeço primeiramente a Deus, que sustentou e guiou todos meus passos até o presente momento e fornecendo as forças necessárias para nunca desistir.

Aos meus pais (Edvaldo e Eliene), irmão (Emanuel) e amigos, os quais participaram efetivamente de toda minha trajetória, oferecendo o suporte emocional e afetivo necessário para que eu pudesse vencer os inúmeros obstáculos ocorridos durante esta desafiadora etapa da minha vida.

Sou muito grato aos meus orientadores, a professora Patricia D. L. Machado e o professor Franklin S. Ramalho. Sou grato pela dedicação, paciência, orientação e total disponibilidade destes para transmitir um pouco do muito que sabem através dos seus preciosos conselhos e *feedbacks*.

Aos professores e demais funcionários do curso de Pós-Graduação em Ciência da Computação da UFCG, que contribuíram direta ou indiretamente para a boa condução da pesquisa realizada e para meu crescimento como aluno e pesquisador.

Aos meus colegas e amigos da UFCG que me ajudaram durante toda essa trajetória, sempre dando a força e o incentivo nos momentos mais necessários. Agradeço aos meus companheiros de laboratório, em especial a Diego Tavares, Anderson Ledo, Natã Melo e Andreza Vieira, pela amizade e constante colaboração. Sou grato especialmente a Augusto Macedo pelo seu esforço e prontidão em me auxiliar por diversos momentos durante a execução deste trabalho.

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de nível Superior (CAPES), pelo apoio financeiro.

Conteúdo

Lista de Símbolos

AFCT - *Automatic Functional Component Testing*

ATL - *Atlas Transformation Language*

CIM - *Computacional Independent Models*

CITM - *Computacional Independent Testing Models*

GQM - *Goal, Question, Metric*

LTS - *Labeled Transition Systems*

MBT - *Model-based Testing*

MDA - *Model Driven Architecture*

MDD - *Desenvolvimento Dirigido por Modelos*

MDE - *Model Driven Engineering*

MDT - *Model-Driven Testing*

MOF - *Meta Object Facility*

OCL - *Object Constraint Language*

OMG - *Object Management Group*

PIM - *Plataform Independent Models*

PITM - *Plataform Independent Testing Models*

PSM - *Plataform Specific Models*

PSTM - *Plataform Specific Testing Models*

QVT - *Query/View/Transformation*

SO - *Sistema Operacional*

SPACES - *SPecification bAsed Component tESter*

SUT - *System Under Test*

RT - *Real Time*

STRs - *Sistemas de Tempo Real*

RTTAG - *Real-Time Test Architecture Generator*

TA - *Timed Automata*

TAIO - *Timed Automata with Inputs and Outputs*

TIOLTS - *Timed Input-Output Labelled Transition Systems*

TLTS - *Timed Labelled Transition Systems*

TTCN-3 - *Testing and Test Control Notation version 3*

UFCG - *Universidade Federal de Campina Grande*

UML - *Unified Modeling Language*

UTP - *UML Testing Profile*

XML - *Extensible Markup Language*

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Na atualidade, um grupo especial de aplicações têm ganhado bastante espaço, crescido em número de utilizadores e em nível de complexidade de recursos, são os chamados Sistemas de Tempo Real (*Real-Time Systems*) - STR [51]. STRs são sistemas computacionais que, além de precisar cumprir os deveres comuns a qualquer software (executar corretamente os requisitos especificados pelo cliente), também devem cumprir estes requisitos dentro de certas restrições de tempo (*deadlines*). Outra característica importante quando trata-se de STRs é o conceito de interrupções. STRs de um modo geral são sistemas reativos onde eventos assíncronos (interrupções) podem acontecer a qualquer momento e o sistema deve ser capaz de tratá-los adequadamente. Estas características especiais dos STRs tornam seu desenvolvimento diferenciado e ainda mais complexo que o desenvolvimento dos softwares tradicionais, pois, cuidados especiais precisam ser tomados para que o cumprimento das restrições de tempo e o tratamento das interrupções aconteça adequadamente. É importante também que estes cuidados sejam levados em consideração durante todo o processo de desenvolvimento.

Os STRs estão hoje inclusos em uma grande variedade de contextos, desde em aparelhos domésticos simplificados (*e.g.* forno microondas e máquinas de lavar) até em sistemas mais complexos e críticos (*e.g.* controladores de voo e monitoramento hospitalar). Devido a tal abrangência, muitos são os estudos para evolução desse tipo de sistema, bem como para o desenvolvimento de ferramentas e técnicas que os aprimorem.

Dada a complexidade dos mesmos, bem como o seu alto custo de desenvolvimento, durante muitas décadas os principais consumidores dos STRs foram os militares (estes, nor-

malmente dispunham de recursos suficientes para construção dos seus sistemas complexos e críticos). Porém, com a evolução dos estudos na área, e a significativa diminuição dos custos de hardware, tornou-se viável que muitas empresas pudessem possuir/produzir sistemas (e produtos) de tempo real e, com isto, que esta área crescesse e se popularizasse. Contudo, ainda hoje é fato que a engenharia de software de tempo real exige habilidades especiais.

No contexto geral, a busca por desenvolver softwares mais robustos, bem como atribuir um maior padrão de qualidade ao produto desenvolvido, tem feito com que a academia busque novos e mais eficientes mecanismos que favoreçam o desenvolvimento de softwares de qualidade.

Paralelamente, nos últimos anos a atividade de teste do software, que por muito tempo foi realizada com poucos cuidados, ou até mesmo não executada, tem recebido maiores atenções, sendo atualmente uma das principais atividades do ciclo de desenvolvimento dos softwares e parte central do processo de Verificação e Validação (V & V). Em especial quando trata-se de sistemas com alto nível de complexidade, e/ou criticidade (*e.g.* STRs), a atividade de testes ganha ainda mais importância, pois defeitos que possam passar despercebidos durante o desenvolvimento podem acarretar problemas graves, tais como danos físicos e materiais ao cliente final ou grandes perdas financeiras para a empresa desenvolvedora do software.

Outra atividade que tem cada vez mais ganhado importância na execução do processo de desenvolvimento de softwares é a atividade de modelagem. Modelar um sistema é uma atividade composta de diversas tarefas relacionadas, envolvendo desde o estudo inicial do sistema a ser modelado, até a apresentação dos resultados obtidos com o modelo para os devidos fins [67]. Este processo tem por objetivo permitir o entendimento por completo da estrutura e do comportamento do que deve ser construído, bem como permitir que evoluções futuras e/ou manutenções do software possam ser realizadas sem maiores dificuldades. Existem diversas maneiras possíveis para modelar um software, estas distinguindo entre si pelo nível de abstração empregado e/ou complexidade de uso. Quanto aos formalismos e notações usadas para modelagem, as opções variam desde representações matemáticas formais até a utilização de textos em linguagem natural para descrever o sistema. Porém, modelagem utilizando componentes gráficos é a forma mais comum no meio computacional para descrever sistemas de software. Dentre as notações gráficas existentes, a mais famosa e mais utilizada,

principalmente para modelagem de softwares orientados a objeto, é a UML (*Unified Modeling Language*) [2].

A UML, que hoje encontra-se na sua versão 2.1, foi desenvolvida e é mantida pela OMG (*Object Management Group*) [34], um consórcio formado por um grupo de conceituadas empresas e universidades que tem como objetivo aprovar padrões abertos para aplicações orientadas a objetos. A UML na sua versão atual é composta por treze diferentes diagramas que objetivam possibilitar a modelagem de toda e qualquer faceta de um software, permitindo a caracterização desde a fase de levantamento dos requisitos até a modelagem do modo como o sistema deve ser instalado no seu ambiente organizacional.

Assim como para qualquer sistema computacional, para os STRs também existe a necessidade de que as atividades de modelagem e testes sejam bem executadas. Esta necessidade fica ainda mais evidente devido à complexidade que envolve o desenvolvimento deste tipo de sistema. É usualmente difícil construir um código que atenda precisamente aos requisitos de tempo. Desta forma, a atividade de modelagem tem sido considerada fundamental no projeto de STRs. Caso a fase de projeto/modelagem não tenha sido bem executada e, consequentemente o sistema não esteja claro o suficiente para os desenvolvedores, mais do que no desenvolvimento de qualquer outro tipo de software, falhas poderão ser incorporadas comprometendo o resultado final. Da mesma forma, mesmo com a construção de modelos precisos, falhas podem ser incorporadas durante a codificação do sistema (principalmente no código *real-time*), não descartando assim a necessidade do uso de testes.

Seguindo com a idéia de aumentar a importância da modelagem no processo de desenvolvimento de software, a OMG desenvolveu a MDA (*Model Driven Architecture*) [45]. MDA é uma arquitetura que guia o processo de desenvolvimento, cujos elementos centrais, ao invés de codificação, são os modelos e as transformações entre modelos. Ou seja, MDA apregoa a ideia que o foco do desenvolvimento de software esteja na construção dos modelos e das regras de transformação, e que, de posse desses artefatos e do uso de engenhos de transformação, o código final do software possa ser gerado automaticamente. Outro ponto importante do uso de MDA é a possibilidade da modelagem do sistema, e consequentemente a construção do próprio sistema, em diferentes níveis de abstração e sem a definição imediata de uma única plataforma de uso. Possibilitando assim que posteriormente possa haver uma migração de plataforma, caso necessário, sem maiores dificuldades, necessitando apenas que

novas regras de transformação sejam construídas.

Já há algum tempo, a academia tem direcionado esforços para integrar a atividade de testes com a de modelagem. Dentre as abordagens que têm obtido bons resultados, destaca-se a conhecida por MBT (*Model-based Testing*) [26]. MBT é uma abordagem para geração automática de casos de teste a partir dos modelos de desenvolvimento. Essencialmente, os modelos usados para essa geração são aqueles relacionados à especificação dos requisitos funcionais do sistema. Os testes gerados servem para averiguar se a implementação está se comportando de acordo com o especificado nos modelos projetados anteriormente. Dentre as vantagens da utilização de MBT têm-se, dentre outras: i) a diminuição do tempo gasto para geração dos testes; ii) a efetividade dos testes gerados; e iii) a possibilidade da reflexão automática de mudanças de requisitos (consequentemente mudanças nos modelos) nos testes. Devido a tais características, muitas são as utilizações de sucesso de MBT dentre elas: Barbosa *et al.* [11], Mingsong *et al.* [55] e Rumpe [64].

Uma das grandes dificuldades encontradas para a realização efetiva da atividade de testes é a necessidade de uma infra-estrutura para a construção, realização e execução dos mesmos. Pensando nesta necessidade, bem como na utilização dos conceitos promissores e da eficiência que MDA apregoa, surgiu uma realização de MBT voltada para a utilização dos princípios de MDA para geração de artefatos de teste, é a chamada MDT (*Model-Driven Testing*) [9]. MDT consiste na idéia que, a partir dos modelos de desenvolvimento e de regras de transformação, haja a geração automática de todos os artefatos necessários para realização da atividade de testes, desde uma arquitetura que permita a execução dos testes, até os casos de teste propriamente ditos. Este conjunto de artefatos pode ser definido em diferentes níveis de abstração, bem como ser independente de uma plataforma específica [50]. Buscando proporcionar a aplicação de MDT, a OMG definiu um perfil UML voltado para projeto de arquiteturas de teste, o *UML Testing Profile* - UTP [9]. Este perfil, apesar de ser o mais utilizado no contexto de modelagens de arquiteturas de teste no nível independente de abstração, ainda peca no sentido de permitir a modelagem completa de ambientes de teste mais específicos, como no caso dos STRs.

Assim como qualquer outro sistema computacional, os STRs devem ser modelados e testados. Porém, a modelagem deste tipo de sistema deve ser realizada de uma forma ainda mais cuidadosa pelo fato que suas características especiais (restrições de tempo e eventos assín-

cronos) também devem estar explícitas nos modelos. Esta nem sempre é uma tarefa simples, pois a maioria das notações usadas para modelagem não conseguem, ou pelo menos não de uma forma simplificada, permitir a representação conjunta da estrutura e do comportamento do sistema deixando claras suas necessidades.

Quanto aos testes em STRs, recentemente alguns pesquisadores têm começado a investigar o uso de MBT nos testes desse tipo de sistema, buscando assim que vantagens importantes que MBT fornece, tais como a aquisição antecipada dos casos de tese (gerados a partir dos modelos), também possam ser alcançadas nos STRs [6; 7]. Porém, até onde conhecemos, não existe nenhum trabalho que aplique MDT neste contexto. Considerando que no contexto de teste dos STRs uma das tarefas mais árduas se refere à montagem e definição da arquitetura de testes para os mesmos, acreditamos então que estudos neste sentido podem proporcionar resultados positivos para esta problemática.

Um dos fatores importantes a serem considerados a respeito da área dos STRs é que, até o momento, não foi estabelecido um padrão para uso de um único formalismo/notação, nem mesmo há uma convergência na forma de como modelá-los. Cada novo trabalho a respeito, sugere uma nova forma de modelagem (ou um novo modelo para uso). O mesmo fato acontece na condução da atividade de testes. Portanto, existe uma carência, principalmente para projetistas iniciantes, de uma ajuda a fim de guiá-los na árdua tarefa de modelagem de STRs.

Quanto à área de testes, existem na literatura diversas proposições de mecanismos para geração de casos de teste, inclusive sobre modelos. Contudo, até onde conhecemos, não há nenhum trabalho que identifique quais os elementos necessários para configuração e realização da atividade de testes (*e.g.* manipuladores de tempo, aferidores de vereditos, etc.) no âmbito dos STRs. Existe então a necessidade da definição de um contexto, uma arquitetura, bem como de uma ferramenta que auxilie o testador na realização da atividade de testes para esse tipo de sistema.

1.1 Objetivo do Trabalho

O objetivo central desse trabalho foi investigar e desenvolver mecanismos para apoiar a execução das atividades de modelagem e teste de STRs. No tocante à modelagem, procurou-se definir diretivas para auxiliar o projetista/testador durante as diferentes etapas e decisões

de modelagem. Quanto aos testes, definir os elementos necessários para a construção de arquiteturas de teste eficientes, bem como proporcionar, quando possível, a automação da geração dos artefatos necessários para a condução da atividade de testes de forma organizada e eficiente. É sabido que a qualidade dos modelos é um dos requisitos básicos para a geração de casos de teste efetivos em abordagens de teste baseadas em modelos. Portanto, é importante que modelagem e geração de testes sejam investigados em conjunto.

O escopo do trabalho se restringiu à modelagem e teste de STRs reativos e não críticos (*soft real-time systems*) em um alto nível de abstração. Para tal, fizemos uso da linguagem UML e de técnicas de MDT no contexto de teste de integração (utilizando o perfil de testes da UML - UTP - *UML Testing Profile* [9]). Neste sentido, os objetivos específicos do trabalho foram os seguintes:

1. Definição de direcionamentos/diretrizes para a modelagem de STRs usando UML. Estas diretrizes explicitam desde quais diagramas os projetistas devem usar e em que momento, bem como propor uma forma padronizada para modelagem de características importantes, como: restrições de tempo e o comportamento dos eventos assíncronos possíveis de acontecer.

2. Adaptação do perfil de testes (UTP) para o contexto RT (*Real-Time*). Para tal, foi necessário definir os elementos essenciais de uma arquitetura de testes para STRs no nível independente de plataforma, nível PITM (*Plataform Independent Testing Models*).

3. Criação do metamodelo da linguagem C. Este metamodelo é essencial para a viabilização do uso de MDT no contexto da maioria das plataformas *real-time* existentes, inclusive na plataforma específica escolhida para uso neste trabalho (FreeRTOS).

4. Definição de um mapeamento entre os elementos de teste do nível PITM para uma plataforma *real-time* (FreeRTOS), na linguagem C. Esta plataforma foi escolhida por ser amplamente utilizada no contexto de desenvolvimento de STRs, devido a sua simplicidade e por ser de código aberto.

5. Desenvolvimento de uma ferramenta que aplique os resultados atingidos pelos objetivos anteriores da seguinte forma: usando os princípios de MDT, consiga gerar auto-

maticamente, a partir de modelos UML (modelados seguindo as diretrizes propostas pelo resultado do primeiro objetivo), a arquitetura de testes, no nível independente de plataforma, para o STR. Em uma segunda etapa, consiga, a partir dos modelos da arquitetura de testes no nível PITM, gerar os modelos da mesma arquitetura no nível PSTM (*Plataform Specific Testing Models*), especificamente para a plataforma FreeRTOS, modelos C.

Na literatura, existem trabalhos que questionam a possibilidade da completa modelagem de STRs utilizando somente UML como notação [14]. Porém, graças as melhorias incluídas nas suas versões recentes, onde novos e robustos mecanismos foram introduzidos na linguagem, bem como com a permissividade que o uso de perfis proporciona, acreditamos que hoje seja viável a utilização de UML neste contexto. Como resultado do primeiro dos objetivos específicos elencados anteriormente procurou-se atestar que é possível o uso de UML neste contexto, sendo resultado desta etapa, inclusive, a definição de padrões (boas práticas) de modelagem desse tipo de sistema que auxiliem seus construtores.

Entendido como construir os modelos de *design*, o segundo objetivo específico foi definido a fim de analisar o contexto dos STRs e com isso, identificar os elementos e estruturas que são importantes para a completa especificação, manipulação e execução da atividade de testes nestes sistemas. Este levantamento foi realizado observando o perfil de testes da UML (UTP). Optou-se então por estender este perfil, visto que é notório que o mesmo pouco se preocupa em dar suporte ao teste de STRs. Portanto, com o cumprimento do segundo objetivo específico, tornou-se possível construir uma arquitetura de testes completa para STRs, arquitetura esta num nível independente de plataforma e condizente com os conceitos de MDA.

A plataforma escolhida para construção e execução do STR foi o sistema operacional FreeRTOS [13]. Esta é uma plataforma amplamente usada e com uma comunidade de desenvolvedores crescente e ativa. O FreeRTOS tem como característica principal ser uma plataforma que requisita que seus programas sejam escritos na linguagem C. Sabendo disto, e entendendo que iríamos necessitar desenvolver regras de transformação entre modelos (uma das características da aplicação de MDA/MDT), observamos a necessidade da existência de metamodelos correspondentes para as linguagens a serem transformadas. Como resultado do terceiro objetivo específico do trabalho, foi construído o metamodelo para a linguagem

C. Ao contrário de UML (linguagem de partida das transformações), não foi encontrado na literatura um metamodelo usável para C, por isso houve a necessidade de construção do mesmo.

Como cada plataforma possui suas próprias peculiaridades, é necessário então mapear os conceitos definidos na arquitetura de testes do nível independente de plataforma, para a plataforma específica. Para tal, como resultado do quarto dos objetivos específicos, regras informais de mapeamento foram construídas para indicar como transformar os modelos PITM para modelos C segundo a plataforma FreeRTOS (modelos no nível PSTM).

Para aplicação automatizada dos conceitos e elementos definidos, o último dos objetivos específicos buscou desenvolver uma ferramenta de apoio, seguindo os princípios de MDT. Esta ferramenta possui dois módulos distintos, estes focando na transformação de modelos em dois níveis de abstração: i) independente de plataforma (transformação PIM-PITM), não incluindo qualquer informação acerca da plataforma ou detalhes de implementação do sistema ou dos testes; e ii) específico de plataforma (transformação PITM-PSTM), que inclui as características específicas da plataforma escolhida (FreeRTOS).

1.2 Contribuições

Como resultado deste trabalho um conjunto de contribuições foram alcançadas, tanto para a área de modelagem, quanto para a área de testes de STRs:

- Um conjunto de diretrizes para o projetista, ajudando-o a conseguir modelar STRs usando UML. Assim, não será necessário utilizar mais de um formalismo/notação para o projeto completo de um STR, bem como a especificação do sistema estará mais clara visto que UML é hoje a principal linguagem de modelagem no meio computacional. Esta contribuição útil principalmente para aqueles projetistas de STRs iniciantes no tocante a produzir modelos claros e padronizados;

- A proposição de elementos para composição de uma arquitetura completa para construção e execução da atividade de testes para STRs. Com a identificação destes elementos necessários para teste, torna-se mais simples a atividade de teste como um todo, bem como empresas e desenvolvedores desse tipo de sistema atingirão um maior

grau de organização, com possibilidade de redução de custos para manutenção dos artefatos de teste;

- O metamodelo para linguagem C. É imprescindível, para aqueles que buscam utilizar os conceitos de MDA/MDT, que existam metamodelos para as linguagem de programação, é a linguagem C é uma das principais;

- O mapeamento dos elementos de teste independentes de plataforma para a plataforma FreeRTOS. Como o FreeRTOS é uma das plataformas mais usadas no contexto de utilização de STRs, este mapeamento facilitará a construção de arquiteturas de teste neste contexto. Como o FreeRTOS é muito semelhante às demais plataformas usadas no contexto dos STRs, as regras de mapeamento concebidas poderão inspirar outras traduções semelhantes, permitindo que outras plataformas também possam ser usadas em aplicações de MDT;

- Um suporte ferramental para auxílio dos testadores na aplicação da atividade de testes de STRs, conseguindo de forma automática a geração dos artefatos necessários para execução dos casos de teste, bem como a geração dos mesmos, em dois níveis diferentes de abstração (PITM, PSTM).

No tocante à modelagem, o impacto do uso de UML ao invés de outros formalismos deverá trazer uma maior clareza para comunicação entre os desenvoledores e facilitar a implementação dos requisitos diferenciados que possuem os STRs. Usando somente UML para modelagem, possivelmente, os modelos do sistema poderão mais facilmente evoluir junto com o mesmo.

Quanto aos testes, o que acontece atualmente é que, normalmente, quando trata-se de testes a nível de integração de sistemas, existe uma grande quantidade de casos de teste a serem executados dificultando assim a definição das estruturas necessárias para a execução, bem como a execução manual/individual dos mesmos. Com a geração automática, bem como com todo o suporte fornecido para execução desta atividade, espera-se obter a redução de tempo e esforço empregado. Outro fator relevante como contribuição para a atividade de testes de STRs foi a definição dos elementos necessários para a execução desta atividade (arquitetura). Até então, pouco tem sido discutido a esse respeito na literatura e a definição

desta arquitetura de testes poderá trazer grandes facilidades na forma de como esta atividade é, e será, conduzida.

1.3 Estrutura da Dissertação

Os próximos módulos deste documento estão estruturados da seguinte forma:

Capítulo 2: Fundamentação Teórica. Apresenta uma descrição de conceitos básicos necessários para compreender melhor este trabalho. Os conceitos descritos estão relacionados à STRs, MBT, MDT, MDA, UTP e FreeRTOS.

Capítulo 3: Diretrizes para Modelagem de STRs. Apresenta a descrição das diretrizes construídas com o objetivo de guiar o projetista de STRs no processo de modelagem dos mesmos usando UML. Ainda neste capítulo são apresentados os estudos de caso que foram aplicados com o objetivo de avaliar as diretrizes propostas, bem como os resultados alcançados e conclusões extraídas destes.

Capítulo 4: Extensão de UTP para STRs. Apresenta o conjunto de extensões desenvolvidas para UTP a fim de deixá-lo propício também para uso no contexto de testes de STRs. Também serão apresentados neste capítulo: como se deu a aplicação dos estudos de caso, o resultado do processo avaliativo dos estudos de caso e a demonstração de como trabalha o primeiro módulo da ferramenta RTTAG.

Capítulo 5: Mapeando Arquiteturas de Teste para o nível PSTM. Apresenta as regras de mapeamento entre os elementos da arquitetura PITM para a plataforma FreeRTOS. Ainda neste capítulo é apresentado o estudo de caso desenvolvido com o objetivo de avaliar as regras de mapeamento, bem como o segundo módulo da ferramenta RTTA é apresentado.

Capítulo 6: Trabalhos Relacionados. Este capítulo apresenta os principais trabalhos relacionados da literatura que utilizam conceitos como MBT, MDT e modelagem e testes de STRs. Ainda neste capítulo são apresentadas argumentações que diferenciem os trabalhos relacionados com o apresentado neste documento.

Capítulo 7: Conclusão e Trabalhos Futuros. Este capítulo apresenta a conclusão deste trabalho através do resumo dos resultados obtidos, bem como uma descrição das perspectivas de trabalhos futuros.

1.4 Considerações Finais do Capítulo

Este capítulo apresentou de maneira sucinta as problemáticas que motivaram a realização deste trabalho. Ficou evidenciado que o trabalho em questão irá atacar duas das grandes áreas dos processos de desenvolvimento de software (modelagem e testes) objetivando preencher lacunas existentes nas mesmas, para o contexto específico de desenvolvimento de STRs reativos e não-críticos. Ainda neste capítulo, foram apresentados os objetivos gerais e específicos do trabalho, as contribuições que a realização do mesmo proporcionou para ambas as áreas atacadas e, por fim, como se dá a composição deste documento.

Capítulo 2

Fundamentação Teórica

Neste capítulo são apresentados, de forma sucinta, os principais conceitos referentes a este trabalho, com o objetivo de fornecer um embasamento teórico ao leitor do mesmo, para que este consiga compreendê-lo em sua totalidade. Nas seções subsequentes serão apresentados os conceitos relacionados a: Sistemas de Tempo Real e *Model-Based Testing* (MBT). Em sequência, serão apresentadas abordagens dirigidas por modelos, tais como a *Model Driven Architecture* (MDA) e *Model Driven Testing* (MDT). Por fim, é apresentada a realização da OMG para proporcionar a aplicação de MDT com UML, o *UML Testing Profile* (UTP) e um sistema operacional *real-time*, o FreeRTOS.

2.1 Sistemas de Tempo Real

No contexto atual, em que a quantidade e a variedade de sistemas computacionais têm crescido a cada ano, pode-se destacar uma categoria dentre esses sistemas, são aqueles nos quais, além da exigência da corretude lógica, existe uma preocupação constante com o cumprimento de restrições de tempo, são os chamados Sistemas de Tempo Real (*Real-Time Systems* - STR [67]).

Esse tipo de sistema tem se popularizado e hoje já se encontra aplicado nas mais variadas situações, desde embutidos em sistemas simples (*e.g.* lavadoras de roupas e celulares), até gerenciando sistemas críticos (*e.g.* aparelhos hospitalares de monitoramento de pacientes e de controle de voo).

Um sistema de tempo real é um sistema de software cujo funcionamento correto depende

dos resultados produzidos por ele e do tempo em que esses resultados são produzidos [67]. Existe uma classificação que subdivide os STRs em dois grupos, segundo sua criticidade:

- *Soft real-time* (sistemas de tempo real leves): são aqueles que devem satisfazer um prazo, mas se um certo número de prazos for perdido o sistema ainda pode ser considerado operacionalmente aceitável. Ou seja, a operação sofrerá "apenas" uma *degradação* se os requisitos de tempo não forem satisfeitos.

- *Hard real-time* (sistemas de tempo real rígidos): são aqueles onde todos seus prazos devem ser cumpridos, sob pena de um resultado inaceitável ocorrer. Ou seja, a operação será *incorreta* se os requisitos de tempo não forem satisfeitos.

A programação usual, a partir de linguagens de programação de alto nível, torna-se insatisfatória para as características dos STRs. Tal fato dá-se porque, em sua maioria, as linguagens de programação possuem poucos, ou não possuem, mecanismos que garantam a implementação de restrições temporais corretamente. Outra dificuldade para a programação de STRs é que os sistemas operacionais usuais não dão a liberdade que o desenvolvedor precisa para que os mesmos gerenciem o tratamento de interrupções de tarefas e possam programar com temporizadores (os sistemas operacionais - SO - fazem esse gerenciamento transparentemente). Por isso, muitas vezes os programadores de STR abrem mão das facilidades que um SO tradicional oferece e trabalham com núcleos simplificados dos mesmos e com linguagens de programação desenvolvidas especificamente para os propósitos *Real-Time* (*e.g. Real-Time Concurrent* C [31]).

Um conceito importante para STRs é a previsibilidade. Um sistema é dito previsível quando, independentemente das variações no nível de hardware, da carga ou de falhas, o comportamento do sistema pode ser antecipado, antes da sua execução [29]. Portanto, para assumir a previsibilidade de um sistema, é necessário conhecer bem seu comportamento antecipadamente. Esta nem sempre é uma tarefa simples de se realizar devido à complexidade atrelada a esses sistemas. Para conhecimento prévio do sistema é necessário considerar características que possam interferir no comportamento e no cumprimento ou não das restrições temporais, como: carga do ambiente, possíveis falhas, arquitetura do hardware, sistema operacional, etc. Estes fatores são muitas vezes difíceis de serem medidos acrescentando ainda maior dificuldade à programação para STRs.

Outra forma de visualizar um STR é como um sistema *reativo* de estímulo/resposta. Mediante a um estímulo recebido, o sistema deve produzir uma resposta coerente. Portanto, o papel dos estímulos são de suma importância, pois, associado a cada estímulo (e a resposta do mesmo) existirão requisitos de tempo. Quanto aos estímulos, estes são classificados da seguinte forma:

- *Periódicos*: ocorrem em intervalos de tempo previsíveis.
- *Aperiódicos*: ocorrem irregularmente e, normalmente, são tratados usando os mecanismos de interrupção do computador.

O problema *Tempo Real* consiste então em especificar, verificar e implementar sistemas ou programas que, mesmo com recursos limitados, apresentam comportamentos previsíveis, atendendo às restrições temporais impostas pelo ambiente ou pelo usuário. Para tratá-lo, diversas metodologias e abordagens têm sido desenvolvidas sob diferentes contextos (*e.g.* [15] e [65]). Porém, existe ainda muito a se evoluir, principalmente no contexto de testes e modelagem desse tipo de sistemas.

2.2 MBT

As boas práticas do desenvolvimento de sistemas computacionais indicam que, antes mesmo de qualquer codificação, é necessário que haja um planejamento do que será construído. Para isso, é realizada a modelagem do sistema durante as fases iniciais do projeto, tanto para facilitar a comunicação entre os membros da equipe, quanto para garantir que o posterior processo evolutivo do software possa ser realizado sem maiores complicações, visto que modelos também fazem parte da documentação de um sistema.

Sabendo da necessidade de modelagem, e que boa parte do esforço e do tempo empregado na atividade de teste é usado na busca por compreender o que o sistema deveria fazer, a abordagem para geração de casos de teste MBT - *Model-Based Test* [26] foi desenvolvida. Esta abordagem utiliza os modelos (representações da estrutura e comportamento do sistema) para diretamente derivar os casos de teste do sistema.

Como a idéia de MBT é derivar casos de teste exclusivamente a partir dos modelos de *design*, faz-se necessário que tais modelos sejam os mais completos possíveis para que, as-

sim, a geração dos casos de teste possa ter resultados satisfatórios, ou seja, modelos pouco expressivos proporcionarão casos de teste pouco efetivos na busca por falhas. Geralmente, a geração de casos de teste a partir dos modelos é acompanhada pelo uso de ferramentas para automatizar este processo. Atualmente, já existem inúmeras ferramentas (*e.g.* [58] e [10]) e algoritmos que realizam a geração e seleção de casos de teste para diferentes formalismos de modelagem. Dentre os principais formalismos usados é possível citar: UML (*Unified Modeling Language*) [2], redes de Petri [63], LTS (*Labeled Transition Systems*)[69], *statecharts*, etc. Dada essa grande variedade de opções, cabe à equipe de testes escolher quais (formalismos e ferramentas) usar de acordo com o contexto no qual o problema está inserido.

MBT é uma abordagem bastante interessante para a atividade de teste de software, pois quando ocorre uma modificação nas especificações, a equipe de testes necessita unicamente modificar os modelos de dados para gerar um novo roteiro de testes [18].

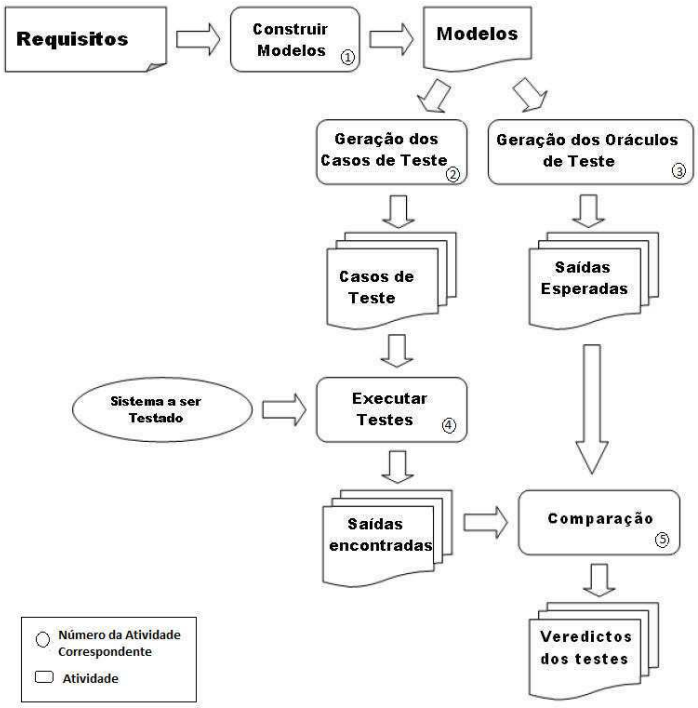

Figura 2.1: Atividades do processo de MBT.

As principais atividades relacionadas no processo de MBT são as seguintes:

1. Construir o modelo: mediante os requisitos do sistema, construir um modelo preciso do mesmo.

2. Gerar os casos de teste: casos de teste são extraídos do modelo para avaliar se o sistema está em concordância com seus requisitos.

3. Gerar oráculos de teste: criar mecanismos que serão usados para gerar o resultado esperado de um caso de teste.

4. Executar os testes: exercitar o sistema com os casos de teste gerados anteriormente.

5. Comparar resultados obtidos com os esperados: os resultados produzidos pelos oráculos serão confrontados com os resultado reais obtidos pela execução do sistema em teste.

As atividades de MBT, descritas anteriormente, bem como a ordem em que estas ocorrem, estão resumidas na Figura 2.1.

2.3 MDA

Model Driven Architecture (MDA) [45] é uma abordagem de desenvolvimento de software definido pela *Object Management Group* (OMG), para desenvolvimento de software, que segue os princípios de *Model Driven Engineering* (MDE) [43]. MDE apregoa ganhos em tempo e redução de custos no desenvolvimento dada a utilização efetiva de modelos. A idéia central é que, com a aplicação de técnicas de MDE, ao invés do foco do desenvolvimento estar na codificação do software, que o esforço esteja na construção dos modelos do sistema e em regras de transformação entre estes modelos, regras estas que serão executadas por engenhos de transformação em diversos níveis de abstração, chegando (automaticamente) até o nível de código.

MDA segue este mesmo princípio. Porém, como este *framework* é sustentado pela OMG, se baseia num conjunto de padrões já bem estabelecidos e difundidos por esta organização. O pricipal objetivo da OMG ao propor a abordagem MDA foi padronizar diversos modelos para que as empresas os usem a fim de representar suas aplicações. Os principais padrões utilizados para sustentar a utilização de MDA são:

1. *Unified Modeling Language 2.0* (UML2), linguagem usada para representação gráfica dos sistemas computacionais. A sua versão atual é composta por treze diferentes

diagramas que se propõem a permitir a modelagem desde a parte estática (*e.g.* diagrama de classes, de componentes, etc) à dinâmica (*e.g.* diagramas de sequência, de atividades, etc) de um sistema computacional.

2. *Object Constraint Language 2.0* (OCL2) [72], linguagem formal usada especificar expressões e restrições. Importante para alcançar níveis de expressividade que UML não consegue atingir e/ou para dissipar ambiguidades.

3. *Query/View/Transformation* (QVT) [60], linguagem para descrever as regras de transformação entre os modelos.

4. *Meta Object Facility* (MOF) [56], linguagem utilizada para descrever meta-modelos.

Os modelos descritos em MDA possuem geralmente quatro níveis de abstração, são eles:

1. CIM (*Computational Independent Model*), são os modelos de domínio. Estes modelos representam os requisitos e o domínio onde o sistema está inserido. São os modelos de maior nível de abstração.

2. PIM (*Platform Independent Model*), são os modelos que já possuem detalhes (estruturais, comportamentais, etc) de como os sistema será computacionalmente. Porém, estes modelos serão livres de detalhes de uma tecnologia (ou plataforma) específica.

3. PSM (*Platform Specific Model*), modelos que além das informações já presentes ao nível PIM também incluem descrições detalhadas e elementos específicos da plataforma.

4. Código, código-fonte executável do sistema.

Regras de transformação são utilizadas para fazer a passagem entre os níveis de modelos. Estas regras podem ser construídas tanto para refinamento de modelos de uma mesma categoria (*e.g.* de PIM para PIM), quanto para mudanças de categoria de modelos (*e.g.* de PIM para PSM), conforme pode ser visto na Figura 2.2. A Figura 2.2, apresenta as transformações mais comuns no contexto de MDA, porém, nada impede que regras de transformações entre níveis não seqüenciais também sejam desenvolvidos (*e.g.* PIM-Code).

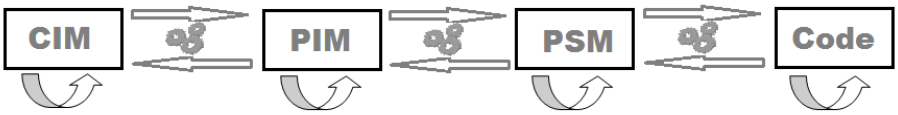

Figura 2.2: Transformações em MDA.

Esta organização de modelos em níveis de abstração permite que, fazendo uso da MDA, uma empresa seja capaz de construir um sistema independente da tecnologia de implementação existente na época, e caso a *posteriori* haja a necessidade de uma mudança de plataforma, essa passagem poderá ser realizada mais facilmente, pois apenas os modelos específicos de plataforma (PSM) sofreriam alterações. Logo, arquiteturas poderão servir por mais tempo para as empresas.

2.4 MDT

É fato a importância que a atividade de teste tem ganhado atualmente, sendo inclusive, já iniciada junto com início da execução do processo de desenvolvimento. Pensando em viabilizar a integração entre as demais atividades do processo de desenvolvimento com a atividade de testes, novas abordagens têm surgido na literatura especializada. Uma destas novas abordagens é a MDT - *Model Driven Testing* (Teste Dirigido por Modelos).

MDT [37; 9] é uma realização de MBT que faz uso de práticas de MDA para a geração automática de artefatos de teste (casos de teste, oráculos, etc) de acordo com regras de transformações pré-definidas, possivelmente a partir de modelos de desenvolvimento. Ou seja, MDT vai além da geração apenas dos casos de teste, possuindo a preocupação em fornecer à equipe de testes uma infra-estrutura para garantir que a atividade possa ser executada satisfatoriamente. Esses artefatos gerados poderão servir para a execução em diferentes plataformas.

Dentre as vantagens da utilização de MDT em relação à MBT, a principal é que, em MBT, os casos de teste são derivados a partir dos modelos de desenvolvimento fracamente conectados, onde estes são normalmente incompletos de informações necessárias para os testes (restrições, casos alternativos, dentre outros). Com a utilização das práticas de MDA, onde modelos são o centro do desenvolvimento, tais informações poderão ser naturalmente

incorporadas.

Seguindo o que indica MDA, MDT também apresenta quatro níveis de modelos de teste que são: CITM - *Computational Independent Test Model*, PITM - *Platform Independent Test design Model*, PSTM - *Platform Specific Test design Model* e *test code*; estes modelos são refinados de um nível para outro fazendo uso de regras e engenhos de transformação [17].

2.5 UTP

A UML [16] provê mecanismos (*e.g.* perfis) para extensão de seus diagramas com o objetivo de adaptá-los a semânticas de domínios específicos. No contexto de testes, a OMG definiu um perfil de testes para a versão 2.0 da UML - *UML Test Profile* (UTP), a fim de facilitar o projeto, visualização, especificação, análise, construção e documentação de artefatos de teste funcional.

UTP cobre amplamente os diferentes conceitos que são abordados em um projeto de teste. Esse perfil foi definido com base no meta-modelo de UML 2.0, visando a integração e reuso de conceitos já existentes, propiciando a integração de MDA e MDT.

O perfil é organizado em quatro grupos de conceitos: i) Arquitetura de Teste (*Test Architecture*), definindo elementos usados para especificar aspectos estruturais relacionados à arquitetura necessária para atividade de testes; ii) Comportamento de Teste (*Test Behavior*), elementos que auxiliam a especificação do comportamento dos testes, seus objetivos, bem como a modelagem de comportamentos importantes como a avaliação da execução dos testes; iii) Dados de Teste (*Test Data*), elementos relacionados à seleção e uso dos dados de teste; e iv) Conceitos de Tempo (*Time Concepts*), elementos que referem-se a conceitos de tempo, tais como restrições ou observações temporais. Cada grupo elenca um conjunto de conceitos que especificam elementos importantes para a construção e execução dos artefatos de teste, e sua utilização conjunta leva a uma especificação mais completa em se tratando de testes.

2.6 FreeRTOS

O FreeRTOS [13] é um mini-*kernel* de um sistema operacional de tempo real, portátil e de código livre usado para o desenvolvimento de aplicações comerciais para sistemas embarcados, com suporte de uma comunidade ativa de usuários. Versões para várias arquiteturas estão disponíveis (*e.g.* x86, ARM9, etc).

Este sistema operacional (SO) foi desenvolvido na linguagem C com a meta de ser pequeno, simples e de fácil uso. O sistema, que está baseado nos conceitos de tarefas e corotinas, permite o rastreamento de execução e a codificação de interrupções no nível de software e, através de estruturas como filas e semáforos realiza a comunicação e sincronização entre as tarefas e interrupções. Em um sistema operacional convencional, cada tarefa é um programa executável sob o controle do sistema operacional, que pode executar apenas uma tarefa de cada vez. No entanto, a troca rápida entre as tarefas faz parecer como se estivessem sendo executadas concorrentemente. Em sistemas de tempo real o escalonamento das tarefas garante a obediência das restrições de tempo e, mais especificamente no FreeRTOS, a política de prioridades permite a previsão do fluxo de execução de acordo com as trocas de contexto entre essas tarefas. Ou seja, possibilita a criação de aplicações com características determinísticas e, por conseguinte, que podem ser testadas de maneira mais rigorosa.

2.7 Considerações Finais do Capítulo

O capítulo que aqui se encerra, apresentou um breve resumo dos principais conceitos abordados neste trabalho. Os conceitos apresentados se subdividem entre: conceitos relacionados a apresentação da área de atuação (subseção *Sistemas de Tempo Real*); conceitos relacionados a abordagens para desenvolvimento com foco em modelagem (subseção *MDA*); a apresentação das abordagens e mecanismos que relacionam testes com modelos (subseções *MBT*, *MDT* e *UML Testing Profile*); e, por fim, a apresentação de um SO de tempo real que é fundamental para a realização prática dos resultados desenvolvidos neste trabalho (subseção *FreeRTOS*).

Capítulo 3

Diretrizes para Modelagem de STRs

3.1 Motivação

Projetar um sistema computacional é sempre uma atividade complexa de ser realizada. Traduzir as idéias oriundas da mente do cliente para uma notação menos abstrata e compreensível pelos desenvolvedores é uma tarefa que requer cuidados especiais, visto que estes artefatos guiarão todo o processo de desenvolvimento do software. Para a classe dos STRs a necessidade de construção de projetos de software mais completos e expressivos é ainda mais evidente, visto que suas características próprias já inferem uma complexidade extra para seu projeto e desenvolvimento.

A UML por si, apesar de ser uma linguagem bastante expressiva, acaba por não fornecer mecanismos para proporcionar a modelagem clara de certos aspectos real-time. A seguir, um sistema exemplo será apresentado e alguns dos aspectos que a UML deixa a desejar no tocante a possibilidade de modelagem e expressividade serão discutidos.

Tomemos por base um sistema de tempo real relativamente simples. Este tem por objetivo central realizar o monitoramento e tomar as decisões cabíveis segundo dados recebidos de um conjunto de sensores, provendo assim a segurança de um prédio. A especificação dos principais requisitos deste sistema, o "Sistema de Alarmes", encontra-se descrita em [67]. Em linhas gerais, o sistema utiliza-se de sensores de janela, de porta e de movimento para averiguar a existência de intrusos no prédio. Confirmada a presença do intruso, um conjunto de medidas de segurança são executadas, as quais incluem o acionamento das luzes na sala onde o intruso está localizado, a realização de uma chamada telefônica para a polícia local

comunicando o arrombamento, etc. Atrelado a estes requisitos, existem restrições temporais que devem ser satisfeitas (*e.g.* as luzes do ambiente invadido devem ser ligadas em no máximo 2 segundos). Outra característica importante deste sistema é a sua necessidade de lidar com eventos aperiódicos (*e.g.* invasão de ambiente e queda de energia).

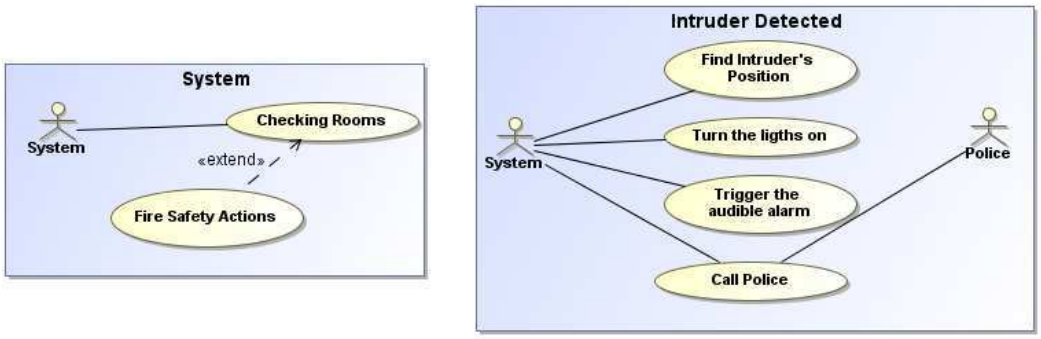

Figura 3.1: Casos de Uso do sistema exemplo.

Apesar da simplicidade deste sistema, com ele já conseguimos vislumbrar exemplos de deficiências da UML, aspectos que a UML por si não nos fornece mecanismos para sua modelagem. Por exemplo, normalmente o primeiro diagrama UML usado para realizar a tradução dos requisitos do sistema em linguagem natural para uma notação menos ambígua é usando Casos de Uso. Porém, como modelar usando este diagrama os requisitos relacionados a eventos aperiódicos? A Figura 3.1 demonstra, usando os elementos existentes neste diagrama, uma possível forma de modelagem do requisito de tratamento para o evento de intruso localizado. Porém, esta forma de modelagem não deixa claro que o comportamento representado é o da captura/tratamento de interrupções e não do comportamento regular do sistema. Mesmo em outros diagramas UML esta dificuldade persiste. Ou seja, a UML, mais especificamente neste caso os diagramas de caso de uso UML, não fornecem mecanismos para diferenciação de representações entre requisitos síncronos e assíncronos, característica de suma importância no tocante a realização de projetos de STRs expressivos.

Da mesma forma, como poderíamos modelar usando os elementos comportamentais da UML a necessidade que, após o tratamento de uma determinada interrupção, o fluxo do sistema deva retornar para a exata configuração a estrutura qual estava antes do lançamento da interrupção?

Questionamentos como estes nos evidenciam a carência que a linguagem UML tem no

tocante a modelagem de forma clara e simplificada de alguns aspectos real-time. Por isso, é evidente a necessidade de complemento desta linguagem, ou mesmo que um auxílio e/ou adaptação possa ser estabelecido a fim de guiar os projetistas neste contexto.

3.2 Solução

Visando permitir a completa modelagem de STRs usando uma linguagem familiar aos desenvolvedores de software (UML), um conjunto de diretrizes foi desenvolvido. Estas diretrizes têm por objetivo auxiliar projetistas de STRs, principalmente iniciantes, no tocante a práticas de modelagem, bem como guiá-los no sentido de como modelar as características especiais destes sistemas de uma forma clara. Para tal, em conjunto com as diretrizes, dois elementos foram criados para auxiliar no processo de modelagem: um perfil UML (*Real-Time Design Profile*) e um pacote auxiliar (*Real-Time Elements*). Vale salientar que tanto os novos elementos desenvolvidos (perfil e pacote auxiliar), bem como as diretrizes estabelecidas, objetivou auxiliar a modelagem de STRs reativos não-críticos no nível independente de plataforma, ou seja, buscamos aqui construir de uma maneira mais expressiva os primeiros modelos que serão usados durante o processo de desenvolvimento. Estes, muito provavelmente deverão ser refinados em etapas posteriores.

3.3 *Real-Time Design Profile*

Para melhor identificar e modelar STRs reativos, decidimos criar um novo perfil, o *Real Time Design Profile*. É sabido que na literatura já existem alguns uso de UML [24] e proposição de perfis que se propõem a auxiliar a modelagem de STRs (*e.g.* MARTE [22] e UML-RT [52]). Porém, estes lidam nas suas especificações com artefatos em nível de abstração mais baixo (*e.g.* semáforos e priorização de processos) do que o que pretendemos trabalhar (nível PIM). Acreditamos que modelos de *design* que incluem informações de baixo nível acabam por ser complexos de construir e findam confundindo seus desenvolvedores/projetistas, especialmente se estes forem iniciantes ou com pouca experiência. Por isso, é importante ter um conjunto de modelos anteriores, com um nível de abstração mais elevado. Para auxiliar neste sentido, o *Real-Time Design Profile* foi desenvolvido. Vale salientar que os elementos do

Real-Time Design Profile serão usados para a definição das diretrizes de modelagem (Seção 3.5).

A Figura 3.2 apresenta graficamente os elementos do perfil desenvolvido.

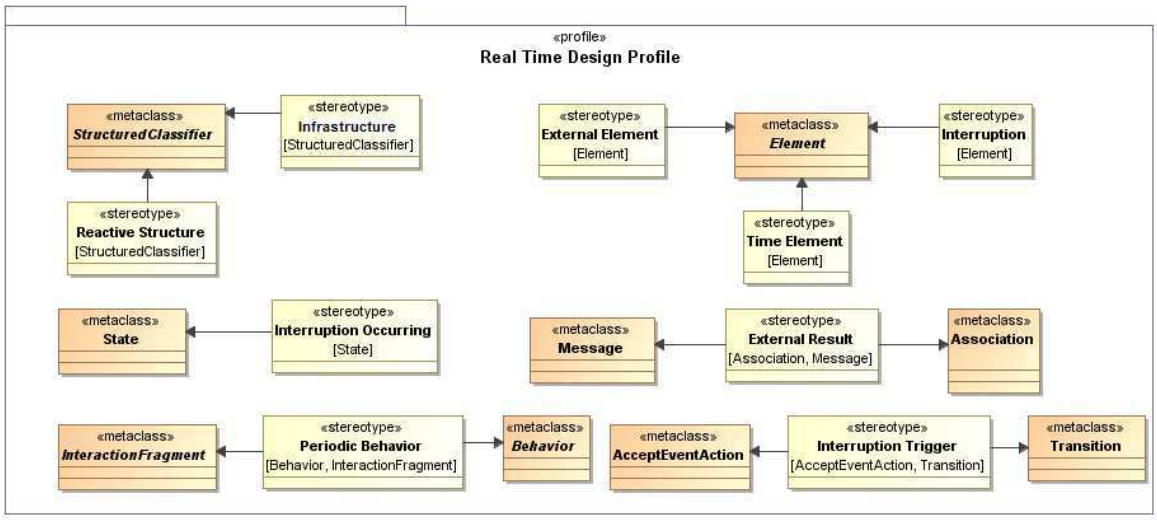

Figura 3.2: O *Real Time Design Profile*.

A seguir, os estereótipos especificados na Figura 3.2 serão melhor explicados individualmente, seguindo a seguinte padronização: primeiramente será fornecido um texto curto explicando as funcionalidades do estereótipo, considerando o momento quando utilizá-lo (*id* Descrição); na sequência será apresentada a justificativa que levou à criação do respectivo estereótipo (*id* Motivação); e, por fim restrições OCL, caso necessário, serão usadas para complementar a definição do estereótipo (*id* Restrições).

Estereótipo: *Infrastructure*

Descrição: Aplicado a estruturas, como Classes e Componentes, que são provenientes do contexto onde o sistema está incluído, ou seja, estas estruturas estão presentes nos modelos de projeto por uma questão organizacional, apenas para torná-lo mais claro. Porém, estes artefatos não necessariamente serão implementados, podendo ser somente reusados da arquitetura (infra-estrutura) cujo sistema está implementado.

Motivação: Muitos dos STRs reusam estruturas provenientes do SO para prover meios de cumprir seus requisitos. É importante identificar tais estruturas nos modelos de *design* para que os projetistas possam entender o que deve ou não ser implementado. Esta identificação

também ajudará a equipe de testes a entender quais objetos devem ser instanciados para a realização dos testes.

Estereótipo: ***Reactive Structure***

Descrição: Aplicado a estruturas que reagirão a estímulos externos e, a partir destes estímulos, executarão os tratamentos adequados. Estas estruturas normalmente possuem processamento independente. Portanto, é necessário que estas sejam tratadas como objetos ativos.

Motivação: A identificação das estruturas ativas permitirá a localização e diferenciação destas das demais que também podem possuir processamento independente, mas que não reagem a estímulos externos. Localizar as estruturas reativas permitirá às equipes de implementação e de testes tratá-las adequadamente, ou mesmo criar objetos que simulem o comportamento de hardwares especiais (*e.g.* sensores).

Restrições:

Código Fonte 3.1: Restrição OCL para o estereótipo *Reactive Structure*

```
Context Reactive_Structure inv:
    self.isActive = true
```

Esta restrição força que a estrutura que possua o estereótipo *Reactive*, também seja necessariamente representada como um objeto ativo.

Estereótipo: ***External Element***

Descrição: Aplicado a elementos que não fazem parte do sistema propriamente, mas devem estar presentes na modelagem para melhor apresentar o comportamento do sistema, bem como a interação com o mesmo. Comumente, os elementos dotados com este estereótipo representam o ambiente externo, ou os usuários da aplicação.

Motivação: STR reativos comumente são baseados em interações entre elementos externos e o sistema. A inclusão de elementos para representação dessas estruturas externas facilitará a modelagem dos requisitos do sistema, tornando os modelos mais claros. Quanto à definição dos testes, esta identificação explícita facilitará que as interações com os elementos externos possam ser simuladas, assim como os dados de entrada que serão consumidos nos testes.

Estereótipo: ***Time Element***

Descrição: Todo e qualquer elemento dotado com este estereótipo representa uma estrutura incluída no projeto especialmente para tratar questões temporais, tais como um relógio do sistema, *timers*, etc.

Motivação: É importante diferenciar estruturas especialmente criadas para ajudar a implementação dos requisitos de tempo, pois estas estruturas deverão merecer maior atenção durante todo o processo de desenvolvimento.

Estereótipo: ***Interruption***

Descrição: Estereótipo aplicado aqueles elemento cuja semântica faz referência ao tratamento de interrupções (eventos assíncronos), podendo ser usado tanto na parte de levantamento de requisitos (*e.g.* nos diagramas de Caso de Uso), quanto na parte comportamental (*e.g.* nas Máquinas de estado).

Motivação: Destacar as estruturas e comportamentos onde ocorrem as interrupções é importante por esta ser uma das características mais diferenciadas dos STRs. Portanto, é importante que esta parte da modelagem seja bastante clara e se destaque da modelagem do restante do comportamento comum do sistema.

Estereótipo: ***Interruption Trigger***

Descrição: Estereótipo que deve ser aplicado a transições que possuam um evento/ação ou condição que desencadeie o lançamento de uma determinada interrupção.

Motivação: Quando pensa-se na realização de testes para averiguar o comportamento de uma interrupção, o motivo pelo qual a interrupção foi lançada é de fundamental importância, principalmente porque o testador necessitará forçar que este "motivo" seja satisfeito de alguma maneira. Destacar nos modelos o evento, ação ou condição que fará com que a interrupção seja lançada é bastante interessante neste sentido, facilitando o trabalho daquele que construirá os casos de teste, e/ou para guiar a execução de algum algoritmo para geração automática dos mesmos.

Estereótipo: ***Interruption Occurring***

Descrição: Este estereótipo será usado em máquinas de estado para sinalizar estados onde

esteja sendo executado o tratamento de certa interrupção.

Motivação: Tratando do comportamento individual de cada estrutura, existirão momentos onde o objeto/componente entrará num fluxo diferenciado do seu comportamento regular, comportamento este que foi proporcionado devido a uma interrupção ter sido lançada. Diferenciar este comportamento é importante para melhor compreensão das características do sistema.

Estereótipo: *External Result*

Descrição: Aplicado a mensagens (nos diagramas de sequência) e/ou associações (nos casos de uso) onde, como resultado, exista uma resposta a um elemento externo ao sistema. Obrigatoriamente esta mensagem/associação deve estar ligada a um artefato com o estereótipo *«External Element»*.

Motivação: Muitas das restrições de tempo de STRs reativos limitam seus *deadlines* a respostas que o sistema deve fornecer. A inclusão destas respostas nos modelos de *design* e a identificação das mesmas com o estereótipo *«External Result»* facilitará a modelagem deste tipo de requisito.

Restrições:

Código Fonte 3.2: Restrição OCL referente ao estereótipo *External Element*

```
Context External_Result inv:
    if self.receiveEvent.getAppliedStereotypes()->select(s | s.name = '
       External Result')->size() > 0 then
        self.source.getAppliedStereotypes()->select(s | s.name = '
           External Element')->size() > 0 OR
        self.target.getAppliedStereotypes()->select(s | s.name = '
           External Element')->size() > 0
```

Esta restrição tem por finalidade forçar que toda mensagem e/ou associação que possua o estereótipo *External Result* esteja ligada a uma estrutura que possua o estereótipo *External Element* aplicado.

Estereótipo: *Periodic Behavior*

Descrição: Aplicado a fragmentos combinados ou comportamentos (*behavior* UML), com

o objetivo de dotá-los com a seguinte semântica: o comportamento incluso dentro destes fragmentos serão repetidos periodicamente. A periodicidade será estabelecida dentro de intervalos de tempo estabelecidos pela restrição atribuída ao operando do fragmento.

Motivação: A UML por si, não oferece um meio eficiente para modelagem de requisitos a serem executados com periodicidade. Por isso, visualizamos a possibilidade de então fazer o uso dos fragmentos combinados e adequá-los ao contexto em questão, apenas dotando-os com um estereótipo específico. Logo, a visualização e compreensão destes requisitos ficará facilitada.

Restrições:

Código Fonte 3.3: Restrição OCL referente ao estereótipo *Periodic Behaviour*

```
Context Periodic_Behavior inv:
    self.operand.oclIsUndefined = false
```

Esta restrição tem por finalidade forçar que todo fragmento referenciado que possua o estereótipo *Periodic Behaviour* possua um valor válido como operando, pois este valor será essencial para caracterizar a periodicidade do comportamento.

3.4 *Real Time Elements*

Durante o estudo realizado para definição das diretrizes de modelagem (que serão apresentadas na próxima subseção), algumas estruturas foram identificadas como importantes no tocante à padronização, boas práticas e/ou essenciais para modelagem das características particulares dos STRs reativos. Estas estruturas foram definidas, principalmente, inspiradas em análises de implementações de STRs já existentes e de aplicações apresentadas em outros trabalhos da área (*e.g.* [12]). O elemento *pacote UML* (*package*) foi escolhido como estrutura de agrupamento dos elementos por este ser um mecanismo facilitador de reuso no contexto de projetos usando UML.

O objetivo da definição deste pacote é auxiliar o projeto de arquiteturas de STRs, proporcionando aos projetistas a identificação e uso de estruturas que comumente devem ser implementadas no contexto desta classe de sistemas. É válido destacar que os elementos deste pacote não estão fortemente conectados aos elementos desenvolvidos no *Real Time*

Design Profile, ambos podendo serem usados independentemente. Porém, sua utilização combinada deverá trazer uma maior expressividade aos modelos de *design* dos sistemas.

Os elementos do pacote *Real Time Elements* são apresentados na (Figura 3.3) e serão melhor descritos na sequência. Vale salientar que estes elementos são usados nas diretrizes de modelagem.

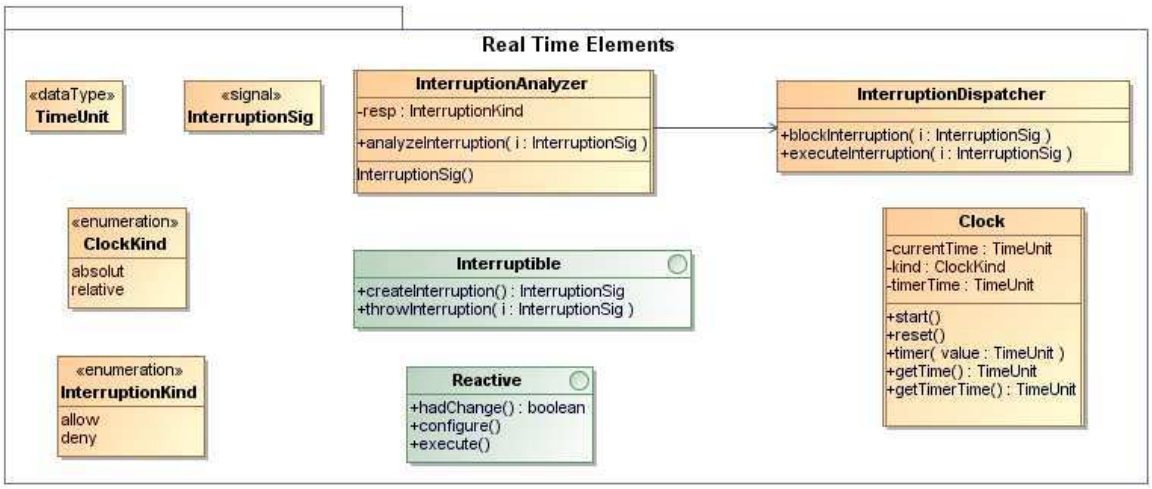

Figura 3.3: O pacote *Real Time Elements*.

Elemento: **Classe *Interruption Analyzer***

Descrição: Estrutura responsável por receber o sinal das interrupções e decidir se estas devem ser tratadas no exato momento do recebimento, ou bloqueadas temporariamente. Esta estrutura, na maioria das vezes já é implementada pelo SO do sistema. Porém, em algumas circunstâncias é preciso implementar estruturas especiais para tratar tais características, por necessidade de um maior controle.

- Atributos:

  - ***resp***: resposta da análise se a interrupção deve ser lançada instantaneamente (*allow*) ou esperar (*deny*).

- Operações:

  - ***analyzeInterruption***: responsável por receber o sinal da interrupção e enviá-lo para a estrutura que dará a possibilidade de tratá-la no momento adequado.

- Recepção de Sinais:

  - ***InterruptionSig***: identificação que a estrutura poderá receber o sinal de uma

interrupção.

Elemento: **Classe *Interruption Dispatcher***

Descrição: Estrutura responsável por conectar cada interrupção recebida com a devida estrutura que irá tratá-la adequadamente.

- Operações:

  - ***blockInterruption***: bloqueia o tratamento da interrupção por um determinado tempo, ou até que o escalonador de processos permita.

  - ***executeInterruption***: responsável por fazer a ligação com a estrutura específica que tratará a interrupção adequadamente.

Elemento: **Classe *Clock***

Descrição: Estrutura responsável pelo controle do tempo no sistema. Não necessariamente esta estrutura será o *clock* físico do sistema. Poderá existir mais de uma estrutura de *clock* nos modelos de *design*, desde que estes sejam *clocks* relativos. Estas estruturas possuem algumas operações que facilitam o tratamento do tempo e possibilita a especificação das restrições temporais.

- Atributos:

  - *currentTime*: corresponde ao tempo corrente no *clock*.
  - *kind*: referente ao tipo do *clock*.
  - *timerTime*: corresponde ao tempo corrente no *timer*.

- Operações:

  - ***start***: responsável por acionar o *clock*.
  - ***reset***: zera a contagem de tempo.
  - ***timer(x)***: aciona uma contagem regressiva de tempo iniciando no valor “x”, recebido como parâmetro.

  - ***getTime e getTimerTime***: retorna o valor atual da contagem de tempo geral e do timer, respectivamente.

Elemento: **Interface *Interruptable***
Descrição: Interface que deve ser implementada por toda estrutura que tenha por característica poder criar e lançar interrupções.

- Operações:
  - ***createInterruption***: responsável por criar o objeto que terá papel da interrupção.
  - ***throwInterruption***: lança a interrupção para a estrutura capaz de receber e tratá-la (*e.g. Interruption Analyzer*).

Elemento: **Interface *Reactive***
Descrição: Interface que deve ser implementada pelas estruturas que têm papel reativo (comumente sensores ou estruturas que desempenhem papel semelhante).

- Operações:
  - ***configure***: realiza as configurações iniciais básicas na estrutura reativa.
  - ***execute***: inicia a execução do processo que comanda a estrutura.
  - ***hadChange***: analisa se houve alguma mudança significativa no meio que necessite de uma resposta/reação do sistema.

Elemento: **Sinal *InterruptionSig***
Descrição: Sinal que representa uma interrupção. Este sinal poderá ser especializado para diferenciar os diferentes tipos de interrupções do sistema.

Elemento: **Enumeração *InterruptionKind***
Descrição: Enumeração que elenca as possibilidades que uma determinada interrupção possa receber (permitida ou bloqueada).

- Literais:
  - ***allow***: o sistema recebeu autorização para tratar a interrupção no momento.
  - ***deny***: o sistema não recebeu autorização para tratar a interrupção no momento.

Elemento: **Enumeração *ClockKind***
Descrição: Enumeração referente às classificações que os *clocks* podem receber com relação ao tempo tratado (absoluto ou relativo).

- Literais:
  - ***absolute***: tempo absoluto, ou seja, o tempo marcado será o que guiará toda execução do sistema.
  - ***relative***: tempo relativo, ou seja, o tempo marcado é relativo a um outro *clock* do sistema que será tomando como referência

Elemento: **Tipo de dado *TimeUnit***
Descrição: Tipo de dado que servirá para padronizar as unidade de tempo usadas em todos modelos.

3.5 Diretrizes de Modelagem

Visando facilitar a atividade de projeto de STRs usando UML, um conjunto de diretrizes de modelagem foi definido. Esta seção do trabalho apresenta estas diretrizes. Por fins de organização, a seção será estruturada da seguinte forma: a primeira diretriz se distingue das demais por sugerir uma seqüência de diagramas UML a serem usados para *design* de STRs; em seqüência, são apresentadas individualmente cada uma das diretrizes para cada respectivo diagrama UML, uma explicação de como aplicá-las, e quais artefatos deverão ser obtidos como resultado de sua aplicação (quando necessário); por fim, ao final da apresentação das diretrizes para cada modelo UML, um exemplo simplificado (*toy example*) da aplicação das mesmas é apresentado para melhor compreensão do que foi explicado anteriormente. A título de identificação, as diretrizes são rotuladas pela letra *D*, acrescidas com a numeração

da mesma (*e.g.* a primeira diretriz está rotulada como **D1**, a segunda como **D2**, etc). É sugerível que estas diretrizes sejam usadas na sequência em que são apresentadas, pois algumas dependem de outras que foram definidas anteriormente.

- **D1 - Sequência de diagramas sugerida para modelagem**
  - Diagramas de Caso de Uso

    Justificativa: Diagramas mais simples e mais usados quando se trata da realização da atividade da modelagem inicial dos requisitos do sistema, bem como da interação destes com os atores. Importante para definição de sujeitos a partir dos requisitos, bem como para apresentar uma visão geral do mesmo.
  - Diagrama de Componentes (caso necessário)

    Justificativa: Este deve ser o diagrama que representará a modelagem estrutural do sistema em altíssimo nível, identificando os principais elementos presentes e suas associações. STRs normalmente são trabalhados a partir de componentes. Caso o sistema a ser modelado seja bastante simplificado, este diagrama poderá ser desnecessário. Porém, por questões organizacionais e de boas práticas, é interessante modularizar via componente, quando possível.
  - Diagrama de Classes

    Justificativa: Será usado para modelar as estruturas em um nível mais baixo de abstração. Os componentes serão decompostos entre classes neste diagrama.
  - Diagrama de Estruturas Compostas

    Justificativa: Este será usado por ser importante, principalmente quando se pensa na criação automática de testes a partir de modelos. Para tal, é importante que estejam claras quais estruturas interagem, como um componente é decomposto e por quais objetivos.
  - Máquina de Estados Comportamentais

    Justificativa: Serão usadas para especificar o comportamento individual de cada componente. Saber quais os possíveis estados que um componente pode ter, bem como quais eventos fazem com que haja a mudança de estados. Estes são fatores de grande importância em se tratando de sistemas reativos.

  - Diagrama de *Overview* de Interação

    Justificativa: É importante ter uma visão geral (em alto nível) de como os diagramas de seqüência estão conectados e, principalmente, quando, e de quais comportamentos as interrupções podem surgir. Este diagrama será usado para fornecer a visão geral do comportamento do sistema como um todo.

  - Diagramas de Sequência

    Justificativa: Estes serão os diagramas chave para demonstração do comportamento e a interação dos objetos do sistema ao nível de classes, tornando possível inclusive a modelagem de restrições de tempo num nível de abstração mais baixo.

- Diretrizes para diagramas de Caso de Uso

  1. *Objetivo*: Permitir, a partir dos modelos de definição e entendimento dos requisitos do sistema, a expressividade de características para modelar eventos periódicos e aperiódicos.
  2. *Solução*:

     - **D2** - O projetista deve decidir quantos diagramas de Caso de Uso e seus elementos (Casos de Uso, atores, etc) julga necessário para modelar o comportamento regular do sistema (comportamento sem a inclusão de eventos aperiódicos).

       Resultado: texto descritivo com o resultado das decisões tomadas.

     - **D3** - O projetista deve, via análise criteriosa dos requisitos, decidir quantos e quais dos requisitos serão tratados e implementados como interrupções (característica importante nos STRs). Ou seja, neste momento serão identificadas as possíveis interrupções que poderão ser lançadas.

       Resultado: lista com as interrupções identificadas.

     - Modelagem dos Casos de Uso correspondentes ao comportamento regular do sistema.

       * **D4** - Caso exista a necessidade da modelagem de um comportamento periódico (*i.e.* comportamento que se repete a cada "x" unidades de tempo), fazer uso de um ator especial chamado "*Time*" (dotado com o

estereótipo *«Time Element»*), e conectá-lo às estruturas que simbolizam o comportamento a ser repetido. A este ator deve ser anexado um comentário UML com o valor de tempo referente à periodicidade.

- **D5** - Caso algum dos atores seja um elemento externo ao sistema, dotá-lo com o estereótipo *«External Element»*.
- **D6** - Caso exista uma comunicação entre o sistema e elementos externos (*«External Element»*), a associação que os conecta deve possuir o estereótipo *«External Result»*.

Resultado D4-D6: conjunto de diagramas de casos de uso que representem somente o comportamento regular do sistema.

– Para cada interrupção presente na lista produzida pela aplicação da diretriz D3:

- **D7** - Definir um Caso de Uso próprio, nomeando-o adequadamente a fim de relacioná-lo ao tratamento da interrupção, e o diferenciando-o dos demais através do estereótipo *«Interruption»*.
- **D8** - Conectar o caso de uso criado na etapa D7 ao caso de uso que será interrompido, através de uma associação *«extend»*.
- **D9** - Incluir, no caso de uso que será interrompido, um *extension point* nomeado com o motivo que leva ao lançamento da respectiva interrupção.
- **D10** - Cada caso de uso referente a uma interrupção (incluídos a partir da aplicação da diretriz D9) deve ser associado (através de associações *«include»*) com casos de uso que representarão o comportamento referente ao tratamento da interrupção. Estes novos casos de uso devem ser incluídos num assunto que seja nomeado de uma forma que deixe evidente a qual interrupção aqueles casos de uso fazem referência.

Resultado D7-D10: Os diagramas produzidos anteriormente, até a aplicação da diretriz D6, agora acrescidos com a modelagem dos requisitos de interrupção.

3. *Exemplo de aplicação*:

Requisitos a serem modelados:

- Um conjunto de sensores deve ser checado a cada 5 segundos.
- Caso algo inesperado seja identificado no ambiente, um alerta deve ser liberado para o administrador do sistema.
- Caso haja uma queda de energia, a troca de bateria deve ser executada imediatamente.

Diretrizes Aplicadas:

- Figura 3.4 (destaque em vermelho): D2, D3, D4, D5, D6, D7, D8, D9
- Figura 3.4 (destaque em laranja): D2, D10, D6
- Figura 3.4 (destaque em verde): D2, D10

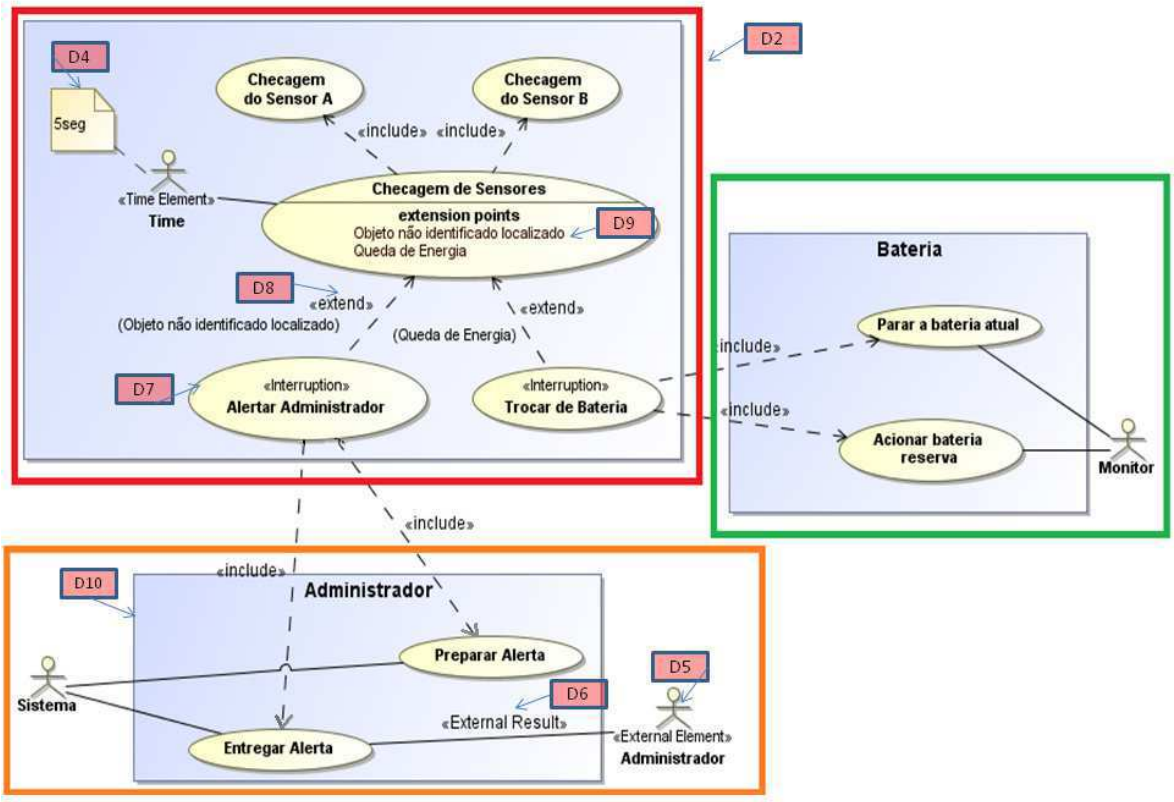

Figura 3.4: Comportamento regular do sistema, acrescido das interrupções.

- Diretrizes para diagramas de Componentes

1. *Objetivo*: Tornar claro quais estruturas são provenientes do contexto arquitetural onde o sistema está inserido e quais são aquelas que terão caráter reativo.

2. *Solução*:

   – **D11** - Apenas um único diagrama de componentes deverá ser criado. Porém, a decisão de quantos e quais componentes e interfaces existirão ficará a cargo do projetista.

   Resultado: Diagrama de componentes do sistema.

   – **D12** - Anotar os componentes que possuem características reativas (*i.e.* reagem a estímulos externos) com o estereótipo *«Reactive Structure»*.

   Resultado: O diagrama produzido na diretriz D11, modificado.

   – **D13** - Especificar aqueles componentes que serão implementados usando um processo independente (geralmente todas as estruturas reativas seguem esta restrição), com a indicação de objeto ativo (barras paralelas nas bordas laterais do componente).

   Resultado: O diagrama produzido na diretriz D11, modificado.

   – **D14** - O projetista pode decidir incluir no projeto estruturas que, apesar de não irem ser implementadas durante o desenvolvimento, são de suma importância para a compreensão do comportamento geral do sistema. Se estas estruturas são provenientes da infra-estrutura a qual o sistema será implementado (*e.g.* sistema operacional, artefatos de hardware), aos componentes que as representam deve ser atribuído o estereótipo *«Infrastructure»*. Esta identificação é importante principalmente para que estas estruturas utilizadas da infra-estrutura possam ser facilmente identificadas e utilizadas adequadamente, tanto durante a implementação do sistema, quanto nos testes do mesmo.

   Resultado D14: O diagrama produzido na diretriz D11, modificado.

3. *Exemplo de aplicação*:

   Decisões de design:

   – Um componente reativo será implementado, este gerenciará os sensores do sistema.

– Para manipulação das interrupções, será reusado do sistema operacional o componente reativo "Gerenciador de Interrupções". Este, receberá a interrupção e enviará um sinal para a estrutura responsável por tratá-la.

Diretrizes Aplicadas:

– Figura 3.5: D11, D12, D13, D14

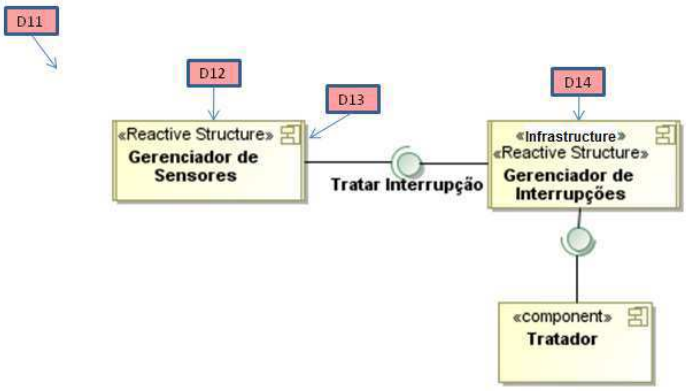

Figura 3.5: Parte do diagrama de componentes do sistema.

- Diretrizes para diagramas de Classe

1. *Objetivo*: Deixar o diagrama de classes mais claro e objetivo, trazendo para esta abstração do projeto estrutural do sistema, elementos que possibilitem a modelagem das características *real-time*.
2. *Solução*:

   – **D15** - Deve-se criar um único diagrama de classes. As decisões de quais classes (também interfaces, operações, enumerações, etc) incluir, ficará a cargo dos projetistas. É interessante utilizar o conceito de pacotes se o diagrama tiver com uma grande quantidade de elementos, ou quando a divisão por domínios seja relevante.

   Resultado: Diagrama de classes do sistema.

   – **D16** - Para manter o padrão usado no diagrama de componentes, dotar com os estereótipos *«Reactive Structure»* e *«Architecture»* as classes que representem objetos reativos e aquelas provenientes da arquitetura, respectiva-

mente.

Resultado: O diagrama de classes da diretriz D15, modificado.

- **D17** - Caso o sistema possua características temporais e eventos aperiódicos, importar o pacote *Real Time Elements* e reusar seus elementos (analisar a seção anterior desde capítulo para melhor compreender a semântica dos elementos deste pacote).

  Resultado: O diagrama de classes da diretriz D15, modificado.

- Para cada interrupção possível de ser lançada (proveniente da lista criada na diretriz D3):

  D18 - Especializar o *signal InterruptionSig* com o tipo da interrupção específica.

  Resultado: O diagrama de classes da diretriz D15, modificado.

- Para cada classe com comportamento reativo:

  D19 - Fazê-la implementar a interface *Reactive*.

  Resultado: O diagrama de classes da diretriz D15, modificado.

- Para cada classe que pode gerar uma interrupção:

  D20 - Fazê-la implementar a interface *Interruptible*.

  Resultado: O diagrama de classes da diretriz D15, modificado.

- Caso o projetista decida por não reusar as estruturas provenientes da arquitetura para realizar a análise e encaminhamento das interrupções:

  * **D21** - Fazer uso das classes *InterruptionAnalyzer* e *InterruptionDispatcher* do pacote *Real Time Elements* (vide seção anterior para melhor compreender a semântica de tais elementos).
  * **D22** - Associar as classes que podem gerar interrupções (classes que seguem a diretriz D19) a classe *InterruptionAnalyzer*.
  * **D23** - Associar a classe *InterruptionDispatcher* àquelas classes responsáveis por tratar a determinada interrupção.

  Resultado D21-D23: O diagrama de classes da diretriz D15, modificado.

3. *Exemplo de aplicação*:

   Decisões de design:

- Não reusar nenhuma estrutura proveniente da arquitetura.
- A única interrupção que poderá ser lançada é a de “Objeto Não Identificado”. A classe que tratará esta interrupção é a “Gerador de Alerta”.

Diretrizes Aplicadas:

- Figura 3.6: D15, D16, D17, D18, D19, D20, D21, D22, D23

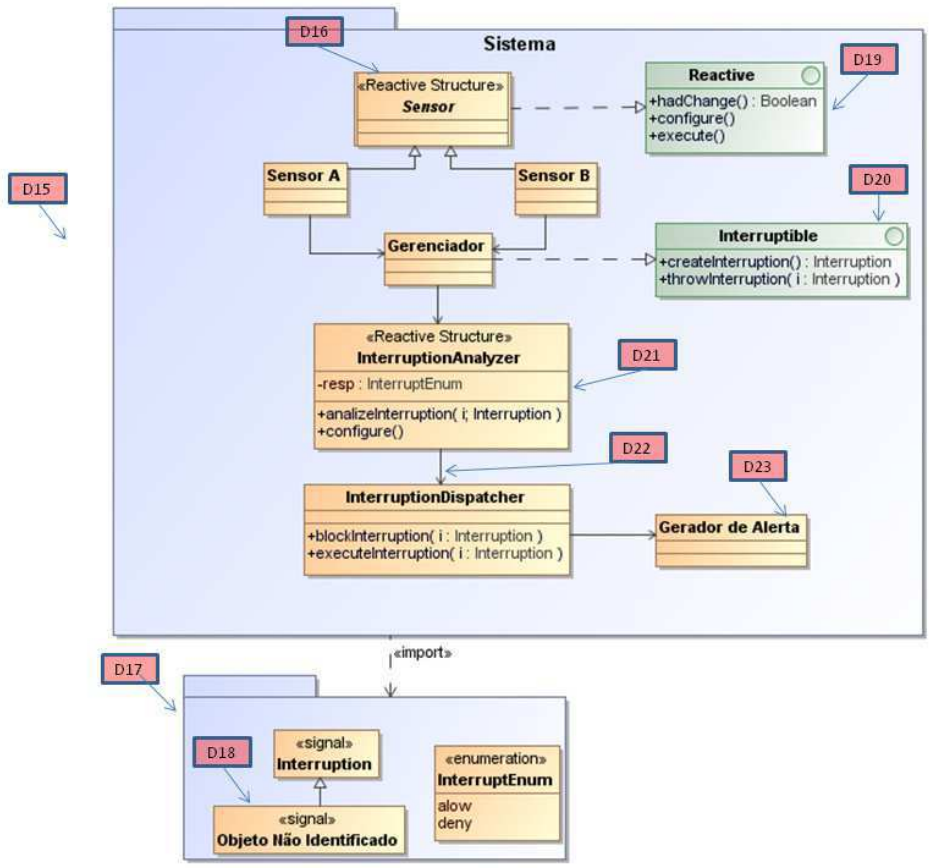

Figura 3.6: Parte do diagrama de classes do sistema.

- Diretrizes para diagramas de Estruturas Compostas

1. *Objetivo*: Melhor esclarecer quais estruturas interagem em um dado componente e identificar como este é decomposto, e por quais objetivos.
2. *Solução*:
   - Para cada componente definido no diagrama de componentes:
     * **D24** - Criar um diagrama de estruturas compostas que corresponda à sua estruturação interna.

∗ **D25** - Identificar o objetivo do componente através de uma colaboração e concedê-la um nome apropriado.

Resultado D24-D25: Um conjunto de diagramas de estruturas compostas.

3. *Exemplo de aplicação*:

   Decisões de design:

   – O componente "Gerenciador de Sensores" é decomposto nas classes "Sensor A", "Sensor B" e "Gerenciador".

   Diretrizes Aplicadas:

   – Figura 3.7: D24, D25

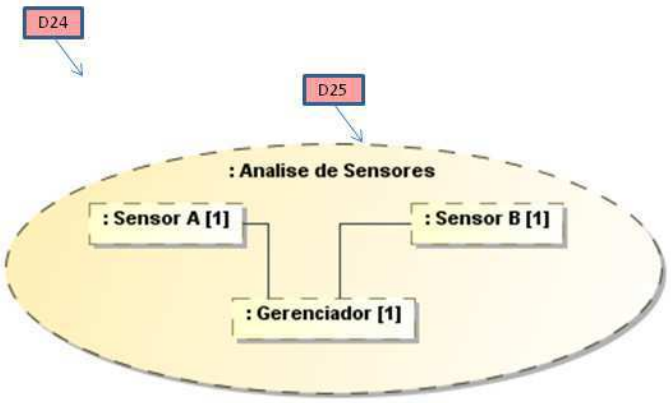

Figura 3.7: Diagrama de Estruturas Compostas do componente Gerenciador de Sensores.

- Diretrizes para Máquinas de Estado

  1. *Objetivo*: Tornar este diagrama mais expressivo no que diz respeito a restrições de tempo e eventos aperiódicos.
  2. *Solução*:

     – **D26** - Deve-se criar uma máquina de estados para cada componente especificado anteriormente e/ou classe que tenha comportamento importante no comportamento geral do sistema.

     Resultado: Conjunto de máquinas de estado do sistema.

     – Caso o componente/objeto trate com eventos periódicos:

     D27 - Fazer uso das palavras chave "*after*" e/ou "*when*" nas guardas das

transições para assim deixar claro quando estes eventos deverão ocorrer.
Resultado: Máquinas de estado da diretriz D26, modificadas.

- **D28** - Usar o estereótipo *«Interruption Trigger»* para sinalizar as transições que possuem eventos ou guardas que indicam a criação ou tratamento de uma interrupção.
Resultado: Máquinas de estado da diretriz D26, modificadas.
- **D29** - Sinalizar o estado onde está sendo executado o tratamento da uma interrupção com o estereótipo *«Interruption Occuring»*.
Resultado: Máquinas de estado da diretriz D26, modificadas.
- **D30** - Incluir o pseudo-estado *Deep History* no estado de onde partirá a interrupção.
Resultado: Máquinas de estado da diretriz D26, modificadas.
- **D31** - Adicionar uma transição partindo do estado que finaliza o tratamento da interrupção para o pseudo-estado *Deep History* (localizado no estado condizente com a diretriz 21).
Resultado: Máquinas de estado da diretriz D26, modificadas.

3. *Exemplo de aplicação*:
Requisitos para o componente Gerenciador de Sensores:
   - A cada 0.5 unidades de tempo, uma verificação de sensores deve ser realizada.
   - Caso um objeto não identificado seja localizado uma interrupção de "Objeto não identificado" deve ser lançada e tratada adequadamente.

   Diretrizes Aplicadas:
   - Figura 3.8: D26, D27, D28, D29, D30, D31

- Diretrizes para diagrama *Overview* de Interação

1. *Objetivo*: Obter uma visão geral do comportamento do sistema e ao mesmo tempo conectar os comportamentos individuais destacando a partir de quais, interrupções podem ocorrer, quais, e como estas serão tratadas.
2. *Solução*:

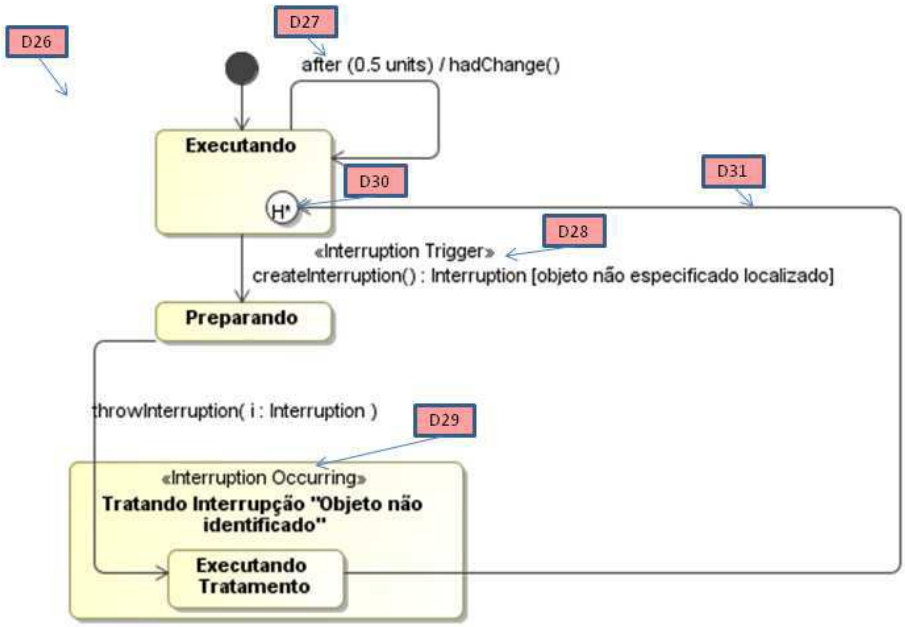

Figura 3.8: Máquina de estados do componente Gerenciador de Sensores.

- **D32** - Criar apenas um diagrama de *overview* para todo o sistema.
  Resultado: Um diagrama de *overview* apresentando em linhas gerais o comportamento do sistema.
- **D33** - Fazer uso de fragmentos referenciados (REF), representativos dos diagramas de seqüência do sistema. Tal modelagem será utilizada para melhor organizar e compreender visualmente o comportamento geral do sistema. A fim de manter a coerência entre os modelos, os nomes dados aos fragmentos REF no diagrama de *overview* devem ser os mesmos dos diagramas de seqüência do sistema.
- **D34** - Fragmentos que representem o tratamento de uma determinada interrupção devem ser dotados com o estereótipo *«Interruption»*.
  Resultado: Diagrama construído na diretriz D32, modificado.
- Para cada interrupção presente na lista produzida na diretriz D3:
  - **D35** - Demarcar onde é possível que esta interrupção ocorra através de regiões de interrupções (*Interruption Regions*).
  - **D36** - Caso uma interrupção possa ser lançada a qualquer momento e em qualquer ponto do sistema, uma *Interruption Region* deve englobar

todas as estruturas que representam o comportamento do mesmo.

- **D37** - Dentro da *Interruption Region*, identificar a interrupção através de um *Accept Signal Action* com o nome da mesma e dotá-lo com o estereótipo *«Interruption Trigger»*.
- **D38** - Conectar o *Accept Signal Action* ao fragmento REF que tem como comportamento o recebimento das interrupções.

  Resultado D35-38: Diagrama construído na diretriz D32, modificado.

3. *Exemplo de aplicação*:

   Requisitos:

   – Dois tipos de interrupções podem ocorrer em situações diferenciadas.

   – Cada interrupção possui um tratamento específico.

   – Existirá um componente gerenciador de interrupções que se encarregará de direcionar a execução para a estrutura correta que irá tratar a interrupção.

   Diretrizes Aplicadas:

   – Figura 3.9: D32, D33, D34, D35, D37

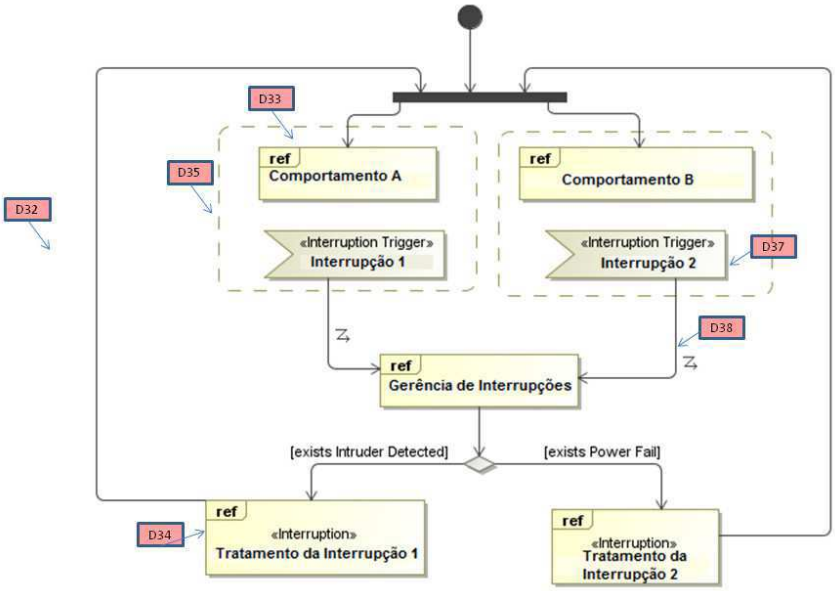

Figura 3.9: Diagrama de Overview do sistema.

- Diretrizes para diagramas de Sequência

  1. *Objetivo*: Tornar o modelo comportamental mais expressivo e permitir a modelagem de requisitos de tempo mais complexos e interações comuns no contexto dos STRs.

  2. *Solução*:

     – **D39** - As decisões de quantos diagramas de seqüência construir, bem como em quais níveis de abstração, ficará a cargo do projetista segundo as necessidades do sistema.

     Resultado: Conjunto de diagramas de seqüência que representam o comportamento do sistema.

     – Em qualquer dos diagramas de seqüencia, caso exista uma interação entre o sistema e o mundo externo:

       * **D40** - Uma linha de vida (geralmente nomeada "Ambiente"ou "Usuário") com o estereótipo *«External Element»* deve ser criada.

       * **D41** - Mensagens de resposta do sistema ao agente externo deverão ser dotadas com o estereótipo *«External Result»*.

     Resultado D40-D41: Diagramas construídos na diretriz C39, modificados.

     – **D42** - Se uma restrição de duração de tempo estiver relacionada à uma ação externa, esta deve incluir uma mensagem de resposta com o estereótipo *«External Result»*.

     Resultado: Diagramas construídos na diretriz C39, modificados.

     – **D43** - Criar uma linha de vida representando a classe *Clock* (pacote *Real Time Elements*), dotá-la com o estereótipo *«Time Element»*, e fazer uso de suas operações, para modelar os comportamentos que envolvem requisitos de tempo.

     Resultado: Diagramas construídos na diretriz C39, modificados.

     – **D44** - Para modelar comportamentos periódicos, fazer uso do fragmento *Loop* (dotado do estereótipo *«Periodic Behavior»*). Este fragmento terá como operando um uso da operação *timer* (classe *Clock*) com o valor temporal da periodicidade. Caso o projetista prefira, a modelagem do comporta-

mento periódico poderá ser estabelecida em um diagrama de sequência em separado.

Resultado: Diagramas construídos na diretriz C39, modificados, ou inclusão de um novo diagrama de sequência.

- **D45** - Sempre que uma interrupção for lançada, usar nas mensagens as operações *createInterruption* e *throwInterruption* (provenientes da interface *Interruption* - pacote *Real Time Elements*).

  Resultado: Diagramas construídos na diretriz C39, modificados.

- **D46** - Usar o tipo de dados *TimeUnit* (pacote *Real Time Elements*) como padrão para todas as unidades de manipulação de tempo.

  Resultado: Diagramas construídos na diretriz C39, modificados.

3. *Exemplo de aplicação*:

   Comportamentos a serem modelados:

   - A estrutura *obj2* gera uma resposta "resp1" para o ambiente quando a operação "op1" for executada.
   - A estrutura *obj2* gera uma resposta "resp2" para o ambiente quando a operação "op2" for executada. Esta resposta deve ser fornecida em no máximo 2 unidades de tempo.
   - A cada 2 unidades de tempo, o objeto *obj2* deve executar o teste "opt4". Se obtiver resposta positiva, a interrupção "Interrupção 1" deve ser lançada.

   Diretrizes Aplicadas:

   - Figura 3.10: D39, D40, D41, D42, D43, D44, D45, D46

3.6 Avaliação

O objetivo desta seção é apresentar a avaliação realizada através de estudos de caso, que teve como finalidade analisar a aplicação das diretrizes no contexto de sistemas reais. Esta análise tem por finalidade encontrar possíveis gargalos/dificuldades na aplicação destas diretrizes, bem como estabelecer quais benefícios foram alcançados para a atividade de modelagem

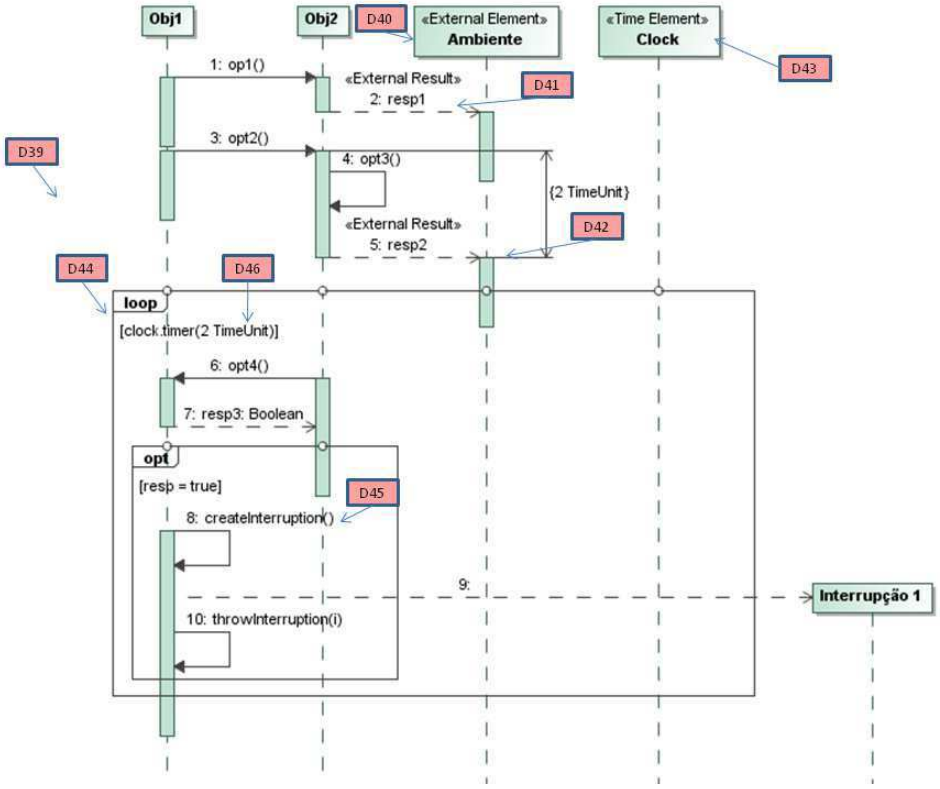

Figura 3.10: Diagrama de Seqüência para comportamento do sistema.

através da sua utilização. Esta avaliação foi executada através da aplicação de estudos de caso baseados em sistemas reais.

3.6.1 GQM

Para realização desde estudo empírico, foi escolhida a utilização da abordagem de mensuração orientada a metas que apóia a definição e implementação de metas operacionais e mensuráveis, a chamada GQM (*Goal*, *Question*, *Metric*) [70]. Esta é uma abordagem largamente adotada na Engenharia de Software para medição de produtos e processos de software. Nas próximas seções são apresentadas as instanciações das fases de Definição e Planejamento (primeiras fases da abordagem GQM) no contexto do nosso estudo, bem como os resultados e conclusões do estudo.

3.6.2 Definição dos Objetivos

Pergunta de Pesquisa

As diretrizes para modelagem de STRs reativos usando UML são realmente úteis e aplicáveis, cumprindo o propósito de melhorar a expressividade e facilitar a atividade de projeto desse tipo de sistema?

Proposições de Estudo

Baseado na execução de um conjunto de estudos de caso, caracterizar:

- Quais as dificuldades de uso das diretrizes para modelagem/projeto de STRs reativos.
- Atribuição de um nível de aplicabilidade das diretrizes no contexto dos STRs.
- Atribuição de um nível de qualidade para os modelos construídos segundo as diretrizes.
- Identificação de quais melhorias poderiam ser estabelecidas para tornar as diretrizes mais úteis.

Objetivo de Estudo

Analisar as diretrizes de modelagem para STRs reativos usando UML
Com o propósito de análise
Com respeito à localizar possíveis gargalos na aplicação das diretrizes e atestar seu papel de melhoria da qualidade na modelagem
Do ponto de vista dos projetistas
No contexto de projeto/desenvolvimento de STRs

Questões e Métricas

Q1: As diretrizes são realmente úteis e necessárias para modelagem de STRs com UML?

- **PRC**: Porcentagem de Requisitos Cobertos quando a modelagem é realizada com UML + diretrizes.

  - R_t - Quantidade total de requisitos do sistema.
  - R_d - Quantidade de requisitos cobertos nos modelos de projeto, modelagem realizada usando UML + diretrizes.
  - $PRC = R_d/R_t$

M1:

- MGPRC: Média geral das PRCs
  $MGPRC = \frac{1}{n}\sum_{i=1}^{n} PCR_i$
  Onde:
  - PRC_i corresponde ao PRC do estudo de caso índice i;
  - n é o número de estudos de caso aplicados.

Q2: Quão aplicáveis são as diretrizes em STRs?

- **PDNA**: Porcentagens de Diretrizes Não Aplicadas
  - Q_{di} - Quantidade total de diretrizes independentes. Entenda-se "diretrizes independentes" por aquelas que sempre podem ser aplicadas, ou seja, não necessitam que exista um requisito específico para que esta seja aplicada (*e.g.* D26).
  - Q_{dd} - Quantidade total de diretrizes dependentes. Entenda-se "diretrizes dependentes" por aquelas que nem sempre podem ser aplicadas, ou seja, dependem que um requisito específico seja satisfeito para que a mesma possa ser aplicada (*e.g.* D27).
  - Q_{ui} - Quantidade de diretrizes independentes usadas na modelagem dos sistemas.
  - Q_{ud} - Quantidade de diretrizes dependentes usadas na modelagem dos sistemas.
  - $PDNA_i = 1 - (\frac{Q_{ui}}{Q_{di}})$
  - $PDNA_d = 1 - (\frac{Q_{ud}}{Q_{dd}})$

M2:

- $MTPDNA_i$: Média total das $PDNA_i$s.
  $MTPDNA_i = \frac{1}{n} \sum_{j=1}^{n} PCR_{ij}$
  Onde:
  - $PDNA_{ij}$ corresponde a $PDNA_i$ do estudo de caso índice j;
  - n é o número de estudos de caso aplicados.

M3:

- $MTPDNA_d$: Média total das $PDNA_d$s.
  $MTPDNA_d = \frac{1}{n} \sum_{i=1}^{n} PCR_{di}$
  Onde:
  - $PDNA_{di}$ corresponde a $PDNA_d$ do estudo de caso índice i;
  - n é o número de estudos de caso aplicados.

Q3 - Existe uma melhoria de qualidade nos artefatos de modelagem com o uso das diretrizes?

- MNQSD: Média das notas de qualidade dos modelos sem o uso das diretrizes.
  $MNQSD = \frac{1}{n} \sum_{i=1}^{n} NS_i$
  Onde:
  - NSi corresponde a nota de qualidade dada pelo grupo i para os artefatos gerados sem as diretrizes;
  - n é o número de grupos participantes do estudo.

- MNQCD: Média das notas de qualidade dos modelos com o uso das diretrizes.
  $MNQCD = \frac{1}{n} \sum_{i=1}^{n} ND_i$
  Onde:
  - NDi corresponde a nota de qualidade dada pelo grupo i para os artefatos gerados com o uso das diretrizes;
  - n é o número de grupos participantes do estudo.

M4:

- **DMQ**: Diferença das médias de qualidade.
  $DMQ = MNQCD - MNQSD$

A Figura 3.11 traz o mapa de questionamentos e métricas (conforme instrui GQM) que foram usadas para atingir o objetivo de estudo.

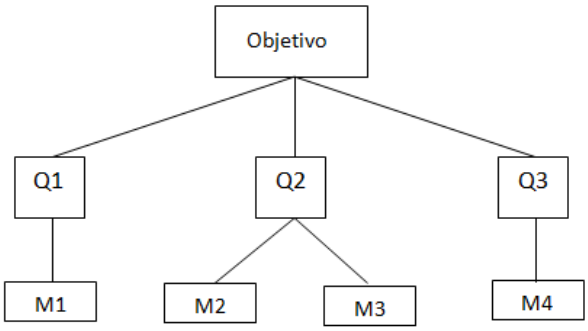

Figura 3.11: Mapa do GQM Proposto.

3.6.3 Planejamento

Seleção do Contexto

Nosso estudo supõe o processo *off-line*, sendo que as especificações dos sistemas (estudos de caso) foram fornecidas aos participantes e as métricas captadas mediante a aplicação de um questionário e a análise dos artefatos gerados. Estas coletas e análises somente foram realizadas após a finalização da aplicação dos estudos de caso.

Os participantes selecionados para participar do estudo foram alunos da graduação em Ciência da Computação da UFCG que cursaram a disciplina de Engenharia de Software II no período 2010.2. Dado que a disciplina possuiu um foco relacionado em aspectos avançados de teste, os resultados deste trabalho foram introduzidos como base para o projeto da disciplina e os artefatos gerados neste projeto serviram como material de análise para estudo experimental.

Os alunos envolvidos no estudo já possuíam experiência prévia no uso geral da notação UML. Quanto à experiência dos participantes com relação ao planejamento e modelagem de STRs, em sua maioria os alunos não tinham contato prévio com este tipo de sistema. Para

suprir tal deficiência algumas aulas foram ministradas com o intuito de fazer com que estes obtivessem uma noção inicial do contexto com que trabalhariam.

Os participantes receberam especificações simplificadas, baseadas em sistemas reais. As competências dos alunos participantes são comparadas às competências dos demais projetistas/desenvolvedores de STRs iniciantes, então, o contexto possui caráter específico.

Resumo do estudo empírico:

- O processo: *off-line*
- Os participantes: alunos de graduação
- Realidade: problemas reais
- Generalidade: específico

Seleção de Casos

Para realização da atividade foram utilizadas três especificações de aplicações RT reativos de diferentes âmbitos (uma aplicação de celular, um sistema de alarmes e uma aplicação de TV digital). As especificações entregues foram todas escritas em linguagem natural, seguindo um padrão informal (descrição do sistema - requisitos não funcionais - requisitos funcionais). As aplicações selecionadas tiveram o objetivo de evidenciar, principalmente, características *real-time* (*e.g.* restrições de tempo e eventos assíncronos). As especificações entregues aos participantes do estudo podem ser visualizadas no Apêndice A deste documento.

Definição das Hipóteses

Hipótese nula (H0): A modelagem obtida com uso das diretrizes cobre menos de 95% dos requisitos dos sistemas.

O valor *95%* foi definido pela necessidade que um STRs possui de cumprir a quase totalidade dos requisitos pré-estabelecidos.

R_t - Quantidade total de requisitos do sistema.

R_d - Quantidade de requisitos cobertos nos modelos de projeto, modelagem realizada usando UML + diretrizes.

***H0*:** $(\frac{R_d}{R_t}) < 0.95$

Métricas Necessárias: MGPRC

Hipótese alternativa (H1): A modelagem obtida com uso das diretrizes cobre mais de, ou pelo menos, 95% dos requisitos dos sistemas.

***H1*:** $(\frac{R_d}{R_t}) \geq 0.95$

Métricas Necessárias: MGPRC

Hipótese nula (H2): A maioria das diretrizes não são aplicáveis para a modelagem de sistemas de tempo real reativos.

H2: ($MTPDNA_i$ ***> 0.5 AND*** $MTPDNA_d$ ***> 0.5)***

Métricas Necessárias: $MTPDNA_d$, $MTPDNA_i$

Hipótese alternativa (H3): A maioria das diretrizes são aplicáveis para a modelagem de sistemas de tempo real reativos.

H3: NOT($MTPDNA_i$ ***> 0.5 AND*** $MTPDNA_d$ ***> 0.5)***

Métricas Necessárias: $MTPDNA_d$, $MTPDNA_i$

Hipótese nula (H4): As diretrizes não melhoram a qualidade dos artefatos de modelagem gerados.

$H4 : DMQ \leq 0$

Métricas Necessárias: DMQ

Hipótese alternativa (H5): As diretrizes melhoram a qualidade dos artefatos de modelagem gerados.

H5: DMQ > 0

Métricas Necessárias: DMQ

Descrição da Instrumentação

Um total de 36 alunos estavam matriculados na disciplina de Engenharia de Software II (período 2010.2). Destes, foram criados seis grupos de seis alunos cada. Cada grupo, numa primeira etapa recebeu a especificação de um STR (alocação realizada aleatoriamente) e deveria modela-lo com a notação que lhes fossem mais familiar. Numa segunda etapa, cada

grupo recebeu então três artefatos: i) a especificação correspondente a um novo sistema (esta segunda especificação deveria ser diferente da recebida na primeira etapa para assim evitar que o grupo já possuísse conhecimento adquirido); ii) um tutorial que listava as diretrizes, bem como demonstrava como aplicá-las; e iii) um questionário a ser respondido pelo grupo com aspectos qualitativos (o questionário aplicado está disponível para visualização no Apêndice B).

Ao final da segunda etapa, cada grupo teve que entregar: i) os artefatos de modelagem; e ii) o questionário devidamente respondido com as análises subjetivas a respeito da participação no estudo.

Seleção de Indivíduos

Como participantes para o estudo, dada a dificuldade em conseguir reais projetistas de STRs que se dispusessem a participar da atividade, utilizou-se um grupo de estudantes da graduação do curso de Ciência da Computação da Universidade Federal de Campina Grande (UFCG). Visto que era necessário que os indivíduos envolvidos já possuíssem certa experiência no uso de UML, foi escolhida a disciplina de Engenharia de Software II [19] para aplicação deste estudo. Os alunos que cursam esta disciplina ou já cursaram a disciplina de Sistemas de Informações II [21] (que ensina o uso de UML de forma avançada para modelagem de sistemas) ou a estavam cursando no momento da aplicação do estudo. Este cuidado foi levado em consideração para assim evitar que problemas como o não conhecimento das notações vinhessem a afetar os resultados do estudo.

A fim de dissipar os problemas do não conhecimento do contexto dos STRs, no início da disciplina algumas aulas foram ministradas com o intuito de fornecer o embasamento teórico para os mesmos. É importante destacar que, diferindo da maioria dos demais grupos, um destes era composto por indivíduos que já possuíam experiência prévia com a modelagem e desenvolvimento de STRs, tornando o conjunto de participantes do estudo um grupo ainda mais genérico e representativo.

Dada a necessidade de participantes com características especiais e visto que o número de possíveis candidatos no contexto da UFCG é bastante reduzido, a escolha dos indivíduos foi realizada através de técnica não-probabilística.

Variáveis

Variável Independente: Especificações de STRs. Mais precisamente, um conjunto de documentos contendo as descrições dos sistemas e de seus requisitos (em linguagem natural).
Variáveis Dependentes:

- Diretrizes pouco úteis ou não aplicáveis;
- Principais dificuldades na utilização das diretrizes.
- Principais benefícios da utilização das diretrizes.
- Nível de melhoria de qualidade na modelagem referente ao uso das diretrizes

Ameaças de Validade

1. **Validade interna**: como mencionado na seção 3.6.3 (*Seleção de Indivíduos*), para o estudo, foi proposto utilizar alunos com experiência em modelagem, provenientes de um curso de graduação, mas, com pouco conhecimento no contexto dos STRs. Assim, assume-se que estes são indivíduos representativos para a população de desenvolvedores/projetistas iniciantes de STRs.

   Para redução de fatores que não são interesse para o estudo e, portanto, para aumentar a validade interna do estudo, aulas sobre STRs foram ministradas, para assim nivelar o conhecimento os candidatos na área.

   Outros problemas que podem afetar a validade interna do estudo e as respectivas soluções planejadas são:

   - As especificações escolhidas como estudos de caso não serem claras o suficiente, proporcionando dúvidas quando à modelagem dos requisitos nos participantes.
     Solução: O planejador do estudo, bem como dois estagiários docentes da disciplina, ficaram responsáveis por sanar quaisquer dúvidas com respeito às especificações recebidas.
   - Os questionários não serem preenchidos corretamente pelos participantes.
     Solução: Ao planejador/condutor dos experimentos também foram reportados os

artefatos de modelagem finais da aplicação dos estudos de caso de cada grupo. Logo, quaisquer inconsistências puderam ser analisadas.

- Os participantes não estarem motivados ou com tempo suficiente para realização dos estudos de caso com a dedicação necessária.
  Solução: a realização deste estudo foi incluída como parte da nota final na disciplina de Engenharia de Software II.

2. **Validade de construção**: as especificações dos sistemas usados para realização do estudo experimental foram provenientes de aplicações *real-time* reativas reais. A fim de estarem adequadas ao escopo do experimento, foram usados sistemas simples ou sistemas complexos foram simplificados.

3. **Validade externa**: como foram mencionados nas seções Seleção de indivíduos e Validade interna, os participantes do estudo em geral podem ser considerados representativos para a população dos desenvolvedores/projetistas de STRs iniciantes.
   Os materiais utilizados no estudo (especificações de STRs reativos) podem ser considerados representativos para a área por serem sistemas reais da área objeto de estudo.
   Características temporais poderiam ser problemáticas para a execução do estudo, visto que a entrega dos resultados do projeto pelos participantes coincidiu com o final do período letivo da instituição, onde os alunos estavam com acúmulo de testes a realizar, diminuindo assim a motivação e disposição dos mesmos com a realização do experimento, principalmente em uma disciplina optativa.

3.6.4 Resultados

A partir dos dados coletados e das métricas calculadas (vide Apêndice C), um conjunto de observações puderam ser estabelecidas, bem como conclusões sobre as hipóteses de estudo puderam ser realizadas.

Quanto às hipóteses H0 e H1:

Dado que a métrica MGPRC obteve o valor de 0.975 (mais de 97%), com variância de 0.00375 e intervalo de confiança $[0.9107; 1.0392]$, com nível de 95%, é possível concluir que é mais provável que a hipótese H0 seja falsa em detrimento da veracidade da hipótese

H1. Ou seja, a quase totalidade dos requisitos fornecidos para as equipes foram passíveis de modelagem com o uso das diretrizes.

Com tais resultados a primeira questão de pesquisa levantada (Seção 3.6.2) pode ser respondida, atestando que as diretrizes são realmente úteis para a modelagem de requisitos.

Possíveis ameaças a estas conclusões:

- Como a verificação da quantidade de requisitos foi realizada mediante computação das respostas fornecidas pelos próprios participantes à questão número 6 do questionário (vide Apêndice B), possivelmente algumas destas respostas podem ter sido fornecidas sem a exatidão necessária para um estudo investigativo, como se é proposto.

- A definição se um requisito foi ou não modelado é uma questão subjetiva. Dependendo do grau de experiência do projetista e do desenvolvedor, é possível que na opinião de um o requisito esteja claramente modelado, enquanto para o outro isto pode não ser verdade. Esta subjetividade está intrínseca à métrica estabelecida, e este pode ser um fator que pode levantar ameaças as conclusões obtidas.

Quanto às hipóteses H2 e H3:

A fim de averiguar se as diretrizes eram realmente aplicáveis no contexto de STRs e assim responder a segunda questão de pesquisa (Seção 3.6.2), foram calculadas as métricas $MTPDNA_i$ e $MTPDNA_d$ que obtiveram os seguintes valores: 0.128 (com variância de 0.00517, e intervalo de confiança [0.0528; 0.2038] com nível de 95%) e 0.363 (com variância de 0.0321, e intervalo de confiança [0.1750; 0.5516] com nível de 95%), respectivamente (vide Apêndice B para maiores detalhes). A partir destes dados é possível atestar que é mais provável que a hipótese H2 seja falsa em detrimento da veracidade da sua hipótese alternativa (H3), visto que ambas métricas $MTPDNA_i$ e $MTPDNA_d$ tiveram valores menores que 0.5, conforme estabelecido na hipótese H3.

Como foi verificado que a maioria das diretrizes foram aplicadas em cada estudo de caso (em média, bem mais que a metade das diretrizes foram aplicadas), é possível responder a segunda questão de pesquisa atestando que sim, as diretrizes estabelecidas têm um bom nível de aplicabilidade (acima de 50%).

Possíveis ameaças a estas conclusões:

- A possibilidade dos sistemas usados como estudos de caso não serem representativos do contexto dos STRs reativos. Buscou-se, dentre as limitadas opções que tinhamos, escolher sistemas de diferentes âmbitos na classe dos STRs reativos. Porém, como esta trata-se de uma classe bastante grande, é possível que alguma importante subcategoria tenha sido desconsiderada, impossibilitando uma maior generalidade nas conclusões estabelecidas.

Quanto às hipóteses H4 e H5:

Tomando por base as notas fornecidas pelas equipes via questionário (questões 4 e 5, Apêndice B) as métricas MNQSD e MNQCD foram calculadas, resultando nos seguintes valores: 8.083 (com variância de 0.6416, e intervalo de confiança [7.242; 8.9239] com nível de 95%) e 8.666 (com variância de 0.9666, e intervalo de confiança [7.6348; 9.6984] com nível de 95%), respectivamente. Estas métricas serviram por base para o cálculo da métrica DMQ que quantificou a melhoria de qualidade dos artefatos de modelagem com o uso das diretrizes. Esta métrica teve como resultado o valor de 0.583 que equivale a uma melhoria de qualidade de aproximadamente 7.3%.

Conforme questionado na terceira pergunta de pesquisa, a partir dos dados obtidos é possível inferir que houve sim uma melhoria na qualidade dos artefatos de modelagem quando estes são gerados usando as diretrizes de modelagem. Logo, é possível atestar que é mais provável que a hipótese H4 seja falsa em favor da veracidade da sua hipótese alternativa (H5).

Possíveis ameaças a estas conclusões:

- Entendendo que as notas foram dadas pelos próprios participantes, e em momentos diferentes, fatores subjetivos podem ter atrapalhado o julgamento e aferição das notas. Dentre esses fatores é possível citar: i) como o estudo foi realizado num momento em que os participantes estavam fortemente envolvidos com outras atividades (*e.g.* exames finais das disciplinas de graduação, *deadlines* de projetos de outras disciplinas, etc), tais fatores podem ter feito com que os mesmo não dedicassem tempo suficiente para aprendizado e consequentemente os resultados não terem sido tão eficientes quanto esperado; ii) o possível baixo interesse dos integrantes do estudo em se dedicar ao projeto, produzindo resultados com qualidade questionável; e iii) alguns dos participantes

Grupos	Dados Coletados		
1	R_t = 20 R_d = 20 PRC = 1	Q_{di} = 31 Q_{dd} = 15 Q_{ui} = 29 Q_{ud} = 7 $PDNA_i$ = 0.06 $PDNA_d$ = 0.53	NS = 7.5 ND = 9
2	R_t = 20 R_d = 17 PRC = 0.85	Q_{di} = 31 Q_{dd} = 15 Q_{ui} = 24 Q_{ud} = 7 $PDNA_i$ = 0.22 $PDNA_d$ = 0.53	NS = 9 ND = 10
3	R_t = 11 R_d = 11 PRC = 1	Q_{di} = 31 Q_{dd} = 15 Q_{ui} = 28 Q_{ud} = 6 $PDNA_i$ = 0.09 $PDNA_d$ = 0.6	NS = 9 ND =7
4	R_t = 12 R_d = 12 PRC = 1	Q_{di} = 31 Q_{dd} = 15 Q_{ui} = 28 Q_{ud} = 11 $PDNA_i$ = 0.09 $PDNA_d$ = 0.26	NS = 8 ND = 9
5	R_t = 20 R_d = 20 PRC = 1	Q_{di} = 31 Q_{dd} = 15 Q_{ui} = 28 Q_{ud} = 9 $PDNA_i$ = 0.09 $PDNA_d$ = 0.4	NS = 7 ND = 8.5
6	R_t = 20 R_d = 20 PRC = 1	Q_{di} = 31 Q_{dd} = 15 Q_{ui} = 24 Q_{ud} = 9 $PDNA_i$ = 0.22 $PDNA_d$ = 0.4	NS = 8 ND = 8.5

Legenda

R_t: Quantidade total de requisitos do sistema.
Rd: Quantidade de requisitos cobertos nos modelos de projeto, modelagem realizada usando UML + diretrizes.
PRC: Porcentagem de Requisitos Cobertos quando a modelagem é realizada com UML + diretrizes.
Q_{di}: Quantidade total de diretrizes independentes.
Q_{dd}: Quantidade total de diretrizes dependentes.
Q_{ui}: Quantidade de diretrizes independentes usadas na modelagem dos sistemas
Q_{ud}: Quantidade de diretrizes dependentes usadas na modelagem dos sistemas.
$PNDA_i$: Porcentagens de Diretrizes Independentes Não Aplicadas.
$PDNA_d$: Porcentagens de Diretrizes Dependentes Não Aplicadas
NS: Nota de qualidade atribuída para os artefatos gerados sem as diretrizes.
ND: Nota de qualidade atribuída para os artefatos gerados com o uso das diretrizes.

Figura 3.12: Valores coletados para o cálculo das métricas.

do estudo já possuíam conhecimento prévio de outras notações de modelagem, e usaram estas na modelagem durante a fase inicial do estudo. Este conhecimento anterior pode os ter direcionado estes a atribuir uma nota de qualidade inferior ao esperado para o artefatos gerados com o uso das diretrizes.

Os valores coletados, bem como os resultados dos cálculos das métricas estão resumidamente apresentados nas tabelas das Figuras X e Y. Maiores detalhes a respeito destes elementos estão presentes no Apêndice C.

Conclusões a respeito dos aspectos subjetivos

Algumas das questões presentes no questionário entregue pelos participantes foram relacionadas à aspectos subjetivos. Estas questões foram pensadas a fim de avaliar o conjunto de diretrizes por outros aspectos que não conseguiriam ser analisados numericamente.

Métrica	Valor
MTPRC	**0.975** Variância = 0.00375 Intervalo de confiança com nível de 95% = [0.9107; 1.0392]
$MTPDNA_i$	**0.128** Variância = 0.00517 Intervalo de confiança com nível de 95% = [0.0528; 0.2038]
$MTPDNA_d$	**0.363** Variância = 0.0321 Intervalo de confiança com nível de 95% = [0.1750; 0.5516]
DQM	**0.583**

Legenda
MTPRC: Média total das PRCs.
$MTPDNA_i$: Média Total das PDNAs independentes.
$MTPDNA_d$: Média Total das PDNAs independentes.
DQM: Diferença das médias de qualidade.

Figura 3.13: Valores das métricas.

A seguir, serão apresentadas algumas das conclusões estabelecidas a partir das respostas fornecidas a estas perguntas, bem como das análises que os condutores do estudo observaram no decorrer da aplicação dos mesmos. São elas:

- Dentre as principais dificuldades apontadas pelos participantes no uso das diretrizes, a mais citada foi o pouco tempo que dispunham para se dedicar ao estudo/aprendizado das mesmas. Este fator pode ter afetado negativamente os resultados alcançados, pois era de suma importância para a boa execução da atividade, que estes participantes compreendessem bem o papel de cada diretriz.

- Todos os participantes atestaram que consideram as diretrizes estabelecidas úteis para modelagem de STRs reativos. Todas as respostas para a questão 11 (Apêndice B) foram entre os níveis "Útil" ou "Bastante Útil", fortalecendo assim, a conclusão tomada a respeito da primeira questão de pesquisa.

- Dentre os comentários mais pertinentes, os participantes indicaram que o uso das diretrizes como um todo custou muito tempo e que seria interessante que as diretrizes fossem mais independentes entre si, possibilitando que, caso necessário, somente parte destas pudessem ser usadas, diminuindo assim o tempo para aprendizado e de aplicação.

- As respostas à pergunta 10 do questionário ("*Na sua opinião, o sistema poderá ser implementado mais facilmente após a atividade de modelagem realizada nesta etapa? Quais fatores lhe direcionam a esta resposta?*") trouxeram maiores indícios que o conjunto de diretrizes realmente cumpre seu papel em ajudar que os artefatos produzidos

tenham mais legibilidade e conseqüentemente isto facilite uma posterior implementação do sistema. Na opinião dos participantes, após a modelagem com o uso das diretrizes, existiu uma maior clareza de entendimento do sistema, isto, segundo eles, permitirá que este seja mais facilmente implementado. Ou seja, a lacuna entre projeto e implementação conseguiu ser diminuída neste sentido. Porém, esta é uma questão que necessita que estudos mais aprofundados sejam realizados para assim comprovar tais suspeitas.

- Especificamente quanto ao grupo que já possuía um perfil mais experiente quanto à modelagem e desenvolvimento de STRs, este se mostrou bastante satisfeito com o resultado da utilização das diretrizes. O grupo evidenciou que, apesar do maior tempo gasto para modelagem, o grau de dificuldade encontrada por eles para modelar usando UML + diretrizes foi efetivamente menor do que utilizando modelos formais sem a adoção e aplicação das diretrizes propostas.

Conclusões Gerais do Estudo

Como conclusões gerais da aplicação dos estudos de caso, é possível analisar que os resultados obtidos dão indícios que os objetivos iniciais, que motivaram a construção das diretrizes foram de fato alcançados, ou seja, o conjunto de diretrizes é útil e usável, os artefatos produzidos com o uso das mesmas foram considerados relevantes e interessantes pelos participantes. Tais conclusões nos dão indícios que estas devam também ser verdadeiras para o universo de projetistas, principalmente iniciantes, de STRs.

Vale salientar que, dada as limitações já previstas (*e.g.* dificuldade em conseguir especificações de STRs e de selecionar indivíduos dispostos a participar do estudo), o estudo teve um caráter limitado. Não foi possível realizar as repetições necessárias para que testes estatísticos pudessem ser usados (*e.g.* teste de hipóteses). Por isso, os resultados alcançados nos fornecem apenas indícios que as conclusões realizadas são coerentes. Porém, um estudo maior nesse sentido deve ser realizado posteriormente para que realmente um caráter de validação possa ser alcançado.

3.7 Considerações Finais do Capítulo

Este capítulo apresentou um conjunto de artefatos desenvolvidos com o intuito de preencher lacunas e tornar mais completos, claros e úteis os elementos de modelagem produzidos para projeto de STRs, no nível independente de plataforma. Os artefatos desenvolvidos foram os seguintes: i) o *Real-Time Design Profile*, perfil UML para identificação e representação dos principais elementos de STRs reativos; ii) o pacote *Real Time Elements*, desenvolvido para identificação e reuso de estruturas comuns para arquiteturas de STRs com o intuito de padronização e auxílio em projetos de sistemas desta classe; e iii) um conjunto de diretrizes de modelagem, desenvolvidas para guiar os projetistas no projeto de STRs e no uso dos artefatos anteriormente apresentados. Ainda neste capítulo foram apresentados os resultados do estudo experimental, bem como o modo como este foi executado. O estudo comprovou a real utilidade dos artefatos desenvolvidos para melhoria da qualidade dos resultados de modelagem de STRs usando UML, principalmente para projetistas inexperientes neste contexto.

Capítulo 4

Extensão de UTP para STRs

4.1 Motivação

Aplicando os conceitos de MBT a modelos de *design* de um sistema podemos obter casos de teste referentes a cenários comportamentais no nível independente de plataforma. A Figura 4.1 apresenta um exemplo de um possível caso de teste para o "Sistema de Alarmes", sistema exemplo que está resumidamente descrito na Seção 3.1. É possível notar que para que este caso de teste possa ser de fato implementável, um conjunto de outros elementos devem existir, para assim fornecer o suporte necessário para esta execução e análises. Como exemplos destes "elementos extras" que devem existir para permitir a execução deste caso de teste é possível citar: um elemento que realize a arbitragem da execução do caso de teste, a pré-configuração dos elementos envolvidos nos testes (*e.g.* monitor do edifício, tratador da interrupção de intruso localizado, etc), o escalonamento e controle dos elementos intanciados para teste, etc. Estes são então alguns dos elementos que compõem uma arquitetura de teste.

UTP (*UML Testing Profile*) é um perfil de testes da UML cujo seu intuito é de auxiliar o projeto de arquiteturas de teste no nível independente de plataforma. Porém, como UTP foi essencialmente concebido para auxilar a atividade de testes de propósito geral, este acaba por pecar no tocante a proporcionar o projeto de arquiteturas de teste para o contexto dos STRs (que possui particularidades e restrições próprias). Como exemplo destas deficiências é possível citar: a permissividade do uso de uma arbitragem *off-line* usando análise de arquivos de *log*, um ambiente simplificado e total controlado para execução dos casos de teste, elemen-

Caso de Teste	
Objetivo:	Verificar se o Monitor do Edifício está sendo avisado caso haja detecção de movimento.
Entradas:	Movimento na área onde o alarme está ativo
Pré-condição:	Alarme funcionando; Energia estável
Pós-condição:	Monitor do Edifício devidamente avisado; Prosseguimento normal da verificação do alarme (duas a cada segundo)
Passos:	(Executando) -> Interrupção [Detecção de movimento] -> (Preparando) -> LançarInterrupção -> (Executando Tratamento) -> (Executando)*

* Mesma configuração que estava antes da interrupção ter sido lançada

Figura 4.1: Exemplo de caso de teste para o Sistema de Alarmes.

tos para forçar o lançamento de interrupções, etc. Tais elementos, dentre outros, não estão presentes na configuração atual estabelecida por UTP, evidenciando assim a necessidade de extensão/melhoramento do mesmo.

4.2 Solução

A fim de tentar atacar a problemática de complementar UTP com o intuido de torná-lo eficaz também para o desenvolvimento de arquiteturas de teste para STRs, desenvolvemos um conjunto de extensões do perfil de teste com o objetivo de torná-lo mais expressivo e útil para o contexto de testes dos STRs reativos.

Nas próximas seções as extensões desenvolvidas são apresentadas, em seguida é apresentado o primeiro módulo da ferramenta RTTAG (responsável pela geração automática dos modelos PITM) e, por fim, são apresentados os estudos de casos aplicados com o objetivo de avaliar as extensões, bem como a ferramenta criada.

4.3 Extensões

Com o intuito de não modificar a semântica já consolidada dos elementos presentes em UTP, foi tomada a decisão de criar um novo pacote de extensões ao perfil, onde, o mesmo conteria os novos elementos que, na verdade, estendem os elementos originais adicionando aos mesmos a semântica necessária para permitir o melhor tratamento para o contexto de testes para STRs. Portanto, o novo pacote criado, o chamado “UTP RT”, se conecta ao pacote original de UTP através de uma relação de *merge*, conforme apresentado na figura

4.2.

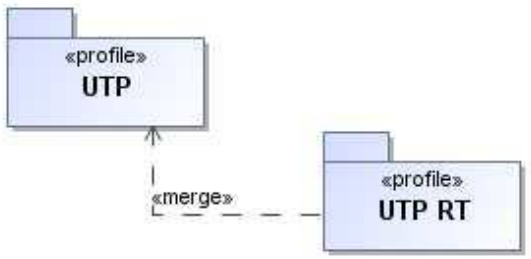

Figura 4.2: Estrutura de pacotes.

A Figura 4.3 apresenta a representação visual do novo pacote criado, bem como do pacote auxiliar anexado, o "*Auxiliar Elements*". O pacote auxilar foi desenvolvido com o objetivo de ajudar organizacionalmente no complemento das extensões criadas ao perfil de testes.

Na próxima seção é apresentado individualmente cada novo elemento (estereótipo, interface, classe, enumeração ou tipo de dado) presente no perfil *UTP RT*. A apresentação terá o seguinte padrão seqüencial: i) o elemento será identificado (*id* Elemento); ii) quando cabível, uma breve descrição do elemento de UTP ao qual o novo elemento irá estender (*id* Descrição do elemento em UTP); iii) a motivação que levou a criação da extensão (*id* Motivação para extensão); iv) como se deu a extensão (*id* Solução); v) regras OCL, quando necessário, para complementar a definição da extensão (*id* Restrições); e vi) um exemplo genérico da aplicação do elemento numa arquitetura no nível PITM.

4.3.1 UTP RT

As extensões apresentadas nesta seção visam, em sua maioria, complementar os elementos já existentes de UTP para o contexto RT. Para tal, a maior parte dos elementos são especializações dos elementos presentes no pacote original de UTP, permitindo assim que semânticas novas sejam agregadas a estes elementos, bem como que o uso dos novos elementos propostos possam ser usados somente caso o sistema a ser testado seja um STR.

Elemento: Estereótipo *Test Context RT*

Descrição do elemento em UTP: Estrutura que coordena toda a configuração e execução da atividade de testes. Armazenando, inclusive, os casos de testes a serem executados.

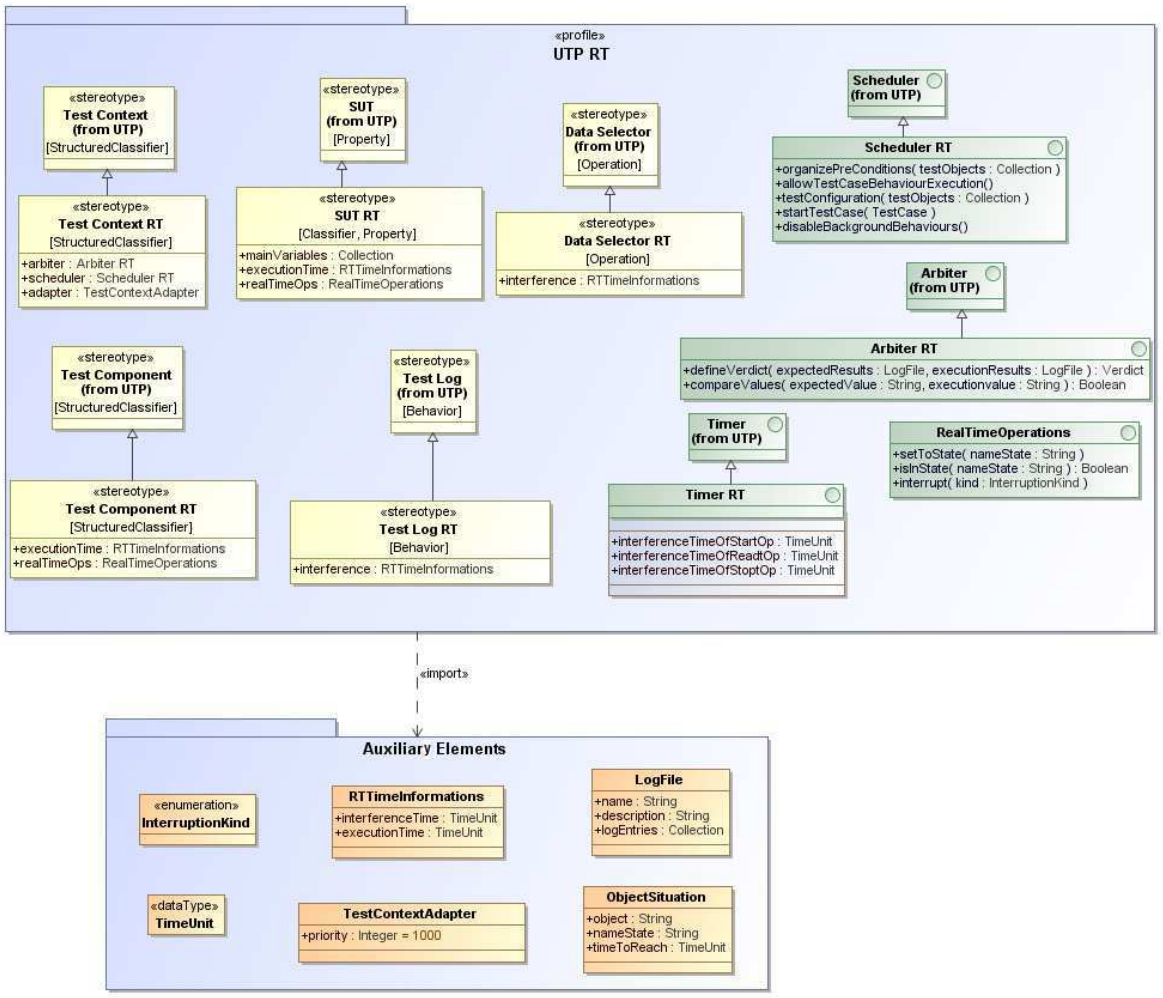

Figura 4.3: Extensões RT para o UTP .

Motivação para extensão: Em uma arquitetura de teste RT é necessário que existam elementos especiais para conduzir a execução e configuração dos testes de uma maneira não-transparente para o testador, principalmente para aferição de vereditos e escalonamento de estruturas envolvidas nos casos de teste. É necessário também, num contexto onde vai-se lidar por diversos momentos com a manipulação de processos, que haja a garantia que a estrutura *Test Context* possua processamento independente e que esta realmente tenha maior prioridade para execução em relação aos demais processos existentes.

Solução: Criação do estereótipo *Test Context RT* que herda as características provenientes do estereótipo *Test Context* (de UTP) acrescido de três novos atributos, são eles:

- *arbiter*: árbitro especial (sobrescrito do árbitro de UTP), usado exclusivamente quando o objetivo dos testes é puramente RT e com isto seja necessário que o mesmo seja um

árbitro passivo (entenda-se arbitro passivo como aquele que só aferirá um veredito ao caso de teste após a completa finalização da execução do mesmo).

- *scheduler*: escalonador especial (sobrescrito do *scheduler* de UTP), estrutura responsável por escalonar toda a atividade de testes, o escalonamento é realizado explicitamente em todas etapas. Esta é uma característica bastante importante, pois em testes RT o testador necessita ter total controle sobre quais processos estão executando a qualquer momento.

- *adapter*: elemento que aferirá à estrutura *Test Context* a prioridade necessária para que esta consiga reger toda atividade de testes.

Restrições:

Código Fonte 4.1: Restrição OCL para o estereótipo *Test Context RT*

```
Context Test_Context_RT inv:
    self.isActive = true
```

Esta restrição força que a estrutura que possua o estereótipo *Test Context RT*, também seja necessariamente representada como um objeto ativo.

Exemplo de Aplicação: Na Figura 4.4 é possível visualizar a classe *ComunicationManager_TestContext*. Esta recebeu o estereótipo *Test Context RT* indicando que foi criada exclusivamente com o propósito de gerenciar a execução da atividade de testes. É possível observar que a classe *Test Context* tem as seguintes características: i) implementa as interfaces *SchedulerRT* e *ArbiterRT* (que serão explicadas posteriormente); ii) possui um conjunto de casos de teste; iii) possui prioridade com valor elevado; iv) é um objeto ativo; e v) possui uma instância do SUT. Desta forma, esta estrutura possui a capacidade de lidar com a execução dos casos de teste.

Elemento: Estereótipo *SUT RT*

Descrição do elemento em UTP: O sistema, ou parte dele, que será testado.

Motivação para extensão: Quando o objetivo dos testes é analisar aspectos RT (*e.g.* se uma restrição de tempo é ou não satisfeita), muitas vezes se faz necessário instrumentar o código

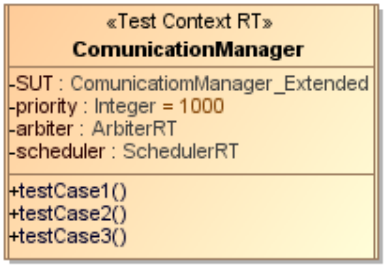

Figura 4.4: Exemplo de aplicação do estereótipo *Test Context RT*.

a ser testado de alguma forma para que os valores de tempo sejam capturados. Normalmente, esta instrumentação acaba por interferir também nos tempos finais de execução, o que pode levar à confusão e/ou erros na atestação dos vereditos. Por isso, é necessário que exista numa arquitetura RT uma investigação preliminar, e que esta deixe claro para as demais estruturas presentes nesta arquitetura, de quanto é este tempo de interferência. É importante também que exista uma análise de quais serão as variáveis, presentes no SUT, cujos valores deverão ser armazenados para posterior análise (método bastante utilizado para teste e aferição de vereditos em STRs).

É também bastante comum nos testes de STRs que o SUT inicie sua execução em diferentes momentos, não necessariamente no seu estado inicial de execução (estados provenientes do comportamento modelado, por exemplo, na sua máquina de estados). Pensando na facilitação desta característica, estas ações seriam melhor demonstradas numa arquitetura de testes independente de plataforma, se estivessem encapsuladas de alguma maneira.

Solução: Criação do estereótipo *SUT RT* que herda as características provenientes do estereótipo *SUT* (de UTP), acrescido das seguintes aspectos:

- *mainVariables*: coleção das principais variáveis presentes no SUT, variáveis estas que deverão ser "vigiadas" e seus valores correntes armazenados durante a execução dos casos de teste para servirem de base para identificação dos vereditos dos mesmos.

- *executionTime*: atributo que deixará explícito o tempo que o SUT gasta, originalmente (não no contexto de testes), para execução. Este, deverá ser utilizado para averiguação e subtração dos tempos de interferência nos casos de teste.

- *realTimeOps*: implementação da interface *RealTimeOperations*, permitindo que os casos de teste possam ser melhor documentados e construídos com o uso destas opera-

ções, e permitindo que casos de teste que não iniciam partindo do estado inicial do SUT possam ser desenvolvidos, acarretando consequentemente em casos de teste menores e mais compreensíveis. Permitirá também a imposição do lançamento de interrupções ao SUT.

Restrições:

Código Fonte 4.2: Restrição OCL para o estereótipo *SUT RT*

```
Context SUT_RT inv:
    self.interferenceTime.oclIsUndefined = true
```

Restrição que força que, obrigatoriamente, o atributo *interferenceTime* esteja definido no modelo.

Exemplo de Aplicação: A Figura 4.5 apresenta a classe *ComunicationManager_Extended*. Esta estende a classe *ComunicationManager* (estrutura original sob teste), adicionando a esta características que possibilitem o teste de aspectos *real-time* (os atributos *executionTime*,*mainVariables* e a interface *RealTimeOperations*). É importante destacar que esta estrutura *SUT RT* deve estar associada às estruturas *Test Components*, para assim permitir que os dados dos testes, bem como sua execução possa ocorrer coerentemente.

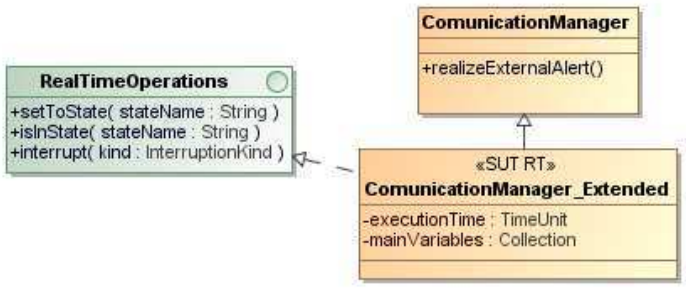

Figura 4.5: Exemplo de aplicação do estereótipo *SUT RT*.

Elemento: Estereótipo *Test Component RT*

Descrição do elemento em UTP: Objeto do sistema que se relacionará com o *SUT*, produzindo e/ou consumindo elementos que provém dos casos de teste. Este elemento deve possuir um conjunto de interfaces que possibilite a comunicação com outros *Test Components* ou com o *SUT*.

Motivação para extensão: Assim como no SUT, é importante identificar especificamente aquelas estruturas que auxiliarão a condução da execução dos casos de testes RT. Como, muitas vezes os *Test Components* são apenas emuladores das estruturas originais que interagem com o SUT, criados unicamente com propósitos de teste, se os testes são RT é importante que estes emuladores possuam claramente um tempo de execução equivalentes aos tempo de execução originais para que questões temporais não sejam atrapalhadas. É interessante também que exista um mecanismo facilitador para que estas estruturas possam ser usadas nos casos de teste em momentos (estados) diferentes do inicial, este tipo de mecanismo poderia ser bastante útil, principalmente, no teste de interrupções (característica bastante comum nos STRs).

Solução: Criação do estereótipo *Test Component RT* que herda as características provenientes do estereótipo *Test Component* (de UTP), acrescido dos seguintes atributos:

- *executionTime*: atributo que deixa claro para a arquitetura de testes o valor, que deve ser cuidadosamente computado anteriormente, do tempo de execução que o *Test Component* deve ter (que será o mesmo da estrutura original ao qual o *Test Component* está emulando).

- *realTimeOperations*: operações que permitirão a imposição e verificação do *Test Component* em seus estados, bem como a imposição do lançamento de uma determinada interrupção.

Restrições:

Código Fonte 4.3: Restrição OCL para o estereótipo *Test Component RT*

```
Context Test Component RT inv:
    self.interferenceTime.oclIsUndefined = true
```

Restrição que força que, obrigatoriamente, o atributo *interferenceTime* esteja definido na arquitetura.

Exemplo de Aplicação: Na Figura 4.6 temos um exemplo de possível *Test Component* para um objetivo de teste *real-time*. Nesta figura é possível visualizar a classe *ControllerLights_Emulator*. A esta, foi aplicado o estereótipo *Test Component RT*. Esta classe

generaliza uma classe já existente (presente nos modelos do PIM), com o intuito de emular seu comportamento. Para tal, foi adicionado o atributo *executionTime* e implementada a interface *RealTimeOperations* que permitirá a manipulação desta estrutura pelos casos de teste.

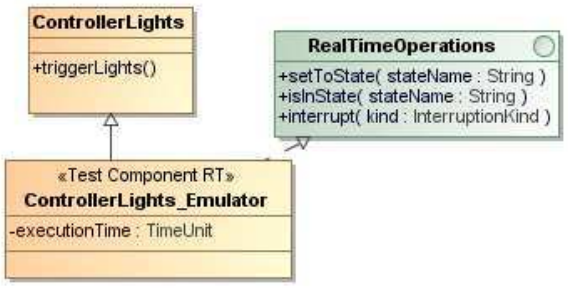

Figura 4.6: Exemplo de aplicação do estereótipo *Test Component RT*.

Elemento: Interface *Arbiter RT*

Descrição do elemento em UTP: Interface composta pelas operações *"setVerdict(v: Verdict)"* e *"getVerdict():Verdict"*. Tem como funcionalidade avaliar o resultado individual do teste a partir dos *Test Components* e atribuir o veredito final. Os resultados individuais são fornecidos ao árbitro via mecanismos de validação que verificam valores de variáveis em tempo de execução.

Motivação para extensão: Na maioria dos testes RT é inviável que a definição dos vereditos dos casos de teste seja realizada em tempo de execução, principalmente porque isto seria uma atividade custosa tanto em relação a recursos quanto a tempo (existiriam grandes interferências de tempo, o que atrapalharia a verificação de restrições nos casos de teste, por exemplo). Para tal, é interessante incluir o conceito de arbitragem passiva no perfil de testes, entenda-se por arbitragem passiva a atribuição de vereditos somente após a execução dos casos de teste, via análise de arquivos de *log*.

Solução: Criação da interface *Arbiter RT*, que herda a semântica proveniente da interface *Arbiter* (do UTP). Esta nova interface vai possuir as seguintes operações:

- *defineVerdict*: operação responsável por fornecer o veredito final de um caso de teste mediante análise/comparação de dois arquivos. O primeiro parâmetro (*expectedResults*), deverá possuir a lista dos valores esperados para as variáveis do sistema e

de tempo, estes valores são definidos e fornecidos pelo testador. O segundo parâmetro (*executionResults*), deverá possuir os valores que foram armazenados em *log* durante a execução do caso de teste. Estes valores, armazenados em *log*, deverão conter o corrente estado das principais variáveis do sistema e o tempo corrente no momento do *logging*.

- *compareValues*: operação que auxiliará a operação *defineVerdict* a cumprir sua função, comparando as *strings* contidas nos arquivos de *log* referentes aos valores alcançados e esperados.

Exemplo de Aplicação: A Figura 4.7 apresenta a interface *Arbiter RT*, propícia para averiguação e aferição de vereditos de casos de teste no contexto *real-time*.

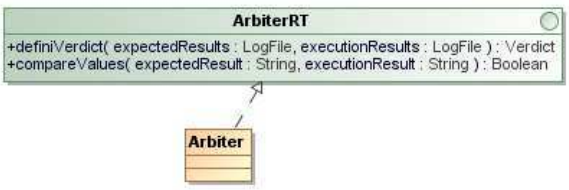

Figura 4.7: Exemplo de aplicação da interface *Arbiter RT*.

Elemento: Interface *Scheduler RT*

Descrição do elemento em UTP: Interface composta pelas operações *"startTestCase()"*, *"finishTestCase(t: TestComponent)"* e *"createTestComponent()"*. Tem por objetivo controlar a execução dos diferentes *Test Components*.

Motivação para extensão: Mais uma vez, o perfil permite pouco controle na execução da atividade de teste, no contexto de testes RT esta é uma falta grave que impossibilita seu uso em plenitude. Para tal, novos mecanismos de controle devem ser criados deixando a cargo do testador a gerência de toda a execução dos testes.

Solução: Criação da interface *Scheduler RT*, que herda a semântica da interface *Scheduler* (de UTP) acrescentando novas operações, que são elas:

- *disableBackgroundBehaviours*: operação responsável por garantir que nenhum outro processo ou serviço esteja em execução, mesmo que em *background*, durante a realização das configurações ou execução dos casos de teste.

- *startTestCase*: operação responsável por iniciar um determinado caso de teste. Esta operação será reponsável por fazer a chamada das demais operações que terão a incumbência de realizar a configuração e execução dos casos de teste.

- *testConfiguration*: operação responsável por instanciar as estruturas necessárias para execução de um caso de teste.

- *organizePreConditions*: operação responsável por, reusando as operações da interface *RealTimeOperations*, configurar as estruturas participantes dos casos de teste da forma necessária para que estas possam ser usadas. Esta operação deve ser executada antes de cada execução de caso de teste, pois, mudando o caso de teste poderão mudar as estruturas participantes, bem como as pré-condições individuais de cada uma.

- *allowTestCaseBehaviourExecution*: operação que transfere o fluxo de execução que está no *Test Context RT* para execução do *SUT RT* ou de algum *Test Component RT*, esta transferência marca a finalização da configuração do caso de teste e inicia a execução do comportamento do mesmo. Na maioria dos casos de teste RT não é possível que o processo *Test Context* continue executando em *background* (isto causaria grandes interferências de tempo e perda de recursos), por isso é importante que esta transferência de fluxo exista, só retornando ao *Test Context* ao final da execução do caso de teste.

Exemplo de Aplicação: A Figura 4.8 apresenta a interface *Scheduler RT*, propícia para gerenciamento dos processos envolvidos na execução dos casos de teste, bem como na realização das configurações necessárias para tal.

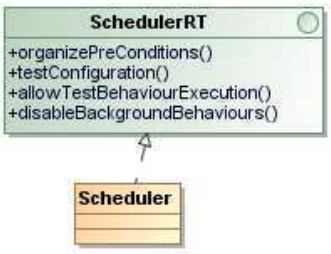

Figura 4.8: Exemplo de aplicação da interface *Scheduler RT*.

Elemento: Interface *Timer RT*

Descrição do elemento em UTP: Mecanismo que gera um evento de *timeout* quando um intervalo de tempo expira em uma dada instância. *Timers* são acessados pelos *Test Components*. Esta interface é composta pelas operações *start*, *read* e *stop*.

Motivação para extensão: Como a interface *Timer* irá manipular com dados de tempo que serão usados durante a execução dos casos de teste, suas operações também, muito provavelmente, causam interferências. Logo, estas interferências devem ser computadas e subtraídas dos tempos totais finais.

Solução: Inclusão da interface *Timer RT*, que herda as características da interface *Timer* (de UTP), acrescentando três novos atributos:

- *interferenceTimeOfStartOp*: atributo que indica o tempo de interferência que a operação *start* causa. Este tempo deve ser computado via análises preliminares.
- *interferenceTimeOfReadOp*: atributo que indica o tempo de interferência que a operação *read* causa. Este tempo deve ser computado via análises preliminares.
- *interferenceTimeOfStopOp*: atributo que indica o tempo de interferência que a operação *stop* causa. Este tempo deve ser computado via análises preliminares.

Exemplo de Aplicação: A Figura 4.9 apresenta a classe que implementa a interface *Timer RT*. Esta classe, estende o papel da interface *Timer*, presente no perfil original de UTP, acrescentando a preocupação da aferição dos tempos de interferência.

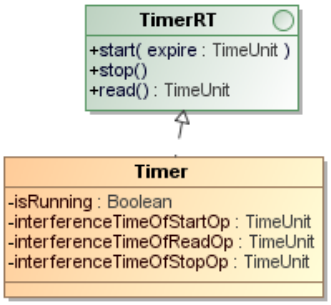

Figura 4.9: Exemplo de aplicação da interface *Timer RT*.

Elemento: Interface *RealTimeOperations*

Descrição do elemento em UTP: Elemento não existente em UTP

Motivação para criação: Existe a necessidade de configuração dos elementos envolvidos nos casos de teste (*SUT RT* e *Test Components RT*) a fim de deixá-los, mais facilmente usáveis nos testes, bem como permitir que estes elementos consigam simular e/ou forçar comportamentos essenciais para a atividade de testes RT.
Solução: Criação da interface *RealTimeOperations* possuindo as seguintes operações:

- *setToState*: operação que força a estrutura que a implementar (SUT e/ou *Test Components*), a ter seu fluxo direcionado até o estado de nome equivalente a *string* passada como parâmetro (*nameState*). Esta operação deve ser implementada pelo testador que, tendo bons conhecimentos do sistema, saberá o que é necessário para que a estrutura consiga alcançar o estado desejado.

- *isInState*: operação complementar a operação *setToState*, realizando a verificação se um estado foi ou não alcançado, ou seja, verifica se o fluxo da estrutura está no determinado estado.

- *interrupt*: operação que força a estrutura que a implementar possa ser interrompida por uma específica interrupção que lhe é indicada através do parâmetro *kind*. Esta operação deverá ser usada quando o caso de teste investiga o tratamento e/ou comportamento de eventos assíncronos.

Exemplo de Aplicação: A Figura 4.10 apresenta a interface *RealTimeOperations*, e a Figura 4.5 uma de suas aplicações. Esta interface proporciona que os elementos de teste RT (*SUT* e *Test Components*) possam ser configurados e utilizados nos casos de teste conforme as necessidades.

Figura 4.10: Interface *Timer RT*.

Elemento: Estereótipo *Data Selector RT*

Descrição do elemento em UTP: Operação que define como valores de dados ou classes de

equivalência são selecionados para serem usados nos testes.

Motivação para extensão: A operação de busca e seleção de dados pode ser uma tarefa bastante custosa e, no contexto de testes RT, este custo deve ser extinto ou ao menos controlado pelos testadores.

Solução: Criação do estereótipo *Data Selector RT*, este herda do estereótipo *Data Selector* (de UTP), adicionando um novo atributo a sua definição:

- *interference*: atributo (que assumirá o nome de *interferenceTime*, vide classe *RTAuxiliarElements* do pacote *Auxiliar Elements*), refere-se ao tempo de interferência que a operação de seleção de dados desperdiçará. Este tempo deve ser previamente investigado e poderá ser subtraído, caso necessário, durante a computação dos vereditos dos casos de teste.

Exemplo de Aplicação: A Figura 4.11 apresenta um exemplo de aplicação do estereótipo *Data Selector RT*. Para tal, no exemplo, uma classe especial foi construída para que valores inteiros válidos para os testes sejam obtidos, especificamente a operação *getIntegerData* executará esta função. Por isso, esta operação recebe o estereótipo *Data Selector*. Como a arquitetura a qual esta operação está inserida é uma arquitetura com propósitos RT, os atributos referentes a análise de tempo (*interferenceTime* e *executionTime*) também são inclusos.

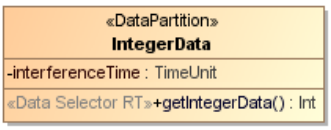

Figura 4.11: Exemplo de uso do estereótipo *Data Selector RT*.

Restrições:

Código Fonte 4.4: Restrição OCL para o estereótipo *Data Selector RT*

```
Context Data_Selector_RT inv:
    self.class.attribute->select((a.oclIsUndefined = true) AND (a | a.
        name = 'interferenceTime' OR
    a. name = 'executionTime'))->size() = 2;
```

Restrição que força que, a classe que possua uma operação com o estereótipo *Data Selector*

RT, possua também os atributos *interferenceTime* e *executionTime* definidos na arquitetura.

Elemento: Estereótipo *Test Log RT*

Descrição do elemento em UTP: Comportamento referente ao armazenamento em arquivos dos passos de execução dos casos de teste.

Motivação para extensão: A utilização de *logs* é algo bastante comum e importante para aferição de vereditos em testes RT. Porém, a realização da atividade de *log* também é intrusiva. Por isso, deve ser analisada cuidadosamente e, caso sua interferência seja maior que a tolerável, seu valor de desperdício deve ser descontado.

Solução: Criação do estereótipo *Test Log RT*, este herda do estereótipo *Test Log* (de UTP), acrescentando um novo atributo:

- *interference*: Valor do tempo de interferência acrescentado à execução do caso de teste pelo cumprimento do comportamento relativo a execução da atividade de *log*.

Exemplo de Aplicação: A Figura 4.12 apresenta o diagrama de seqüencia correspondente ao comportamento de uma possibilidade de realização de *log*. Como este comportamento se trata de um *log*, que será usado diversas vezes durante a atividade de testes num contexto RT, a este, é aplicado (mais especificamente a seu elemento *Interaction* - elemento UML que compõe o diagrama de seqüencia) o estereótipo *Test Log RT*. Com isto, o testador deve lembrar de inferir o tempo que a aplicação deste comportamento interferirá para a execução dos testes.

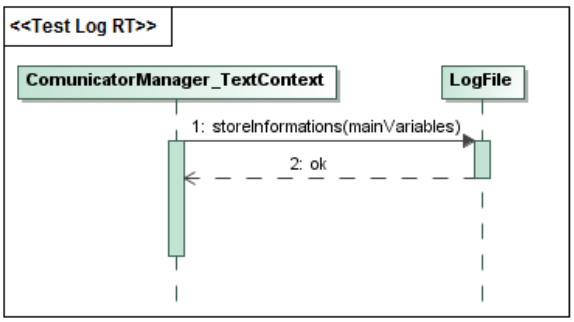

Figura 4.12: Exemplo de aplicação estereótipo *Test Log RT*.

4.3.2 Pacote *Auxiliar Elements*

Os elementos deste pacote (Figura 4.3) cumprem o papel de complementar a composição do perfil *UTP RT*, auxiliando na sua definição (*e.g.* classes *RTAuxiliaryInformations* e *TestContextAdapter*) ou ajudando na padronização dos elementos usados na definição e execução dos casos de teste (*e.g.* classes *LogFile* e *ObjectSituation*). A seguir, os elementos deste pacote serão apresentados individualmente. A apresentação seguirá o seguinte padrão: i) o elemento será identificado (*id* Elemento); ii) será apresentada a motivação que levou a criação do elemento (*id* Motivação); iii) apresentação de como se deu a solução para a problemática levantada na motivação (*id* Solução); e iv) regras OCL, quando necessário, para complementar a definição do elemento (*id* Restrições).

Elemento: Classe *RTTimeInformations*

Motivação: É necessário identificar e deixar claro nos elementos de teste RT aqueles cujo, devido a inclusão de instrumentações para análise, acabam por afetar nos tempos de execução gastos. Estes devem ser conhecidos pelos demais elementos da arquitetura (*e.g.* o árbitro) para que a aferição dos vereditos ocorra corretamente.

Solução: Inclusão de uma nova classe que possui os seguintes atributos:

- *interferenceTime*: variável que armazena o tempo extra desperdiçado para realização de alguma atividade incluída exclusivamente para propósitos de teste.
- *executionTime*: responsável por armazenar o tempo de processamento que a estrutura gasta originalmente, se a inclusão dos elementos de teste.

Restrições:

Código Fonte 4.5: Restrição OCL para a classe *RTAuxiliaryInformations*

```
Context RTAuxiliaryInformations inv:
    self.executionTime.oclIsUndefined = true
    XOR
    self.interferenceTime.oclIsUndefined = true
```

Elemento: Classe *TestContextAdapter*

Motivação: É imprescindível que a estrutura *Test Context* possua controle sobre todos os

processos e estruturas que participarão da atividade de testes.

Solução: Inclusão de uma nova classe que possui o atributo *priority*. Este, estabelece uma alta prioridade para a estrutura *Test Context RT*, a fim de que esta tenha o comando do fluxo de execução e assim possa controlar a realização da atividade de testes. Por *default* o atributo já inicia com prioridade 1000, este valor inteiro foi escolhido por ser um valor alto e normalmente acima das prioridades usadas na maioria dos STRs em diversas plataformas.

Elemento: Classe *LogFile*

Motivação: Como a atividade de *log* é bastante difundida e usada no meio de testes para STRs, uma padronização de como este *log* é realizado seria bastante interessante.

Solução: Inclusão de uma nova classe que padroniza o modo de como os *logs* devem ser armazenados. Para tal, a classe possui três atributos, são eles:

- *name*: nome identificador do arquivo de *log*.
- *description*: descrição do objetivo pelo qual este arquivo de *log* foi criado.
- *logEntries*: conjunto de entradas de *log* armazenadas no arquivo.

Elemento: Classe *ObjectSituation*

Motivação: A título organizacional, é necessário que exista uma padronização de como os elementos envolvidos nos testes devem estar configurados inicialmente.

Solução: Inclusão de uma nova classe que possui três atributos a serem instanciados, são eles:

- *object*: nome do objeto/estrutura ao qual o *objectSituation* está tratando.
- *nameState*: nome do estado que representa a configuração inicial ao qual o objeto/estrutura deve estar configurado para ser válido no caso de teste (pré-condição do objeto).
- *timeToReach*: tempo estimado para que o objeto/estrutura original atinja o estado/configuração desejada.

Elemento: Enumeração *InterruptionKind*

Motivação: Muitas vezes, num contexto de testes RT, é necessário que o testador force que uma determinada interrupção seja lançada. Saber quais os tipos de interrupções possíveis é imprescindível para assim entender qual e como lançar determinada interrupção nos momentos propícios.

Solução: Criação de uma enumeração para agrupar os nomes das possíveis interrupções do sistema a serem lançadas durante a atividade de teste.

Diretriz para uso: Os nomes das possíveis interrupções devem ser incluídos como literais na enumeração.

Elemento: Tipo de Dado *TimeUnit*

Motivação: É importante padronizar a unidade que os elementos relacionados a tempo usam.

Solução: Criação de um tipo de dados específico. Todo elemento, estrutura ou comportamento que manipule com tempo, deve então fazer uso deste tipo de dado como unidade de medida. O tipo *TimeUnit* herda de *Integer* (pacote Pimitive Types da UML) seus valores e operações, tornando a especificação do tipo mais completa e clara.

4.4 Suporte Ferramental

Com o intuito de proporcionar automação na geração das arquiteturas de teste (conforme apregoa MDT) para STRs *soft* e reativos, foi desenvolvida uma ferramenta que foi denominada pelo acrônimo RTTAG (*Real-Time Test Architecture Generator*). A seguir, são apresentadas as principais informações a respeito desta ferramenta

Objetivo: Auxiliar a equipe de testes, gerando automaticamente o máximo possível dos elementos da arquitetura de testes para STRs *soft* e reativos, em diferentes níveis de abstração. *Características Gerais*: A ferramenta foi implementada através da construção de um conjunto de regras de transformações de modelos. Estas regras foram escritas usando a linguagem ATL - *Atlas Transformation Language* [42]. Apesar de ATL não ser o padrão

proposto pela OMG para construção de transformações (a OMG sugere o uso da linguagem QVT [1]), esta escolha foi realizada porque, atualmente ATL pode ser considerada o padrão informal para desenvolvimento de regras de transformação, possuindo uma comunidade bastante ampla de desenvolvedores e inúmeros casos de sucesso com seu uso [68].

Arquitetura: A ferramenta RTTAG, cuja arquitetura está representada através do diagrama de componentes da Figura 4.13, é composta essencialmente por dois módulos distintos, são eles:

- ***PIM2PITM***: este módulo recebe como entrada os modelos de *design* do STR, construídos usando UML e condizentes com as diretrizes de modelagem (estabelecidas na Seção 3.5). Esta característica está representada pela interface *iUML_Diagrams_using_the_guidelines*, requisitada pelo componente PIM2PITM. Estes modelos passam por uma verificação de metamodelo (interface *iUMLMetamodelCheckling*), são copiados para o modelo de saída (interface *iCopyElements*) e posteriormente passam pelo processo de geração dos modelos da arquitetura de testes no nível independente de plataforma - PITM (interface *iPITMTestArchitectureGeneration*). A arquitetura fornecida como saída será composta por instanciações dos elementos de *UTP RT*, em conjunto com as extensões desenvolvidas (Seção 4.3). Ainda no módulo PIM2PITM, um algoritmo de geração de casos de teste é executado (interface *iTestCaseGeneration*), onde este, via caminhamento sobre a máquina de estados da estrutura SUT fornecerá um conjunto de casos de teste que é anexado a arquitetura. O pseudo código do algoritmo implementado está descrito no Código Fonte 4.6. Maiores detalhes sobre a execução deste algoritmo pode ser visualizado em [5].

Código Fonte 4.6: Pseudo código referente ao algoritmo de geração de casos de teste implementado na ferramenta RTTAG (Módulo PIM2PITM)

```
TestCaseGeneration (SM: StateMachine)
        BEGIN
            testCasesSet = {};
             transitions = SM.transitions ();
             testCase = new SequenceDiagram (new lifeLineTester(), new
                 lifeLineTestable());
             currentState = SM.initialState;
             currentTransition = searchForTransitionBeginningWith (
                 currentState);
```

```
        fragmentInitialized = FALSE;

        WHILE Transitions != {} DO
            source = currentTransition.source;
            target = currentTransition.target;
            IF (currentTransition.guard != NULL AND fragmentIniiialized
                = FALSE) THEN
                testCasesSet.add (testCase);
                testCase = new SequenceDiagram ();
                testCase.add (createSetToStateMessage(source));
                testCase.add (createIsInStateMessage(source));
                testCase.add (new OPTFragment(currentTransition.guard.
                    condition));
                fragmentInitialized = TRUE;
                GOTO WHILE;
            END IF

            IF (currentTransition.event != NULL) THEN
                testCase.add (createEventMessenge (currentTransition.
                    event));
            END IF

            testCase.add (createIsInStateMessage(target));
            transitions.remove(currentTransition);
            currentTransition = searchForTransitionBeginningWith (
                target);

            IF (currentTransition = NULL) THEN
                testCasesSet.add (testCase);
                testCase = new SequenceDiagram (new lifeLineTester(),
                    new lifeLineTestable());
                fragmentInitialized = FALSE;
                currentTransition = getRandomTransition (transitions);
            END IF
        END WHILE
    END.
```

- ***PITM2PSTM***: módulo que recebe como entrada os modelos PITM (provenientes da exe-

cução do módulo PIM2PITM) e fornece como resultado os modelos da mesma arquitetura de testes, agora no nível PSTM (interface *iPSTMTestArchitectureGeneration*). Mais especificamente, saída deste módulo será a representação da arquitetura de testes para a plataforma FreeRTOS na linguagem C (checado através da interface *iCMetamodelChecking*).

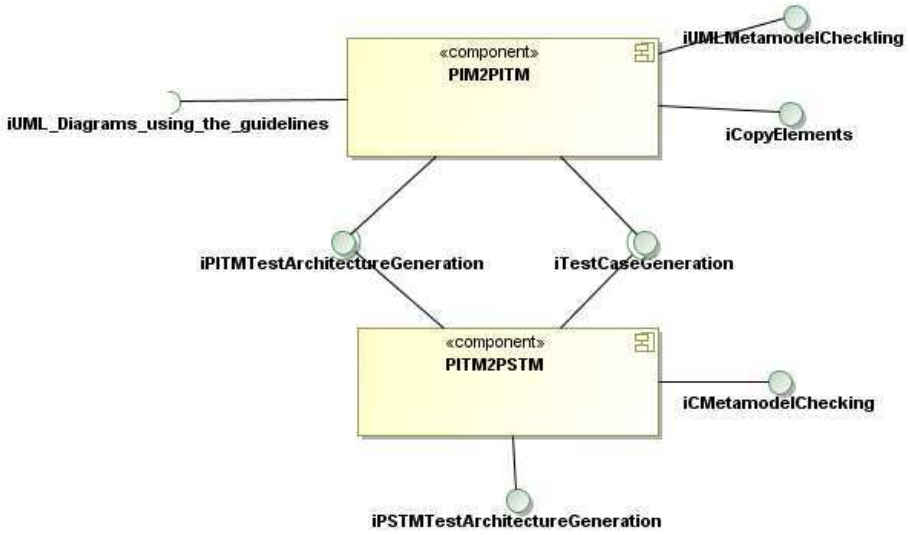

Figura 4.13: O *Arquitetura da ferramenta RTTAG*.

As regras de transformação ATL referentes aos módulos PIM2PITM e PITM2PSTM estão disponíveis para visualização e análise no seguinte endereço eletrônico `https://mestradotool.googlecode.com/svn/trunk/Mestrado/transformations/`.

4.5 Avaliação das Extensões

Esta seção apresenta o processo avaliativo que foi realizado com o intuito de averiguar se as extensões propostas são eficientes no tocante a auxiliar o projeto de arquiteturas de teste mais completas e efetivas para o contexto de STRs, no nível independente de plataforma. Para tal, foram aplicados estudos de caso usando o primeiro módulo da ferramenta RTTAG. Ao utilizamos os resultados da ferramenta RTTAG para realização deste estudo avaliativo pretendemos, além de avaliarmos o conjunto de extensões propostas para UTP, também averiguar a coerência dos resultados fornecidos pelo primeiro módulo da ferramenta.

Como já dito anteriormente, a atividade de testes de STRs carece de cuidados especiais, e é imprescindível que uma arquitetura de testes neste contexto considere seus aspectos diferenciados, inclusive no nível PITM. A fim de avaliarmos se a configuração de arquiteturas de testes proposta (usando UTP + extensões) efetivamente tornou a mesma, no nível PITM, mais expressiva, um conjunto de requisitos foi definido. Os elementos deste conjunto de requisitos podem ser considerados como essenciais e, por isso, é imprescindível que estejam presentes em arquiteturas de teste de tempo real. Esta lista de requisitos foi definida baseada na experiência adquirida na modelagem e testes de STRs, bem como na análise de trabalhos da literatura que: usam UTP para construção de arquiteturas de teste (*e.g.* [9; 35; 30]); tratam da geração de arquiteturas de teste (*e.g.* [50; 7]); ou trabalhos que se referem a mecanismos de teste para STRs (*e.g.* [54; 53; 49]). Os requisitos identificados foram:

1. Deve existir uma estrutura/elemento controlador da execução testes;
2. É importante que a identificação SUT seja clara e evidente;
3. Deve-se identificar todos os elementos participantes dos casos de testes (além do SUT e da estrutura controladora);
4. Deve existir uma estrutura capaz de realizar a arbitragem dos casos de teste;
5. Deve existir um elemento que seja capaz de organizar a forma como cada elemento da arquitetura é usado nos testes (SUT ou outros artefatos de teste). Este, interferindo minimamente na execução dos testes e no seu tempo de execução;
6. Devem existir mecanismos para computar tempo;
7. Devem existir mecanismos para aferição de tempos de interferência;
8. Devem existir mecanismos para verificação de restrições de tempo;
9. Devem existir mecanismos para permissividade de arbitragem *off-line*;
10. Devem existir mecanismos para configuração das pré-condições dos casos de teste;
11. Deve existir um mecanismo para estabelecimento e apresentação dos resultados da execução dos casos de teste;

12. Devem existir mecanismos para forçar o lançamento de interrupções.

A fim de analisarmos o conjunto de extensões para UTP, propostas nas seções anteriores, três estudos de casos foram executados. Nestes, a partir dos modelos PIM, foram geradas automaticamente arquiteturas de teste para estes sistemas através do uso do primeiro módulo da ferramenta RTTAG. De posse desses modelos, foram realizadas análises para verificação se os requisitos importantes para construção de uma arquitetura de teste para STRs de qualidade (elencados anteriormente) estavam ou não presentes nas arquiteturas geradas. A partir destas análises, foi possível aferir conclusões a respeito do conjunto de extensões proposto.

A título de ilustração, a seguir será apresentada a localização, ou identificação de ausência, dos elementos citados na lista apresentada anteriormente, para arquiteturas de teste gerada para o estudo de caso "Sistema de Alarmes" (especificação do sistema no Apêndice A). É importante ressaltar que o mesmo processo que será apresentado na seqüencia foi também realizado para os demais estudos de caso (uma aplicação de celular e para uma aplicação de TV Digital - especificações no Apêndice A) e em ambos os casos os resultados alcançados foram semelhantes.

Os modelos PIM do estudo de caso *Sistema de Alarme* estão presentes para visualização no Apêndice D deste documento. Estes modelos foram os que serviram de entrada para o primeiro módulo da ferramenta RTTAG. Como resultado, foi obtido um conjunto de modelos representando a arquitetura de testes do sistema, dentre estes modelos, temos: um diagrama de classes (Figura 4.14), o diagrama de estrutura compostas do elemento *Test Context* (Figura 4.15), e um conjunto de diagramas de seqüência correspondente aos casos de teste extraídos da máquina de estado do componente SUT (*e.g.* Figura 4.17).

Na Figura 4.14, é possível visualizar que ao diagrama de classes original do *Sistema de Alarmes* foi anexado o pacote *TestPackage*. Este pacote inclui a configuração estrutural da arquitetura de testes, gerada para o sistema em questão.

Na Figura 4.16, o pacote *TestPackage* está visualmente ampliado a fim de proporcionar a melhor localização dos elementos. Neste, é possível localizar a estrutura *ComunicatorManager_TestContext*, esta estrutura foi criada exclusivamente para gerenciamento, configuração e análise da execução dos testes. Com isto, o requisito número um da lista foi satisfeito. É possível visualizar também, que foi adicionada à esta estrutura características RT, tais como a identificação do elemento como objeto ativo e a preocupação em dotá-lo com uma priori-

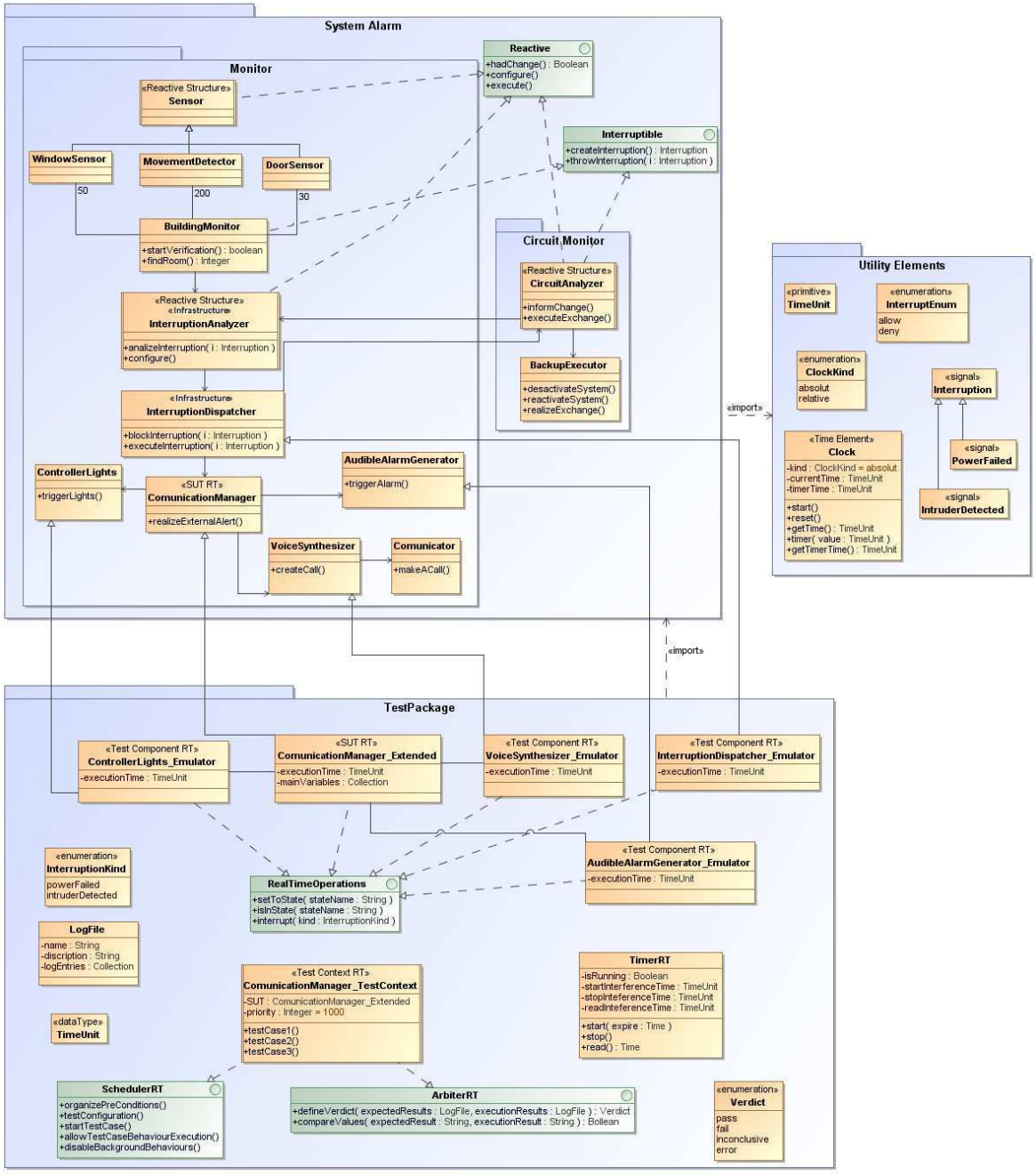

Figura 4.14: O *Diagrama de Classes estendido do* Sistema de Alarmes.

dade elevada para deixar claro que esta será a primeira estrutura a executar. Portanto, além de cumprir o requisito de existência do elemento, a este foi adicionado um conjunto de novos aspectos que tornaram sua definição mais completa.

Outra nova estrutura foi criada, a *ComunicationManager_Extended*, que estende a classe SUT da vez (a ferramenta RTTAG exige que um SUT seja escolhido a cada execução), e,

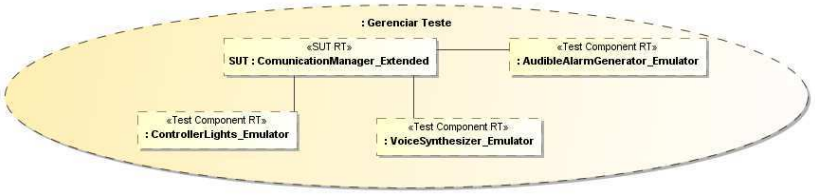

Figura 4.15: O *Diagrama de Estruturas Compostas da classe* ComunicatorManager_TestContext.

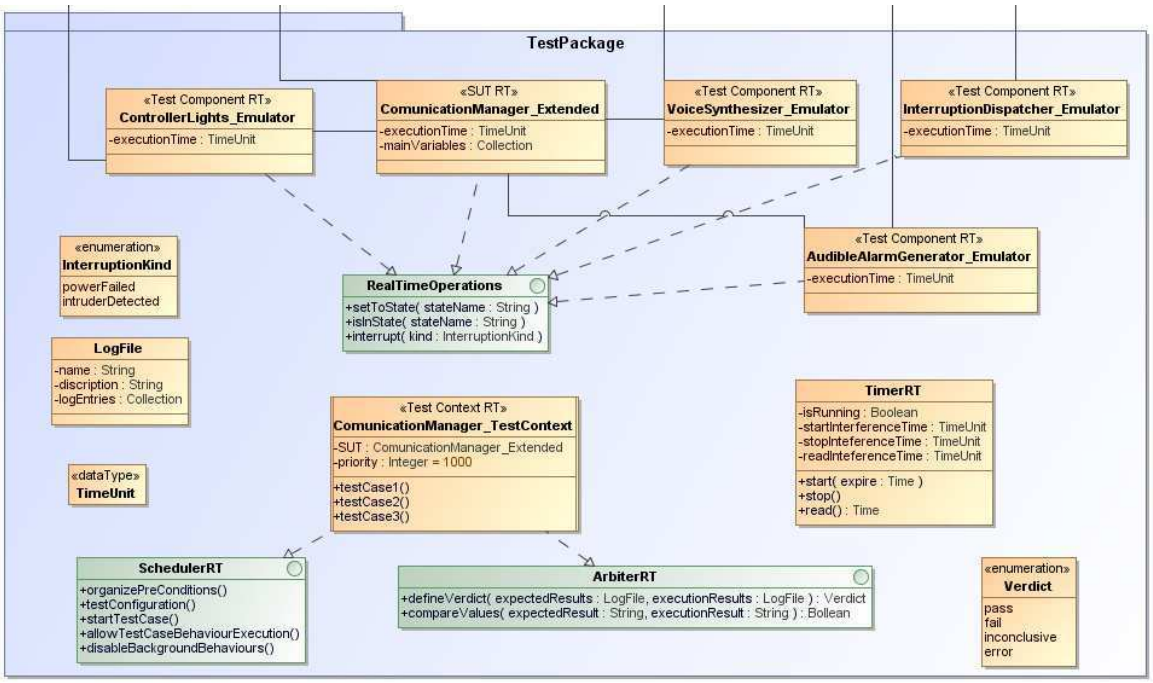

Figura 4.16: Pacote arquitetural de testes que foi anexado ao diagrama de classes original do PIM.

acrescenta a esta os elementos necessários para adequá-la ao contexto RT. Portanto, na arquitetura de testes, a estrutura SUT está claramente identificada através do estereótipo *SUT RT*, cumprindo o segundo requisito da lista. Além disto, um outro benefício foi adquirido, com a inclusão de um novo elemento para ser o SUT nos testes (estendendo o SUT original), os elementos de teste ficam independentes dos elementos originais. Logo, extensões necessárias para realização dos testes, principalmente testes RT, podem ser realizadas exclusivamente nos elementos de teste. É importante notar que a classe SUT implementa a interface *RealTimeOperations*, permitindo assim que testes mais complexos (*e.g.* testes envolvendo interrupções) sejam organizadamente planejados.

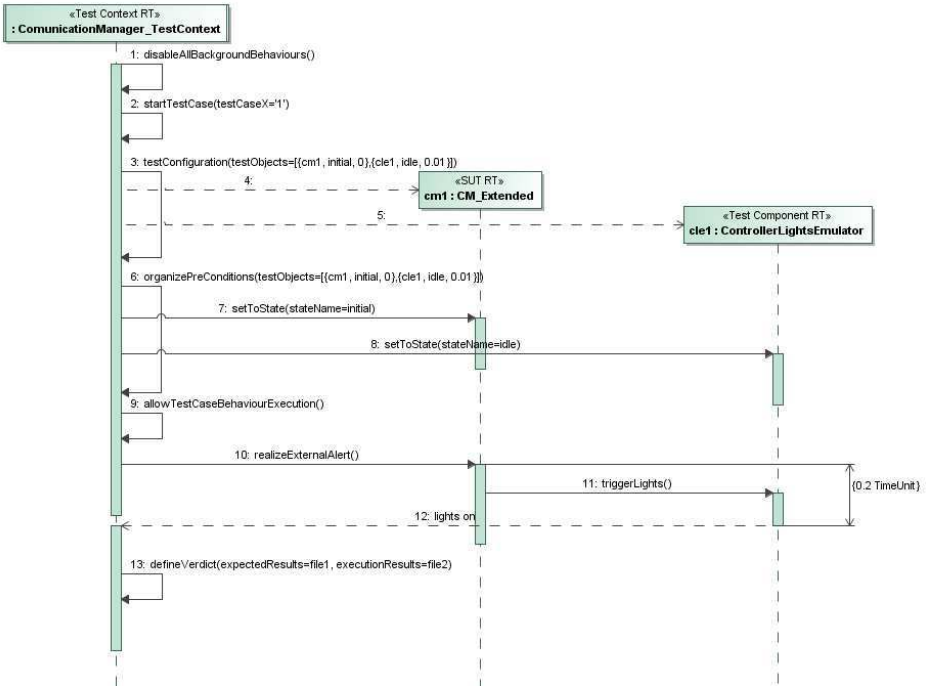

Figura 4.17: Exemplo de caso de teste extraído da máquina de estados do componente *External Comunicator*.

Como nosso objetivo é trabalhar com teste de integração, os demais elementos que participam dos testes, além do SUT, são tratados como *Test Components RT*. Ou seja, para toda estrutura que se relacione de alguma forma com o SUT foi criada uma classe no pacote *TestPackage* estendendo a classe original do PIM. Estas novas classes, neste exemplo, serão emuladores das classes originais. Além da identificação das estruturas participantes dos testes (cumprindo assim o requisito três da lista), por tratarmos de testes RT, cada elemento *Test Component RT* criado implementa a interface *RealTimeOperations*. Com isto, foi adicionada a estas estruturas a possibilidade de serem usadas nos casos de teste já pré-configuradas, ou seja, não somente no seu momento inicial de configuração, bem como possibilitar seu uso no contexto de testes de interrupções. Parte do uso das operações da interface *RealTimeOperations* pode ser visualizada na Figura 4.17 (*e.g.* mensagens 7 e 8).

Quanto ao requisito número quatro da lista, a classe *ComunicatorManager_TestContext* (estrutura *Test Context*) implementa a interface *ArbiterRT*, que tem o objetivo de proporcionar meios para a realização da arbitragem passiva (ou *off-line*) dos casos de teste RT. Logo, tanto os requisitos quatro como nove da lista, foram cobertos nesta arquitetura. Também, na Figura 4.17, referente a um dos casos de testes gerados para esta arquitetura, é

possível verificar o modo como são usadas uma das operações da interface *ArbiterRT* (mensagem 13). Com isso, além de fornecer a estrutura responsável pela arbitragem, a execução da ferramenta RTTAG também gera uma maneira *default* para arbitragem dos casos de teste.

Semelhantemente, a classe *ComunicatorManager_TestContext* também implementa a interface *Scheduler RT*. Com isso, é adicionado a esta estrutura um conjunto de novas características, entre elas o controle da criação e configuração dos objetos de teste, organização das pré-condições dos casos de teste e controle total dos processos envolvidos na execução. Como as operações da interface *Scheduler RT* são utilizadas antes da execução do comportamento real do caso de testes, a configuração do ambiente afetará minimamente no tempo de execução dos mesmos (fator de grande importância para o contexto de testes RT). Na Figura 4.17 é apresentada a forma como as operações devem ser usadas para cada caso de teste (mensagens 1, 2, 3, 6 e 9). Desta forma, os requisitos cinco e dez, foram cobertos.

O requisito seis se refere à existência de estruturas que permitam a computação de tempo durante a execução dos casos de teste. Por esta ser uma questão bastante específica, ou seja, no nível independente de plataforma pode não ser útil a inclusão de um novo elemento para demonstrar como se realiza a computação do tempo, não são encontrados nas arquiteturas em questão elemento que cubra especificamente a restrição seis, pois acreditamos que esta seja uma questão mais adequada de ser tratada nos níveis específicos de plataforma.

Mecanismos para aferição de tempos de interferência foram incluídos nos diversos elementos de teste adicionados. Na classe que estende o SUT (com o estereótipo *SUT RT*), nas classes *Test Components* (com o estereótipo *Test Component RT*), nos atributos da classe *TimerRT*, etc. Em outros elementos esta preocupação também está inclusa (*e.g. Data Selector*, *Test Log*). Porém, questões como a forma como são selecionados os dados do teste, bem como onde inserir as chamadas à função de *log* são bastantes específicas e relacionadas ao interesse do testador, a ferramenta RTTAG não gera estes elementos nos seus artefatos, mas as extensões cobrem também tais características. Portanto, o requisito sete também está coberto.

A fim de verificar restrições de tempo, a classe *TimerRT* foi incluída, permitindo que o tempo seja verificado no momento inicial e final do comportamento que exibe a restrição a ser verificada. Portanto, o requisito oito também está coberto da arquitetura gerada. Porém, como pode ser notado no caso de teste da Figura 4.17, este elemento não foi usado na geração

dos casos de teste, pois neste foram considerados testes onde a arbitragem e verificação é realizada via *log* após a finalização da execução.

Quanto ao requisito onze, a interface elementos *ArbiterRT* e a enumeração *Verdict* (de UTP) cumprem parcialmente o papel de estabelecimento dos resultados dos casos de teste. Porém, a maneira como estes resultados serão apresentados ao usuário/testador não está presente na arquitetura, pois, as formas como isto pode ser realizado pode variar (*e.g.* visualmente, textualmente, etc) ficando a cargo do testador escolher a maneira mais adequada de projetar e implementar tal requisito.

Forçar o lançamento de uma interrupção é de suma importância para o teste de STRs. Para tal, a operação *interrupt* da interface *RealTimeOperations* e a enumeração *InterruptionKind* foram acrescentadas para cumprir esse papel na arquitetura. Com tais elementos, é possível incluir nos casos de teste a expressividade de que, a qualquer momento (o momento será escolhido pelo testador), o elemento que esteja em execução, seja o SUT ou um *TestComponent*, seja interrompido e o tratamento desta específica interrupção seja testado. Portanto, o requisito doze também está coberto na arquitetura definida.

Conclusões Gerais

A partir das análises apresentadas anteriormente, foi possível estabelecer um conjunto de conclusões a respeito das extensões propostas para UTP: i) os elementos adicionados, bem como a execução do módulo PIM2PITM da ferramenta RTTAG, proporcionam a construção de uma arquitetura de testes de qualidade e completa. A localização, quase total dos requisitos listados (11 dos 12), na arquitetura gerada, nos proporciona o sentimento de que houve um acréscimo na expressividade e qualidade dos elementos projetados; ii) a definição de todos os elementos necessários para a realização da atividade de testes, bem como a definição de casos de testes para estes sistemas, como o exemplificado na Figura 4.17 nos fornece indícios que a passagem da arquitetura definida no nível PITM para o nível PSTM, ou mesmo diretamente para código, poderá ser realizada mais facilmente; e iii) os elementos acrescentados ao perfil de testes (as extensões), conjecturanto a partir da análise visual e *feeling*, trouxeram melhor representatividade para a arquitetura no contexto dos STRs.

É importante destacar que outros dois estudos de caso foram também aplicados (uma aplicação de celular e uma de TV Digital). Estes, não estão presentes neste capítulo, por

questões de simplificação didática. Porém, a execução dos mesmos nos forneceram resultados semelhantes aos alcançados com o *Sistema de Alarmes*. As arquiteturas de testes geradas para os dois outros estudos de caso, estruturalmente se assemelham com a presente no pacote *Test Package* (Figura 4.16), variando na composição dos elementos, bem como nos casos de teste gerados (por se tratarem de sistemas diferentes). Estes estudos de caso foram planejados a fim de atestar a consistência dos resultados alcançados, bem como a efetividade dos artefatos gerados com a execução do primeiro módulo da ferramenta RTTAG, fato que se confirmou.

Devido a limitações de tempo e de recursos, um estudo mais abrangente não pôde ser efetuado no tocante a analisar o quão úteis são as extensões propostas no contexto de criação de arquiteturas de testes para STRs. Porém, as conjecturas apresentadas anteriormente, bem como a análise visual nos permite observar que os modelos de testes ficaram mais expressivos e que, diversos elementos importantes para uma arquitetura RT, mas que o UTP original não incluía (*e.g.* arbitragem passiva, preocupação com interferências de tempo, etc) foram incorporadas com as extensões. Tais indicações nos dão o sentimento que, apesar de não realizado um estudo comprobatório aprofundado, as extensões são realmente bastante úteis e trouxeram melhorias para o perfil de testes da UML.

4.6 Considerações Finais do Capítulo

Este capítulo apresentou um conjunto de extensões ao perfil de testes da UML. As extensões desenvolvidas têm por finalidade dotar o perfil de testes com mecanismos próprios para o contexto de projeto de arquiteturas de teste para STRs. Para tal, diversos elementos arquiteturais e comportamentais existentes em UTP foram expandidos semanticamente, bem como um conjunto de novos elementos foram propostos. Para auxiliar a atividade de projeto de arquiteturas de testes RT, o primeiro módulo da ferramenta RTTAG foi apresentado. Este módulo automatiza o processo de geração destas arquiteturas, incluindo também um conjunto de casos de teste gerados a partir de caminhamentos sob máquinas de estado. Finalizando, foi apresentada como se deu a realização dos estudos de caso que demonstraram a utilidade e aplicabilidade das extensões propostas anteriormente.

Capítulo 5

Mapeando Arquiteturas de Teste para o nível PSTM

O perfil de testes da UML (UTP) é o mais propagado mecanismo para modelagem de arquiteturas de testes no nível independente de plataforma (PITM). Anteriormente neste trabalho, UTP foi estendido para também dar suporte à construção de PITMs para STRs (Capítulo 4). Porém, a utilização de UTP, mesmo o estendido (UTP RT - Capítulo 4), limita-se unicamente ao contexto independente de plataforma, deixando a cargo do projetista de teste o trabalho de mapear seus conceitos para o ambiente de execução real. A fim de propiciar o uso do perfil em larga escala, é imprescindível: i) demonstrar a implementabilidade de seus conceitos em diferentes plataformas; e ii) dispor de padrões e/ou diretrizes que guiem a transformação de seus elementos em conceitos dependentes de plataforma. Em particular, sistemas embarcados possuem inúmeras restrições, tornando esta transformação difícil e suscetível a erros.

Neste capítulo será apresentado um conjunto de mapeamentos de conceitos entre os elementos do perfil de testes estendido (UTP RT) e os recursos disponíveis em um *framework* operacional para sistemas embarcados, o FreeRTOS [13], com o intuito de contribuir com a investigação dos dois desafios anteriormente mencionados. O FreeRTOS foi escolhido como plataforma específica por ser um sistema operacional bastante usado no contexto de STRs embarcados, possuindo uma comunidade bastante ativa de desenvolvedores, e por ser uma plataforma embarcada com severas restrições de recursos, nos fornecendo então um exemplo extremo das dificuldades encontradas para realização deste tipo de mapeamento.

O mapeamento apresentado neste trabalho tende a ser o mais completo possível devido ao

fato de englobar todos os elementos presentes no perfil de testes da UML (assim como outros trabalhos realizam, para contextos diferentes *e.g.* [73]) e ainda suas extensões propostas para o contexto RT (descritas no Capítulo 4 deste documento).

5.1 Mapeamentos

Assim como na especificação do perfil de testes, as regras de mapeamento desenvolvidas também foram subdivididas segundo os mesmos grupos de conceitos (*Test Architecture*, *Test Behavior*, *Test Data* e *Time Concepts*). Para cada elemento dos grupos foi realizada uma análise de como poderia ser efetuada sua implementação, se existiriam adaptações a serem feitas, etc. Na seqüencia, serão apresentados estes mapeamentos. A apresentação das regras seguirá o seguinte padrão: primeiramente o nome do elemento ou conceito será identificado (*id* Elemento), em sequência será apresentada a definição do mesmo segundo UTP ou UTP RT (*id* Definição) e, por fim, como pode ser sua implementação segundo o FreeRTOS (*id* Mapeamento).

Parte dos mapeamentos que serão apresentados a seguir, foi publicado em [3].

Antes da apresentação efetiva dos mapeamentos, é importante destacar que, para tratar questões como a instrumentação do código para captura de informações, aferição de tempo de interferência, bem como a observação das principais variáveis do SUT e *logging*, foi escolhido utilizar o auxílio da API proposta em [53]. Esta, tem por objetivo fornecer o suporte técnico para a execução de testes para STRs embarcados.

5.1.1 Test Architecture

Elemento: *SUT*

Definição: O sistema, ou parte dele, que será testado.

Mapeamento: É fato que o sistema original deve ser desenvolvido em C e seguindo as restrições impostas pela plataforma *FreeRTOS*. Neste contexto, os elementos que podem ser testados ficam restritos a uma rotina, um processo, um comportamento (conjunto de processos que interagem) ou todo o sistema. A observação das variáveis se dará através do uso das funções *xSetObserverFields* e *xEndObserverWork* (da API [53]), as chamadas a estas funções serão incluídas estrategicamente pelo testador no código do SUT. As operações

para manipulação de estados do SUT (interface *RealTimeOperations*) são mapeadas por uma rotina acrescentada ao SUT. Esta rotina conterá um comando *switch* onde, cada *case* possuirá os comandos a serem executados para que determinado estado seja atingido, ou que uma interrupção seja lançada.

A Figura 5.1 apresenta visualmente parte deste mapeamento. Mais especificamente, a Figura 5.1-b apresenta parte do código do comando *switch* (presente no corpo da função *setToState*, conforme indica o mapeamento). No caso, este código foi inserido manualmente pelo testador para forçar que a estrutura SUT consiga alcançar o estado *1* da sua máquina de estados (correspondente ao estado que representa que um invasor foi detectado). A Figura 5.1-c apresenta como se dá o uso da das funções *xSetObserverFields* e *xEndObserverWork* (provenientes da API escolhida para auxiliar o processo de teste). No caso, as funções foram usadas para instrumentar o código do SUT para que a variável *roomNumber* pudesse ser observada durante a execução dos casos de teste.

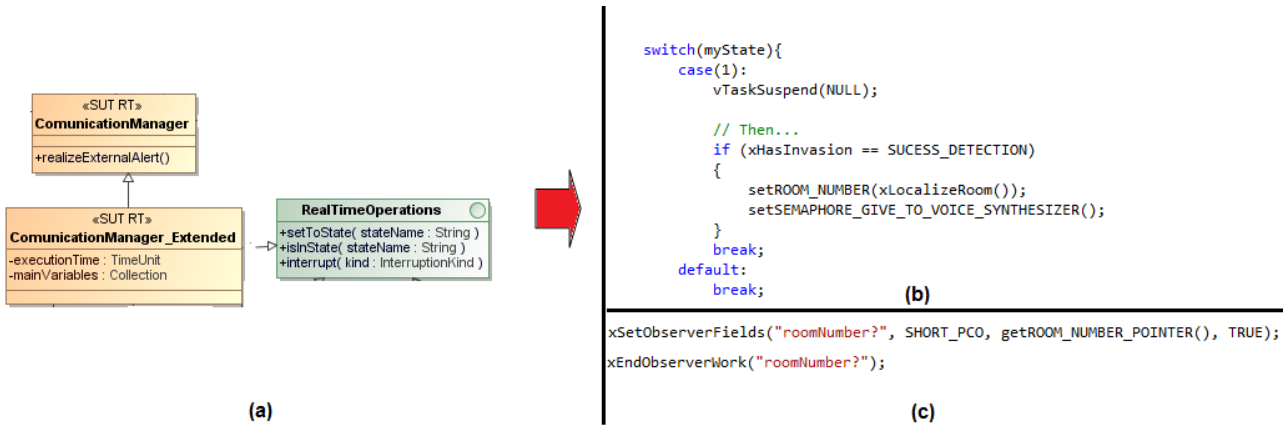

Figura 5.1: (a) Exemplo de um SUT no nível PITM. (b) e (c) Parte do código representativo do mapeamento.

Elemento: ***Test Context***

Definição: Classe que possui os casos de teste (operações) e cujo diagrama UML de estruturas compostas define a configuração necessária para execução dos testes.

Mapeamento: Devido à necessidade de adaptação, por lidarem com paradigmas diferentes (UTP usa conceitos de orientação a objetos e o *FreeRTOS* segue o paradigma imperativo), o *Test Context* é mapeado como uma rotina auxiliar (*"vStartTest"*) que chamará as rotinas que executarão a configuração de teste (instanciam e preparam todos os elementos necessários

para execução dos testes, *e.g SUT* e *Test Components*), e por um processo principal para execução dos casos de teste. Este processo (*"vTestContext"*), guiará o fluxo de execução da atividade de testes realizando as chamadas aos casos de teste (*Test Cases*). No entanto, para que possua o controle da execução, este processo deverá inicialmente possuir a maior prioridade entre todos, garantindo ser ele o primeiro a executar.

A Figura 5.2 apresenta visualmente como se dá parte do mapeamento apresentado. Na Figura 5.2-b é possível visualizar a rotina auxiliar *vStartTest* e o código a ser executado pelo processo *vTestContext*. Ainda nesta figura é possível observar a chamada das funções referentes aos três casos de teste (corpo do processo *vTestContext*), elencados no modelo da Figura 5.2-a. A criação e atribuição da prioridade adequada ao processo *TestContext* é realizada pelas rotinas *vTaskScheduler* e *vTaskStartScheduler* (Figura 5.2-b, rotina *vStartTest*), o corpo destas rotinas não estão apresentados na figura por questões de simplificação didática.

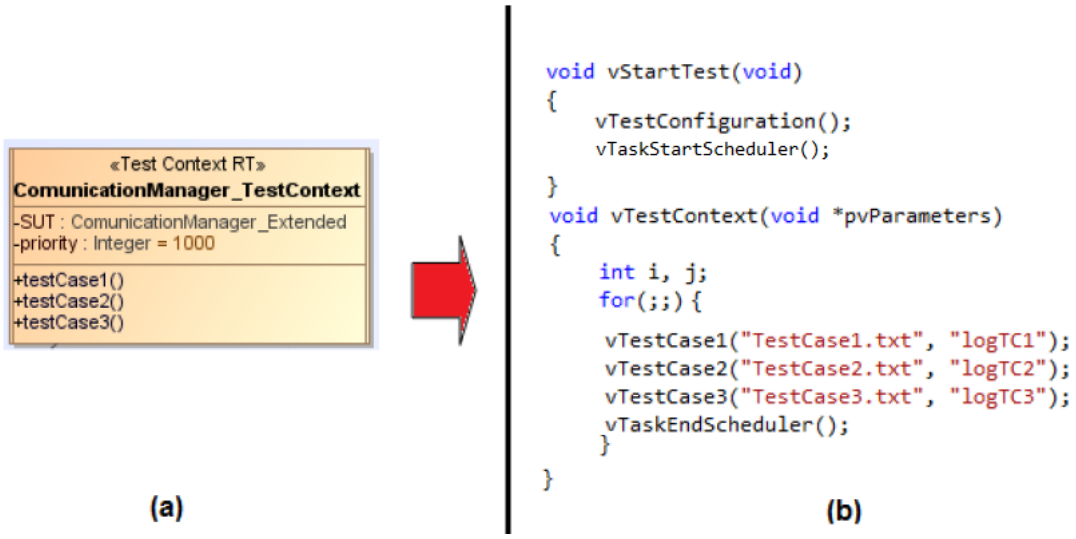

Figura 5.2: (a) Exemplo de um Test Context no nível PITM. (b) Parte do código representativo ao mapeamento.

Elemento: *Test Component*

Definição: Objeto do sistema que se relacionará com o *SUT*, produzindo e/ou consumindo elementos que provém dos casos de teste. Deve possuir um conjunto de interfaces que possibilite, via conectores, a comunicação com outros *Test Components* ou com o *SUT*.

Mapeamento: Conhecendo de antemão quem será o *SUT*, no contexto do *FreeRTOS*, qual-

quer processo que interaja com este *SUT* de alguma forma será classificado como um *Test Component*. Caberá ao testador decidir se os *Test Components* serão os mesmos processos usados pela aplicação original (se estes estiverem disponíveis) ou se serão emuladores criados especificamente para propósitos de teste (solução aconselhada principalmente quando trata-se de teste de integração). Caso seja decidido utilizar processos emuladores, é importante que estes modifiquem corretamente as variáveis essenciais para a execução do *SUT*, bem como possuam um *delay* com o mesmo tempo que o processo emulado gastaria para executar. Com isso, casos de teste que envolvem tempo também poderão ser realizados.

Para permitir que o comportamento dos casos de teste sejam executados com uma seqüencia determinística de passos, é necessário que os processos *Test Component RT* possuam como último comando a ser executado a chamada da rotina *"vTaskPrioriySet"*, recebendo como parâmetro a prioridade pré-estabelecida na fila *"testCaseExecutionPrioritiesQueue"*, forçando assim a mudança de prioridade do mesmo. Da mesma forma como foi estabelecido no mapeamento do SUT, um comando *switch* deve ser adicionado ao processo *Test Component*, a fim de permitir a manipulação dos estados do mesmo.

A Figura 5.3 apresenta visualmente como se dá parte do mapeamento apresentado. Mais especificamente, na Figura 5.3-b, é apresentado o código do processo *Test Component* (*vSensorManagerTask_TComponent*) que também implementa a operação *setToState* (Figura 5.3-a) através do comando *switch*. Ainda na Figura 5.3-b é possível visualizar a chamada a rotina *"vTaskPrioriySet"*, conforme indicado no mapeamento.

Elemento: *Arbiter*

Definição: Interface composta pelas operações *"setVerdict(v: Verdict)"* e *"getVerdict():Verdict"*. Tem como funcionalidade avaliar o resultado individual do teste a partir dos *Test Components* e atribuir o veredito final. Os resultados individuais são fornecidos ao árbitro via *Validation Actions* (mecanismos que verificam valores de variáveis em tempo de execução).

Mapeamento: Como o *FreeRTOS* é um SO desenvolvido com o intuito de possibilitar a programação de sistemas de tempo real, logo, não é aconselhável que o árbitro seja defina os vereditos em tempo de execução (como sugerem os exemplos presentes na especificação do UTP), caso contrário este poderia interferir drasticamente no tempo gasto, podendo invalidar

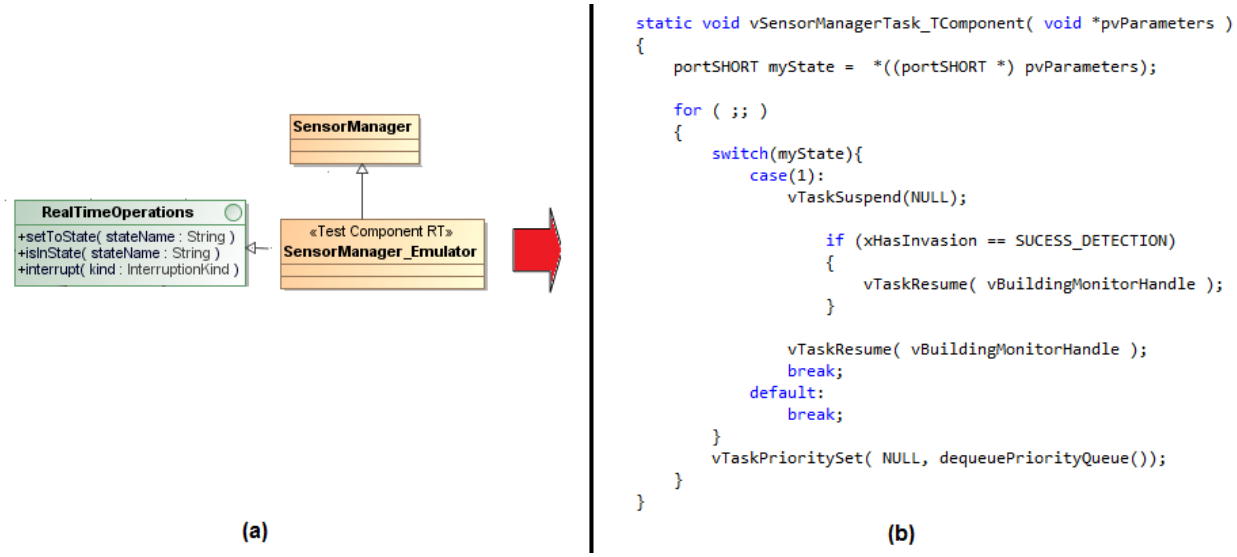

Figura 5.3: (a) Exemplo de um *Test Component* no nível PITM. (b) Parte do código representativo ao mapeamento.

o caso de teste. Devido a tais circunstâncias, mapeamos o árbitro para uma rotina, presente no mesmo módulo do processo *Test Context RT*, que será acionada após a execução de cada caso de teste. A rotina *"xArbiter"* tem por funcionalidade comparar resultados coletados durante a execução dos casos de teste (armazenados em *log*) com os resultados esperados, para assim poder inferir um veredito ao caso de teste. É importante que a rotina de *xArbiter* conheça as características do sistema (*e.g.* o tempo gasto para armazenamento em *log*), para que esses tempos de interferência possam ser subtraídos do tempo total.

A Figura 5.4 apresenta visualmente como se dá o mapeamento parcial da estrutura (o algoritmo de comparação, presente na rotina, irá variar de acordo com a forma como arquivos de *log* de do caso de teste estão estruturados).

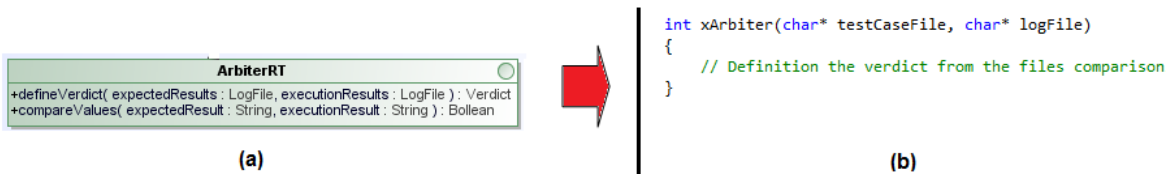

Figura 5.4: (a) Arbitro no nível PITM. (b) Parte do código representativo ao mapeamento.

Elemento: *Scheduler*

Definição: Interface composta pelas operações *"startTestCase()"*, *"finishTestCase(t: TestComponent)"* e *"createTestComponent()"*. Tem por objetivo controlar a execução dos diferentes *Test Components*.

Mapeamento: O controle dos *Test Components*, bem como do *SUT*, durante a execução dos casos de teste acontecerá via troca de prioridades, forçando assim que a ordem para execução dos processos (pré-estabelecida pelo testador) seja deterministicamente seguida. Portanto, para implementação de tal característica, uma rotina especial (*"vScheduler"*) será executada antes da execução do comportamento de cada caso de teste. Esta rotina terá o propósito de criar a fila *"testCaseExecutionPrioritysQueue"* que contém, em ordem, os valores das mudanças de prioridade que os processos terão que realizar para assim compor o comportamento de caso de teste. O Código Fonte 5.1 apresenta o pseudocódigo de um algoritmo simplificado que tem o objetivo de preencher a fila *"testCaseExecutionPrioritysQueue"* (este algoritmo é apenas uma sugestão de implementação). O algoritmo escolhido deverá estar implementado na rotina *"vScheduler"*.

Código Fonte 5.1: Pseudocódigo referente ao algoritmo que compõe a fila *"testCaseExecutionPrioritysQueue"*

```
SchedulerRoutine (testCaseProcessOccurences: ARRAY)
    BEGIN
        VARIABLES
            prioritysArray : ARRAY [testCaseProcessOccurences.length -
                1];
            testCaseExecutionPrioritysQueue: queue[
                testCaseProcessOccurences.length - 1];

        testCaseProcessOccurences <- readTheProcessesOccurences(
            testCaseFile);
        FOR I <- 0 TO prioritysArray.length DO
            BEGIN
                prioritysArray [I] = searchNextOccurence (
                    testCaseProcessOccurences[I], I+1,
                    testCaseProcessOccurences);
            END;
        testCaseExecutionPrioritysQueue <- prioritysArray;
    END.
```

```
INTEGER searchNextOccurence (value: STRING, I: INTEGER,
    testCaseProcessOccurences: ARRAY)
    BEGIN
        FOR J <- I TO testCaseProcessOccurences.length DO
            BEGIN
                IF testCaseProcessOccurences[J] = value THEN
                    RETURN | J - testCaseProcessOccurences.length |;
            END;
        RETURN -1;
    END.
```

Elemento: ***Test Configuration***

Definição: Demonstra como a coleção de objetos se relaciona entre si e suas conexões com o *SUT*.

Mapeamento: Segundo UTP, dentre os modelos de teste, deve existir um diagrama de estruturas compostas do *Text Context* onde este representará o *Test Configuration*. Para facilitar o mapeamento para o contexto do *FreeRTOS* é necessário que este diagrama seja também refinado, demonstrando como se dá a relação dos processos participantes. A observação de quais processos interagem com o *SUT* será importante para identificação dos *Test Components*. É necessário também que processos que não interagem diretamente como *SUT*, mas que lhe causam interferência devido suas características comportamentais (*e.g.* processos periódicos) estejam presentes neste diagrama. Em relação ao mapeamento direto para código, deverá existir uma rotina ("*vTestConfiguration*") a ser executada pela rotina auxiliar do *Test Context* ("*vStartTest*") para instanciação e pré-configuração das pré-condições dos casos de teste (criação dos processos do *SUT*, *Test Components*, etc). A Figura 5.5 apresenta visualmente como se dá o mapeamento da configuração de testes.

Elemento: ***Utility Part***

Definição: Parte do sistema de teste que representa os diversos componentes que ajudam os *Test Components* a realizar seus comportamentos de teste.

Mapeamento: Qualquer processo, módulo ou rotina que um dado *Test Component* faça uso

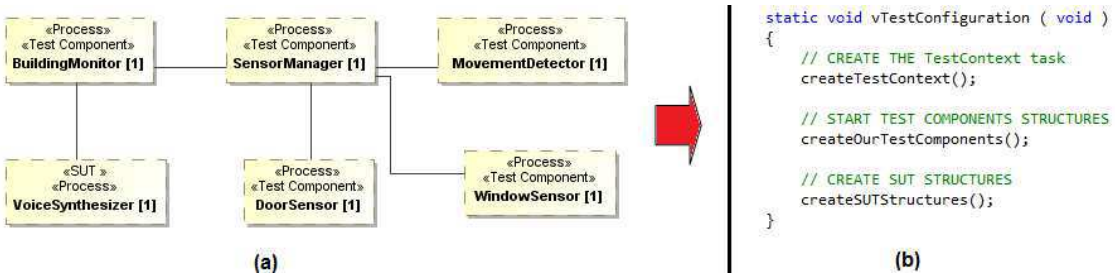

Figura 5.5: (a) Diagrama de estruturas compostas da estrutura *Test Context*. (b) Parte do código representativo ao mapeamento.

para executar seu comportamento, pode ser considerado um *Utility Part*.

5.1.2 Test Behavior

Elemento: *Test Control*

Definição: Indica a ordem de invocação dos casos de teste no *Test Context*.

Mapeamento: A especificação de como o *SUT* deverá ser testado ficará subentendida pela da ordem das chamadas dos dados casos de teste no processo *Test Context*, bem como pelos correspondentes *Test Cases*.

Elemento: *Test Case*

Definição: Concretização do propósito de teste. Sempre retorna um veredito.

Mapeamento: Será uma rotina única (*"vTestCase"*), a qual será chamada exclusivamente pelo processo *Test Context*. A cada nova chamada desta rotina, um arquivo diferente com a ordem de execução do caso de teste (proveniente dos diagramas de seqüência referentes ao comportamento dos casos de teste no nível PITM) será passado como parâmetro. Logo, cada chamada à rotina *"vTestCase"* executará um diferente *Test Case*. Como primeira tarefa, a rotina chama *"vScheduler"* para preencher corretamente a fila *"testCaseExecutionPrioritiesQueue"* (de acordo com a ordem recebida via arquivo). Após este passo, a ordem de execução do caso de teste estará definida. É importante garantir que o *SUT* e os *Test Components* modifiquem principalmente variáveis globais do sistema e que, a cada modificação destas variáveis, armazene-se em *log* os valores atuais e o tempo corrente.

Este armazenamento permitirá a posterior definição do vereditos pelo *Arbiter*. A Figura 5.6 apresenta visualmente como se dá o mapeamento.

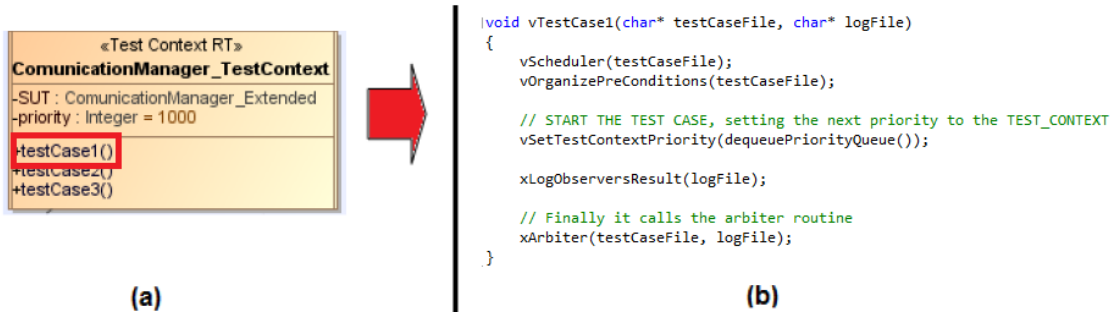

Figura 5.6: (a) Diagrama de estruturas compostas da estrutura *Test Context*. (b) Parte do código representativo ao mapeamento do *Test Case*.

Elemento: *Test Invocation*

Definição: Um caso de teste pode ser chamado com parâmetros específicos e dentro de um contexto específico. Os *Test Invocations* levam à execução do caso de teste.

Mapeamento: A invocação dos casos de teste será realizada pelo processo *Test Context*. A ordem da execução poderá ser visualizada diretamente pelo código do *Test Context* ou via leitura do *log* da execução.

Elemento: *Test Objective*

Definição: Comportamento que descreve informalmente o propósito do caso de teste.

Mapeamento: Comentários serão adicionados antes da chamada de cada caso de teste, estes explicando o propósito a ser testado no dado caso de teste.

Elemento: *Stimulus*

Definição: Dados de teste enviados para o *SUT*, a fim de controlá-lo para assim fazer avaliações sobre o mesmo.

Mapeamento: Não existirá mapeamento direto para este elemento. Os estímulos farão parte da implementação dos casos de teste e dos *Test Components*. Porém, estes estímulos poderão estar encapsulados na implementação das rotinas referentes *RealTimeOperations* (*setToState*, *interrupt*) forçando que o SUT ou Test Component receba os estímulos

necessários para a condução dos casos de teste.

Elemento: *Observation*

Definição: Dados de teste que refletem as reações do *SUT* e são utilizados para avaliações. É tipicamente o resultado de um estímulo enviado ao *SUT*.

Mapeamento: A observação dos resultados da execução dos casos de teste será realizada indiretamente, visto que os resultados dos estímulos provocarão alterações nas variáveis do sistema e estas serão armazenadas em *log* para posterior análise.

Elemento: *Coordination*

Definição: *Test Components* concorrentes (e potencialmente distribuídos) devem ser coordenados funcionalmente e em relação ao tempo a fim de garantir execuções de teste determinísticos e replicáveis. A coordenação é feita de forma explícita com a troca de mensagens entre os componentes ou implicitamente, com mecanismos de ordenação geral.

Mapeamento: Devido ao mecanismo desenvolvido, que utiliza a troca de prioridades dos processos que compõem os *Test Cases*, o determinismo e a replicabilidade dos testes é garantida. Por isso, não se faz necessário a existência de um elemento ou estrutura especial para tratar tal característica.

Elemento: *Default*

Definição: Mecanismo usado para especificar como responder ao recebimento de mensagens que não estão explicitamente modeladas na especificação.

Mapeamento: Este elemento será parcialmente mapeado. Isto se deve ao fato de que, como o FreeRTOS é um SO para execução de aplicações de tempo real, este tipo de aplicação tem que ser bem especificada e esta especificação deve tratar qualquer mensagem ou evento que possa ocorrer (inclusive interrupções) durante a execução. Portanto, parte do comportamento que o elemento *Default* especificaria para testes, não necessitará de mapeamento, pois já deverá estar bem descrito nos modelos e no código da aplicação. Porém, situações que não foram identificadas no projeto podem levar ao lançamento de uma exceção inesperada. Tal situação será tratada por uma rotina especial que a capturará, tal situação fará com que o caso de teste seja inferido como *inconclusive*.

Elemento: ***Verdict***

Definição: Cada caso de teste retorna um veredito. As possibilidades de vereditos são *pass*, *fail*, *inconclusive* e *error*.

Mapeamento: Com já dito anteriormente, a rotina *"xArbiter"* irá determinar o veredito do caso de teste (via comparações através de arquivos de *log*). Cada possível veredito possui o seguinte mapeamento: i) *pass*, quando todos os valores armazenados em *log* estão de acordo com os valores esperados pelo testador; ii) *fail*, algum dos valores armazenados em *log* não está de acordo com os valores esperados pelo testador; iii) *inconclusive*, algo inesperado aconteceu durante a execução do caso de teste, seja uma exceção lançada, um *loop* infinito, etc; e iv) *error*, algum erro proveniente da plataforma aconteceu que levou ao não término da execução do caso de teste.

Elemento: ***Validation Action***

Definição: Ação que serve para avaliar o estado da execução de um caso de teste, avaliando as observações *SUT* e/ou características adicionais. A ação de validação é realizada por um *Test Component* e define um veredito local.

Mapeamento: No contexto em questão, as ações de validação são mapeadas para uma única chamada à rotina *"xArbiter"* ao final da execução do caso de teste. Não existirão definições locais de vereditos, apenas um veredito final para cada caso de teste.

Elemento: ***Test Log/ Log Action***

Definição: Armazenamento em arquivo dos passos de execução dos casos de teste.

Mapeamento: A rotina *"xLogObserversResult"* será utilizada para armazenamento em *log* de cada modificação importante nas variáveis globais do sistema. Cada execução desta rotina armazenará as variáveis em questão e o tempo corrente para posterior análise. O tempo gasto para execução da rotina de armazenamento em *log* deverá ser descontado no tempo final de execução (tarefa realizada pela rotina do *Arbiter*).

5.1.3 Time Concepts

Elemento: *Timer*

Definição: Mecanismo que gera um evento de *timeout* quando um intervalo de tempo expira em uma dada instância. *Timers* pertencem aos *Test Components*.

Mapeamento: Os dados de tempo, bem como restrições, serão analisados pela rotina do *Arbiter* que verificará se uma dada restrição de tempo foi ou não satisfeita.

Elemento: *Timezone*

Definição: Mecanismo de agrupamento para *Test Components* segundo características de tempo. Cada *Test Component* pertence a um dado *timezone*. *Test components* no mesmo *timezone* possuem o mesmo tempo por referência.

Mapeamento: Devido restrições impostas pela plataforma, todos os *Test Components* pertencerão ao mesmo *timezone*, ou seja, existirá somente uma referência de tempo durante toda a execução dos casos de teste. *Clocks* relativos não serão considerados.

5.1.4 Test Data

Até o presente momento os elementos deste subgrupo (*Wildcards*, *Data Pool*, *Data Partitions*, *Data Selector* e *Coding Rules*) não são diretamente mapeáveis para o contexto do *FreeRTOS*. Tal fato deve-se à simplicidade deste SO e a necessidade de não interferência em relação ao tempo (por tratar de sistemas de tempo real). Caberá ao testador incluir diretamente nos casos de teste seus dados, ou forçar que os *Test Components* produzam dados adequados para execução dos testes.

5.2 Suporte Ferramental

Conforme anteriormente já apresentado (Seção 4.4), a ferramenta RTTAG foi criada com o objetivo de auxilar os testadores de STRs, automatizando a criação de arquiteturas de testes em diferentes níveis de abstração. O segundo módulo desta ferramenta (PITM2PSTM - vide Figura 4.13) realiza a transformação entre os modelos de teste independentes de plataforma (PITM) para os modelos de teste equivalentes no nível específico de plataforma (PSTM), ou

seja, transforma os modelos da arquitetura de testes resultante da execução do primeiro módulo da ferramenta (PIM2PITM), em modelos C da mesma arquitetura segundo a plataforma de execução escolhida, no caso o FreeRTOS.

O módulo PITM2PSTM nada mais é do que a formalização das regras de mapeamentos informais apresentadas anteriormente através da utilização de regras ATL. Desta forma os mapeamentos, antes de caráter apenas informal, passam a ser também formalizados. Como resultado destas aplicações serão fornecidos um conjundo de arquivos .xmi com a representação da arquitetura de testes segundo o metamodelo de C (desenvolvido também neste trabalho e apresentado no Apêndice E deste documento).

As regras de transformação ATL referentes ao módulo PITM2PSTM estão disponíveis para visualização no seguinte endereço eletrônico: `https://mestradotool.googlecode.com/svn/trunk/Mestrado/transformations/PITM2PSTM.atl`.

5.3 Avaliação dos Mapeamentos

Com o intuito de avaliar a aplicabilidade dos mapeamentos, bem como sua efetividade em gerar arquiteturas de testes úteis no contexto de STRs segundo a plataforma do FreeRTOS, um estudo de caso foi executado. O principal objetivo deste estudo foi avaliar a completude dos mapeamentos propostos no tocante à analisar quanto de trabalho extra o testator terá de executar para construir uma arquitetura de testes completa e utilizável. Para tal a seguinte metodologia foi aplicada.

1. Uma arquitetura de testes de um STR, no nível PITM, foi escolhida. Os modelos desta arquitetura foram provenientes da execução do módulo PIM2PITM da ferramenta RTTAG.

2. Os modelos PITM (escolhidos na execução da etapa 1) serviram diretamente como entrada para o módulo PITM2PSTM da ferramenta RTTAG.

3. Os modelos C, referentes a arquitetura testes e resultantes da etapa anterior da metodologia, foram transformados para seu código C equivalente (processo manual).

Processo simplificado visto que os modelos C gerados pela ferramenta fornecem a estrutura completa do código C equivalente.

4. Analisou-se os resultados obtidos na etapa anterior e identificou-se os trechos que ainda necessitavam de complemento de código para que a arquitetura de testes fosse realmente usável.

A seguir, serão apresentados mais detalhes a respeito das etapas da metodologia seguida, bem como serão apresentados os artefatos resultantes da aplicação dos mapeamentos.

A fim de cumprir a primeira etapa da metodologia anteriormente descrita, a arquitetura escolhida para participar do estudo de caso, no nível PITM, foi a arquitetura de teste do *Sistema de Alarmes*. Este mesmo sistema foi utilizado no estudo realizado para avaliação das extensões de UTP (Seção 4.5). Esta escolha foi realizada por já possuirmos no grupo o código relativo ao simulador deste sistema. Tal fato nos permitiu avaliar a arquitetura de testes e executar efetivamente casos de teste reais.

Na segunda etapa, os modelos PITM serviram como entrada para o módulo PITM2PSTM da ferramenta RTTAG. Como resultado desta etapa, um conjunto de arquivos no formato XMI foram obtidos. Estes modelos C representam a arquitetura de testes equivalente aos modelos PITM, para a plataforma FreeRTOS (nível PSTM). A Figura 5.7 apresenta parte da representação da arquitetura de testes gerada como resultado do módulo PITM2PSTM, para o estudo de caso *Sistema de Alarmes* (representação gráfica fornecida pelo *plug-in* ATL-DT do ambiente de desenvolvimento Eclipse [25]). A Figura 5.8 apresenta a representação do mesmo arquivo mostrado na Figura 5.7, desta vez no formato .xmi. Em ambas figuras (5.7 e 5.8) está em destaque a modelagem da rotina representativa do caso de teste (resultante do mapeamento do elemento *Test Case*, Seção 5.1.2).

Como o resultado da execução do módulo PITM2PSTM gera modelos C, é necessário então que posteriormente exista a execução de uma transformação modelo-texto para que finalmente seja obtido o código executável da arquitetura de testes. Este não é o escopo atual de atuação da ferramenta RTTAG, esta finaliza seu trabalho com a construção dos modelos PSTM (uma possível evolução da ferramenta poderá incluir as regras de transformação entre modelos e código textual C). Porém, a título de visualização dos resultados produzidos pela ferramenta, o código resultante do que seria a transformação dos modelos PSTM para texto

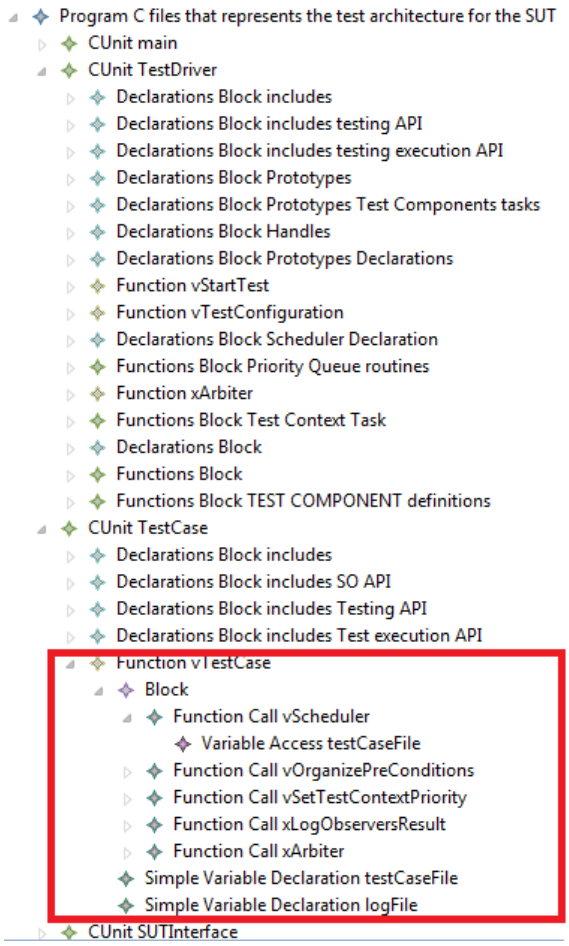

Figura 5.7: Representação gráfica de parte dos modelos PSTM do “Sistema de Alarmes”.

foi manualmente produzido e, juntamente com o código de simulação da aplicação *Sistema de Alarme* está disponível para análise no seguinte endereço eletrônico: `http://gmf.ufcg.edu.br/~everton/AlarmSystemFreeRTOSCaseStudy.zip`.

Com a realização da passagem dos modelos PSTM para código, realizada manualmente conforme previsto na metodologia, foi observada a necessidade de inclusão de um conjunto de complementos para tornar o código totalmente executável. As inclusões foram necessárias por tratarem de lacunas que a ferramenta RTTAG não consegue resolver ou que os mapeamentos não cobrem, foram as seguintes: i) a instrumentação do código SUT. É necessário que o testador, conhecendo bem o sistema, inclua manualmente chamadas às rotinas da API, proposta em [53], em locais estratégicos, para assim observar as variáveis que considera importantes para análise da execução dos testes; ii) a implementação do código equivalente às operações *setToState*, *isInState* e *interrupt* (interface *RealTimeOperations*) no SUT e nos

processos *Test Components*. Cada sistema terá uma implementação diferente para tais operações, o testador, mais uma vez de posse de seus conhecimentos a respeito do sistema, deverá implementar estas operações de maneira coerente, para que estas possam ser usadas nos casos de teste; e iii) o corpo da rotina *xArbiter*, como explicado no mapeamento da estrutura *Arbiter*, dependendo de como forem configurado os arquivos de *log* e/ou capturados as informações das variáveis, a função que realiza a arbitragem será implementada de uma maneira diferente.

Após a inclusão do código referente aos pontos acima elencados, um caso de teste simples, que tinha por objetivo testar a execução do comportamento de interrupção, foi planejado. Ou seja, foram identificadas as variáveis a serem analisadas, o código do SUT foi instrumentado e os arquivos de estabeleciam a ordem de execução dos processos envolvidos nos casos de teste (arquivos de entrada da rotina *Test Case*) foram construídos. Após isto, o caso de teste em questão foi executado sem maiores dificuldades. Com isto, é possível observar que, com o mapeamento realizado através do uso da ferramenta RTTAG, juntamente com o código complementar, foi possível construir uma arquitetura de testes executável no contexto do FreeRTOS.

A partir da análise dos artefatos gerados com a execução do módulo PITM2PSTM da ferramenta RTTAG para o estudo de caso, foi possível estabelecer as seguintes conclusões gerais: i) o módulo PITM2PSTM realiza com satisfatoriedade o seu objetivo de gerar, de forma automática, os elementos da arquitetura de testes no nível PSTM; ii) os artefatos gerados pelo módulo foram considerados úteis e corretos, sendo a representação em código do modelo gerado executável e necessitando de poucas modificações; iii) o mapeamento realizado para a plataforma FreeRTOS foi coerente, no escopo do estudo de caso, preservando a semântica da arquitetura original independentes de plataforma; e iv) apesar de não completo, ainda exige um trabalho extra do testador para complementar o código dos artefatos produzidos, o mapeamento juntamente com a ferramenta RTTAG trouxe um ganho em tempo de construção (graças a automatização) e diminuição da dificuldade da construção de arquiteturas de teste RT para o contexto do FreeRTOS.

Mais uma vez, as limitações de tempo e de recursos impediram que um estudo mais abrangente pudesse ter sido realizado a fim de analisar com profundidade o quão corretas e úteis são as regras de mapeamento entre os elementos de UTP para o contexto do FreeRTOS.

Porém, os resultados conseguidos através da realização do estudo de caso nos fornece indícios que o mapeamento é realmente correto e efetivo, cumprindo o objetivo pelo qual foi projetado.

5.4 Considerações Finais do Capítulo

O capítulo aqui finalizado apresentou um conjunto regras informais para mapeamento dos elementos do perfil de testes da UML (e sua extensão proposta no Capítulo 4 deste documento), para o contexto de um SO de tempo real, o FreeRTOS. Para tal, além do mapeamento proposto, que foi formalizado através de regras de transformação entre modelos escritas na linguagem ATL (módulo PITM2PSTM da ferramenta RTTAG), foi desenvolvido também um metamodelo para a linguagem C (Apêndice E). Finalizando, foi apresentado neste capítulo um estudo de caso realizado com o objetivo de avaliar o quão completo é o conjunto de mapeamento proposto, bem como analisar os resultados fornecidos pelo segundo módulo da ferramenta RTTAG. Os resultados do estudo de caso apresentaram indícios que os mapeamentos propostos realmente cumprem seu objetivo de auxiliar a passagem dos modelos de teste do nível PITM para PSTM, proporcionando a construção de arquiteturas de teste para uma plataforma de STRs, mesmo esta sendo uma das mais restritivas, sem acrescentar grandes dificuldades neste processo.

Figura 5.8: Parte de um dos arquivos .xmi que representam os modelos PSTM do “Sistema de Alarmes”.

Capítulo 6

Trabalhos Relacionados

Neste capítulo é apresentada uma seleção de trabalhos, referentes à uma pesquisa na literatura especializada, que refletem o estado da arte no contexto da geração automática de casos de teste a partir de modelos, bem como tratam sobre o que se tem produzido a respeito de modelagem e testes para sistemas de tempo real. No decorrer da apresentação dos trabalhos relacionados, serão realizadas análises críticas dos mesmos, buscando, quando possível, localizar lacunas existentes, verificar pontos positivos que ajudaram no direcionamento do trabalho desenvolvido e comparações dos mesmos com o trabalho apresentado neste documento.

6.1 MBT

É notório o crescente uso de MBT e também está claro que muitos dos esforços da área de testes têm sido direcionados em aperfeiçoar esta abordagem e propor melhorias para a mesma. Pensando nisto, e buscando analisar o que tem sido produzido cientificamente na área, foi selecionado um conjunto de artigos que tem por foco o uso de MBT, ou alguma aplicação desta, para assim traçar um panorama da pesquisa nesta área. São exemplos dos trabalhos analisados: Ernits *et al.* [28], Hartmann *et al.* [36], Perez *et al.* [62], Barbosa *et al.* [11], Neto *et al.* [57], Mandrioli *et al.* [54], Orozco *et al.* [61], Dalai *et al.* [18], etc. A partir destes trabalhos, alguns foram selecionados e serão melhor discutidos por mostrarem-se mais interessantes pelos seguintes motivos: i) utilizarem modelos UML; ii) apresentarem resultados práticos e/ou ferramentas de automação; ou iii) trazer um panorama

geral da pesquisa na área. Na seqüencia, os artigos em questão são apresentados.

Barbosa *et al.* [11] propõem um método de teste funcional para componentes (*Automatic Functional Component Testing* - AFCT). O método é baseado em estratégias e técnicas de MBT e faz uso de diagramas UML (diagramas de classe, de sequência e de estados) juntamente com restrições OCL (*Object Constraint Language*) para derivar casos de teste de forma automática. Para garantir a aplicabilidade do método proposto, no trabalho também é apresentada a ferramenta SPACES (*SPecification bAsed Component tESter*). SPACES tem por funcionalidade gerar casos de teste para componentes na linguagem Java e dar suporte para execução e análise dos dados obtidos com os testes. Dentre as limitações do trabalho temos: i) a ferramenta SPACES trabalha somente com diagramas da UML 1.4, tornando-se desatualizada visto que a UML atualmente já se encontra na sua versão 2.1; e ii) o artigo em questão não fornecesse dados que indiquem o quão eficiente a metodologia proposta é em relação a outras existentes na literatura, nem trata de classes especiais de sistemas, como os STRs.

A defasagem do trabalho de Barbosa *et al.* com relação a evolução de UML é ainda mais problemática quando se objetiva a geração de casos de teste para o contexto de STRs. Em sua versão 1.4 (usada no artigo), a UML carecia em mecanismos que permitissem a modelagem de aspectos RT, carência esta que foi diminuída nas suas novas versões. Ainda no trabalho de Barbosa *et al.*, o nível de teste usado é diferente do empregado no nosso trabalho (trabalhamos com teste de integração e eles com teste de unidade). O nível de dificuldade em gerar testes de integração, bem como sua arquitetura, é bem maior que com testes de unidade, visto que para testes de unidade já existem diversos *frameworks* (*e.g.* Junit) que auxiliam este processo.

Ernits *et. al.* [28] desenvolveram um trabalho um pouco atípico, porém bastante interessante. Trata-se de um trabalho que tem por objetivo demonstrar a aplicabilidade de MBT no contexto das aplicações Web (domínio pouco investigado com MBT) utilizando um *framework* de teste. Ao invés da utilização de modelos gráficos ou textuais (estratégia tradicional), os modelos usados para representar as aplicações são escritos em C# e os serviços Web são acessados através de bibliotecas .NET. Trata-se de um trabalho com resultados aplicados a um projeto real de uma grande empresa, trazendo assim maior confiabilidade na escalabilidade da técnica e dos resultados obtidos. Como resultados alcançados, o trabalho faz refe-

rência a defeitos descobertos nas aplicações, que possivelmente passariam desapercebidos pelas técnicas tradicionais de teste, e, principalmente, a significativa redução de custos que a aplicabilidade de MBT trouxe neste contexto. Como ponto negativo desta técnica destaca-se o tempo necessário para aprendizado do novo formalismo de modelagem. Tal característica traz um *overhead* à atividade de testes que não seria necessário caso modelos já conhecidos pelos desenvolvedores (tais como UML ou *Statecharts*) fossem utilizados.

A forte conexão da solução de Ernits *et. al.* com os modelos textuais usados dificulta sua utilização. Caso modelos de *design* independentes de plataforma fossem utilizados (*e.g.* usando UML), este acomplamento estaria reduzido, bem como existiria uma maior independência com relação aos demais artefatos de projeto e de teste. Esta foi uma das nossas preocupações, escolhendo a linguagem UML como padrão para modelagem de STRs e utilizando seus artefatos para geração das arquiteturas de teste.

Por fim, o trabalho de Neto *et al.* [57] foi escolhido por ser um artigo bem diferenciado da maioria. Seguindo os princípios da Engenharia de Software Experimental, os autores preocuparam-se não somente em propor uma nova abordagem MBT, mas, ao contrário, tiveram por objetivo buscar na literatura trabalhos que já propunham novidades na área e realizaram uma análise comparativa entre estes. Tal característica facilita que empresas ou instituições, que desejem integrar MBT na realização das suas atividades de testes, não se vejam perdidas em meio as inúmeras propostas existentes. Com esse trabalho comparativo é possível escolher a realização que melhor se encaixa aos objetivos da empresa/instituição sem grandes dificuldades. Para realização deste estudo investigativo foram selecionados, dentre 202 iniciais, 78 artigos a serem analizados segundo a metodologia formal de catalogação Revisão Sistemática (*Systematic Review* [44]). Cada trabalho foi classificado dentro de categorias que se distiguiam segundo a forma de representação usada nos modelos, bem como segundo o formalismo usado. Em sequência, um conjunto de características, que os autores julgaram importantes, foram estabelecidas para assim buscar uma classificação de cada trabalho a ser analisado para trazer um carater comparativo para os mesmos. Dentre as diversas características comparativas utilizadas, a título de exemplificação, temos as seguintes: i) nível de abstração dos testes; ii) escopo a ser aplicado; iii) existência de ferramentas de suporte; iv) nível de automação; v) critério para geração dos testes; e vi) complexidade.

O trabalho ainda fez análises a respeito dos dados obtidos. Dentre as conclusões apresen-

tadas, os autores destacam que cerca de 66% dos trabalhos usam MBT aplicada para testes de sistema. Outra das conclusões obtidas pela pesquisa reflete um dos principais princípios de MBT que é a automação, cerca de 64% dos trabalhos pesquisados usam ferramentas (na maioria não 100% automáticas) para auxiliar a execução das técnicas. Em relação aos principais formalismos utilizados, a maior parte dos trabalhos utilizaram os modelos UML (*Statecharts*, diagramas de Classe e de Sequência), seguidos por Máquinas de Estados Finitos e as especificações *Z*. O trabalho também apontou um conjunto de deficiências nas técnicas estudadas tais como: i) poucas trataram de como usar MBT para modelagem e teste dos requisitos não-funcionais dos sistemas; ii) a quase inexistência da realização de análises experimentais bem fundamentadas nos trabalhos sobre MBT; e iii) abordagens MBT normalmente não são integradas com o processo de desenvolvimento de software.

Tal trabalho mostrou-se interessante por reunir um grupo significante de trabalhos de pesquisa na área de MBT e de forma sistemática compará-los. Desta forma, foi possível verificar que modelos UML são ainda bastante utilizados neste contexto e que a deficiência "iii" pode ser facilmente dissipada com o uso de MDT, como a nossa solução pretende. Apesar do trabalho apresentado neste documento se tratar de uma utilização de MDT, a análise de trabalhos relacionados à MBT mostra-se importante por esta (MDT), ser essencialmente uma realização de MBT. Trabalhos como o de Neto *et al.* [57] são interessantes por trazer um panorama geral da área, bem como mostrarem quais notações e ferramentas são as mais usadas e em quais contextos.

6.2 MDT

Javed et al. [41] propõem um método, fazendo uso do *framework* MDA e de regras de transformação, para geração de casos de teste no nível de teste de unidade. O método propõe que o comportamento do sistema seja modelado através do uso de sequências de chamadas a métodos - SCM (subconjunto dos elementos do metamodelo do diagrama de sequência UML). A partir destes modelos, regras de transformação escritas em Tefkat [48] são usadas para derivar os casos de teste em modelos *xUnit*, ou seja, independentes de plataforma. Por fim, outro conjunto de regras de transformação (escritas em MOFScript [59]) são usadas para fazer a passagem dos modelos de teste independentes de plataforma para código concreto de

teste em Java (seguindo o *framework* JUnit [40]). Este trabalho mostra-se interessante por trazer de forma clara uma aplicação dos conceitos de MDA para testes e por fazer fortemente a separação dos modelos de teste em níveis de abstração, contribuindo assim para ganhos em portabilidade, interoperabilidade, rápido desenvolvimento, manutenção, etc. Porém, a solução apresentada falha por não aplicar por completo os conceitos de MDA/MDT. Isto pode ser constatado por não trazer como formalismo de modelagem UML e sim apenas parte do metamodelo de seu diagrama de sequência.

Dai [17] cria uma metodologia que demonstra como é possível aplicar os conceitos pertencentes à UTP sobre sistemas modelados com UML, objetivando assim automatizar a completa construção de artefatos de teste. Esta metodologia é concretizada através de regras de transformação escritas em QVT [1]. Este trabalho é interessante por conseguir agrupar numa única metodologia os principais conceitos de MDT, principalmente a preocupação na geração de todos os artefatos necessários para a execução da atividade de testes na sua plenitude. Outro ponto positivo no trabalho é o uso dos formalismos defendidos pela OMG para aplicação de MDA tais como: modelos UML, para modelagem do sistema; e QVT, para desenvolvimento das regras de transformação. Como pontos fracos deste trabalho tem-se que a definição das regras de transformação ainda não está completa, por não distinguir grupos de sistemas e suas características especiais na sua metodologia, e que ainda não existem ferramentas que ofereçam suporte aos conceitos do perfil UTP.

Semelhantemente ao trabalho de Dai, Zander *et al.* [73] fazem uso do perfil de teste da UML (UTP). Porém, neste artigo o objetivo é transformar, via regras de transformação, os modelos de teste a nível PIM (aplicando UTP) em código TTCN-3 (*Testing and Test Control Notation version 3* [33]) no nível PSTM, passível de execução. Este trabalho tem como ponto positivo conseguir mapear a semântica da maioria das estruturas presentes ao UTP para uma linguagem de teste já bastante utilizada e passível de execução (TTCN). Porém, a solução torna-se inviável no contexto dos STRs por TTCN-3 ainda não dar suporte ao teste deste tipo de sistemas.

Lima *et al.* [50] apresenta uma solução baseada em MDT para geração automática de arquiteturas de teste como o propósito de checagem das interações entre componentes KobrA [8], cliente e servidor. Neste trabalho também é proposto um perfil UML (perfil BIT) que ajuda a definir quais elementos devem estar presentes durante a atividade de teste e di-

recionam a ferramenta (também parte da solução apresentada) para geração dos artefatos de teste. Este trabalho mostra-se importante por trazer uma realização efetiva de MDT, apresentando inclusive, resultados concretos e uma ferramenta para uso. Outro ponto positivo é a clara distinção que os autores fazem dos modelos de testes em níveis de abstração, conceito este importante para a aplicabilidade de MDT. Como pontos limitantes desta pesquisa pode-se destacar o escopo da solução apresentada. Os resultados alcançados são úteis unicamente no contexto de teste de componentes (não aplicável a componentes RT) e, unicamente se estes forem desenvolvidos segundo a metodologia KobrA.

O Desenvolvimento Dirigido por Modelos (DDM) tem por objetivo colocar a modelagem e a construção de regras de transformação como centro do processo de desenvolvimento de software. Fazer uso de DDM juntamente com MDT, apesar de ambas trabalharem essencialmente com modelos, não é uma tarefa simples. Ambas técnicas possuem particularidades e especificidades que dificultam a integração das duas num único processo. Alves *et al.* [4] discutem essas dificuldades e apresentam direcionamentos para solução dessas problemáticas em diversos contextos, bem como apresentam uma proposta concreta para integração entre DDM e MDT. As questões levantadas no trabalho de Alves *et al.* são bastante interessantes pois permitem que empresas que se proponham a usar MDD e MDT consigam atingir seu objetivo sem maiores dificuldades e com isso atingir os benefícios que ambas técnicas podem oferecer.

6.3 Modelagem e Testes em STRs

Com Respeito à Modelagem de STRs

A maioria das notações usadas para modelagem de STRs preocupam-se unicamente com a modelagem da parte comportamental dos mesmos. Dentre os mais usados, e que melhor empregam a questão da modelagem do tempo, existem os *Timed Labelled Transition Systems* (TLTS) e os *Timed Input-Output Labelled Transition Systems* (TIOLTS). Ambos são extensões do já tradicional formalismo usado para modelar comportamento de sistemas, o LTS (*Labelled Transition Systems*). Tanto TLTS quanto TIOLTS possuem representação simplificada e conseguem modelar as restrições temporais através das suas transições dotadas com ações (discretas e *time-elapsing*). Trabalhos como o de Krichen & Tripakis [46] e de El-

Nouaary *et al.* [27] usam tais representações para modelagem de STRs, inclusive utilizando-os como modelos de entrada para ferramentas que têm por objetivo a geração automática de artefatos. Seguindo linha semelhante aos TLTS e TIOLTS, existem os *Timed Automata* (TA) e os *Timed Automata with Inputs and Outputs* (TAIO), distinguindo dos TLTS e TIOLTS por serem baseados na teoria dos automatos e não em LST. Muitos dos trabalhos da área de *Model-Checking* para STR fazem uso de TA e TAIO (*e.g.* Hessel *et al.* [39] e Krichen & Tripakis [47]).

Os formalismos de modelagem para STRs anteriormente apresentados assemelham-se entre si por objetivarem unicamente a modelagem da parte comportamental dos sistemas. Porém, apesar de conseguirem modelar (com certas restrições) os requisitos temporais, estes formalismos falham na modelagem de eventos (principalmente assíncronos) e por não incorporarem também mecanismos de modelagem da face estrutural do sistema, necessitando assim que algum outro formalismo também tenha que ser empregado para preencher esta deficiência.

Trabalhos e estudos recentes têm sido desenvolvidos para introduzir o formalismo de modelagem da UML na área dos STRs. Por muito tempo os projetistas e desenvolvedores de STRs não acreditavam que seria possível somente usando UML modelar todos os requisitos que os STRs possuem. Porém, com as novas versões de UML, a utilização de perfis especializados e de OCL, esta ideia tem começado a tornar-se possível.

Trabalhos como o de Selic & Rumbaugh [66] (escrito no ano de 1998) já incentivava ao uso de UML no contexto de STRs. Neste artigo em particular, os autores descrevem um conjunto de construções que facilitam o *design* de softwares para arquiteturas *Real-Time*, estas especificadas em UML. Mesmo fazendo uso de uma versão antiga da UML (UML 1.4) os autores conseguem através de exemplos práticos explicar como é possível modelar parte das faces estruturais e comportamentais de um STR complexo. Outros trabalhos, como o de Jong [20], fazem uso de UML juntamente com uma linguagem auxiliar (geralmente uma linguagem formal) para modelar STRs críticos.

Buscando complementar a linguagem e torná-la capaz de ser usada sozinha para modelagem de STRs, a OMG incorporou, nas versões posteriores da UML, estruturas para facilitar a modelagem de requisitos temporais (*e.g* diagrama de tempo, elementos de duração, etc). Esta evolução da UML contribuiu também para preenchimento de outras lacunas existentes.

Estudos têm sido realizados para o desenvolvimento de perfis próprios para modelagem de requisitos específicos dos STRs. Dentre os perfis mais conhecidos e mais utilizados tem-se: i) o UML-RT, que provê o conceito de objetos ativos e se propõe a descrever aplicações concorrentes e distribuídas. Trabalhos como Lyons [52] e Douglass [23] incentivam o uso deste perfil apresentando seus benefícios; e ii) MARTE, que tem por objetivo prover mecanismos necessários para modelagem dos requisitos funcionais e não funcionais dos STRs (*e.g.* o trabalho de Demathieu *et al.* [22]). Os perfis (UML-RT e MARTE) se preocupam com a modelagem de requisitos RT num nível de abstração mais baixo (manipulando por exemplo com semáforos), contrapondo com o objetivo das diretrizes e do perfil desenvolvido no nosso trabalho que tiveram o objetivo da modelagem dos requisitos de STRs num nível de abstração mais elevado.

A Figura 6.1 apresenta resumidamente uma tabela comparativa entre os trabalhos acima citados e o por nós desenvolvido, considerando aspectos de modelagem de STRs.

Trabalho	Notação	Característica
Nosso Trabalho	UML + *Real Time Design Profile* + Pacote *Real Time Elements* + Diretrizes	Modelagem de diversos aspectos RT num alto nível de abstração. Uso de somente uma notação (UML).
Selic & Rumbaugh	UML 1.4	Modelagem somente de alguns aspectos comportamentais.
Jong	UML + Linguagem Formal	Preocupação de modelagem de STRs críticos.
Lyons	UML + UML-RT	Baixo nível de abstração.
Douglass	UML + UML-R	Baixo nível de abstração.
Demathieu *et al.*	UML + Marte	Baixo nível de abstração.

Figura 6.1: Tabela comparativa entre os trabalhos relacionados à modelagem de STRs e o trabalho apresentado neste documento.

Com Respeito à Testes de STRs

Com a difusão do uso de STRs foi necessário investir em mecanismos que garantissem a integridade dos mesmos e que demonstrassem que estes são confiáveis o suficiente para uso. Diversos trabalhos foram desenvolvidos no sentido de garantir que os modelos dos sistemas estão corretos (*model checking*), como o de Gowen [32]. Porém, garantir que o modelo do sistema está correto nada infere a respeito do software final, diversos *bugs* podem ser inseridos durante a fase de implementação do software. Por tais motivos, como em qualquer software, existe a necessidade da aplicação da atividade de testes também em STRs.

Devido às suas características especiais, a atividade de testes de STRs torna-se ainda mais complexa. Os testes de um STR devem ter preocupações que vão além da verificação da corretude dos resultados, mas também na análise do *quando* estes resultados são obtidos. Outra grande dificuldade em relação a testes de STRs é o ambiente para execução dos mesmos. Pois, na maioria das aplicações *real-time* existe a necessidade de lidar com diversos processos simultâneos, alguns de forma síncrona outros de forma assíncrona, tal situação é complexa de ser controlada.

Neste contexto alguns trabalhos têm direcionado esforços para melhorar a atividade de testes para STRs e conseqüentemente aferir maior credibilidade aos produtos elaborados. Seguindo a idéia de utilizar os modelos de *design* para geração de artefatos de teste, os seguintes trabalhos apontam resultados nesta linha: Hessel [38], Li *et al.* [49], Andrade & Machado [6] e Andrade *et al.* [7].

Hessel [38] teve o objetivo de reunir da literatura, um conjunto de técnicas de teste que poderiam ser aplicadas a STRs, implementá-las numa ferramenta e avaliá-las em um estudo de caso de proporções industriais. Como resultado das análises, as experiências mostraram que o problema de encontrar uma boa abstração para o sistema é a principal dificuldade para uma ferramenta de MBT neste contexto. Quando aplicadas às técnicas em modelos com alto nível de abstração, os testes gerados muitas as vezes não testavam suficientemente o sistema. Em contrapartida, se os modelos usados possuem baixo nível de abstração, a geração dos testes tornavam-se tão complexas quanto testar diretamente o sistema.

Li *et al.* [49] propõe uma abordagem para geração de casos de teste em STRs orientada a propriedades. O artigo define uma estratégia de teste para propriedades de segurança. Como formalismo de modelagem são usados os *Time-Enriched Statecharts* que na execução da abordagem são transformados para máquinas de estado estendidas para posteriormente serem derivados os casos de teste. [49] foca somente em linguagens de especificação deixando de lado outros conceitos importantes que devem ser levados em consideração durante a execução da atividade de testes.

Andrade & Machado [6] apresentam uma extensão aos *Input/Output Symbolic Transition Systems* (IOSTS), que por sua vez já são uma extensão de LTS para permitir a representação de interrupções (eventos assíncronos) em sistemas reativos. Esta adaptação permite que a geração de casos de teste possa ser realizada em alto nível sem a preocupação com a explosão

de espaços de estados (a solução encontrada foi fazer uso de testes simbólicos) e a inclusão do teste das interrupções. Este trabalho se diferencia exatamente por fornecer uma forma para modelagem e teste de interrupções, grande dificuldade para testes de STRs.

Andrade *et al.* [7] discutem em seu trabalho problemas inerentes à construção de modelos de teste e à geração automática de casos de teste para sistemas de tempo real embarcados com interrupções na plataforma do SO FreeRTOS. Questões como "Qual o modelo de teste mais apropriado?" e "Quais melhorias devem ser desenvolvidas no ambiente de execução do FreeRTOS?" são tratadas no trabalho. Uma proposta de solução às problemáticas é apresentada onde, partindo de modelos UML, estes são transformados em *Symbolic Transition Systems* - STSs (o trabalho apresenta, em linguagem natural, um conjunto de regras de transformação). Por fim, os STSs do sistema servem de entrada para a ferramenta STG que deriva automaticamente os casos de teste para o sistema. Este trabalho mostra-se interessante por tratar de questões relevantes no contexto do teste de STRs em um ambiente de execução real e também por conseguir, usando UML, tratar de forma sucinta a questão da modelagem e teste de interrupções.

A Figura 6.2 apresenta resumidamente uma tabela comparativa entre os trabalhos acima citados e o por nós desenvolvido, considerando aspectos de teste de STRs.

Trabalho	Abordagem	Notação	Característica
Nosso Trabalho	MDT	UML + UTP RT + Modelos C	Proposta de uma arquitetura de teste independente e específica de plataforma para STRs. Utiliza o FreeRTOS como plataforma específica.
Hessel	MBT	-	Demonstra a aplicabilidade de MBT para STRs.
Li *et al.*	MBT	Time-Enriched Statecharts	Avaliação somente de propriedades de segurança.
Andrade & Machado	MBT	IOSTS	Apresenta um mecanismo para modelagem e teste de interrupções.
Andrade *et al.*	MBT	UML 2.0 + IOSTS	Tratam da geração de casos de teste para testar o FreeRTOS.

Figura 6.2: Tabela comparativa entre os trabalhos relacionados à testes STRs e o trabalho apresentado neste documento.

Modelagem e testes em STRs atualmente ainda são áreas recentes em pesquisa. Existem hoje poucas linhas de pesquisa preocupadas em fornecer soluções para problemas neste contexto e consequentemente ainda são poucas as ferramentas de apoio existentes neste sen-

tido. Devido a tal fato, estas ainda são áreas desafiadoras e com grandes possibilidades de evolução.

6.4 Considerações Finais do Capítulo

Este capítulo apresentou um conjunto de trabalhos que serviram de base para o desenvolvimento do trabalho apresentado neste documento. À medida que os trabalhos foram apresentados, quando necessário, foram realizadas as críticas relacionadas aos mesmos, bem como apontados os pontos que o distinguem do nosso trabalho.

Capítulo 7

Conclusões

O trabalho apresentado neste documento se propõe a preencher lacunas existentes na execução das atividade de modelagem e testes de STRs *soft* e reativos, fazendo uso de técnicas que valorizam o emprego de modelos, mais especificamente usando MDA e MDT. Para atingir tal objetivo, um conjunto de artefatos foi desenvolvids. No tocante à modelagem de STRs, foi desenvolvido um conjunto de diretrizes para auxiliar os projetistas na modelagem de STRs usando a notação UML. No total foram descritas 46 diretrizes reunindo, desde indicações de quais diagramas usar, até boas práticas para modelagem das características diferenciadas que esta classe de sistemas tem (*e.g.* eventos aperiódicos). Em segundo plano, ainda referente à modelagem de STRs, um perfil UML foi definido (o *Real Time Design Profile*), com o intuito de auxiliar a construção de modelos de *design* mais claros e facilitar a aplicação das diretrizes de modelagem. Para avaliação dos elementos produzidos nesta primeira etapa do trabalho, um conjunto de estudos de caso foram planejados e executados utilizando um conjunto de indivíduos que mesclava projetistas inexperientes e experientes, que fizeram uso das diretrizes propostas. Mediante análise dos resultados quantitativos coletados, a partir dos artefatos gerados pelos projetistas participantes, e dos resultados qualitativos capturados através das respostas a um questionário, foi possível atestar que as diretrizes e artefatos criados cumpriram seu papel de auxiliar projetistas de STRs, principalmente aqueles inexperientes, na tarefa de produzir modelos de *design* mais claros e com menos dificuldades.

No tocante à atividade de testes de STRs, também focando em STRs *soft* e reativos, foram desenvolvidos neste trabalho: i) um conjunto de extensões ao perfil de testes da UML (UTP). Estas extensões (UTP RT) foram criadas devido a necessidade de complementar o perfil para

as necessidades de teste dos STRs, bem como para melhor construção de arquiteturas de teste para este contexto; ii) um conjunto de regras de mapeamento informais para transformação dos elementos de uma arquitetura de testes no nível independente de plataforma (modelada usando UTP RT) para a plataforma do FreeRTOS, levando em consideração suas limitações; e iii) um suporte ferramental (ferramenta RTTAG) foi construído com o intuito de, usando o que apregoa MDT, conseguir gerar automaticamente arquiteturas de teste válidas para STRs, em diferentes níveis de abstração.

Outro artefato secundário produzido neste trabalho, e que acreditamos ser também uma contribuição relevante para a área de MDA/MDT foi a construção do metamodelo da linguagem C (Apêndice E). Este metamodelo foi construído com o objetivo de permitir o desenvolvimento da ferramenta RTTAG. Porém, qualquer ferramenta ou regra de transformação entre modelos (artefatos preponderantes no contexto de MDA) que tenha como ponto de partida ou de destino, modelos da linguagem C, poderá a partir de agora utilizar o metamodelo criado. Até então, atividades relacionadas a transformação de/para modelos C ficavam impossibilitadas, ou limitadas, pela falta de um metamodelo consistente e completo para esta importante linguagem.

7.1 Limitações

Como o trabalho desenvolvido, em sua maior parte, trata-se da proposições de novos, ou extensão de existentes, mecanismos para preenchimento de lacunas na execução das atividades de modelagem e teste de STRs, estas novidades necessitam ser melhor avaliadas e testadas sua eficiência. Para assim garantir que estes artefatos (diretrizes de modelagem, extensões de UTP, regras de mapeamento entre UTP e FreeRTOS) são realmente úteis e aplicáveis neste contexto. Devido limitações de tempo e recursos, foram executados apenas estudos de caso neste sentido. Estes estudos de caso nos forneceram indícios que os elementos propostos são realmente efetivos naquilo em que se propõem a atuar. Porém, um estudo experimental mais abrangente poderia demonstrar efetivamente os benefícios alcançados com sua utilização, bem como suas deficiências.

Outra limitação do nosso trabalho trata da ferramenta RTTAG. Até o momento, a ferramenta finaliza seu trabalho com composição dos modelos no nível PSTM. Como a ferra-

menta essencialmente propõauxiliar no cumprimento dos passos da abordagem MDT, a sua aplicação plena deveria finalizar com a geração do código C equivalente a arquitetura desenvolvida no nível PSTM. Esta transformação modelo-texto ainda não está implementada na ferramenta.

Dado que o número de sistemas exemplo que utilizamos nos estudos de caso é bastante restrita (apenas três sistemas foram usados), não podemos garantir que os resultados deste trabalho sejam aplicáveis para toda e qualquer subcategoria dos STRs reativos *soft*. Procuramos variar os domínios das aplicações utilizadas, mas, dado que a classe de STRs é bastante vasta, alguma de suas subcategorias pode não ter sido analisada. Sendo esta mais uma limitação do trabalho, a aplicação de mais estudos de caso, com a maior variabilidade de domínios possível, bem como a utilização no contexto real das técnicas propostas para o contexto de modelagem, poderia dissipar, ou diminuir, tal limitação.

7.2 Trabalhos Futuros

Com a conclusão deste trabalho, vislumbra-se um conjunto de possíveis trabalhos futuros para continuação do mesmo. Dentre eles é possível destacar:

- A realização de um processo de validação mais abrangente. Com relação às diretrizes de modelagem, um conjunto maior de especificações e/ou participantes poderiam ser usados. Para as extensões de UTP, arquiteturas de teste poderiam ser criadas com e sem o uso das extensões e posteriormente outros testadores poderiam aferir notas qualitativas comparativas para as mesmas. Com relação às regras de mapeamento entre os elemenentos de UTP e o FreeRTOS, arquiteturas de teste de outros STRs poderiam ser mapeadas para posterior análise.

- As diretrizes de modelagem poderiam ser revistas e/ou estendidas a fim de que também possam ser usadas no contexto de outros tipos de STRs (*e.g.* STRs *hard*). Uma análise do impacto do uso de UML no contexto destas outras classes de STRs também poderia ser interessante de ser realizada.

- A ferramenta RTTAG poderia ser estendida no sentido de também realizar a transformação final entre os modelos da arquitetura PSTM para código C propriamente.

Com esta extensão, a ferramenta estaria cumprindo todo o processo estabelecido pelas metodologias MDT, ou seja, conseguindo gerar, a partir dos modelos de *design*, arquiteturas de testes em diversos níveis de abstração, refiná-las até, por fim, chegar ao nível de código.

- Mapear os elementos de UTP para outras plataformas de STRs, além do FreeRTOS. Tal tarefa poderá ser realizado através de estenções de trabalhos da literatura que já propõem mapeamentos entre UTP e linguagens como TTCN-3. Tal trabalho porporcionará que também as nossas extensões ao UTP possam ser incorporadas a estes mapeamentos.

Bibliografia

[1] Metaobject facility (mofmof) 2.0 query/view/transformation specification. disponível em: <http://www.omg.org/mof/>. Technical report, Object Management Group.

[2] Uml superstructure, v2.1.1. disponível em: <http://www.omg.org/spec/uml/2.1.2/>. Technical report, OMG, 2007.

[3] E.L.G. Alves, A.Q. Macedo, W.L Andrade, P.D.L. Machado, and F Ramalho. Mapeando modelos de teste em utp para a plataforma freertos. *Proceedings of IV Brazilian Workshop on Systematic and Automated Software Testing (SAST 2010)*, pages 23–31, 2010.

[4] E.L.G. Alves, P.D.L. Machado, and F Ramalho. Uma abordagem integrada para desenvolvimento e teste dirigido por modelos. *Anais do 2nd Brazilian Workshop on Systematic and Automated Software Testing (SBES/SAST 2008)*, pages 74–83, 2008.

[5] E.L.G. Alves, F. Ramalho, and P.D.L. Machado. Automatic Generation of Built-in Contract Test Drivers. *A ser publicado*, 2011.

[6] W.L. Andrade and P.D.L. Machado. Interruption Testing of Reactive Systems. In *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19-21, 2009 Revised Selected Papers*, page 37. Springer, 2009.

[7] W.L. Andrade, P.D.L. Machado, E.L.G. Alves, and D.R. Almeida. Test Case Generation of Embedded Real-Time Systems with Interruptions for FreeRTOS. In *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19-21, 2009 Revised Selected Papers*, page 54. Springer, 2009.

[8] C. Atkinson, J. Bayer, and C. Bunse. *Component-based product line engineering with UML*. Addison-Wesley Professional, 2002.

[9] P. Baker, Z.R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams. *Model-driven testing: Using the UML testing profile*. Springer-Verlag New York Inc, 2007.

[10] DL Barbosa, WL Andrade, PDL Machado, and JCA Figueiredo. SPACES–Uma Ferramenta para Teste Funcional de Componentes. *XVIII Simpoósio Brasileiro de Engenharia de Software XI Sessão de Ferramentas*, page 55, 2004.

[11] D.L. Barbosa, H.S. Lima, P.D.L. Machado, and J.C. Figueiredo. Automating functional testing of components from UML specifications. *International Journal of Software Engineering and Knowledge Engineering*, 17(3):339–358, 2007.

[12] R. Barry. Real Time Application Design Using FreeRTOS in small embedded systems, Disponível em:< http://www.docstoc.com/docs/39822853/Real-Time-Application-Design-Using-FreeRTOS-in-small-embedded >, 2003.

[13] R. Barry. FreeRTOS-a free RTOS for small embedded real time systems, Disponível em:< http://www.freertos.org/ >, 2006.

[14] L.B. Becker. *Um método para abordar todo o ciclo de desenvolvimento de aplicações tempo real*. Tese de Doutorado - UFRGS-Instituto de Informatica - Programa de Pós-Graduação em Computação, 2003.

[15] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Readings in hardware/software co-design*, page 147, 2001.

[16] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[17] Z.R. Dai. Model-driven testing with UML 2.0. *Computer Science at Kent*, page 179, 2004.

[18] SR Dalai, A. Jain, N. Karunanithi, JM Leaton, CM Lott, GC Patton, and BM Horowitz. Model-based testing in practice. In *International Conference on Software Engineering ICSE*, pages 285–295, 1999.

[19] Disciplina de Engenharia de Software 2. Disponível em:< https://groups.google.com/group/esii-ccc >. *Acesso em*, 2011.

[20] G. De Jong. A UML-based design methodology for real-time and embedded sytems. In *Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society Washington, DC, USA, 2002.

[21] Disciplina de Sistemas de Informação 2. Disponível em:< http://www.dsc.ufcg.edu.br/ franklin/disciplinas/2011-1/SI2/>. *Acesso em*, 2011.

[22] S. Demathieu, F. Thomas, C. André, S. Gérard, and F. Terrier. First experiments using the UML profile for marte. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 50–57. IEEE Computer Society, 2008.

[23] B.P. Douglass. *Real-time UML: developing efficient objects for embedded systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[24] B.P. Douglass. *Real-time design patterns: robust scalable architecture for real-time systems*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[25] IDE Eclipse. Disponível em:< http://www. eclipse. org>. *Acesso em*, 19, 2005.

[26] I.K. El-Far and J.A. Whittaker. Model-based software testing. *Encyclopedia of Software Engineering*, 2001.

[27] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. Timed wp-method: Testing real-time systems. *IEEE Trans. Softw. Eng.*, 28(11):1023–1038, 2002.

[28] J. Ernits, R. Roo, J. Jacky, and M. Veanes. Model-Based Testing of Web Applications using NModel. *TESTCOM/FATES*, 2009.

[29] J.M. Farines, J.S. Fraga, and R.S. Oliveira. Sistemas de tempo real. *Florianópolis: Departamento de Automacao e Sistemas-Universidade Federal de Santa Catarina*, 2000.

[30] M. Fontoura, W. Pree, and B. Rumpe. *The UML profile for framework architectures*. Addison-Wesley Professional, 2002.

[31] N. Gehani and K. Ramamritham. Real-time Concurrent C: A language for programming dynamic real-time systems. *Real-Time Systems*, 3(4):377–405, 1991.

[32] LD Gowen. Specifying and verifying safety-critical software systems. In *Computer-Based Medical Systems, 1994., Proceedings 1994 IEEE Seventh Symposium on*, pages 235–240, 1994.

[33] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, 42(3):375–403, 2003.

[34] Object Management Group. Catalog of omg modeling and metadata specifications, disponível em:< http://www.omg.org/technology/documents/modeling_spec_catalog.htm >, 2006.

[35] J.J. Gutiérrez, M.J. Escalona, M. Mejías, and J. Torres. An approach to generate test cases from use cases. In *Proceedings of the 6th international conference on Web engineering*, pages 113–114. ACM, 2006.

[36] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 60–70. ACM New York, NY, USA, 2000.

[37] R. Heckel and M. Lohmann. Towards model-driven testing. *Electronic Notes in Theoretical Computer Science*, 82(6):33–43, 2003.

[38] A. Hessel. Model-based test case generation for real-time systems, disponível em: <http://www.hessel.nu/publications/ahlic.pdf>. 2007.

[39] A. Hessel, K.G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using uppaal. *Lecture Notes in Computer Science*, 4949:77, 2008.

[40] T. Husted and V. Massol. Junit in action. *Greenwich: Manning Publications*, 2003.

[41] AZ Javed, PA Strooper, and GN Watson. Automated generation of test cases using model-driven architecture. In *Proceedings of the Second International Workshop on Automation of Software Test*, page 3. IEEE Computer Society, 2007.

[42] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.

[43] S. Kent. Model driven engineering. *Lecture notes in computer science*, pages 286–298, 2002.

[44] B. Kitchenham. Procedures for performing systematic reviews. *Keele, United Kingdom, Keele University*, 33, 2004.

[45] A. Kleppe, J. Warmer, and W. Bast. MDA Explained. The Practice and Promise of the Model Driven Architecture, 2003.

[46] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[47] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *In 11th International SPIN Workshop on Model Checking of Software (SPIN04), volume 2989 of LNCS*, pages 109–126. Springer, 2004.

[48] M. Lawley and J. Steel. Practical declarative model transformation with tefkat. In *Satellite Events at the MoDELS 2005 Conference*, pages 139–150. Springer, 2006.

[49] S. Li, J. Wang, W. Dong, and Z.C. Qi. Property-Oriented Testing of Real-Time Systems. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, page 365. IEEE Computer Society, 2004.

[50] H.S. Lima, F. Ramalho, P.D.L. Machado, and E.L. Galdino. Automatic generation of platform independent built-in contract testers. *Anais do Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS 2007), Campinas*, page 14, 2007.

[51] J.W.S. Liu. *Real-time systems*. Prentice Hall, 2000.

[52] A. Lyons. UML for real-time overview. *Objectime Ltd*, 1998.

[53] AQ Macedo, WL Andrade, D. Rodrigues, and P. Machado. Automating Test Case Execution for Real-Time Embedded Systems. *on Testing Software and Systems: Short Papers*, page 37, 2010.

[54] D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems (TOCS)*, 13(4):365–398, 1995.

[55] C. Mingsong, Q. Xiaokang, and L. Xuandong. Automatic test case generation for UML activity diagrams. In *Proceedings of the 2006 international workshop on Automation of software test*, page 8. ACM, 2006.

[56] OMG MOF. OMG Meta Object Facility Specification v1. 4, Disponível em: < http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF>, 2002.

[57] D. Neto, C. Arilo, R. Subramanyan, M. Vieira, and G.H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 31–36. ACM, 2007.

[58] S.C. Nogueira, E.G. Cartaxo, D.G. Torres, E. Aranha, and R. Marques. Model based test generation: A case study. In *Workshop on Systematic and Automated Software Testing. Workshop on Systematic and Automated Software Testing*, 2007.

[59] J. Oldevik. MOFScript User Guide. Disponível em: <http://www.modelbased.net/mofscript/docs/MOFScript-User-Guide.pdf>, 2006.

[60] QVT OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Disponível em: <http://www.omg.org/spec/QVT/>, 2008.

[61] A.M.S. Orozco, K. Oliveira, F. Oliveira, and A.F. Zorzo. Derivação de Casos de Testes Funcionais: Uma Abordagem Baseada em Modelos UML. *Revista Eletrônica de Sistemas de Informação ISSN 1677-3071*, (1), 2009.

[62] I. Perez, E. Martins, and J.E. Viégas. Uso de Modelos da UML em Testes de Componentes. In *Proceedings of VIII Workshop de Teste e Tolerância a Falhas (WTF 2007). SBRC, Belém–PA*, 2007.

[63] JL Peterson. Petri Net Theory and the Modeling of Systems. *Prenctice-Hall, INC., Englewood Cliffs, NJ 07632, 1981, 290*, 1981.

[64] B. Rumpe. Model-based testing of object-oriented systems. *Lecture notes in computer science*, pages 380–402, 2003.

[65] S. Schneider. *Concurrent and real-time systems: the CSP approach*. Wiley, 2000.

[66] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. *Lecture Notes in Computer Science*, pages 250–260, 1998.

[67] I. Sommerville. *Engenharia de software, 8a edição*. Prentice-Hall, 2007.

[68] AtlanMod team. Disponível em:<http://www.emn.fr/z-info/atlanmod/index.php/Main_Page>. *Acesso em*, 2011.

[69] RJ Van Glabbeek. Labelled Transition Systems. *language*, 50:1.

[70] R. Van Solingen and E. Berghout. *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*. McGraw-Hill Cambridge, UK, 1999.

[71] A. S. Vieira. *Identificação de Diretrizes para a Construção de Meta-modelos na Infra-estrutura de MDA*. Dissertação de Mestrado - Universidade Federal de Campina Grande (UFCG), Campina Grande, 2010.

[72] J. Warmer and A. Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[73] J. Zander, Z.R. Dai, I. Schieferdecker, and G. Din. From U2TP models to executable tests with TTCN-3-an approach to model driven testing. *Lecture Notes in Computer Science*, 3502:289–303, 2005.

Apêndice A

Especificações fornecidas aos participantes do estudo experimental referente a avaliação das diretrizes de modelagem de STRs com UML

- **Aplicação Arquivo TV**

  Descrição: A aplicação a ser desenvolvida, "Arquivo TV", permitirá ao telespectador assistir vídeos disponibilizados pela emissora, onde o mesmo pode interagir com diversas funcionalidades. A principal funcionalidade disponível será o agendamento em uma determinada data e horário. Ao iniciar-se, o aplicativo verificará se existe algum agendamento já efetuado, através da consulta de um arquivo de configuração em XML, também verificará se o telespectador optou por terminar de assistir a um vídeo já iniciado anteriormente. Caso algum dos vídeos agendados for excluído do servidor ou a data do agendamento for expirada, o telespectador será informado e o agendamento excluído. O telespectador poderá navegar pela aplicação através de um menu principal, podendo requisitar ajuda para navegação, pausar, consultar informações, controlar o áudio, alternar e adiantar ou retroceder o vídeo que está sendo exibido. Os vídeos são armazenados de acordo com sua classificação em pacotes em um diretório do servidor da emissora, onde os telespectadores poderão acessá-los através de um sub-menu para assistirem de imediato ou efetuar um agendamento. No agendamento, o telespectador

escolhe o pacote e posteriormente o vídeo pertencente ao mesmo, após isso, o telespectador define o horário que deseja agendar, sendo que esta informação do horário fica gravada em um arquivo XML. Quando o vídeo agendado estiver para começar o telespectador será avisado por meio de mensagem na tela, perguntando se o mesmo deseja assisti-lo ou continuar vendo a programação corrente. Em caso de resposta negativa, o agendamento é cancelado; em caso de resposta positiva, o vídeo agendado será iniciado, interrompendo a programação atual. Finalizando o vídeo agendado, o vídeo que estava sendo executado anteriormente (o interrompido) prossegue executando a partir do momento exato ao qual foi interrompido. O Telespectador poderá agendar no máximo dez vídeos e também poderá excluir qualquer agendamento efetuado.

O administrador do servidor da emissora poderá adicionar ou remover pacotes e vídeos sem que o código fonte do aplicativo sofra alteração. Para adicioná-los ou removê-los, basta o administrador acessar o sistema administrativo do aplicativo ArquivoTV e fazer as devidas alterações, o administrador também poderá incluir e remover as informações sobre os vídeos adicionados. Todas as informações existentes serão armazenadas em arquivos configurados em XML.

Requisitos: Figuras A.1, A.2, A.3, A.4, A.5.

Requisitos Não Funcionais				
Nome	*Restrição*	*Categoria*	*Desejável*	*Permanente*
NF1: Definir cores usadas no aplicativo	Somente serão usadas as cores vermelha, branca, preta, amarela e azul.	Interface	(X)	( )
NF2: Definir nome e logotipo da aplicação	O aplicativo se chamará "Arquivo TV" com o logotipo baseado nas cores da aplicação.	Interface	( )	(X)
NF3: Utilizar o *Middleware* MHP	O *middleware* instalado no *set-top box* deverá ser o MHP	Software	( )	(X)

Figura A.1: Requisitos da aplicação Arquivo TV (Parte 1).

Requisitos Funcionais	
F1: Exibir Vídeos Sequencialmente	Oculto (X)
Descrição: Os vídeos serão exibidos em sequência caso o telespectador não interrompa a programação corrente. Havendo interrupção, a sequência continuará a partir do vídeo escolhido pelo telespectador.	
F2: Mostrar/ Ocultar Menu Principal	Evidente (X)
Descrição: O Menu Principal aparecerá na tela da aplicação toda vez que for chamado, onde o telespectador pode navegar pelos seus sub-menus. O telespectador também poderá ocultá-lo quando ele não estiver sendo usado. Restrição: O aparecimento e/ou desaparecimento do Menu Principal deve ser realizado em no máximo 1seg a partir da requisição do usuário.	
F3: Mostrar/ Ocultar Menu Agenda	Evidente (X)
Descrição: O Menu de Ajuda aparecerá na tela toda que o telespectador chamá-lo. O telespectador também poderá ocultá-lo. Restrição: O aparecimento e/ou desaparecimento do Menu Agenda deve ser realizado em no máximo 1seg a partir da requisição do usuário.	

Figura A.2: Requisitos da aplicação Arquivo TV (Parte 2).

F4: Pausar Vídeo	Evidente (X)
Descrição: O telespectador poderá pausar a execução do vídeo corrente e continuar a execução do vídeo.	
F5: Mostrar Informações dos Vídeos	Evidente (X)
Descrição: O telespectador poderá requisitar as informações do vídeo corrente. Restrição: O aparecimento e/ou desaparecimento das informações do vídeo deve ser realizado em no máximo 1seg a partir da requisição do usuário.	
F6: Mostrar Listagem de Pacotes	Evidente (X)
Descrição: O telespectador poderá listar os pacotes de vídeos existentes no servidor através do sub-menu Pacotes.	
F7: Mostrar Listagem de Vídeos	Evidente (X)
Descrição: O telespectador poderá listar os vídeos existentes no servidor através do sub-menu Vídeos, onde é mostrado os vídeos do pacote corrente.	
F8: Agendar Vídeos	Evidente (X)
Descrição: O telespectador poderá agendar um vídeo através do sub-menu Agendar Vídeos, informando o pacote, vídeo e o horário que deseja assistir. Restrição: O limite máximo de agendamentos é 10.	

Figura A.3: Requisitos da aplicação Arquivo TV (Parte 3).

- **Aplicação Alarme de Intruso**

  Descrição: O Alarme de Intruso é um sistema de software capaz de controlar um

F9: Encerrar Aplicativo	Evidente (X)
Descrição: O telespectador poderá encerrar o aplicativo através do sub-menu Sair.	
F10: Adicionar Vídeos e ou Pacotes	Evidente (X)
Descrição: O Administrador do servidor poderá adicionar pacotes e ou vídeos através do sistema administrativo do aplicativo ArquivoTV.	
F11: Remover Vídeos e ou Pacotes	Evidente (X)
Descrição: O Administrador do servidor poderá remover pacotes e ou vídeos através do sistema administrativo do aplicativo ArquivoTV, onde as suas respectivas informações também serão removidas.	
F12: Adicionar Informações	Evidente (X)
Descrição: O Administrador do servidor poderá adicionar informações sobre os vídeos através do sistema administrativo do aplicativo ArquivoTV quando os criar.	
F13: Alterar Áudio	Evidente (X)
Descrição: O telespectador poderá aumentar ou diminuir o áudio através das teclas '+' e '-' do controle remoto do *set top-box*. Restrição: A alteração do volume deve ser realizada em no máximo 50ms.	
F14: Gravar Agendamento	Oculto (X)
Descrição: O sistema irá gravar em um arquivo XML as informações sobre o agendamento efetuado pelo telespectador.	

Figura A.4: Requisitos da aplicação Arquivo TV (Parte 4).

F15: Ler Agendamento	Oculto (X)
Descrição: O sistema irá ler o arquivo XML que contém as informações sobre os agendamentos toda vez que for iniciado.	
F16: Cancelar Agendamento	Evidente (X) Oculto (X)
Descrição: O sistema irá cancelar o agendamento do aplicativo se o telespectador negar a execução do vídeo na data e horário agendado. O agendamento também é cancelado caso a data marcada para o agendamento do vídeo estiver expirada, ou se o usuário excluí-lo através do aplicativo.	
F17: Exibir Vídeo Interrompido	Evidente (X)
Descrição: O telespectador poderá interromper a execução do vídeo ao encerrar o aplicativo e desejar continuar assistindo o vídeo no momento em que foi interrompido quando iniciar o aplicativo novamente.	

Figura A.5: Requisitos da aplicação Arquivo TV (Parte 5).

sistema de alarme contra intrusos em prédios comerciais. Ele utiliza vários tipos diferentes de sensores, incluindo detectores de movimento em salas individuais, sensores de janelas nas janelas do térreo que detectam quando uma janela é quebrada, e sensores de porta que detectam a abertura das portas do corredor. Trata-se de um sistema

de tempo real *soft*, pois seus sensores não necessitam uma alta precisão na detecção de eventos.

Quando um sensor detecta a presença de um intruso, o sistema automaticamente aciona polícia local por telefone e, usando um sintetizador de voz, informa o local do alarme. As luzes são acesas na sala onde o sensor foi acionado e dispara um alarme sonoro. O sistema de sensores é alimentado pela energia elétrica convencional, mas é equipado com uma bateria para o caso de falha de energia. A falha de energia é detectada por elemento monitor de circuito.

Há dois tipos de estímulos a serem processados:

1. Queda de energia: Gerado pelo monitor de circuito. A resposta a esse estímulo é a mudança de alimentação para uma bateria de reserva pela sinalização a um dispositivo eletrônico de comutação de potência.
2. Alarme de intruso: Estímulo gerado por um dos sensores do sistema. A resposta a esse estímulo é a identificação do número da sala ao qual o sensor foi ativado, realização de uma chamada a polícia, iniciar o sintetizador de voz para controlar a chamada, acionar o alarme sonoro de intruso e ligar as luzes do prédio na área.

Existirão três tipos de sensores que devem ser analisados periodicamente, cada um com um processamento independente. Existirá um sistema dirigido por interrupções para manusear a falha de queda de energia e para mudança de alimentação.

Requisitos: Figuras A.6, A.7, A.8, A.9.

- **Aplicação Celular Simples**

  Descrição: O Celular Simples trata-se de um aparelho celular capaz de realizar/receber chamadas telefônicas e enviar/receber mensagens de texto. Nele, é possível gerenciar os contatos telefônicos através de sua agenda telefônica. Nesta agenda é possível armazenar, editar e apagar contatos. Além de poder armazenar contatos na própria agenda telefônica, também é possível armazenar o número do remetente de uma mensagem ou armazenar o número de um telefone digitado na tela inicial. A qualquer momento é possível cancelar o armazenamento do novo contato.

Requisitos Funcionais	
F1: Detecção de movimento	Oculto(X)
Descrição: Cada sensor de movimento deverá verificar periodicamente a presença de algum movimento e avisar ao monitor do edifício se algum movimento foi detectado. Restrição: Cada sensor de movimento fará duas verificações de movimento por segundo.	
F2: Detecção de abertura de porta	Oculto(X)
Descrição: Cada sensor de porta deverá verificar periodicamente a situação de sua porta (aberta ou fechada) e avisar ao monitor do edifício. Restrição: Cada sensor de porta fará duas verificações da situação de sua porta por segundo.	
F3: Detecção de janela quebrada	Oculto(X)
Descrição: Cada sensor de janela deverá verificar periodicamente a situação de sua janela (quebrada ou não quebrada) e avisar ao monitor do edifício. Restrição: Cada sensor de janela fará duas verificações da situação de sua janela por segundo.	
F4: Queda de energia	Oculto(X)
Descrição: Ao ocorrer uma queda de energia o monitor de energia deverá: i) interromper a execução do sistema de alarmes; ii) sinalizar o dispositivo eletrônico de comutação de potência para mudança de alimentação da alimentação da energia convencional para a bateria de reserva; e iii) reativar o sistema de alarmes. Restrição: O processo de troca de alimentação deve ser realizada em no máximo 100 ms.	

Figura A.6: Requisitos da aplicação Alarme de Intruso (Parte 1).

F5: Mudança de alimentação para bateria de reserva	Oculto(X)
Descrição: Uma vez que o monitor de energia tenha solicitada a mudança de alimentação da energia convencional para bateria de reserva o dispositivo eletrônico de comutação de potência deve mudar a alimentação da energia imediatamente. Restrição: O processo de troca de alimentação deve ser realizado em no máximo 50ms.	
F6: Retomada da energia convencional	Oculto(X)
Descrição: Após a energia elétrica convencional estar novamente disponível o monitor de energia deverá: i) interromper a execução do sistema de alarmes; ii) sinalizar o dispositivo eletrônico de comutação de potência para mudança de alimentação da bateria de reserva para a energia convencional; e iii) reativar o sistema de alarmes. Restrição: O intervalo de tempo decorrente do momento da interrupção da execução do sistema de alarmes até a reativação do sistema de alarmes não deverá ultrapassar 100ms.	
F7: Mudança de alimentação para energia convencional	Oculto(X)
Descrição: Uma vez que o monitor de energia tenha solicitada a mudança de alimentação da bateria de reserva para a energia convencional o dispositivo eletrônico de comutação de potência deve mudar a alimentação da energia imediatamente. Restrição: O processo de troca de alimentação deve ser realizado em no máximo 50ms.	

Figura A.7: Requisitos da aplicação Alarme de Intruso (Parte 2).

Para realizar uma chamada o usuário pode tanto digitar o número na tela inicial ao qual deseja realizar a chamada quanto escolher um contato na agenda telefônica para realizar a ligação. Outra forma de realizar chamada é a partir de uma mensagem de

F8: Chamada ao responsável pelo prédio	Evidente(X)
Descrição: Quando a bateria atingir 20% da sua carga máxima o sistema de alarmes deve realizar uma chamada telefônica para o responsável pelo prédio e uma mensagem de voz sintetizada deve ser reproduzida alertando quanto ao nível baixo da bateria. Restrição: A chamada ao responsável pelo prédio deve ser realizada 2 segundos após o nível da bateria de reserva ter atingido 20% de sua carga máxima.	
F9: Alarme sonoro	Evidente(X)
Descrição: Após ter sido detectada a presença de um intruso por qualquer um dos sensores um alarme sonoro deverá ser disparado. Restrição: O alarme sonoro deverá ser disparado 0,5 segundo após a detecção de um intruso.	
F10: Controle de luzes	Evidente(X)
Descrição: Após identificada a sala a qual foi invadida, as luzes da área em torno do local invadido deverão ser acesas, ou seja, as luzes da sala e as luzes próximas ao sensor ativado pelo invasor. Restrição: As luzes deverão ser acesas 0,5 segundo após a sala ter sido identificada.	

Figura A.8: Requisitos da aplicação Alarme de Intruso (Parte 3).

F11: Sintetizar voz para alerta de intruso	Oculto(X)
Descrição: Após ter sido detectada a presença de um intruso por qualquer um dos sensores uma mensagem de voz deve ser sintetizada alertando a presença de um intruso e informando a sala que foi invadida. Restrição: A mensagem de voz sintetizada deve estar disponível em 4 segundos após um intruso ter sido detectado por um sensor.	
F12: Chamada à polícia para alerta de intruso	Evidente(X)
Descrição: Após ter sido detectada a presença de um intruso por qualquer um dos sensores uma chamada telefônica para a polícia deve ser realizada e uma mensagem de voz sintetizada deve ser reproduzida alertando a presença de um intruso e informando a sala que foi invadida. Restrição: A chamada à polícia deve ser realizada 2 segundos após um intruso ter sido detectado por um sensor.	

Figura A.9: Requisitos da aplicação Alarme de Intruso (Parte 4).

texto. Ao ler uma mensagem de texto o usuário poderá escolher a opção de realizar uma chamada para o remetente. A qualquer momento a ligação poderá ser cancelada, inclusive antes que a conversa por voz seja estabelecida. A qualquer momento pode ocorrer a chegada de uma chamada. Quando ocorre a chegada de uma chamada o usuário pode optar por atender ou recusar a ligação.

Para enviar uma mensagem o usuário pode optar por responder uma mensagem previamente recebida ou por enviar através da central de mensagens do aparelho. Para responder uma mensagem o usuário pode fazê-lo abrindo a mensagem e escolhendo a opção de responder. Para enviar através da central de mensagens o usuário deve ir

até a central de mensagens e escolher a opção de criar mensagem, posteriormente, o usuário deve adicionar um ou mais destinatários para o qual deseja enviar a mensagem. Para adicionar um destinatário o usuário deve apertar o botão de adicionar destinatário e em seguida escolher a opção de digitar um número telefônico ou escolher o contato na lista de contatos. Depois de digitado o(s) número(s) telefônico(s) e/ou escolhido o(s) contato(s) da lista de contatos, o usuário deve então digitar a mensagem de texto que será enviada e então escolher a opção de enviar mensagem. A qualquer momento o usuário pode sair da opção de criar mensagem. Caso a mensagem de texto esteja vazia, o aparelho deve sair da opção de criar mensagem sem dar qualquer aviso ao usuário. Caso a mensagem de teste não esteja vazia o usuário deverá ser questionado se a mensagem de texto deve ser salva nos rascunhos ou não. A qualquer momento pode ocorrer a chegada de uma mensagem. Quando uma mensagem chega o celular deve exibir uma mensagem perguntando se o usuário deseja ler a mensagem ou não.

Requisitos: Figuras A.10,A.11, A.12, A.13.

Requisitos Funcionais	
F1: Armazenar um novo contato	Evidente(X)
Descrição: O celular deve permitir que um novo contato possa ser armazenado através da agenda telefônica. Restrições: 1. O número máximo de contatos suportado pelo aparelho é de 1000 contatos. Após esse limite nenhum novo contato poderá ser mais armazenado. Caso o usuário tente adicionar um novo contato e a memória estiver cheia o celular deverá apresentar uma mensagem de memória cheia que ficará presente na tela durante 2 segundos. 2. Todos os campos obrigatórios devem ser preenchidos. Se pelo menos um campo obrigatório não for preenchido o celular deverá apresentar uma mensagem de erro durante 2 segundos na tela quando o usuário tentar armazenar o novo contato.	
F2: Editar contato	Evidente(X)
Descrição: O celular deve permitir que o número de remetente de uma mensagem possa ser armazenado como um novo contato. Restrições: 1. O número máximo de contatos suportado pelo aparelho é de 1000 contatos. Após esse limite nenhum novo contato poderá ser mais armazenado. Caso o usuário tente adicionar um novo contato e a memória estiver cheia o celular deverá apresentar uma mensagem de memória cheia que ficará presente na tela durante 2 segundos. 2. Todos os campos obrigatórios devem ser preenchidos. Se pelo menos um campo obrigatório não for preenchido o celular deverá apresentar uma mensagem de erro durante 2 segundos na tela quando o usuário tentar armazenar o novo contato.	

Figura A.10: Requisitos da aplicação Celular Simples (Parte 1).

F3: Armazenar um novo contato a partir de número digitado na tela inicial	Evidente(X)
Descrição: O celular deve permitir que um novo contato possa ser armazenado após digitado um número telefônico na tela inicial. Restrições: 1. O número máximo de contatos suportado pelo aparelho é de 1000 contatos. Após esse limite nenhum novo contato poderá ser mais armazenado. Caso o usuário tente adicionar um novo contato e a memória estiver cheia o celular deverá apresentar uma mensagem de memória cheia que ficará presente na tela durante 2 segundos. 2. Todos os campos obrigatórios devem ser preenchidos. Se pelo menos um campo obrigatório não for preenchido o celular deverá apresentar uma mensagem de erro durante 2 segundos na tela quando o usuário tentar armazenar o novo contato.	
F4: Editar contato	Evidente(X)
Descrição: O celular deve permitir que um contato já existente possa ser alterado. Restrições: 1. Nenhum dos campos obrigatórios pode ficar vazio ao término da edição do contato. Se pelo menos um campo obrigatório for vazio o celular deverá apresentar uma mensagem de erro durante 2 segundos na tela quando o usuário tentar finalizar a operação de edição.	

Figura A.11: Requisitos da aplicação Celular Simples (Parte 2).

F5: Apagar contato	Evidente(X)
Descrição: O celular deve permitir que um contato seja excluído da agenda telefônica.	
F6: Realizar chamada via agenda telefônica	Evidente(X)
Descrição: O celular deve permitir que o usuário possa realizar chamadas através da sua agenda telefônica.	
F7: Realizar chamada a partir de uma mensagem de texto	Evidente(X)
Descrição: O celular deve permitir que o usuário possa realizar chamadas para o remetente de uma mensagem de texto a partir da própria mensagem de texto.	
F8: Realizar chamada digitando um número telefônico na tela inicial	Evidente(X)
Descrição: O celular deve permitir que o usuário possa realizar chamadas digitando um número telefônico na tela inicial.	
F9: Receber chamada	Evidente(X)
Descrição: O celular deve estar pronto para receber uma chegada de chamada a qualquer instante. Quando ocorre a chegada de uma chamada o usuário pode optar por atender ou recusar a ligação. Restrições: 1. Ao término da chamada, seja ela aceita ou não pelo usuário, o celular deve retornar ao mesmo ponto onde estava antes de ocorrer à chegada da chamada em 50ms sem mudança no seu contexto. 2. Caso o usuário não atenda ou rejeite a chamada em no máximo 30 segundos, o celular optará automaticamente por rejeitar a chamada.	

Figura A.12: Requisitos da aplicação Celular Simples (Parte 3).

F10: Envio de mensagens	Evidente(X)
Descrição: O celular deve permitir que o usuário possa enviar mensagens de texto para um ou mais contatos. Restrição: 1. Caso o usuário não digite nenhuma mensagem de texto, uma mensagem de erro deve ser exibida na tela durante 2 segundos no momento em que o usuário tentar enviar a mensagem vazia. Caso contrário, a mensagem deverá ser enviada normalmente, posteriormente o celular deverá exibir uma mensagem na tela de envio bem sucedido durante 2 segundos e então retornar a tela de central de mensagens em 50ms. 2. A mensagem de texto tem um tamanho máximo de 160 caracteres. Caso a mensagem digitada pelo usuário exceda esse tamanho então cada 160 caracteres excedentes será uma mensagem a mais enviada.	
F11: Receber mensagem	Evidente(X)
Descrição: O celular deve estar pronto para receber uma chegada de mensagem a qualquer instante. Quando ocorre a chegada de uma mensagem o usuário pode optar por ler ou não ler a mensagem naquele momento. Restrições: 1. Caso o usuário opte por ler a mensagem, a mensagem deve ser exibida e quando o usuário sair da opção de leitura de mensagem o celular deve encaminhar a mensagem para a caixa de entrada como lida e retornar ao ponto onde estava sem mudança no seu contexto. 2. Caso o usuário opte por não ler a mensagem, esta deverá ser encaminhada para a caixa de entrada como não lida e retornar ao ponto onde estava sem mudança no seu contexto.	

Figura A.13: Requisitos da aplicação Celular Simples (Parte 4).

Apêndice B

Questionário aplicado para realização da avaliação subjetiva da aplicação das diretrizes de modelagem de STRs com UML

Tabela B.1: *Questionário fornecido durante o estudo*

Questionário
Integrantes da Equipe:
Sistema:
1) Qual seu grau de experiência com a o uso de UML2: ( ) Iniciante ( ) Intermediário ( ) Experiente
2) Qual seu grau de experiência no projeto/desenvolvimento de sistemas de tempo real? ( ) Iniciante ( ) Intermediário ( ) Experiente
3) Já cursou a disciplina de Sistemas de Informação II? (Sim/Não/Cursando)
4) Qual nota (0 a 10) vocês dariam para os modelos construídos na fase 1 do projeto? Leve em consideração aspectos como clareza, legibilidade, cobertura de requisitos, facilidade de implementação, etc.
5) Qual nota (0 a 10) vocês dariam para os modelos construídos na fase 2 do projeto? Leve em consideração aspectos como clareza, legibilidade, cobertura de requisitos, facilidade de implementação, etc.
6) Na sua opinião, todos os requisitos do sistema foram claramente modelados com o uso das diretrizes de modelagem? (Sim/Não)
(Caso a resposta da questão anterior seja Não) 7) Identifique os requisitos que você considera que não foram claramente modelados: 8) Justifique o porquê da impossibilidade, e/ou dificuldade, de modelagem dos requisitos citados na questão 7.
9) Quais foram suas principais dificuldades em cumprir esta etapa do projeto?
10) Na sua opinião, o sistema poderá ser implementado mais facilmente após a atividade de modelagem realizada nesta etapa? Quais fatores lhe direcionam a esta resposta?
11) Levando em consideração as seguintes características: esforço de modelagem, qualidade dos artefatos gerados e facilidade de uso; como vocês classificariam o uso das diretrizes de modelagem no contexto de modelagem de RTSs? ( ) Inútil ( ) Pouco Útil ( ) Útil ( ) Bastante Útil
12) Caso deseje, deixe algum comentário pessoal a respeito de como foi sua experiência na realização desta etapa do projeto ou sugestão de mudança nas diretrizes ou tutorial.

Apêndice C

Dados coletados e cálculo das métricas para os estudos de caso referentes a avaliação das diretrizes de modelagem de STRs com UML

Dados obtidos a partir da realização dos estudos:

- Grupo 1 (Sistema - Arquivo TV):

  Dados para cálculo da M1:

  - $R_t = 20$
  - $R_d = 20$
  - $PRC = \frac{20}{20} = 1 = 100\%$

  Dados para cálculo da M2:

  - $Q_{di} = 31$
  - $Q_{dd} = 15$
  - $Q_{ui} = 29$
  - $Q_{ud} = 7$
  - $PDNA_i = 1 - \frac{29}{31} = 0.06$

- $PDNA_d = 1 - \frac{7}{15} = 0.53$

Dados para cálculo da M3:

- $NS = 7,5$
- $ND = 9$

- Grupo 2 (Sistema - Arquivo TV):

Dados para cálculo da M1:

- $R_t = 20$
- $R_d = 17$
- $PCR = \frac{17}{20} = 0.85 = 85\%$

Dados para cálculo da M2:

- $Q_{di} = 31$
- $Q_{dd} = 15$
- $Q_{ui} = 24$
- $Q_{ud} = 7$
- $PDNA_i = 1 - \frac{24}{31} = 0.22$
- $PDNA_d = 1 - \frac{7}{15} = 0.53$

Dados para cálculo da M3:

- $NS = 9$
- $ND = 10$

- Grupo 3 (Celular Simples):

Dados para cálculo da M1:

- $R_t = 11$
- $R_d = 11$
- $PCR = \frac{11}{11} = 1 = 100\%$

Dados para cálculo da M2:

- $Q_{di} = 31$
- $Q_{dd} = 15$
- $Q_{ui} = 28$
- $Q_{ud} = 6$
- $PDNA_i = 1 - \frac{28}{31} = 0.09$
- $PDNA_d = 1 - \frac{6}{15} = 0.6$

Dados para cálculo da M3:

- $NS = 9$
- $ND = 7$

- Grupo 4 (Sistema - Alarme de Intrusos):

  Dados para cálculo da M1:

  - $R_t = 12$
  - $R_d = 12$
  - $PCR = \frac{12}{12} = 1 = 100\%$

  Dados para cálculo da M2:

  - $Q_{di} = 31$
  - $Q_{dd} = 15$
  - $Q_{ui} = 28$
  - $Q_{ud} = 11$
  - $PDNA_i = 1 - \frac{28}{31} = 0.09$
  - $PDNA_d = 1 - \frac{11}{15} = 0.26$

  Dados para cálculo da M3:

  - $NS = 8$

  - $ND = 9$

- Grupo 5 (Sistema - Arquivo TV):

  Dados para cálculo da M1:

  - $R_t = 20$
  - $R_d = 20$
  - $PCR = \frac{20}{20} = 1 = 100\%$

  Dados para cálculo da M2:

  - $Q_{di} = 31$
  - $Q_{dd} = 15$
  - $Q_{ui} = 28$
  - $Q_{ud} = 9$
  - $PDNA_i = 1 - \frac{28}{31} = 0.09$
  - $PDNA_d = 1 - \frac{9}{15} = 0.4$

  Dados para cálculo da M3:

  - $NS = 7$
  - $ND = 8.5$

- Grupo 6 (Sistema - Arquivo TV):

  Dados para cálculo da M1:

  - $R_t = 20$
  - $R_d = 20$
  - $PCR = \frac{20}{20} = 1 = 100\%$

  Dados para cálculo da M2:

  - $Q_{di} = 31$
  - $Q_{dd} = 15$

- $Q_{ui} = 24$
- $Q_{ud} = 9$
- $PDNA_i = 1 - \frac{24}{31} = 0.22$
- $PDNA_d = 1 - \frac{9}{15} = 0.4$

Dados para cálculo da M3:

- $NS = 8$
- $ND = 8.5$

Cálculo das métricas

- **M1**

  MTPRC: Média da PRC Total

  $MTPRC = \frac{1+0.85+1+1+1+1}{6} = 0.975$

  Variância $(_{MTPRC}) = 0.00375$

  Intervalo de confiança com nível de 95%$= [0.9107; 1.0392]$

- **M2**

  $MTPDNA_i$: Média Total das PDNA independentes

  $MTPDNA_i = \frac{0.06+0.22+0.09+0.09+0.09+0.22}{6} = 0,128$

  Variância $(_{MTPDNAi}) = 0.00517$

  Intervalo de confiança com nível de 95%$= [0.0528; 0.2038]$

- **M3**

  $MTPDNA_d$: Média Total das PDNA dependentes

  $MTPDNA_d = \frac{0.53+0.53+0.06+0.26+0.4+0.4}{6} = 0,363$

  Variância $(_{MTPDNAd}) = 0.0321$

  Intervalo de confiança com nível de 95% = [0.1750; 0.5516]

- **M4**

  DMQ: Diferença das médias de qualidade

  MNQSD = $\frac{7.5+9+9+8+7+8}{6} = 8.083$

  Variância $(_{MNQSD}) = 0.6416$

Intervalo de confiança com nível 95%$= [7.242; 8.9239]$

MNQCD $= \frac{9+10+7+9+8.5+8.5}{6} = 8.666$

Variância $(_{MNQSD}) = 0.9666$

Intervalo de confiança com nível 95%$= [7.6348; 9.6984]$

DMQ $= MNQCD - MNQSD = 0.583$

Apêndice D

Diagramas de *Design* do *Sistema de Alarmes*

Para execução do estudo de caso descrito na Seção 4.5, um conjunto de diagramas UML (seguindo as diretrizes de modelagem) foram criados segundo a especificação do *Sistema de Alarmes*, presente no Apêndice A. Neste apêndice, estes diagramas serão apresentados na seguinte ordem: Use Cases, Componentes, Classes, Estruturas Compostas, Máquinas de Estado, Overview, Sequência.

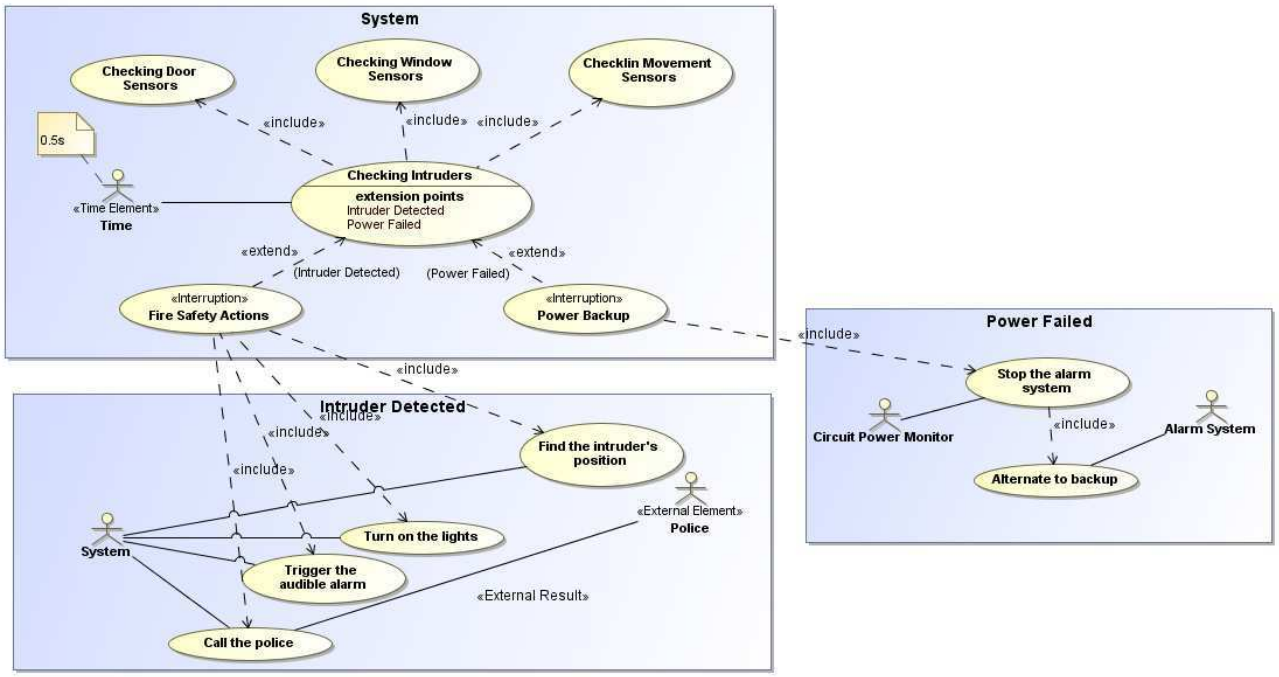

Figura D.1: Diagramas de Use Case do *Sistema de Alarmes*.

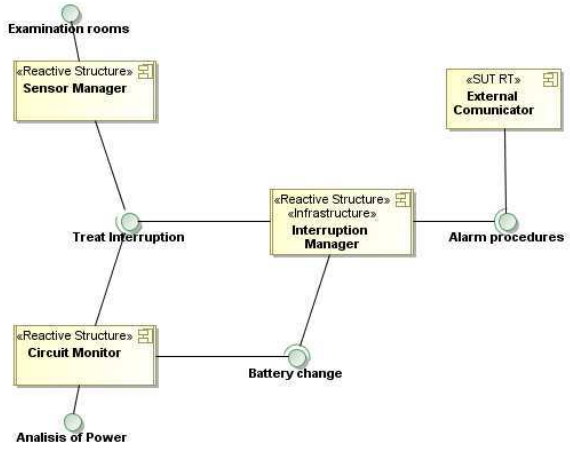

Figura D.2: Diagrama de Componentes do *Sistema de Alarmes*.

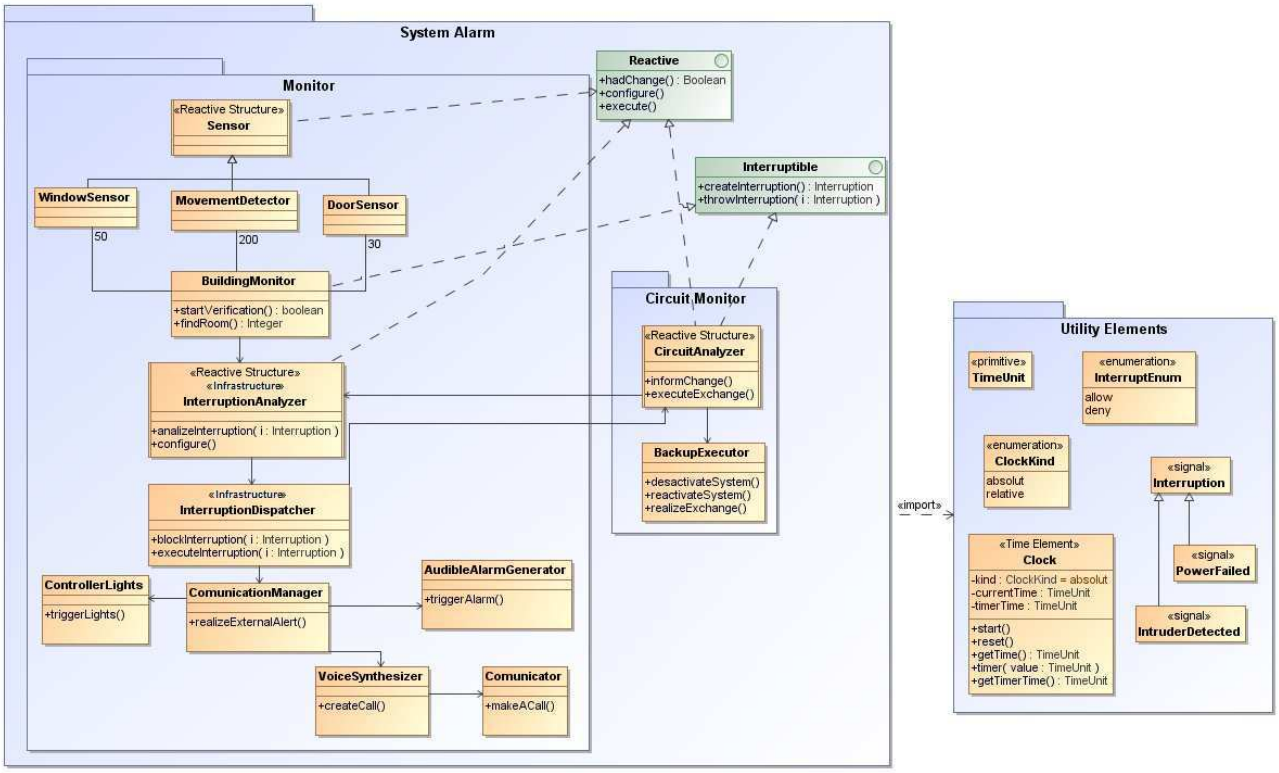

Figura D.3: Diagrama de Classes do *Sistema de Alarmes*.

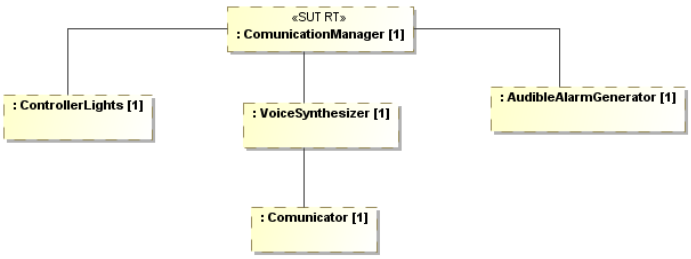

Figura D.4: Diagrama de Estruturas Compostas do componente *External Comunicator*.

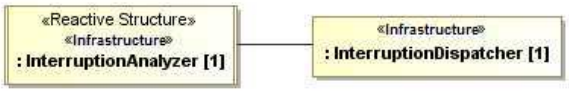

Figura D.5: Diagrama de Estruturas Compostas do componente *Interruption Manager*.

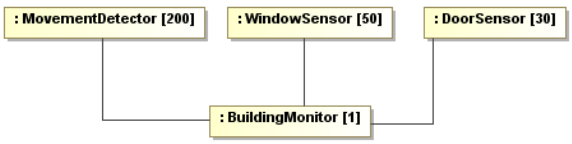

Figura D.6: Diagrama de Estruturas Compostas do componente *Sensor Manager*.

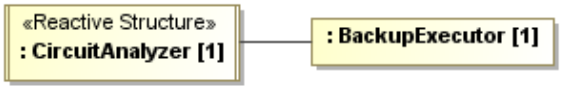

Figura D.7: Diagrama de Estruturas Compostas do componente *Circuit Monitor*.

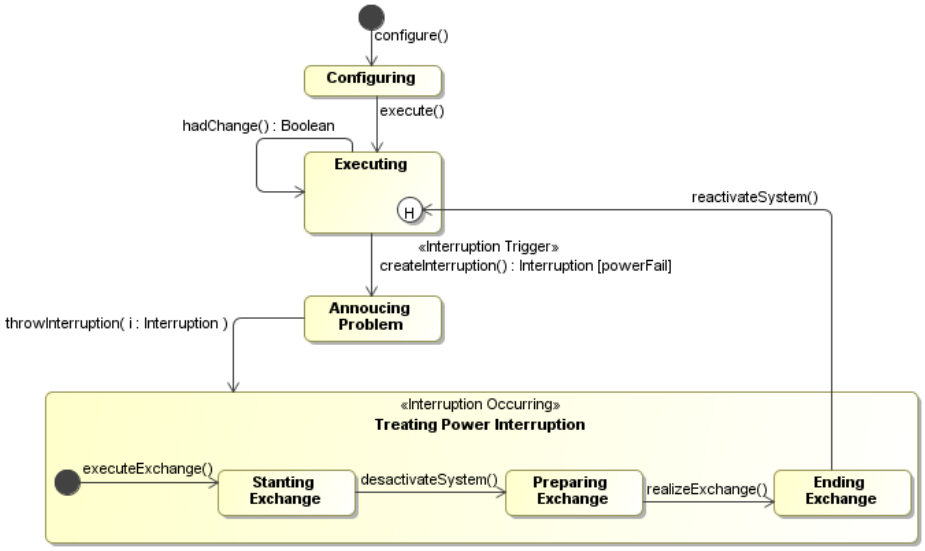

Figura D.8: Máquina de Estados do componente *Circuit Monitor*.

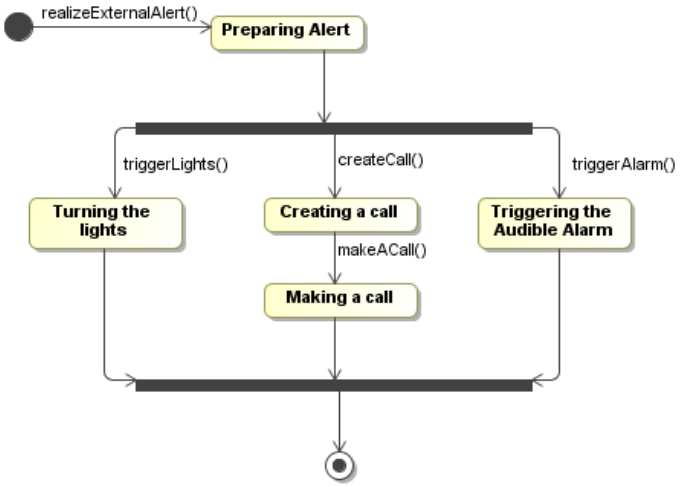

Figura D.9: Máquina de Estados do componente *External Comunicator*.

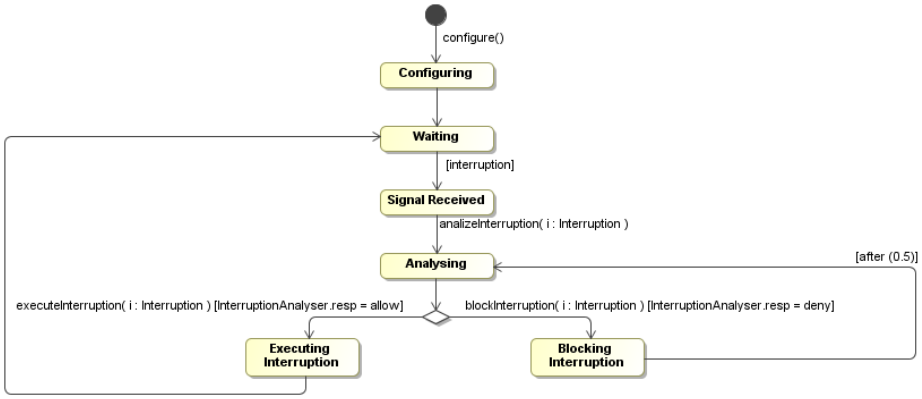

Figura D.10: Máquina de Estados do componente *Interruption Manager*.

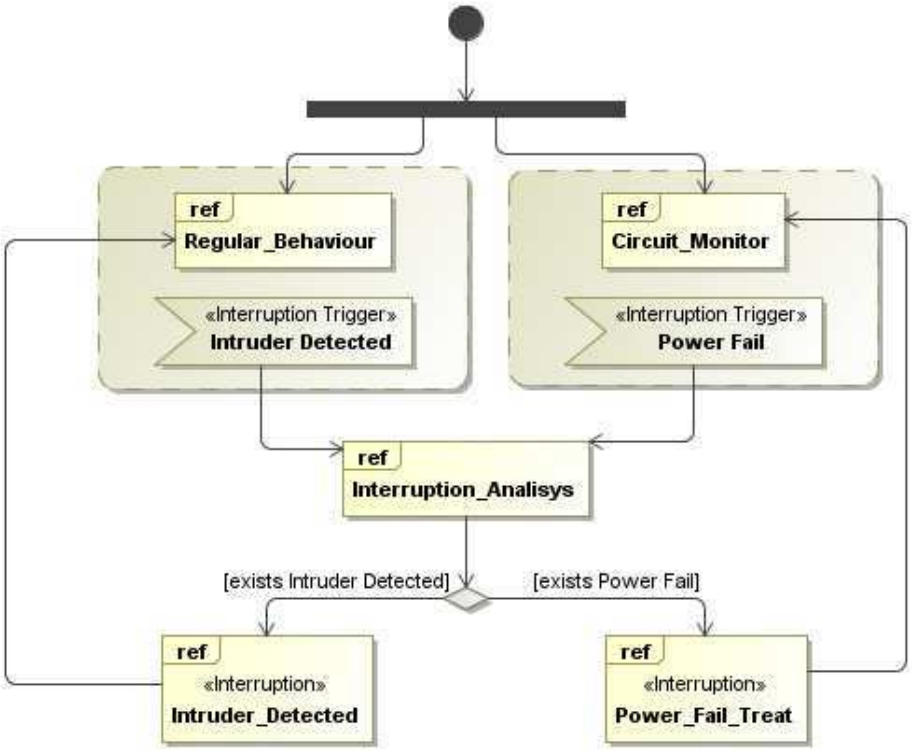

Figura D.11: Diagrama de Overview do *Sistema de Alarmes*.

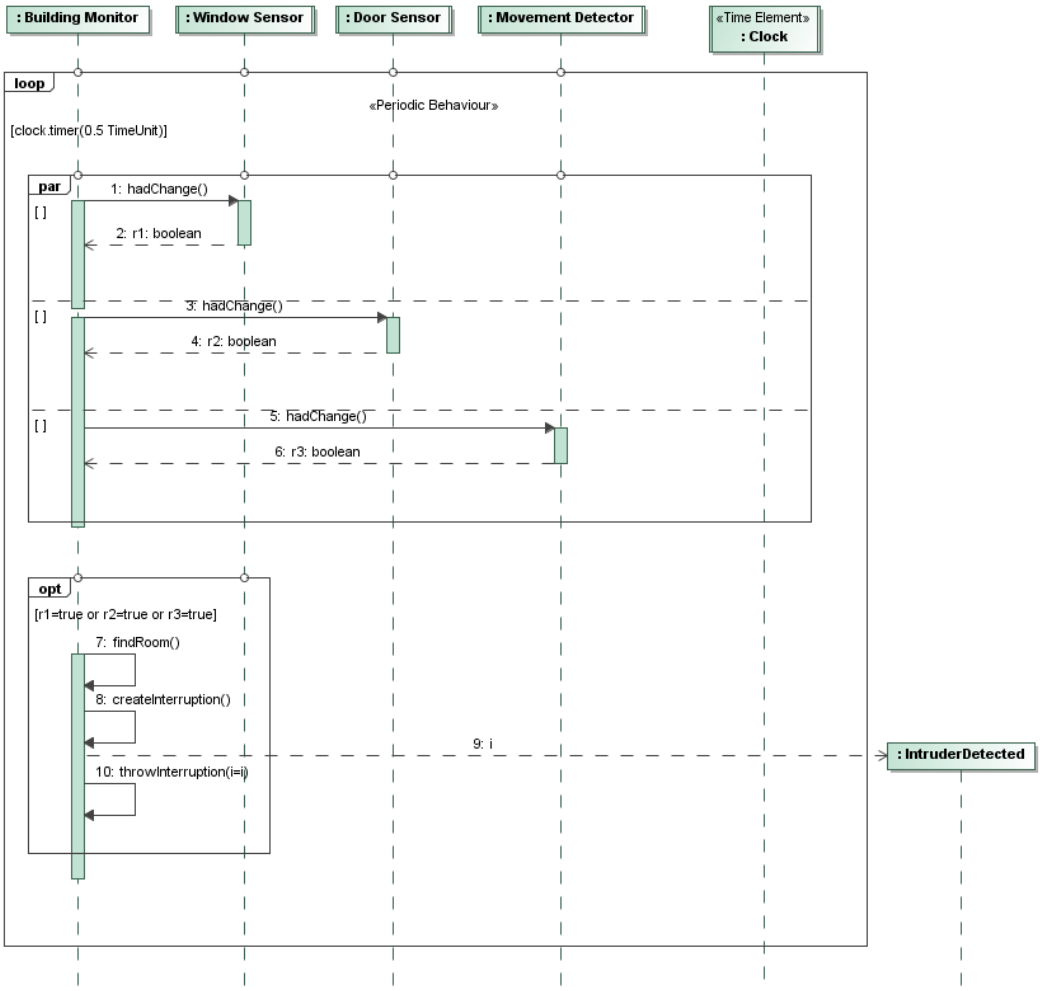

Figura D.12: Diagrama de Sequência referente ao fragmento referenciado *Regular Behaviour*.

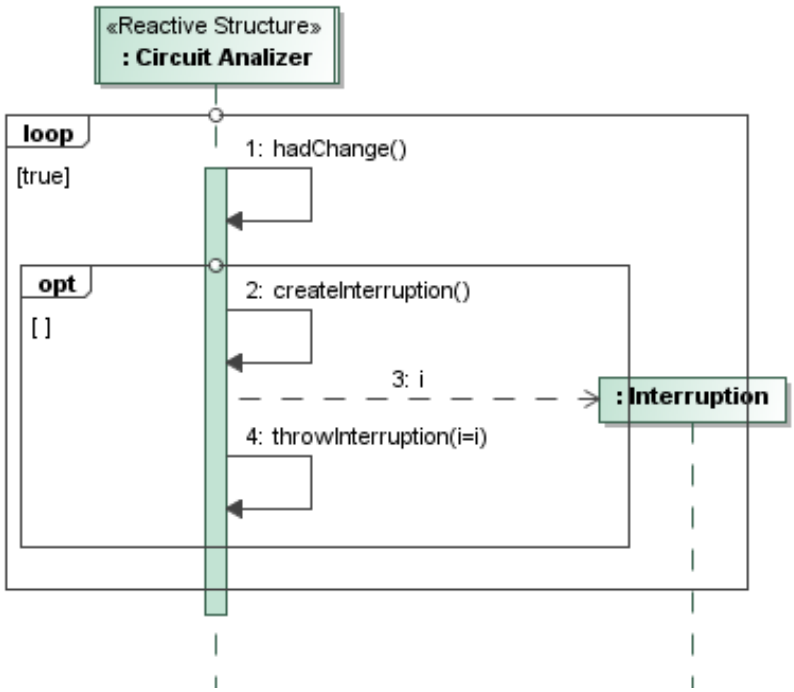

Figura D.13: Diagrama de Sequência referente ao fragmento referenciado *Circuit Monitor*.

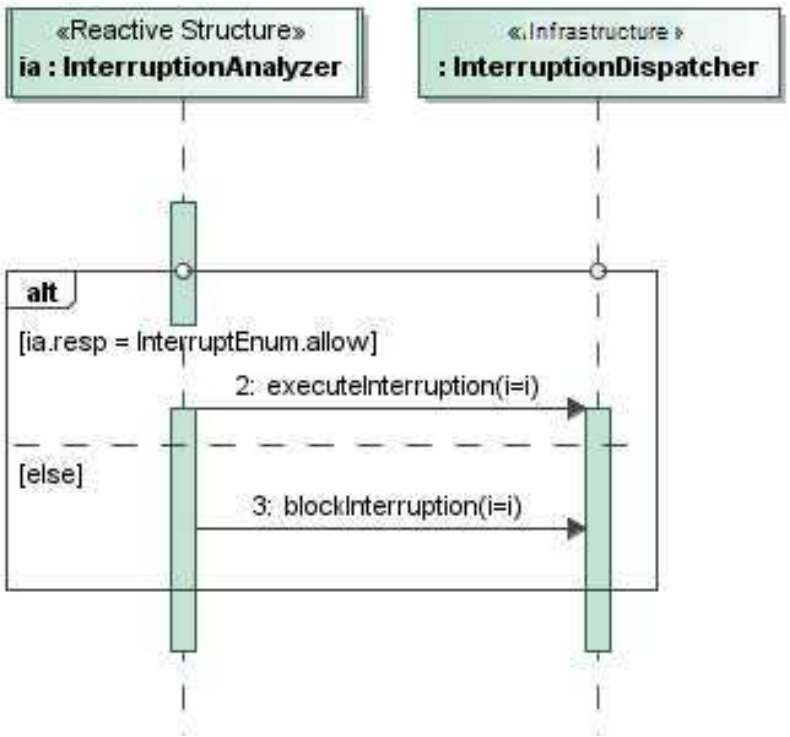

Figura D.14: Diagrama de Sequência referente ao fragmento referenciado *Interruption Analisys*.

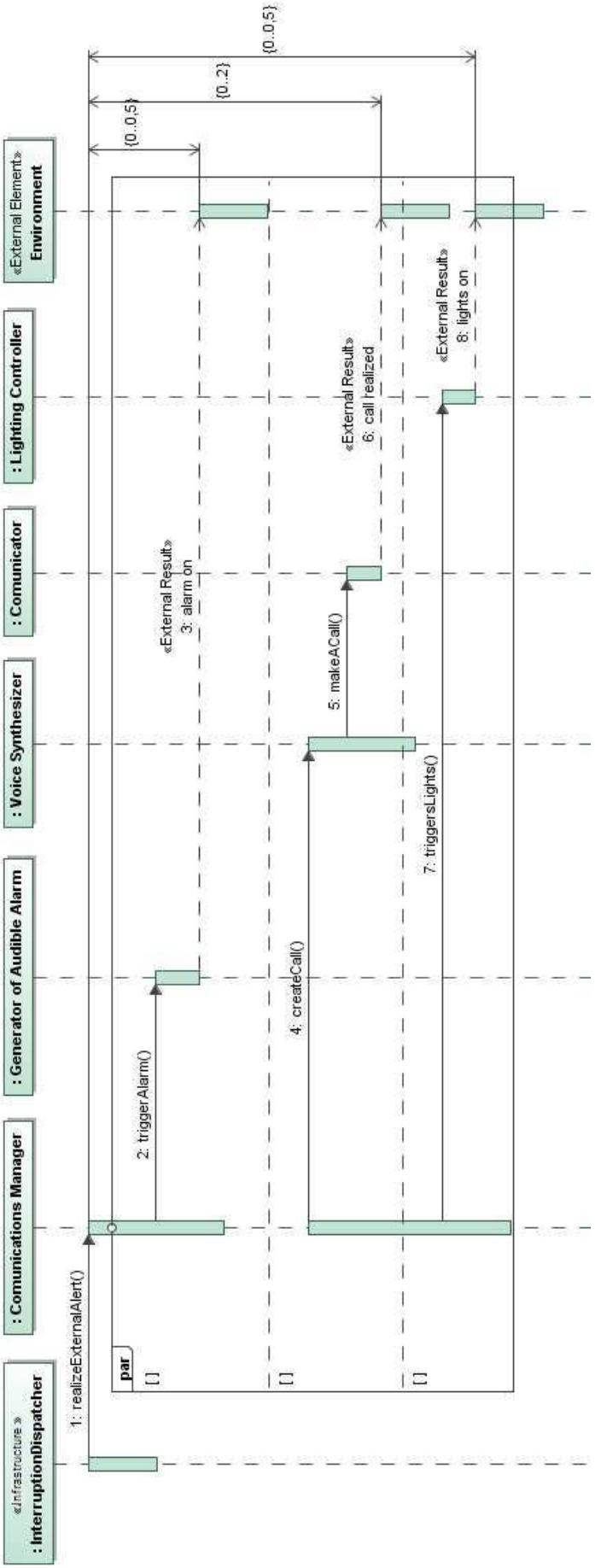

Figura D.15: Diagrama de Sequência referente ao fragmento referenciado *Intruder Detected*.

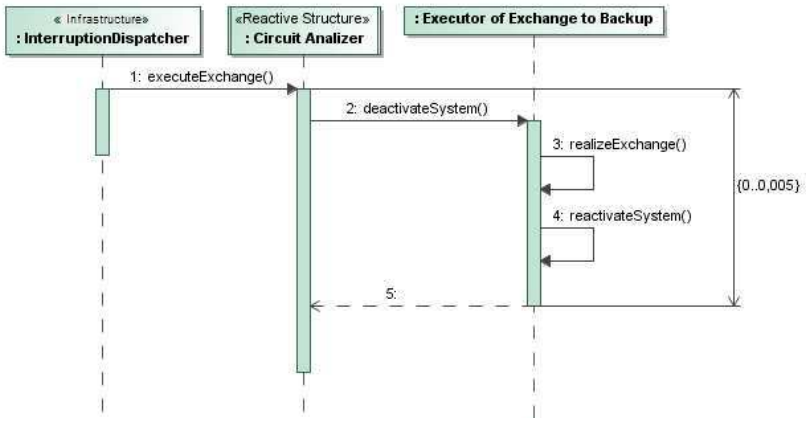

Figura D.16: Diagrama de Sequência referente ao REF *Power Fail Treat*.

Apêndice E

Metamodelo para a linguagem C

É sabido que um dos pontos preponderantes para a realização/execução de abordagens como MDA e MDT é a necessidade da utilização de metamodelos para a criação e execução de suas regras de transformação, e que, neste trabalho foi escolhido como plataforma de execução o FreeRTOS (SO que executa programas em C). Logo, existiu então a necessidade de utilização de um metamodelo representativo desta linguagem. Inicialmente, foi realizada uma busca na literatura, bem como dentre os desenvolvedores. Porém, até onde pudemos investigar, não existe um metamodelo para a linguagem C que seja expressivo o suficiente para ser usado neste contexto. Devido a tal fato, foi necessário construir um metamodelo para representar a linguagem C.

Este apêndice apresenta o metamodelo que foi construído. É válido destacar que o metamodelo em questão foi construído seguindo as diretrizes estabelecidas por [71], estas focam em melhorar a qualidade de metamodelos, deixando-o mais legível, manutenível e primando na organização.

É importante destacar que o autor do metamodelo não é um especialista na linguagem C. Logo, algum elemento pode ter passado despercebido e conseqüentemente não ter sido incluído. Porém, o objetivo da construção deste metamodelo não foi concebê-lo completo em sua plenitude, mas, contribuir construindo um artefato realmente expressivo e usável para o contexto de MDA/MDT. É de interesse que este metamodelo possa ser usado e estendido por outros desenvolvedores. Para tal, o mesmo será disponibilizado em um repositório de metamodelos aberto à desenvolvedores.

E.1 Estrutura de Pacotes

O metamodelo desenvolvido, devido ao seu tamanho e complexidade organizacional, foi subdividido em pacotes. A Figura E.1 apresenta visualmente os pacotes definidos, bem como a maneira como estes se relacionam entre si. Os pacotes criados são os seguintes:

- Main
- Abstractions
- Types
- Declarations
- CompilationDirectiveDeclarations
- Commands
- Expressions
- Sequencers
- Enumerations

Nas próximas seções cada pacote será apresentado individualmente, quando cabível, um exemplo de aplicação de um ou mais elementos deste pacote será demonstrado, bem como as restrições OCL necessárias para a definição dos elementos serão apresentadas.

E.1.1 Pacote Main

A Figura E.2 representa o pacote Main. Este pacote foi criado para agrupar os elementos mais gerais que podem existir em programas C.

A título de exemplificação de possíveis instâncias dos elementos usados neste pacote, a Figura E.3 apresenta o código de um arquivo .c. Neste exemplo de código temos destacado as instâncias dos elementos *C_Unit*, *DeclarationsBlock* e *Function*.

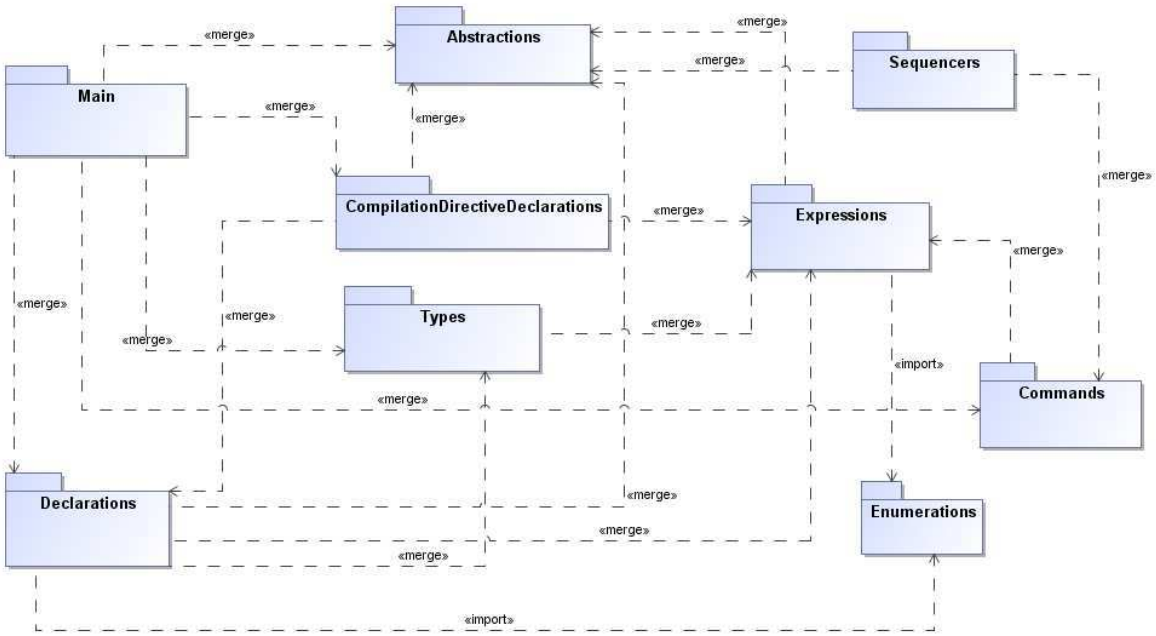

Figura E.1: Estrutura de pacotes do metamodelo da linguagem C.

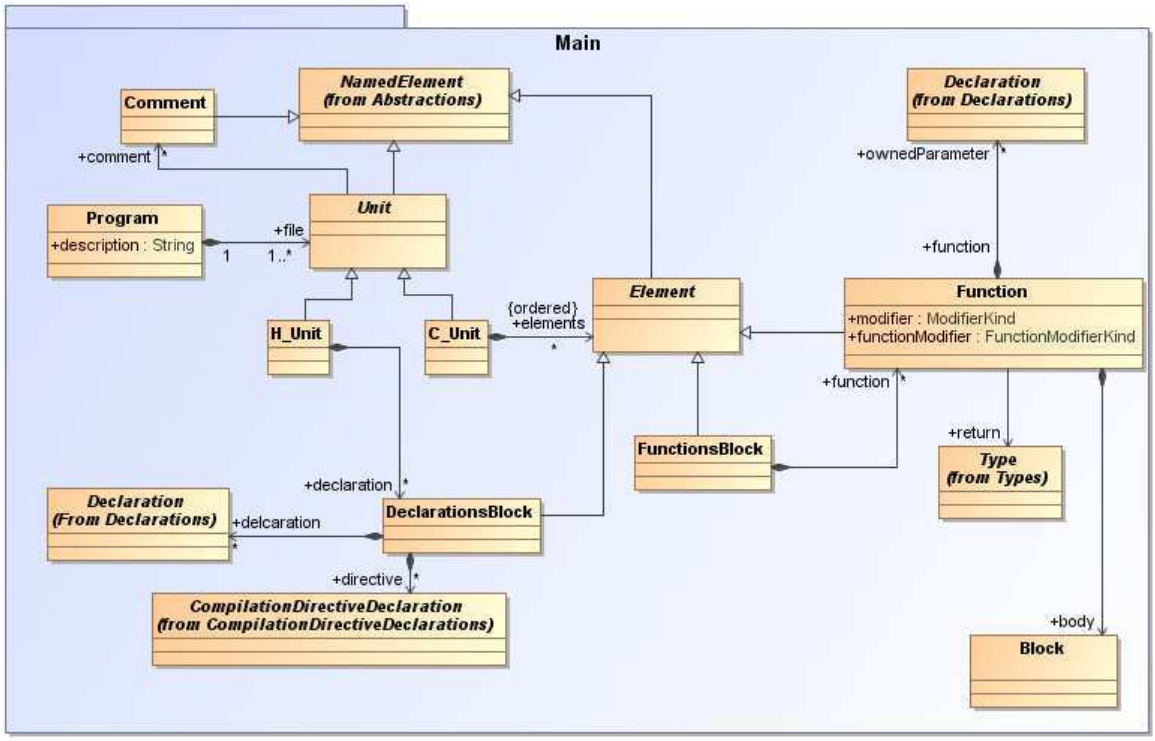

Figura E.2: Pacote Main.

E.1.2 Pacote Abstractions

O pacote *Abstractions* (Figura E.4) foi criado com o objetivo de melhor organizar os elementos presentes no metamodelo. Seguindo uma das diretrizes presentes no trabalho de [71]

Figura E.3: Exemplo de instanciação de alguns elementos do pacote Main.

elementos abstratos foram criados com a intuito de tornar o metamodelo mais claro e facilitar futuras extensões do mesmo.

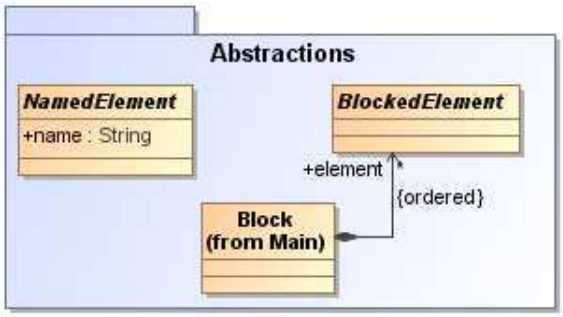

Figura E.4: Pacote Abstractions.

E.1.3 Pacote Types

O pacote *Types* (Figura E.5) apresenta a hierarquia de tipos presente na linguagem C. Como C é uma linguagem fortemente tipada diversos elementos dos outros pacotes irão fazer referencia a um determinado tipo presente no pacote *Types*. O tipo *FromHeader*, apesar de não ser propriamente um tipo de C, foi incluído para possibilitar que os elementos referenciem um tipo presente um determinado arquivo .h e não no .c corrente.

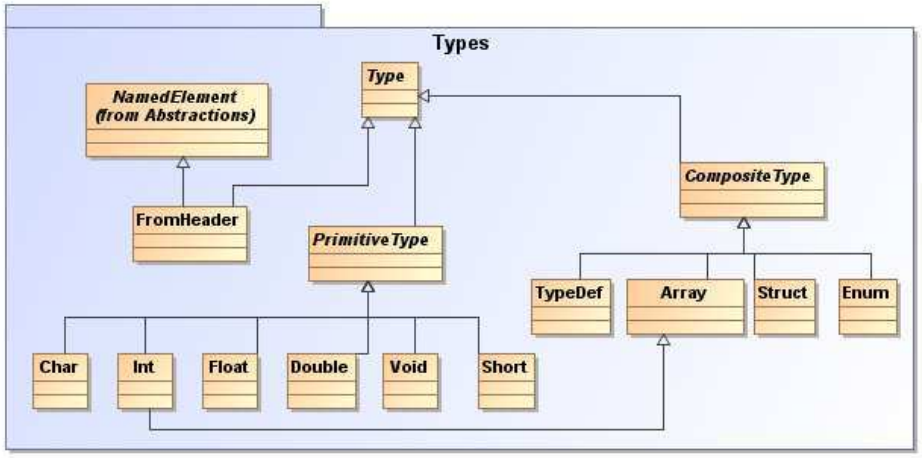

Figura E.5: Pacote Types.

E.1.4 Pacote Declarations

O pacote *Declarations* E.6 agrupa os elementos que representam as possíveis formas de declaração que a linguagem C permite, desde declaração de variáveis (simples ou compostas), até enumerações, constantes, etc.

A Figura E.7 apresenta como exemplo a instanciação dos elementos do metamodelo referentes a declaração de uma variável simples.

Restrições:

Código Fonte E.1: Restrição OCL para o pacote *Declarations*

```
Context VariableDeclaration:
        if self.isAPointer = true then
                self.numberOfPointers > 0
        endif

Context ArrayDeclaration:
        self.dimensions > 0

Context ArrayDeclaration:
        self.elementType.oclTypeOf(PrimitiveType)

Context TypeDefDeclaration:
        self.type.oclTypeOf(PrimitiveType)
```

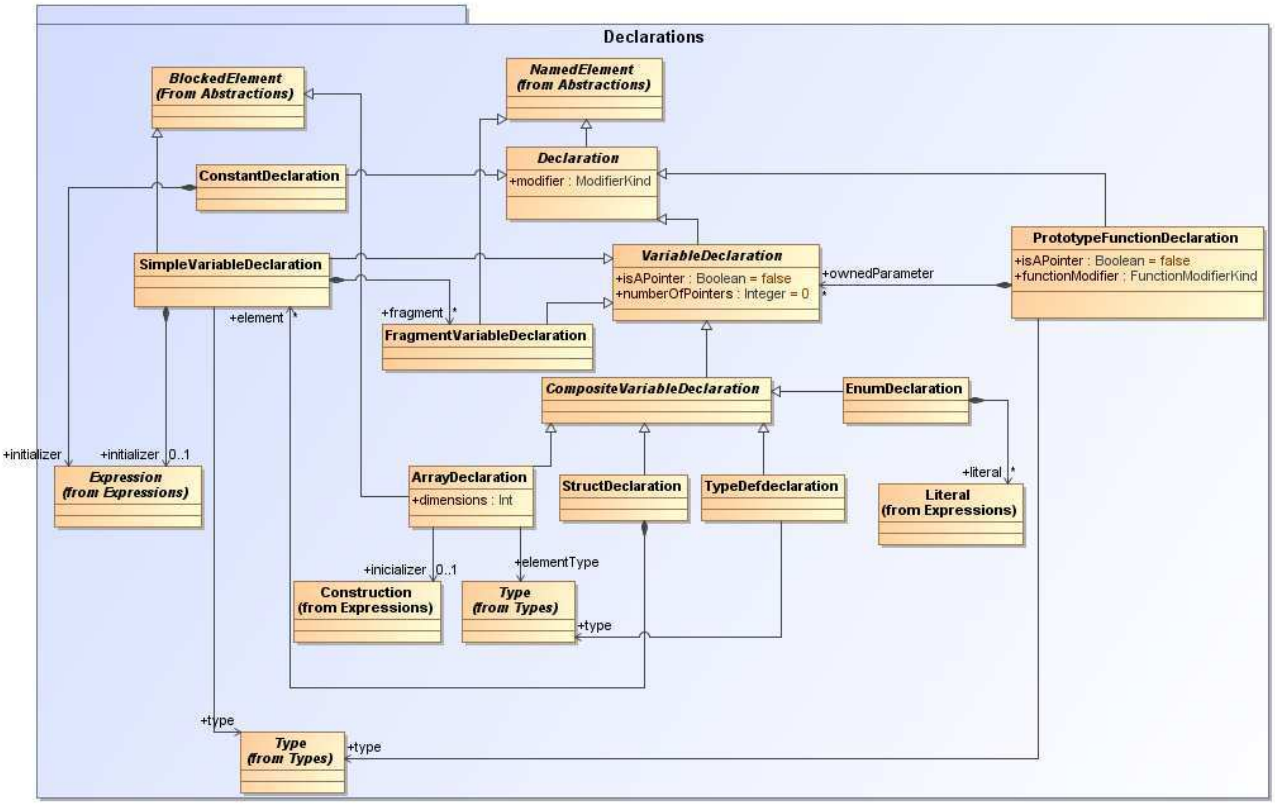

Figura E.6: Pacote Declarations.

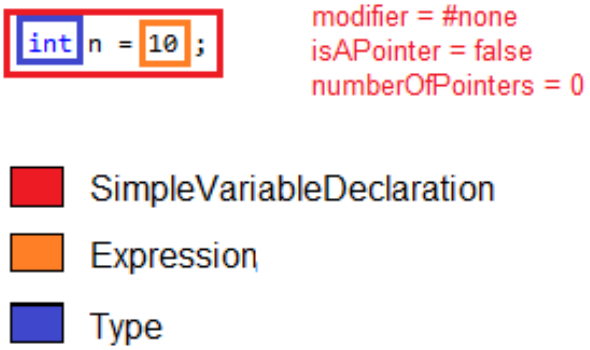

Figura E.7: Exemplo de instanciação de alguns elementos do pacote Declarations.

E.1.5 Pacote CompilationDirectiveDeclarations

O pacote *CompilationDirectiveDeclarations* (Figura E.8) apresenta os elementos necessários para a declaração das diretivas de compilação de C, muito comuns e usadas em praticamente todos programas.

A Figura E.9 apresenta um exemplo de instanciação de alguns elementos do pacote *CompilationDirectiveDeclarations* com o intuito de representar o trecho de código referente a declaração de uma diretiva *include*.

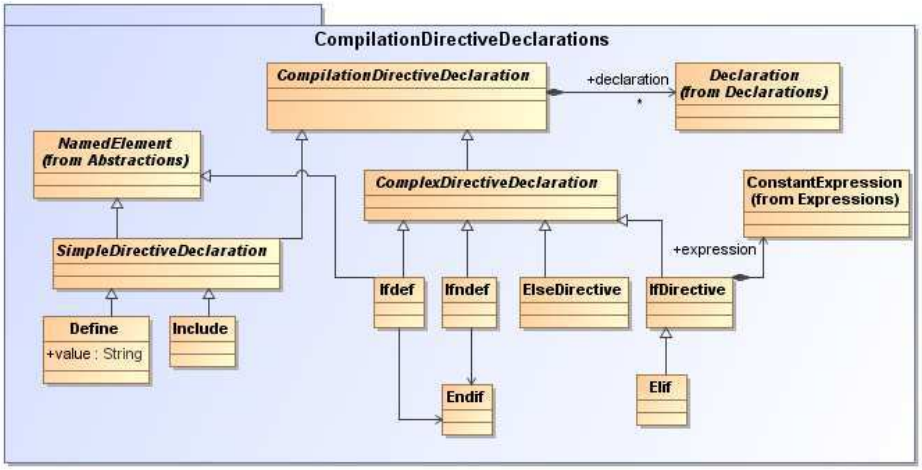

Figura E.8: Pacote CompilationDirectiveDeclarations.

Figura E.9: Exemplo de instanciação de alguns elementos do pacote CompilationDirectiveDeclarations.

Restrições:

Código Fonte E.2: Restrição OCL para o pacote *CompilationDirectiveDeclarations*

```
Context Endif:
        if Ifdef.allInstances()->size < 0 then
                self.oclIsUndefined() == true

Context Else:
        if If.allInstances()->size < 0 then
                self.oclIsUndefined() == true

Context Elif:
```

```
if If.allInstances()->size < 0 then
        self.oclIsUndefined() == true
```

E.1.6 Pacote Commands

A Figura E.10 apresenta a estrutura do pacote *Commands*. Este pacote agrupa os possíveis comandos existentes na linguagem C.

A Figura E.11 exemplifica a instância do comando *IfCommand* no contexto de um trecho de código C.

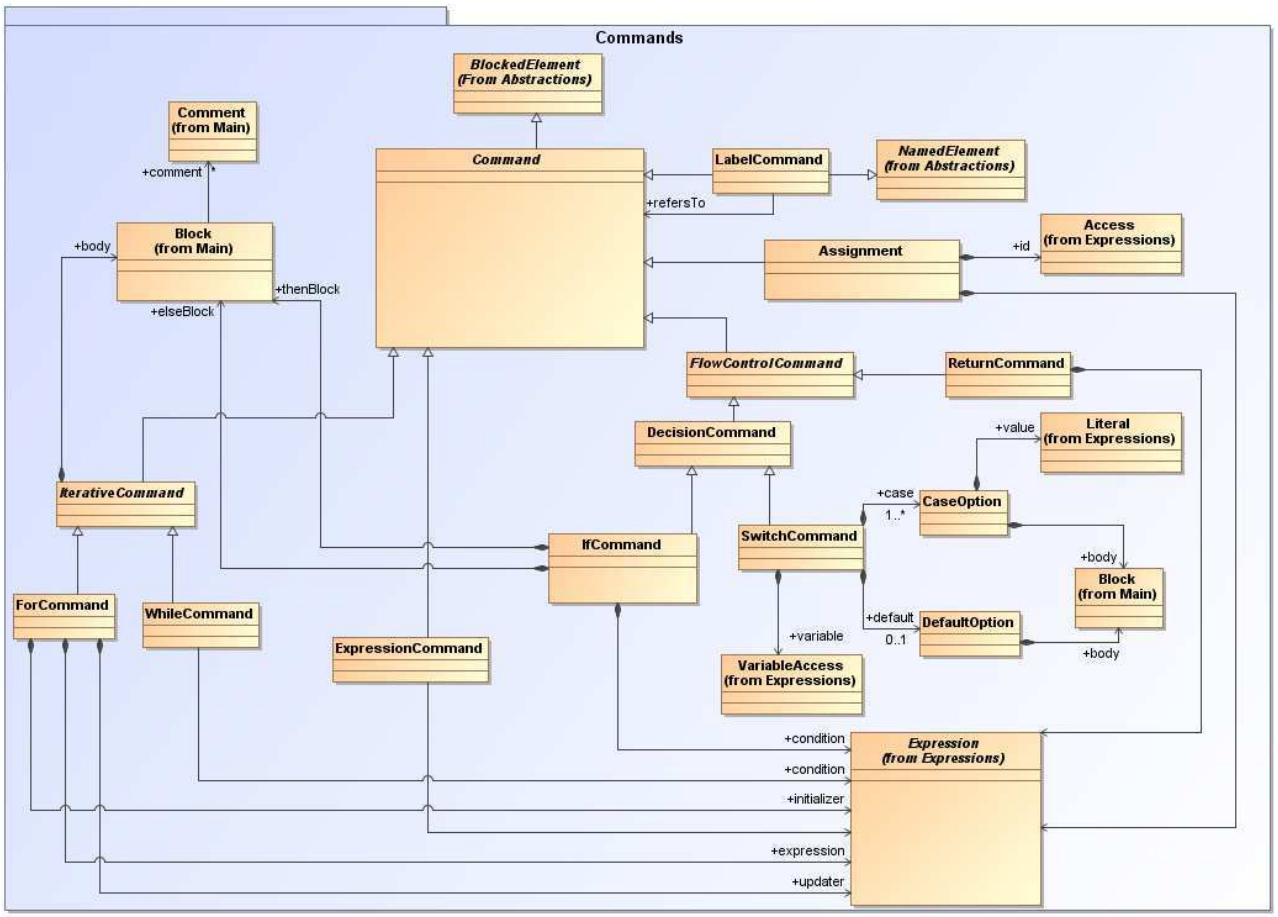

Figura E.10: Pacote Commands.

Restrições:

Código Fonte E.3: Restrição OCL para o pacote *Commands*

```
Context IfCommand:
```

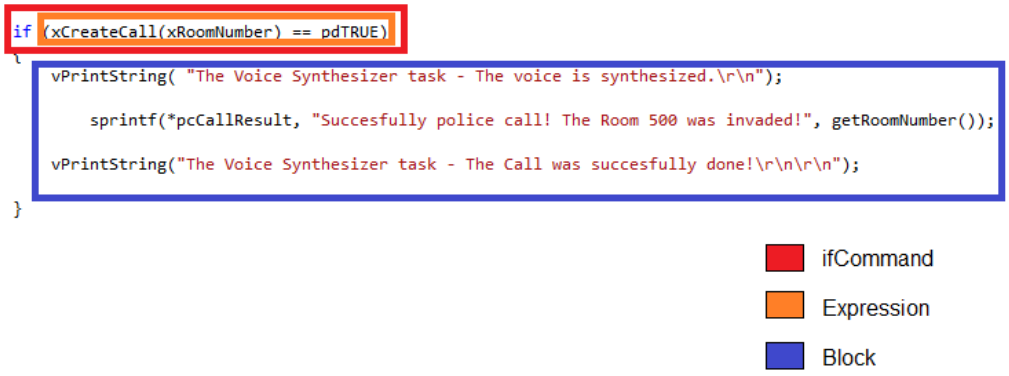

Figura E.11: Exemplo de instanciação de alguns elementos do pacote Commands.

```
        self.condition.oclTypeOf(ConditionalExpression)

Context WhileCommand:
        self.condition.oclTypeOf(ConditionalExpression)

Context ForCommand:
        self.expression.oclTypeOf(ConditionalExpression)
```

E.1.7 Pacote Expressions

O pacote *Expressions* (Figura E.12) reúne as expressões que podem existir num programa C, bem como demonstra com quais elementos estas expressões se relacionam.

Na Figura E.13 demonstra a utilização de alguns dos elementos presentes no pacote *Expressions* para representação de um trecho simples de código.

Restrições:

Código Fonte E.4: Restrição OCL para o pacote *Expressions*

```
Context CastExpression:
        self.elementType.oclTypeOf(Primitive Type)
```

E.1.8 Pacote Sequencers

O pacote *Sequencers* (Figura E.14) foi criado para reunir e diferenciar dos demais os elementos que realizam mudanças de fluxo de execução, mais especificamente os elementos *Goto* e *Break*.

E.1.9 Pacote Enumerations

A Figura E.15 apresenta o pacote *Enumerations*. Este reúne o conjunto de enumerações que são usadas pelos elementos dos demais pacotes do metamodelo.

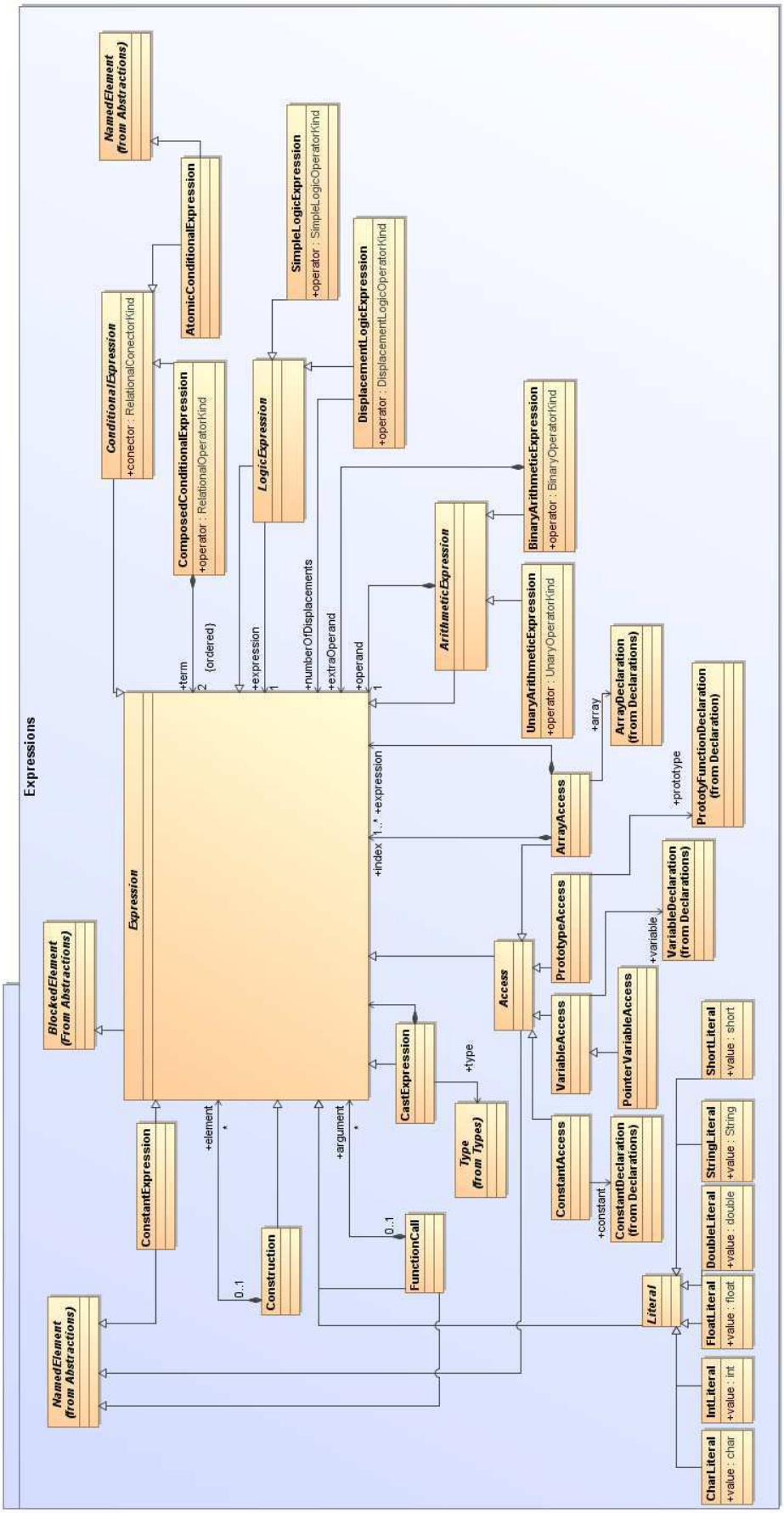

Figura E.12: Pacote Expressions.

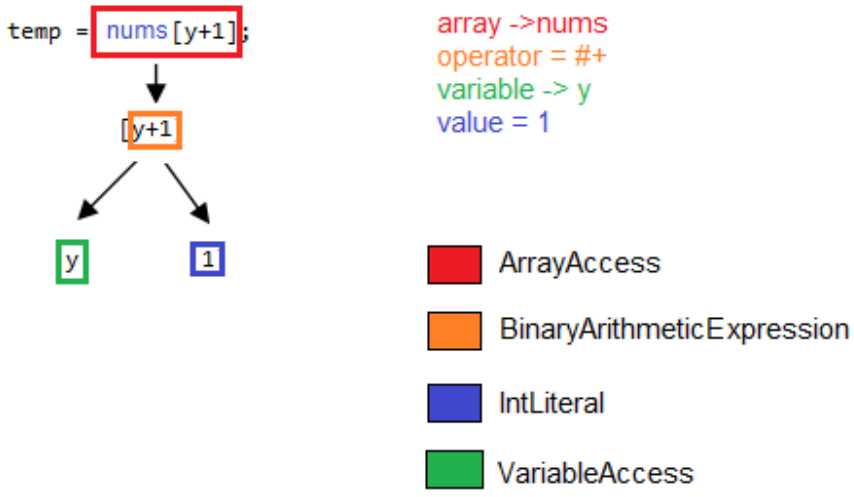

Figura E.13: Exemplo de instanciação de alguns elementos do pacote Expressions.

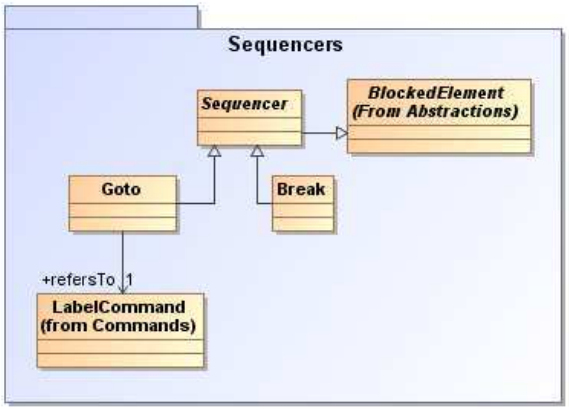

Figura E.14: Pacote Sequencers.

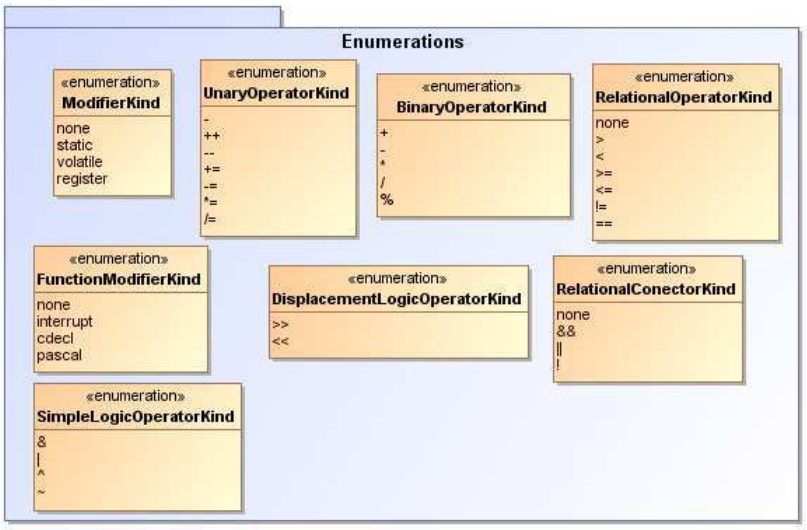

Figura E.15: Pacote Enumerations.