UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

LEANDRO ALEXANDRE FREITAS

# Programação de Espaços Inteligentes Utilizando Modelos em Tempo de Execução

Goiânia
2017

**TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR AS TESES E DISSERTAÇÕES ELETRÔNICAS NA BIBLIOTECA DIGITAL DA UFG**

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

**1. Identificação do material bibliográfico**: **[ ] Dissertação** **[X] Tese**

**2. Identificação da Tese ou Dissertação**

Nome completo do autor: Leandro Alexandre Freitas

Título do trabalho: Programação de Espaços Inteligentes Utilizando Modelos em Tempo de Execução

**3. Informações de acesso ao documento:**

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.

Leandro Alexandre Freitas
Assinatura do (a) autor (a)

Data: 18 / 05 / 2017

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

LEANDRO ALEXANDRE FREITAS

# Programação de Espaços Inteligentes Utilizando Modelos em Tempo de Execução

Tese apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Doutor em Doutorado em Ciência da Computação UFG e UFMS.

**Área de concentração:** Sistemas Distribuídos.

**Orientador:** Prof. Fábio Moreira Costa

**Co-Orientador:** Prof. Ricardo C. A. Rocha

Goiânia
2017

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Alexandre Freitas, Leandro
Programação de Espaços Inteligentes Utilizando Modelos em Tempo de Execução [manuscrito] / Leandro Alexandre Freitas. - 2017.
151, CLI f.: il.

Orientador: Prof. Fábio Moreira Costa; co-orientador Ricardo C. A. Rocha.
Tese (Doutorado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2017.
Bibliografia. Apêndice.
Inclui siglas, mapas, símbolos, gráfico, tabelas, algoritmos, lista de figuras, lista de tabelas.

1. Espaços Inteligentes. 2. Modelos em Tempo de Execução. 3. Linguagem de Modelagem Específica de Domínio. 4. Máquina de Execução de Modelos. 5. Programação centrada no usuário. I. Moreira Costa, Fábio, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

## Ata de Defesa de Tese de Doutorado

Aos quatro dias do mês de abril de dois mil e dezessete, no horário das nove horas, foi realizada, nas dependências do Instituto de Informática da UFG, a defesa pública da Tese de Doutorado do aluno Leandro Alexandre Freitas, matrícula no. 2011 1223, intitulada **"Programação de Espaços Inteligentes Utilizando Modelos em Tempo de Execução"**.

A Banca Examinadora, constituída pelos professores:

Prof. Dr. Fábio Moreira Costa – INF/UFG - orientador

Prof. Dr. Ricardo Couto Antunes da Rocha – DCC Catalão/UFG - coorientador

Prof. Dr. Francisco José da Silva e Silva – DEINF/UFMA

Prof. Dr. Jó Ueyama – ICMC/USP

Prof. Dr. Ronaldo Alves Ferreira – FACOM/UFMS

Prof. Dr. Fabrízzio Alphonsus Alves de Mello Nunes Soares - INF/UFG

emitiu o resultado:

( ) Aprovado

(X) Aprovado com revisão

(A Banca Examinadora deve definir as exigências a serem cumpridas pelo aluno na revisão, ficando o orientador responsável pela verificação do cumprimento das mesmas.)

( ) Reprovado

com o seguinte parecer: precisam ser revisados: a definição do problema, o capítulo de avaliação e o capítulo de trabalhos relacionados, conforme observações anexas.

Prof. Dr. Fábio Moreira Costa

Prof. Dr. Ricardo Couto Antunes da Rocha

Prof. Dr. Francisco José da Silva e Silva

Prof. Dr. Jó Ueyama

Prof. Dr. Ronaldo Alves Ferreira

Prof. Dr. Fabrízzio Alphonsus A. M. Nunes Soares

**Leandro Alexandre Freitas**

Doutorando em Ciência da Computação pela Universidade Federal de Goiás (UFG), Mestre em Ciência da Computação pela UFG (2011) e bacharel em Ciência da Computação pela UFG (2008). Atualmente é professor em regime de dedicação exclusiva pelo Instituto Federal de Goiás e participou do projeto de pesquisa M@ture, do Instituto de Informática da UFG, em cooperação com o centro de pesquisa internacional INRIA. Seus interesses de pesquisa concentram-se nas áreas de Redes de Computadores e Sistemas Distribuídos, com atuação nos seguintes temas: Computação Ubíqua, Computação Sensível ao Contexto, Plataformas de Middleware, Middleware Dirigido por Modelos (MDE), Middleware em Tempo de Execução (Models@Run.Time), Redes de Sensores sem Fio e QoS para Internet do Futuro.

A Deus, meu Senhor e tudo, a Jesus, meu Salvador, a Maria, minha eterna mãe. À minha família, presente do céu, em especial meu pai Mário, que está nos céus, minha mãe Maria das Graças, meu irmão Luciano, a minha cunhada Vivi e minha sobrinha, Maitê. Por fim, a minha amável Jordane Gabriella.

# Agradecimentos

Em primeiro lugar, quero agradecer a Deus por ter me dado saúde e sustento durante essa caminhada. Quero agradecer também a Maria sua Mãe, que sempre intercedeu por mim durante estes anos de estudo. Ao meu pai Mário, que deu sua vida para que eu e meu irmão sempre estudássemos nas melhores escolas, mesmo não tendo tido a oportunidade de estudar. Continue intercedendo por nós aí do céu, pai, te amo. A minha mãe Graça, que juntamente com meu pai, acordava cedo para trabalhar e dar sustento a nossos estudos. Te amo, mãe. A meu irmão Luciano, que sempre esteve do meu lado, me dando forças e incentivando. Te amo, meu irmão. Aos meus familiares, em particular meu padrinho, que me incentivou nos primeiros passos na computação, Manoel Alexandre. Quero agradecer também a meu professor e orientador, Fábio Costa, por tamanha paciência e dedicação comigo. Também quero agradecer a meu co-orientador, professor Ricardo Rocha, por toda sua atenção e também paciência ao me mostrar o caminho. Também existem inúmeros professores que passaram por minha vida, e quero lhes dedicar este trabalho, pois eles construíram meus alicerces: Augusto Venâncio, Vagner Sacramento, Humberto Longo, Auri e Fábio Nogueira. A meus amigos de faculdade e da vida, em especial, Júnio César, Raphael de Aquino, André Coimbra, Marcelo Quinta, Marcos Roriz, Marcos Alves, Marcelo Fortes, Adalberto Júnior e Halley Wesley. Obrigado a todos vocês! Quero agradecer também a meus colegas do Instituto Federal de Goiás, em particular Alan Keller e Victor Hugo Lopes, que me deram palavras de motivação para concluir este trabalho e a minha amiga, Letícia Sateles. Também quero agradecer à Fundação de Amparo à Pesquisa do Estado de Goiás, FAPEG, que financiou este projeto. Por fim, àquela que cuidou de mim nos momentos mais difíceis de minha vida e sempre me encheu de paz quando mais precisava. Ela é minha amada, minha Rosa de Saron, Jordane Gabriella. Te amo!

(...) Mas a vida, a vida, a vida,
a vida só é possível
reinventada. (...)

**Cecília Meireles**,
*Poesia: Reinvenção*.

# Resumo

Alexandre Freitas, Leandro. **Programação de Espaços Inteligentes Utilizando Modelos em Tempo de Execução**. Goiânia, 2017. 149p. Tese de Doutorado Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

O crescimento e a popularização cada vez maior da conectividade sem fio e dos dispositivos móveis, tem permitido a construção de espaços inteligentes que antes eram vislumbrados apenas na proposta de computação ubíqua do cientista da Xerox PARK, *Mark Weiser*. Esses espaços inteligentes são compostos por diversos recursos computacionais, como dispositivos, serviços e aplicações, além de usuários, que devem ser capazes de se associar a esses recursos. Entretanto, a programação destes ambientes é uma tarefa desafiadora, uma vez que os espaços inteligentes possuem uma natureza dinâmica, os recursos se apresentam de forma heterogênea e é necessário que as interações entre usuários e dispositivos sejam coordenadas. Neste trabalho desenvolvemos uma nova abordagem para programação de espaços inteligentes, por meio de modelos em tempo de execução. Para isso, propomos uma linguagem de modelagem de alto nível, denominada *Smart Space Modeling Language* (2SML), em que o usuário é capaz de modelar o espaço inteligente com todos os elementos que dele podem fazer parte. Esse modelo desenvolvido pelo usuário é interpretado e realizado no espaço físico por uma máquina de execução de modelos, denominada *Smart Space Virtual Machine* (2SVM), cujo desenvolvimento é parte deste trabalho.

**Palavras–chave**

Espaços Inteligentes, Modelos em Tempo de Execução, Linguagem de Modelagem Específica de Domínio, Máquina de Execução de Modelos

# Abstract

Alexandre Freitas, Leandro. **Smart spaces programming using Models at Runtime**. Goiânia, 2017. 149p. PhD. Thesis Relatório de Graduação. Instituto de Informática, Universidade Federal de Goiás.

The growth and popularization of wireless connectivity and of mobile devices has allowed the development of smart spaces that were previously only envisaged in the approach proposed by *Mark Weiser*. These smart spaces are composed of many computational resources, such as devices, services and applications, along with users, who must be able to associate with these features. However, programming these environments is a challenging task, since smart spaces have a dynamic nature, resources are heterogeneous, and it is necessary that interactions between users and devices are coordinated with one another. In this work, we present a new approach for smart spaces programming using *Models@RunTime*. In this regard, we propose a high level modeling language, called *Smart Spaces Modeling Language* (2SML), in which the user is able to model the smart space with all elements that can be part of it. Such models are developed by the users, interpreted and effected in the physical space by a model execution engine, called *Smart Space Virtual Machine* (2SVM), whose development is part of this work.

**Keywords**

Smart Spaces, Models@RunTime, Domain Specific Modeling Language, Models Execution Machine

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Glossário

| | |
|---|---|
| **2SB** | 2S Broker |
| **2SML** | Smart Space Modeling Language |
| **2SVM** | Smart Space Virtual Machine |
| **CML** | Communication Modeling Language |
| **CWM** | Common Warehouse Metamodel |
| **DSL** | Domain-Specific Language |
| **DSML** | Domain-Specific Modeling Language |
| **EBAC** | Event Based Access Control |
| **ECM** | Environment Context Model |
| **EMF** | Eclipse Modeling Framework |
| ***i*-DSML** | *interpreted* Domain-Specific Modeling Language |
| **IoT** | Internet of Things |
| **M@RT** | Models@RunTime |
| **MDA** | Model-Driven Architecture |
| **MDE** | Model-Driven Engineering |
| **MOF** | Meta-Object Facility |
| **NCB** | Network Communication Broker |
| **OWL** | Ontology Web Language |
| **RFID** | Radio-Frequency IDentification |
| **ROOD** | Resource-Oriented and Ontology-Driven Development |
| **SE** | Synthesis Engine |
| **SOM** | Smart Object Model |
| **U-2SI** | User-2S Interface |
| **U-C2SM** | User-Centric 2S Middleware |
| **UbiXML** | Ubiquitous XML |
| **UCM** | User Communication Interface |
| **UCM** | User-centric Communication Middleware |

**UML** Unified Modeling Language
**WoT** Web of Things

CAPÍTULO 1

# Introdução

O conceito de ubiquidade tem se tornado cada vez mais real em nossos dias, graças à convergência, disseminação, e popularização de tecnologias de rádio, dos microprocessadores e dos dispositivos digitais, aliados aos novos paradigmas da Internet das Coisas (*Internet of Things* - IoT) [9] e Web das Coisas (*Web of Things* - WoT) [58]. Dispositivos móveis e estacionários coordenam-se entre si de tal forma a fornecer aos usuários acesso a novos serviços que tem por objetivo aumentar as capacidades humanas [34]. Estes dispositivos são tipicamente tratados como objetos inteligentes (*smart objects*), e possuem capacidade computacional, de comunicação, de sensoriamento e de armazenamento [44].

Sistemas computacionais estão se tornando cada vez mais parte de nosso dia a dia, e a interação das pessoas com o mundo real tem se tornado ainda mais determinada pelos sistemas de computação ubíqua [111]. A computação ubíqua é um paradigma de interação usuário-computador e essa tecnologia é integrada de forma transparente a ambientes físicos para auxiliar as pessoas na realização de suas tarefas diárias de forma contínua e onipresente [133, 134]. Com isso, permite-se explorar, de forma mais efetiva, a integração crescente dos dispositivos de computação com o mundo físico [122]. Um tipo particular de ambiente de computação ubíqua é representado pelos espaços inteligentes (*smart space*), que são áreas físicas delimitadas, tais como, salas, andares e prédios, permeados por serviços computacionais que orquestram a infraestrutura do ambiente, orientado a um ou mais aspectos da computação ubíqua [107, 108], com mobilidade e pervasividade.

Espaços inteligentes são ambientes altamente dinâmicos e interativos e devem ser capazes de permitir que usuários executem aplicações e serviços nos mais diversos dispositivos. O conjunto de recursos computacionais presentes nestes ambientes oferecem a possibilidade de criação de espaços interativos que podem oferecer as mais diversas experiências aos usuários. Podemos citar, por exemplo, espaços inteligentes em bibliotecas, museus, estações de metrô, assistência domiciliar à saúde (*homecare*) [96], salas de aula em escolas de universidades, entre outros. Além disso, um mesmo ambiente físico pode ser utilizado para diversos propósitos. Por exemplo, em um mesmo espaço físico de uma casa, podemos ter um ambiente criado para *homecare* ou para ajuda às pessoas nas suas

mais diversas atividades diárias. Em uma mesma sala de uma universidade, podemos ter em um dado momento atividades relacionadas a aula ou mesmo uma reunião.

Embora tenhamos uma diversidade de possibilidades de cenários de espaços inteligentes, a complexidade, heterogeneidade e volatilidade destes ambientes representam desafios significativos para o seu desenvolvimento, o que torna difícil programá-los. Estes ambientes devem ter a capacidade de acomodar dinamicamente usuários e dispositivos que entram e saem do ambiente em virtude da mobilidade, além de gerenciar as aplicações que executam nestes recursos. Em um ambiente ideal, sistemas de computação ubíqua deveriam permitir aos usuários mover suas atividades de um ambiente para outro. Além disso, os usuários devem ser capazes extrair o máximo proveito dos recursos dentro de um determinado ambiente, mesmo em situações que outros usuários e dispositivos entram e saem desse ambiente, e à medida que os recursos mudam [133].

A programação de um espaço inteligente compreende as seguintes tarefas:

- Configurar os objetos inteligentes para aplicações no espaço inteligente e permitir que os usuários façam uso desses recursos;
- Coordenar e manter as interações entre todos os elementos do espaço inteligente;
- Lidar com mudanças de contexto que ocorrem no ambiente, tratando os eventos que ocorrem no espaço inteligente e executando ações para cada um deles;
- Definir diferentes finalidades para um mesmo espaço físico;
- Gerenciar as aplicações que estão executando nos dispositivos dos usuários e do ambiente.

A Engenharia Dirigida por Modelos (*Model-Driven Engineering* - MDE) [69, 15] é uma abordagem que pode ser utilizada no desenvolvimento de sistemas ubíquos. Ela considera modelos como os principais artefatos no desenvolvimento de sistemas. Dentre as sub-áreas da engenharia dirigida por modelos, Modelos em Tempo de Execução (*Models at Runtime* - M@RT) [16] estendem a aplicabilidade dos procedimentos desenvolvidos na engenharia dirigida por modelos se adequando a ambientes em tempo de execução. Para a abordagem M@RT, modelos são entidades de primeira classe, disponíveis em tempo de execução.

Em virtude da alta dinamicidade dos espaços inteligentes, torna-se necessária a adaptação desses ambientes de computação ubíqua às mudanças que ocorrem em tempo de execução. Neste sentido, os modelos em tempo de execução surgem como uma abordagem promissora para lidar com tais ambientes, uma vez que proveem representações apropriadas dos elementos de um sistema, o que proporciona sua adaptação e evolução sem a necessidade de interromper a execução do ambiente. Os modelos de tempo de execução estão relacionados diretamente à reflexão computacional [76], uma vez que ambos definem representações que refletem um sistema em execução e que possuem uma relação de causalidade com este mesmo sistema.

## 1.1 Justificativa

Os ambientes de computação ubíqua são caracterizados por um alto grau de heterogeneidade, haja visto a grande variedade de dispositivos atuais existentes, entre eles sensores, atuadores e demais interfaces de entrada e saída, com as mais diversas capacidades e funcionalidades [97]. Estes dispositivos entram e saem dos espaços inteligentes, e devem permitir aos usuários presentes nestes ambientes executarem as mais diferentes aplicações e serviços.

Graças a este alto grau de dinamicidade destes ambientes, é necessário que as interações, síncronas e assíncronas, que ocorrem entre usuários, dispositivos, aplicações e serviços, ocorram de maneira coordenada.

Outra característica dos ambientes de computação ubíqua é a captura dinâmica das informações do ambiente. Isso permite a esses ambientes executarem ações apropriadas para cada mudança de contexto neles realizada, com o objetivo adaptá-los automaticamente [84].

A mobilidade de usuários, dispositivos e aplicações, é outro aspecto a ser tratado nestes ambientes. Isso ocorre em virtude de que cada um destes elementos do espaço inteligente podem se mover-se intra e inter-ambientes [84].

Além disso, outra característica destes ambientes é a autonomia, que é a capacidade de controlar suas próprias ações e ser autossuficiente. Esta característica permite aos ambientes de computação ubíqua reduzirem a complexidade de gerenciamento do ambiente sob o ponto de vista do usuário. Isso ocorre, pois o próprio ambiente é responsável por sua configuração que deve ocorrer de maneira automática.

Em virtude de todas estas características, é necessário haja um mecanismo responsável por lidar com cada uma delas e permita aos usuários que têm conhecimentos específicos sobre os ambientes de computação ubíqua, programarem estes espaços inteligentes.

## 1.2 Definição do Problema

Programar espaços inteligentes é uma tarefa desafiadora, uma vez que estes ambientes possuem uma diversidade de elementos, dentre eles usuários, dispositivos, aplicações e serviços, que devem ser orquestrados para interagir entre si de maneira coordenada. Além disso, esses ambientes devem ser capazes de identificar usuários e dispositivos que entram e saem desses espaços inteligentes executando as mais diferentes aplicações. Essas aplicações podem mover-se entre os dispositivos de forma a acompanhar o usuário em suas atividades. Tendo isso em vista, realizamos uma revisão da literatura que nos permitiu identificar quatro desafios relacionados à programação desses ambientes: *i)* Pro-

gramação de espaços inteligentes centrada no usuário; *ii)* Implantação automática de espaços inteligentes em tempo de execução; *iii)* Re-configuração do estado de execução das aplicações em tempo de execução; e *iv)* Mudança de finalidade do espaço inteligente em tempo de execução.

### 1.2.1 Programação de Espaços Inteligentes Centrada no Usuário

A programação de espaços inteligentes demanda muito esforço dos usuários programadores desses ambientes, uma vez que a maior parte dos mecanismos utilizados para realizar esta tarefa exigem conhecimentos aprofundados de programação. A programação centrada no usuário visa abstrair do desenvolvedor aspectos técnicos relativos ao domínio a ser programado e reduzir o esforço requisitado do programador para criar estes ambientes computacionais [111]. Sendo assim, a programação centrada no usuário tem por objetivo oferecer aos usuários que tem conhecimentos específicos do domínio de espaços inteligentes, abstrações mais próximas dele que permitem facilitar a atividade de programação destes ambientes.

Existe uma diversidade de trabalhos que oferecem plataformas de *middleware* mais gerais e permitem a programação destes ambientes, como por exemplo *OpenCom* [27], *ACE ORB* (TAO) [113], OpenCorba [82], *FlexiNet* [61], *AspectIX* [60], entre outros; além de outras plataformas de *middleware* e *frameworks* mais específicos, como por exemplo os trabalhos de *Corredor et al*. [30], *Gilman et al*. [52], *Soldatos et al*. [117], *Gouin et al*. [53], entre outros.

De uma maneira geral, as soluções propostas para programação de espaços inteligentes é feita de maneira *ad hoc* e em nível de implementação, ou seja, de acordo com as tecnologias empregadas e a arquitetura de suporte oferecida [55]. Desta forma, não são oferecidas abstrações em um nível mais alto para a programação destes ambientes. O objetivo da programação centrada no usuário é oferecer simplicidade operacional para os usuários que tenham conhecimentos específicos de espaços inteligentes, de tal forma que a complexidade das tecnologias subjacentes seja mascarada pelo mecanismo desenvolvido [81].

### 1.2.2 Implantação Automática de Espaços Inteligentes em Tempo de Execução

De uma maneira geral, as abordagens encontradas na literatura exigem que a implantação da programação dos espaços inteligentes seja feita de maneira manual [52, 3, 126, 106, 50] nos dispositivos.

Essas soluções exigem que cada um dos dispositivos do ambiente seja configurado manualmente pelo programador do ambiente para que possa participar do espaço

inteligente. Desta forma, surge a necessidade da criação de mecanismos que configurem dispositivos e aplicações destes ambientes de computação ubíqua sem a intervenção do usuário.

### 1.2.3 Re-configuração do Estado de Execução das Aplicações em Tempo de Execução

Durante seu processo de execução, as aplicações podem assumir diferentes estados, por exemplo, iniciada, pausada, resumida ou finalizada. A manipulação destes estados requer mecanismos que possam acessar a aplicação e definir esses comportamentos em tempo de execução de acordo com as necessidades do espaço inteligente, bem como dos usuários que fazem uso delas.

Por exemplo, para um determinado espaço inteligente, uma aplicação pode ter a necessidade de inicializar automaticamente em todos os dispositivos móveis dos usuários assim que eles entram no ambiente de computação ubíqua. Outro exemplo, seria a mudança de localização de um usuário que está executando sua aplicação em um dispositivo e se aproxima de outro com melhores configurações. Neste caso, para mover uma aplicação de um dispositivo para outro é necessário que ela seja parada, tenha seu estado salvo e, logo após isso, seja enviada para o outro dispositivo. Neste novo dispositivo, seu estado deve ser recuperado e a aplicação iniciada novamente.

### 1.2.4 Mudança de Finalidade do Espaço Inteligente em Tempo de Execução

Mudar a finalidade de um espaço inteligente é definir um novo comportamento para um ambiente de computação ubíqua. Para realizar esta operação, as abordagens em geral exigem que o sistemas sejam reinicializados ou mesmo parados, de tal forma que possa ser implantada a nova programação no ambiente. Desta forma, temos como desafio a criação de mecanismos que permitam a definição de uma nova programação para o espaço inteligente sem a necessidade de parar o sistema para implantá-la em cada um dos dispositivos do ambiente.

A definição do problema permitiu que identificássemos o principal desafio de nossa tese, que é facilitar a programação de espaços inteligentes para os usuários que tenham conhecimentos específicos destes ambientes. Isto implica que devemos ter uma ferramenta que ofereça abstrações de mais alto nível para a programação destes ambientes, por meio de uma linguagem, e um mecanismo que permita o processamento e implantação desta programação nos dispositivos do ambiente.

## 1.3 Objetivos

O objetivo deste trabalho é propor uma abordagem para programação de espaços inteligentes centrada no usuário. Ao longo do trabalho, demonstraremos que os modelos em tempo de execução (*Models at Runtime* - M@RT) oferecem representações apropriadas para elementos dos ambientes de computação ubíqua, conseguindo expressar e manipular tanto a estrutura quanto o comportamento do sistema em execução.

Com o intuito de demonstrar a viabilidade da proposta, este objetivo é concretizado a partir dos seguintes objetivos específicos:

- Identificar os principais elementos pertencentes aos ambientes de computação ubíqua;
- Fornecer uma linguagem de modelagem de espaços inteligentes, denominada, *Smart Spaces Modeling Language* (2SML), que permite programar estes ambientes e seus mais diversos elementos;
- Oferecer uma máquina de execução de modelos, denominada, *Smart Spaces Virtual Machine* (2SVM), para processamento e implantação dos modelos no ambiente de computação ubíqua;
- Implementar e avaliar a linguagem de modelagem e a máquina de execução de modelos para espaços inteligentes.

## 1.4 Contribuições

As contribuições desta tese são derivadas dos objetivos que foram apresentados na Seção 1.3 e podem ser resumidas da seguinte forma:

1. **Uma abordagem centrada no usuário para programação de ambientes de computação ubíqua**. A abordagem de modelos em tempo de execução nos permitiu representar os elementos dos espaços inteligentes de tal forma que conseguimos expressar e manipular a estrutura desses ambientes (composta por usuários, objetos inteligentes e aplicações) e seu comportamento;
2. **Especificação de uma linguagem de modelagem de espaços inteligentes, e por consequência, de seu metamodelo, que permite definir tipos de usuários, aplicações ubíquas, objetos inteligentes e políticas para o ambiente**. Para definir esta linguagem, nós identificamos os principais elementos pertencentes aos espaços inteligentes e criamos um metamodelo para ela, que contém representações de cada um deles. A fim de validar a definição da linguagem, nós a validamos por meio de sua implementação.

3. **Criação de uma máquina de execução para processamento e implantação automática dos modelos no ambiente criados com a linguagem de modelagem**. A máquina de execução processa os modelos desenvolvidos pelo usuário e os implanta no ambiente de computação ubíqua. Entretanto, para que isso seja feito é necessário que diferentes instâncias dessa máquina executem nos dispositivos presentes no espaço inteligente e orquestrem as mais diversas interações entre usuários, aplicações e estes mesmos dispositivos. Em nosso trabalho, nós especificamos esta máquina e a validamos através de sua implementação.

## 1.5 Organização da Tese

O restante da tese está organizado da seguinte forma: no Capítulo 2, descrevemos a Fundamentação Teórica, que nos fornece os alicerces para todos os conceitos utilizados em nosso trabalho. No Capítulo 3, detalhamos a linguagem de modelagem definida para lidar com a programação de espaços inteligentes e, no Capítulo 4, descrevemos a arquitetura da máquina de execução de modelos. No Capítulo 5, apresentamos a implementação da linguagem de modelagem, 2SML, e de sua máquina de execução 2SVM. O Capítulo 6 apresenta um estudo de caso em que descrevemos exemplos de ambientes de espaços inteligentes e avaliamos a capacidade da linguagem de modelar estes ambientes, bem como a capacidade da 2SVM de executar esses modelos. O capítulo também apresenta uma avaliação do tempo de processamento das operações relacionadas à execução de modelos. Concluímos com o Capítulo 7, que discute os trabalhos relacionados, e o Capítulo 8, que apresenta a conclusão, destacando os resultados e contribuições, bem como trabalhos futuros.

CAPÍTULO 2

# Fundamentação Teórica

A Computação Ubíqua concentra-se na integração de recursos computacionais de maneira despercebida e sem rupturas *seamlessly*, colaborando com as tarefas do cotidiano das pessoas [133, 39, 85]. Ela visa, por meio desta integração, tornar invisível a utilização da computação para o usuário [100, 110].

Com a recente popularização de dispositivos computacionais móveis, como *smartphones* e *tablets*, combinados à infraestrutura de espaços inteligentes, temos o cenário ideal para a realização dos conceitos da computação ubíqua [67, 124, 115]. Este capítulo apresenta um referencial teórico sobre a computação ubíqua que nos permitirá compreender o problema a ser resolvido, a programação de espaços inteligentes. Além disso, apresenta os principais conceitos e definições acerca de Modelos em Tempo de Execução. Esta abordagem foi utilizada em nosso trabalho para lidar com a programação de espaços inteligentes, uma vez que os modelos oferecem representações mais apropriadas dos elementos de um sistema em execução [47].

O capítulo está estruturado da seguinte forma: a Seção 2.1 apresenta os conceitos da Computação Ubíqua, bem como o de Espaços Inteligentes e Programação de Espaços Inteligentes. Na Seção 2.2, apresentamos os conceitos de Modelos em Tempo de Execução, Metamodelos e Linguagens Específicas de Domínio. Por fim, a Seção 2.3 descreve o conceito de Máquinas de Execução de Modelos e na Seção 2.4 está a conclusão do capítulo.

## 2.1 Computação Ubíqua

A Computação Ubíqua tomou forma graças ao trabalho de *Mark Weiser* [133], que vislumbrou a possibilidade de integrar a computação às atividades do dia a dia, tornando seu uso invisível para o usuário. De acordo com esse conceito, a computação estaria permeada nos objetos do ambiente físico do usuário, ao contrário de requerer dispositivos computacionais tradicionais para interação, como por exemplo, teclado e mouse. Segundo *Mark Weiser*, o foco da computação ubíqua está no usuário, extrapolando o dispositivo computacional [133].

A pesquisa na área de computação ubíqua tem por objetivo criar um ambiente inteligente com dispositivos computacionais embarcados e com conectividade em rede, fornecendo aos usuários um acesso transparente aos serviços [105]. Neste contexto, destacam-se algumas áreas de pesquisa importantes que foram introduzidas nos últimos anos e que possuem forte relação com a computação ubíqua, como Inteligência Ambiente (*Ambient Intelligence*) [17], Internet das Coisas (*Internet of Things*) [138] e Sistemas Ciberfísicos (*Cyber Physical Systems*) [10].

Um ambiente de computação ubíqua possui elementos de computação e comunicação que estão em grande parte "ocultos" do usuário [101, 135]. Estes ambientes também devem permitir que diversos dispositivos executem uma mesma aplicação, além de serem capazes de dar suporte a vários serviços. As tecnologias empregadas podem não ser facilmente visíveis, sendo usadas ou incorporadas na construção de toda a infraestrutura [102], o que contribui para o aspecto da ubiquidade. Uma das ideias centrais da computação ubíqua é levar o computador "para longe das estações de trabalho" e torná-lo difusamente disponível no ambiente [36]. Nesse sentido, o computador afasta-se do ambiente de escritório e ocupa uma ampla variedade de ambientes domésticos e de trabalho, trazendo consigo aspectos de mobilidade e cooperação para o primeiro plano [12, 13].

Existem algumas características que são importantes para demonstrar as particularidades da computação ubíqua:

- *Mobilidade*:
  - De aplicação: a aplicação move-se de um dispositivo para o outro, adaptando-se às suas características;
  - De usuário: a mobilidade de usuário está relacionada à capacidade do sistema de continuar provendo um serviço ao usuário que deixa de utilizar um dispositivo e passa para outro;
  - De dispositivo: esta mobilidade está relacionada à capacidade do sistema de continuar provendo um serviço ao usuário após a movimentação do dispositivo.
- *Diversidade*: uma aplicação pode ser acessada através de dispositivos heterogêneos, como *laptops*, *tablets* e *smartphones*. Estes dispositivos podem apresentar mais de uma visão da mesma aplicação, como por exemplo, *i)* O tamanho de tela pode variar de acordo com o dispositivo; e *ii)* A interface de interação com o usuário pode ser sensível ao toque (*touchscreen*), como no caso de *tablets* ou *smartphones*, ou até mesmo utilizar comandos de voz, como em televisões inteligentes (*Smart TVs*) e assistentes pessoais.
- *Sensibilidade ao contexto*: as aplicações respondem a mudanças de contexto. Segundo [37, 2, 23, 123], contexto é qualquer informação relevante para caracterizar a

situação de entidades em uma relação usuário-computador, exemplo disso é a localização de um usuário, que pode ser inferida a partir da interface *Global Positioning System* (GPS) de seu *smartphone*;

- *Comportamento autonômico*: algumas tarefas são realizadas de forma autônoma, ou seja, sem nenhum tipo de intervenção do usuário, tornando-se invisíveis a ele [70, 65];
- *Interação*: os dispositivos devem ser capazes de se comunicar, trocar informações e interagir entre si.

Os ambientes de computação ubíqua são extremamente voláteis [131, 22, 1], uma vez que são altamente dinâmicos e mudam de maneira imprevisível. Os dispositivos desses ambientes possuem conectividade volátil, já que a maior parte deles estão conectados por redes sem fio, mais suscetíveis a desconexão. Outra característica desses ambientes é a interação espontânea, ou seja, as associações entre os elementos rotineiramente alteram entre eles [71, 31].

Apesar de todos os benefícios apresentados pela computação ubíqua, existe uma série de desafios que devem ser tratados [64, 44, 80, 110]:

- Como utilizar *middleware*, arquiteturas e modelos de programação para programar espaços inteligentes de forma dinâmica e eficaz?
- Como os usuários podem interagir com os espaços inteligentes de maneira intuitiva e ubíqua, por meio dos mais diversos dispositivos, tais como, *wearables*, *smartphones*, *displays* públicos, entre outros?
- Como gerenciar os mais diversos objetos inteligentes (*smart objects*) e usuários que entram e saem do espaço inteligente?
- Como lidar com as mais diversas aplicações que podem executar no ambiente?

### 2.1.1 Espaços Inteligentes

A crescente popularização dos dispositivos móveis e a facilidade de acesso à internet por meio de conexões móveis de terceira e quarta gerações, está tornando o mundo cada vez mais conectado. Isso, combinado com a Internet das Coisas (*Internet of Things* - IoT) [9, 77, 132] e *Web* das Coisas (do inglês, *Web of Things* - WoT) [58, 56, 57], traz como potencial a capacidade de interligar, monitorar e controlar remotamente diversos dispositivos do cotidiano, tais como *smartTVs*, interruptores de lâmpadas, alarmes residenciais, fechaduras, ar-condicionado, entre outros, reforçando os conceitos da computação ubíqua.

A fim de reduzir os problemas relacionados às ações que os usuários realizam em um ambiente físico, uma vez que não é possível prever que tipo de operações que

cada um irá desempenhar, a redução do escopo de um ambiente de computação ubíqua a áreas delimitadas permite restringir e limitar os usuários a comportamentos particulares [32]. Essa delimitação da área física facilita a construção de serviços de computação ubíqua sobre a infraestrutura do ambiente, além de auxiliar no conhecimento prévio dos elementos computacionais existentes [108]. Atualmente, os termos mais utilizados para designar esse tipo de ambiente são espaço inteligente (*smart space*) ou ambiente inteligente (*smart environment*). Nesta tese, utilizaremos o termo espaço inteligente.

Espaços inteligentes permitem a aquisição de conhecimento a partir do ambiente e a adaptação dos elementos presentes no sentido de se tirar melhor proveito dos recursos [29, 24, 78, 103]. Para fazer isso, sensores presentes no ambiente monitoram e coletam informações do ambiente físico e ações são realizadas por atuadores com base em decisões tomadas por um mecanismo de raciocínio. Esses mecanismos de raciocínio filtram e gerenciam grandes quantidades de informações que trafegam nos espaços inteligentes diariamente. Seu papel pode ser dividido em duas partes [29, 78, 103]: *i)* Modelar as informações coletadas do ambiente em conhecimento abstrato que seja útil para o espaço inteligente; e *ii)* Raciocinar sobre esse conhecimento para apoiar de maneira eficaz as atividades do dia a dia dos usuários.

Espaços inteligentes estendem a computação para os ambientes físicos e permitem que os diferentes dispositivos neles presentes ofereçam suporte às operações que serão desempenhadas pelos usuários de maneira coordenada, com base em suas preferências e na situação atual do ambiente físico [116, 104, 59]. Assim, em um nível de abstração elevado, os espaços inteligentes são tidos como ambientes de computação ubíqua que entendem e reagem às necessidades humanas [86]. Lupiana *et al*. [86] propõe a seguinte definição para espaços inteligentes:

**Espaço inteligente** *é um ambiente computacional e sensorial altamente integrado que efetivamente raciocina sobre os contextos físico e de usuário do espaço para agir transparentemente com base em desejos humanos*.

De acordo com Lupiana *et al*., espaços inteligentes são altamente integrados quando estão saturados com dispositivos de computação ubíqua e sensores completamente incorporados às redes sem fio. O raciocínio efetivo, por sua vez, pode ser atingido por meio de um mecanismo pseudo-inteligente para o ambiente como um todo, e não somente para dispositivos ou elementos individuais. Já o contexto de usuário refere-se a perfis individuais, políticas, localização atual e *status* de mobilidade. Por fim, a transparência está relacionada às ações humanas e ao suporte a mobilidade sem que, para isso, seja necessária a interação direta deste mesmo usuário.

Espaços inteligentes podem ter diferentes comportamentos, de acordo com as

necessidades dos usuários. Por exemplo, uma casa, repleta de recursos computacionais, pode apresentar - em um dado momento - ajustes para os usuários realizarem suas atividades cotidianas de uma família, ou mesmo, para receber amigos para uma festa.

### 2.1.2 Programação de Espaços Inteligentes

Programar um espaço inteligente é definir uma finalidade/intenção para ele. Para isto, é necessário determinar a estrutura e o comportamento para os elementos do ambiente, entre eles usuários, dispositivos, aplicações e políticas. Na estrutura, criamos os tipos dos elementos que farão parte do espaço inteligente e definimos como eles se relacionarão entre si. Sendo assim, temos papéis do usuário, tipos de objetos inteligentes e tipos de aplicações ubíquas. O relacionamento entre os elementos do espaço inteligente é definido por meio de associações estabelecidas entre eles. Desta forma, diferentes finalidades de espaços inteligentes podem ser definidas para um ambiente em que os relacionamentos são criados de maneira distinta entre seus elementos, ou seja, para um mesmo conjunto de elementos, podemos ter diferentes programações do espaço inteligente, uma vez que as associações entre estes elementos definem como será a interação entre cada um deles.

No comportamento, definimos as ações que serão realizadas pelos elementos do ambiente diante das mudanças de contexto que neles ocorrerem. Por exemplo, quando um usuário entra em um ambiente, este espaço inteligente pode executar ações responsáveis por ligar dispositivos e iniciar aplicações, além de diversas outras ações. Sendo assim, as políticas são responsáveis por dirigir o comportamento dinâmico do ambiente e também por definir uma finalidade para o ambiente.

A programação de espaços inteligentes tem por objetivo definir configurações para os dispositivos, de tal forma que eles possam ser integrados ao ambiente [62] e possam interagir com os usuários e executar um conjunto de aplicações. Quando um novo objeto inteligente entra em um ambiente, é necessário que operações sejam realizadas, integrando-o ao espaço inteligente de forma a ser utilizado pelos usuários. Essa configuração consiste em habilitar os sensores e atuadores do dispositivo, disponibilizar as aplicações para uso, e permitir que ele possa interagir com os demais objetos inteligentes e usuários do ambiente [79]. Disponibilizar as aplicações para uso consiste de permitir que elas possam ser executadas no dispositivo e que este objeto inteligente consiga manipular um conjunto de operações relacionadas a essas aplicações. Por exemplo, um determinado espaço inteligente pode ter como requisito a inicialização de determinadas aplicações assim que o usuário entra no ambiente. Desta forma, o fato de uma aplicação estar instalada no objeto inteligente não significa que ela está pronta para uso. Isso ocorre, pois também necessário que o dispositivo consiga manipular um conjunto delimitado de

operações destas aplicações.

A programação de um espaço inteligente deve ser feita de maneira adaptável, em que o usuário pode definir uma finalidade para aquele ambiente físico, e depois disso, redefini-la para o mesmo ambiente. Uma sala - em dado momento - comporta-se como sala de aulas, enquanto em outra situação, ela pode se comportar como sala de reuniões. Desta forma, temos o reaproveitamento dos recursos do ambiente, podendo portanto, definir uma série de comportamentos distintos para ele. Um ambiente físico é caracterizado pelo conjunto de usuários, objetos inteligentes e aplicações ubíquas em execução nele. Já os espaços inteligentes definem configurações para estes ambientes físicos, que podem se comportar das mais diversas maneiras, de acordo com o interesse do programador do ambiente de computação ubíqua.

## 2.2 Modelos em Tempo de Execução

Ambientes de computação ubíqua - em particular espaços inteligentes - possuem alto grau de dinamismo relacionado aos recursos, à conectividade de rede, às plataformas de *hardware* e *software*, aos requisitos dos usuários (incluindo QoS) e ao contexto, entre outros. A capacidade de adaptação dinâmica desses ambientes, sem a necessidade de parar, reconfigurar ou reinicializar o sistema, é uma das motivações para a utilização da reflexão computacional [89, 14, 87].

Modelos em tempo de execução (*Models@Run.Time* - M@RT) [16, 43, 83] estendem a aplicabilidade dos métodos desenvolvidos na engenharia dirigida por modelos (*Model-Driven Engineering* - MDE) [112] para ambientes em tempo de execução. Para a abordagem M@RT, modelos e metamodelos são entidades de primeira classe, disponíveis em tempo de execução.

Modelo em tempo de execução é uma auto-representação causalmente conectada associada a um sistema, que enfatiza sua estrutura, comportamento ou objetivos, na perspectiva do espaço do problema [16]. Neste caso, os modelos representam e são ligados ao sistema, de tal forma que constantemente *espelham* o atual estado e comportamento desse sistema. Por exemplo, se o sistema mudar - suas representações também devem muda - e vice-versa. Sendo assim, os modelos são um caminho pelo qual desenvolvedores e outros agentes podem compreender, configurar e modificar a estrutura e o comportamento de um sistema em execução [47]. Os modelos em tempo de execução estendem a aplicabilidade das técnicas da engenharia dirigida por modelos para ambientes em tempo de execução [16]. Existe uma grande variedade de dimensões que vêm sendo investigadas pela área de modelos em tempo de execução [16]:

- **Estrutura *e* Comportamento**: os modelos [114] tendem a se concentrar na estrutura e em aspectos comportamentais do sistema. Modelos estruturais enfatizam a

forma como um software é construído. Os modelos comportamentais preocupam-se com a execução do sistema, podendo ser representados em termos de fluxos de eventos;

- **Procedural *e* Declarativa**: os modelos procedurais refletem a estrutura ou comportamento do sistema, detalhando ações a serem executadas ou a constituição estrutural do sistema. Os modelos declarativos, por sua vez, descrevem a tarefa a ser executada sem se preocupar com detalhes de como o modelo será executado pelo mecanismo de execução de modelos;
- **Funcional *e* Não-Funcional**: de uma forma geral, a construção de modelos é baseada nas funcionalidades do sistema. Entretanto, existe também a necessidade de que modelos capturem e representem características não-funcionais. Por exemplo, modelos em tempo de execução podem ser usados para otimizar aspectos da execução do sistema;
- **Formal *e* Informal**: modelos podem ser baseados em formalismos matemáticos ou podem ser derivados de modelos de programação ou abstrações específicas de domínio. Modelos formais têm como vantagem dar suporte ao raciocínio automatizado sobre o estado e o comportamento do sistema. Entretanto, eles podem não capturar ou expressar adequadamente os conceitos do domínio de uma forma simples e acessível aos usuários finais. Modelos informais, por outro lado, são mais acessíveis aos usuários, porém tendem a ser mais inconsistentes e incompletos, se comparados aos modelos formais.

Modelos em tempo de execução são uma evolução da reflexão. Um sistema é reflexivo se mantém uma auto-representação, causalmente conectada com seu estado e comportamento, sendo capaz de refletir mudanças no sistema computacional [88]. A auto-representação permite que a estrutura interna e o comportamento de um sistema possam ser inspecionados e adaptados.

Assim como ocorre com os modelos de desenvolvimento tradicionais, modelos em tempo de execução fornecem apoio ao raciocínio sobre o sistema. Modelos em tempo de execução também podem ser utilizados para dar suporte ao monitoramento e controle do estado dinâmico dos espaços inteligentes durante a execução, ou até mesmo para entender os fenômenos comportamentais desses sistemas em tempo de execução.

### 2.2.1 Metamodelos

Um metamodelo é um modelo que representa uma linguagem de modelagem [114, 109, 40, 120]. Metamodelos contém generalizações de determinados domínios que podem ser expressos por meio de linguagens de modelagem. Um modelo é uma

instância de um metamodelo. Modelos são, portanto, formalizados através de linguagens de modelagem, cujas construções são definidas por um metamodelo.

Linguagens de modelagem, assim como linguagens formais, são definidas por sua sintaxe e sua semântica. A sintaxe de uma linguagem de modelagem é dividida em sintaxe concreta, que é representada por uma notação textual ou gráfica, e sintaxe abstrata, representada pelos conceitos disponíveis na linguagem e seus relacionamentos. A semântica também é tratada de forma separada, como semântica estática, que define critérios para que os modelos sejam considerados válidos e a semântica dinâmica, que define os significados dos elementos de um modelo no momento em que estão sendo executados. De acordo com *Thomas et al.* [125], a sintaxe abstrata e a semântica estática de uma linguagem de modelagem são formalizadas por meio de um metamodelo.

Um metamodelo também é um modelo. Logo, ele também é construído por meio de uma linguagem de modelagem. Esta metalinguagem é definida através de um metamodelo, que neste caso é um meta-metamodelo. O *Object Management Group* (OMG) padronizou a construção de modelos e metamodelos através da arquitetura de modelagem chamada *Meta-Object Facility* (MOF) [94]. A especificação MOF define sua arquitetura em quatro camadas, em que os elementos em uma determinada camada são definidos como instâncias de elementos da camada superior. A Figura 2.1 ilustra esta arquitetura, em que cada elemento da camada inferior é uma instância de um elemento da camada superior.

A camada superior M3 integra a especificação da MOF e representa o meta-metamodelo, que é utilizado para construção de metamodelos. Esta camada é denominada de modelo MOF. O modelo da camada M3 permite construir metamodelos que descrevem a sintaxe abstrata e a semântica estática das linguagens de modelagem [125]. Este modelo formaliza suas próprias abstrações, o que elimina a necessidade de um nível superior.

A camada M2 contém os metamodelos, que podem ser utilizados para criar modelos de determinados domínios. Como exemplo, podemos citar a UML e a *Common Warehouse Metamodel* (CWM), que também foram padronizadas pela OMG. A camada M1 da MOF é composta por modelos que descrevem sistemas usando as definições presentes no nível M2. Finalmente, a camada M0 possui os elementos que formam o sistema em execução, e que são criados a partir das definições presentes em M1.

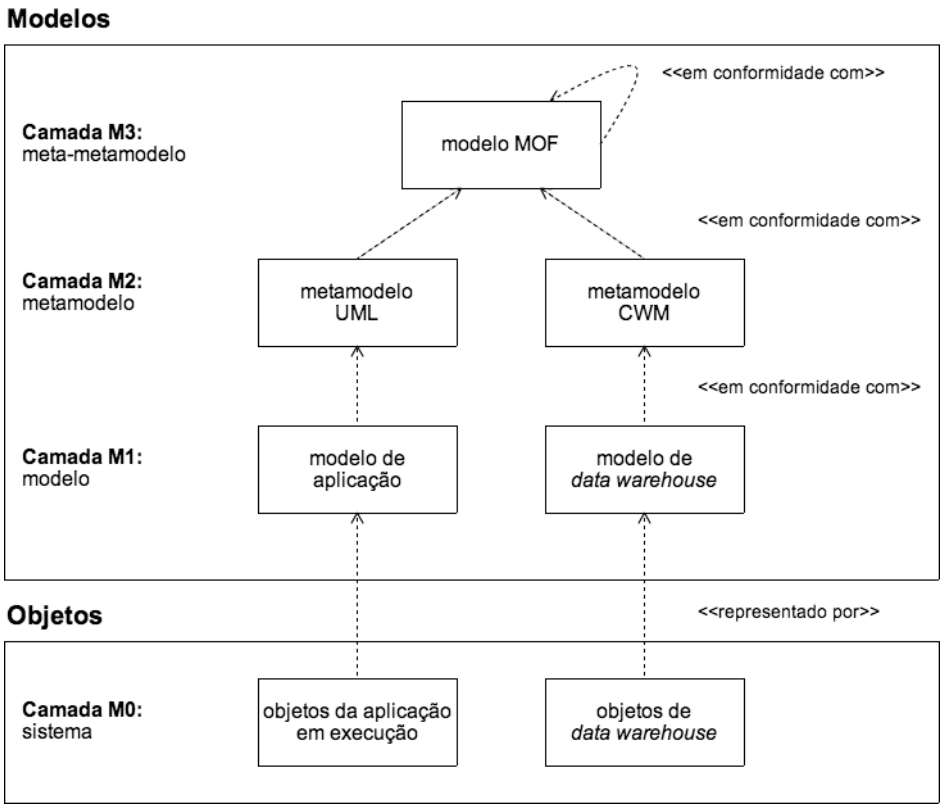



**Figura 2.1:** *Arquitetura de metamodelagem da MOF (figura adaptada de [93])*

### 2.2.2 Linguagens Específicas de Domínio

Linguagens Específicas de Domínio (*Domain-Specific Languages* - DSL) são linguagens de programação ou especificação projetadas com a finalidade de prover abstrações para a resolução de problemas em um determinado domínio [46, 128, 130]. Além disso, elas são projetadas para ser úteis para um conjunto específicos de tarefas [121, 136]. Diferentemente de linguagens de propósito geral, como, Java, C++, e C# que são empregadas para um grande conjunto de tarefas em domínios variados, as DSLs normalmente são restritas a um conjunto limitado de tarefas. Sendo assim, as linguagens específicas de domínio apresentam um maior poder de expressividade na solução de problemas em seu domínio específico [90, 129, 74]. A utilização de conceitos e notações adequadas, próximas ao domínio do problema, permitem que especialistas de domínio ou até mesmo usuários finais sejam capazes de construir aplicações.

Domínios, para DSLs, podem ser considerados áreas de interesse delimitadas. Domínios podem ser caracterizados como técnicos e de negócio [120, 68]. Domínios técnicos são horizontais, como a construção de interfaces de usuário, persistência de dados e comunicação, entre outros. Domínios de negócio são verticais, como seguros, telefonia e comércio varejista, entre outros, e estão relacionados a atividades econômicas e profissionais.

Com a utilização de técnicas de metamodelagem, é possível especificar construções de linguagens que são denominadas Linguagens de Modelagem Específicas de Domínio (*Domain-Specific Modeling Languages* - DSMLs). De acordo com [18, 28], DSLs são linguagens de modelagem representadas de forma textual enquanto DSMLs são representadas de forma gráfica.

Linguagens de modelagem específicas de domínio permitem formalizar estrutura, comportamentos e necessidades dentro de domínios específicos. Criar uma DSML envolve definir um metamodelo (sintaxe abstrata e semântica estática), uma ou mais sintaxes concretas, e uma semântica dinâmica [120, 25, 51].

A sintaxe abstrata define os conceitos do domínio, os relacionamentos entre eles e restrições que limitam a criação de modelos. A sintaxe abstrata é o componente mais importante de uma DSML. De acordo com Ferreira *et al*. [42], "é comum encontrar DSMLs sem definições formais de sua semântica ou sem uma representação concreta, mas sua sintaxe abstrata é imperativa". A sintaxe concreta mapeia os conceitos em elementos que são utilizados para expressar os modelos e pode ser textual, gráfica ou uma junção dessas duas.

A semântica estática determina os critérios para a boa formação (*well-formedness*) de uma linguagem e dos modelos definidos por meio dela. Por exemplo, nas linguagens de programação, as variáveis devem ser declaradas e isso não pode ser determinado pela sintaxe abstrata ou concreta. Já a semântica dinâmica define o significado dos modelos a serem executados.

Como ocorre com linguagens de programação, DSMLs podem ser compiladas ou interpretadas [90]:

- DSML Compilada: as construções na DSML são traduzidas para as construções da linguagem base e para chamadas de bibliotecas;
- DSML Interpretada (*interpreted DSML* - *i*-DSML): as construções são identificadas e interpretadas usando uma semântica dinâmica e processadas por uma máquina de execução [21].

DSMLs representam aspectos estruturais e comportamentais de *software* e sistemas [25]. Por exemplo, aspectos estruturais descrevem o significado dos modelos. Já aspectos comportamentais o conjunto de possíveis ações para as situações identificadas. Existem várias abordagens para a descrição da semântica dinâmica de linguagens de modelagem, que podem ser agrupadas nas seguintes categoria [26]:

- Tradução: a semântica da linguagem é definida por meio da tradução de seus conceitos em conceitos de outra linguagem cuja semântica já é estabelecida;

- Operacional: diz como modelos descritos na linguagem são processados. A semântica operacional é incorporada ao interpretador ou mecanismo responsável pelo processamento e execução de modelos;
- Extensão: possibilita a descrição da semântica como uma extensão de outra linguagem;
- Denotacional: seu propósito é associar objetos matemáticos, tais como números ou funções com os conceitos da linguagem.

DSMLs podem ser adaptadas para corresponder à semântica e sintaxe do domínio da aplicação, ao contrário das linguagens de programação de propósito geral, que raramente conseguem expressar os conceitos naturais de uma aplicação e os objetivos reais do projeto. Além disso, DSMLs permitem aos usuários finais descreverem suas próprias aplicações, pois apresentam construções de alto nível e mais de sua experiência.

## 2.3 Máquina de Execução de Modelos

O propósito de uma máquina de execução de modelos é "executar" ou "animar" o modelo, passando por seus estados de evolução, passo-a-passo [20, 48]. Enquanto a sintaxe abstrata e a semântica estática de uma DSML são descritas por um metamodelo, sua semântica dinâmica é geralmente embutida na máquina de execução. Existem diversas ferramentas padronizadas que empregam técnicas de modelagem para a construção de metamodelos. Entretanto, para a definição da semântica dinâmica, essas ferramentas são limitadas [5].

O processamento de modelos em tempo de execução requer, geralmente, da máquina de execução de modelos, capacidades para analisar, negociar e transformar os modelos em diferentes representações, bem como, para sua adaptação em tempo de execução. Apesar das máquinas de execução de modelos serem desenvolvidas para diferentes DSMLs, sua construção apresenta diversas características comuns, como por exemplo, comparar e transformar modelos, avaliar políticas e regras, adaptar em tempo de execução, entre outros.

A Figura 2.2 ilustra a interação de uma máquina de execução de modelos com o modelo desenvolvido pelo usuário, as políticas definidas para o ambiente e as mudanças de contexto que nele ocorrem, o modelo em tempo de execução e o sistema gerenciado pela máquina. Os modelos são criados pelo usuário e contêm a intenção dele em relação ao sistema. Estes modelos podem ser atualizados gerando uma nova entrada para a máquina.

A máquina de execução interpreta o modelo e gera comandos de controle para o sistema, definindo uma configuração de execução. O sistema, por sua vez, pode gerar eventos e o serviço de contexto sinalizar algumas situações, como por exemplo,

mudanças de contexto. As políticas, assim como situações de contexto, também podem ser modeladas e servem como entrada para a máquina de execução de modelos. Entretanto, elas não são modeladas no modelo do usuário, pois utilizam sua própria linguagem de modelagem. Por exemplo, podem ser utilizadas para modelar as políticas do sistema, regras do tipo *Event-Condition-Action* (ECA) [4]. Além disso, o modelo está causalmente conectado com o sistema modelado, e é mantido e atualizado a partir de situações e mudanças de contexto que acontecem no ambiente.

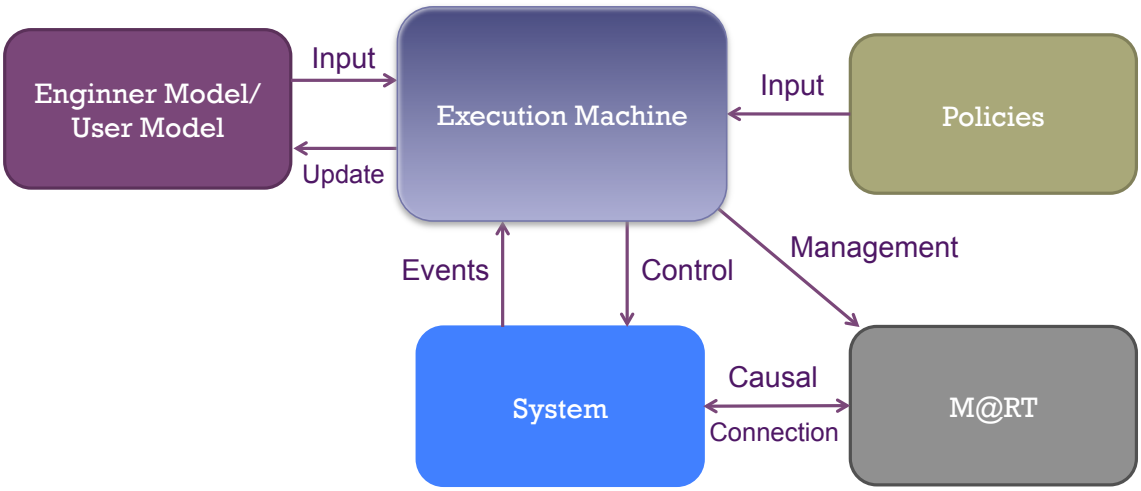



**Figura 2.2:** *Interação da máquina de execução com modelo do usuário, políticas e contexto, modelo em tempo de execução e o sistema*

### 2.3.1 A Máquina Virtual de Comunicação

A Máquina Virtual de Comunicação (*Communication Virtual Machine* - CVM) é uma máquina de execução de modelos para especificação e realização de aplicações de comunicação [35]. Como a CVM é uma máquina dirigida por modelos, ela funciona através do processamento de modelos descritos por meio de uma linguagem específica de domínio, denominada Linguagem de Modelagem de Comunicação (*Communication Modeling Language* - CML). A abordagem utilizada pela CVM não se apoia na geração de códigos, uma vez que a própria CVM é responsável por interpretar os modelos descritos na linguagem CML.

Por meio de um modelo descrito em CML, a CVM é capaz de realizar um serviço de comunicação sem a intervenção do usuário. Além disso, um modelo CML pode ser modificado ao longo da execução, pois a CVM consegue identificar mudanças e adaptar o processo de comunicação de maneira a atender os novos requisitos dos usuários envolvidos. Sendo assim, modelos CML são considerados como modelos de desenvolvimento e de tempo de execução [28].

CML é uma *i*-DSML que possui dois tipos de modelos, um para representar o "programa" (configurações e estruturas de controle) e outro para representar os "dados" (entidades) do domínio. Os termos "programa" e "dados" são utilizados em analogia à tradicional visão de sistemas de computação [99]. A sintaxe abstrata da CML é formada pelo esquema de controle (*control schema*), usado para representar o programa; e pelo esquema de dados (*data schema*), responsável por representar os dados. O esquema de controle representa a configuração da aplicação definida pelo usuário, que consiste em zero ou mais estruturas de controle e tipos de dados do domínio. O esquema de dados contém uma ou mais instâncias de dados para os tipos definidos no esquema de controle. O esquema de controle, juntamente com o esquema de dados, forma um esquema de domínio (*domain schema*). O termo "esquema" é utilizado na CVM para representar modelos *i*-DSML.

A CVM baseia-se em uma arquitetura em camadas, em que cada camada é responsável por realizar uma tarefa na sessão de comunicação. Essas tarefas podem compreender a modelagem da sessão de comunicação, a síntese, o estabelecimento e coordenação da comunicação, entre outros [35]. Um modelo descrito em CML é fornecido pelo usuário (ou aplicação) e é sucessivamente processado e transformado pelas camadas da CVM, de modo que sejam gerados comandos para os componentes que realizam os serviços. A Figura 2.3 apresenta a arquitetura da CVM e suas camadas descritas em seguida.

- Interface de Comunicação com o Usuário (*User Communication Interface* - UCI): representa a interface de interação do usuário ou aplicação com a CVM e provê meios para definição e gerenciamento de modelos em CML. Os usuários solicitam por meio desta camada as sessões de comunicação, descritas em forma de modelo;
- Mecanismo de Síntese (*Synthesis Engine* - SE): é responsável pela negociação de um modelo em CML, com os demais modelos dos outros participantes da sessão de comunicação. O mecanismo de síntese também realiza a transformação do modelo em um *script* de controle de comunicação, que contém a lógica para estabelecer o processo de comunicação. Esses *scripts* são executados pela camada seguinte;
- *Middleware* de Comunicação Centrado no Usuário (*User-centric Communication Middleware* - UCM): tem por objetivo executar o *script* de controle recebido pela camada de síntese e coordenar as sessões de comunicação. Esta camada também trata de políticas de segurança e qualidade de serviço, entre outras, relacionadas à comunicação;
- Intermediador de Comunicação em Rede (*Network Communication Broker* - NCB): esta camada provê uma interface de comunicação independente de tecnologia para ser utilizada pela camada de *middleware*. Sua função é ocultar da camada superior a heterogeneidade e complexidade envolvida na interação entre os componentes

que realizam a comunicação. Logo, a NCB recebe as requisições da camada de *middleware* e intermedia o acesso aos *frameworks* de comunicação, de maneira que realizem a solicitação recebida.

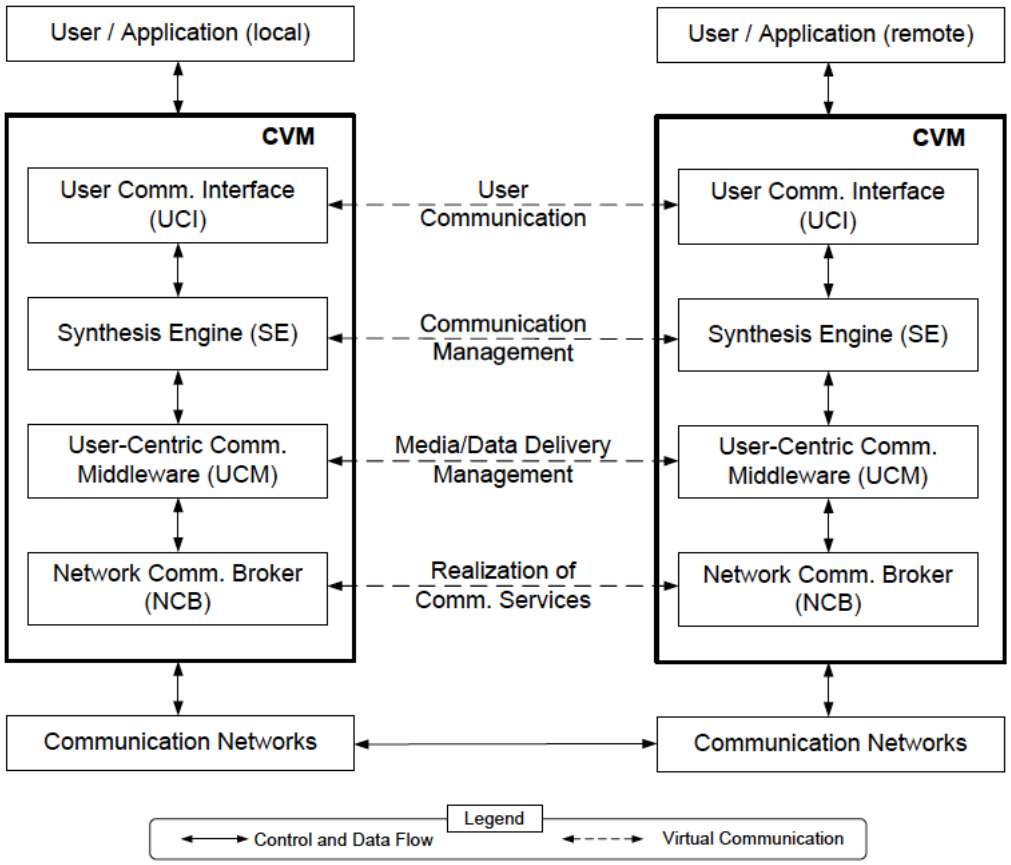



**Figura 2.3:** *Arquitetura em camadas da CVM [35]*

A arquitetura da 2SVM foi baseada na arquitetura da máquina virtual de comunicação CVM, também utilizada na arquitetura de outro domínio de aplicação, o de *Smart-Microgrids* com a MGridVM [8]. Essa arquitetura foi escolhida por apresentar uma maneira efetiva de processar modelos desenvolvidos pelo usuário e lidar modelos em tempo de execução [35, 137, 7, 6].

## 2.4 Conclusão

Este capítulo teve por objetivo apresentar a fundamentação teórica da tese, por meio dos conceitos mais comuns utilizados no trabalho, incluindo motivação e desafios encontrados na área de espaços inteligentes. Inicialmente, o capítulo discorre sobre os aspectos gerais da computação ubíqua, assim como dos espaços inteligentes e programação destes ambientes.

Após isso, o capítulo apresenta as definições dos modelos em tempo de execução, dimensões em que esta abordagem tem sido investigada e sua relação com os sistemas reflexivos. Descrevemos ainda sobre os metamodelos, a arquitetura de metamodelagem MOF (especificação da OMG) e as linguagens específicas de domínio, especificadas a partir de metamodelos. O capítulo expõe ainda o conceito das máquinas de execução de modelos, que têm como propósito executar e animar os modelos desenvolvidos pelos usuários, em particular, a arquitetura da CVM, que foi utilizada como base para a criação da máquina de execução de modelos criada em nosso trabalho.

CAPÍTULO 3

# A Linguagem de Modelagem 2SML

Linguagens específicas de domínio (*Domain-Specific Language* - DSL) têm como objetivo resolver problemas específicos e, por isso, são focadas em domínios particulares [129]. Elas são conhecidas como *little languages*, pois são comparativamente menores que linguagens de propósito geral [127]. Elas possuem foco na expressividade, e quando comparadas com linguagens de propósito geral, fornecem uma solução mais apropriada para domínios de aplicação bem definidos.

Uma DSL, é uma linguagem de programação que herda restrições e premissas de determinado domínio [46]. Em nosso trabalho, fizemos uma diferenciação quanto às categorias de linguagens específicas de domínio: *i)* Linguagens baseadas em texto, denominadas DSLs [119]; e *ii)* Linguagens de modelagem gráficas, comumente chamadas de *Domain-Specific Modeling Language* (DSML).

Neste capítulo, descrevemos a linguagem de modelagem para espaços inteligentes denominada *Smart Spaces Modeling Language* (2SML). Ela permite que usuários especialistas no domínio de espaços inteligentes programem o comportamento de espaços inteligentes sem a necessidade de conhecimentos de programação de mais baixo nível, como aqueles exigidos por linguagens de propósito geral. A criação da linguagem 2SML envolveu a definição de seu metamodelo, composto por sua sintaxe abstrata e sua semântica estática, bem como a definição de sua sintaxe concreta e sua semântica dinâmica.

Este capítulo está organizado como segue: Na Seção 3.1, apresentamos uma visão geral com definições e conceitos sobre a linguagem e sobre os elementos do espaço inteligente. Na Seção 3.2 apresentamos a definição da linguagem de modelagem 2SML e na Seção 3.3 temos a definição do metamodelo da linguagem. Na Seção 3.4 discutimos sobre as políticas e na Seção 3.5, apresentamos exemplos de modelos desenvolvidos com a 2SML. Por fim, na Seção 3.6 temos a conclusão do capítulo.

## 3.1 Visão Geral

### 3.1.1 Definições

Esta seção apresenta as definições necessárias para compreensão de nosso trabalho. Nela, nós definimos os conceitos dos elementos do metamodelo da linguagem de modelagem, entre eles, papel do usuário, objeto inteligente, aplicações ubíquas, políticas e restrições de modelagem.

- Papel do usuário (`UserRole`): é um tipo de usuário que possui características em comum e pode utilizar os mais diversos dispositivos do espaço inteligente para executar suas aplicações. Como característica desse usuário temos o nome do papel definido;
- Objeto inteligente (`SmartObject`): é um tipo de dispositivo que possui um nome e é formado por um conjunto de características, denominadas *features*. Essas *features* possuem um nome, um tipo, uma categoria e uma descrição. O tipo diz se a característica é um hardware ou um software. Já a categoria diz se esta característica é um sensor, um atuador ou entrada/saída. Por fim, a descrição permite definir uma particularidade da *feature*;
- Aplicação ubíqua (`UbiquitousApplication`): é um tipo de aplicação que possui a propriedade ubíqua de mobilidade dentro do espaço inteligente. Esta aplicação, portanto, pode se mover entre os mais diversos objetos inteligentes de forma a acompanhar o usuário em suas atividades no ambiente. As aplicações ubíquas possuem nome, versão, sistema operacional em que estão habilitadas para executarem.

Para lidar com as mais diversas interações entre os elementos do espaço inteligente, nós utilizamos as políticas. Elas dirigem o comportamento dinâmico do espaço inteligente e são descritas por meio de regras *Event-Condition-Action* (ECA) [95, 11], em que, para cada evento, temos uma condição que deve ser avaliada a fim de engatilhar ou não uma ação. Essas políticas possuem níveis de prioridade de execução, de tal forma que algumas políticas devem ser executadas antes de outras. Isso permite lidar também com conflitos entre políticas, de tal forma que a prioridade irá orientar a ordem de execução delas.

Por fim, temos no metamodelo da linguagem as restrições de modelagem, que impedem que determinadas associações entre os elementos definidos no modelo sejam realizadas. Por exemplo, um determinado papel do usuário pode ser impedido de utilizar um tipo de aplicação ubíqua ou de objeto inteligente, a fim de atender as regras de utilização dos recursos do espaço inteligente.

### 3.1.2 Visão Geral dos Conceitos

A fim de definir responsabilidades para os usuários que lidam com a linguagem de modelagem 2SML, bem como para aqueles que participam do espaço inteligente, definimos três tipos de usuários: *i)* Engenheiro do Sistema (ES); *ii)* Usuário do Sistema (US); e *iii)* Usuário do Espaço Inteligente (UEI).

O Engenheiro do Sistema é um usuário especialista em modelagem de espaços inteligentes. Ele é responsável por definir as regras de acesso aos recursos de um espaço inteligente e também por facilitar a criação de modelos pelos usuários do sistema. Neste último caso, ele cria blocos de construção básicos, que são utilizados pelos usuários do sistema para criar novos tipos de `UserRoles`, `SmartObjects` e `UbiquitousApplications`. As regras de acesso aos recursos, que denominamos restrições de modelagem, impedem que o usuário do sistema crie associações entre determinados elementos do modelo. Por exemplo, o engenheiro do sistema pode definir uma restrição de uso dos dispositivos do ambiente *câmera* e *microfone* para usuários visitantes, devido a uma política interna de segurança da organização. Assim, estes usuários tornam-se impedidos de fazer uso destes dispositivos, como, gravar uma atividade da organização por meio de alguma aplicação ubíqua. O ES também é capaz de definir, em seu modelo, políticas que guiam o comportamento dinâmico do ambiente em tempo de execução, ou seja, mudanças de contexto do espaço inteligente são tratadas por meio de políticas. Para fazer isso, o ES utiliza políticas que são criadas por meio de regras ECA, disponibilizadas na linguagem de modelagem 2SML.

O Usuário do Sistema é responsável por programar o espaço inteligente. Ele utiliza os tipos básicos, definidos no modelo do engenheiro do sistema, para assim criar os elementos de seu modelo. Desta forma, no modelo do engenheiro do sistema temos os tipos básicos enquanto no modelo do usuário do sistema temos sub-tipos desses tipos básicos. Por exemplo, o ES pode ter definido em seu modelo o tipo `smartphone`, e o US pode especializar este tipo em um `smartphoneAndroid`. Logo, os elementos do modelo do US são subtipos daqueles definidos no modelo do ES. O US pode também definir em seu modelo políticas ECA, a fim de dirigir o comportamento dinâmico do espaço inteligente.

O Usuário do Espaço Inteligente é aquele presente no ambiente de computação ubíqua, que utiliza seus recursos computacionais pessoais, assim como recursos do ambiente, e ainda interage com os demais elementos presentes no espaço inteligente. Este usuário é uma instância de algum papel de usuário criado pelo engenheiro do sistema ou pelo usuário do sistema em seus modelos.

## 3.2 Definição da 2SML

A definição da linguagem 2SML foi baseada na arquitetura de metamodelagem da OMG [94], descrita no Capítulo 2. Ela é definida no nível M2 e seu uso é feito através de elementos no nível M1. A Figura 3.1 ilustra a definição da linguagem de modelagem 2SML.

Na camada M2, temos o metamodelo da 2SML, composto por sua sintaxe abstrata e sua semântica estática. Na sintaxe abstrata, temos a definição dos conceitos da linguagem, seus relacionamentos e restrições. Já na semântica estática temos os critérios que um modelo sintaticamente correto deve satisfazer para que seja possível definir um significado para ele. Nesta camada, temos um único metamodelo dividido em três partes: *Core-Metamodel*, *System-Engineer Metamodel* e *System-User Metamodel*.

O *Core-Metamodel* apresenta elementos comuns aos metamodelos do engenheiro e do usuário do sistema. Portanto, tanto o *System-Engineer Metamodel* quanto o *System-User Metamodel* fazem referência ao *Core-Metamodel*.

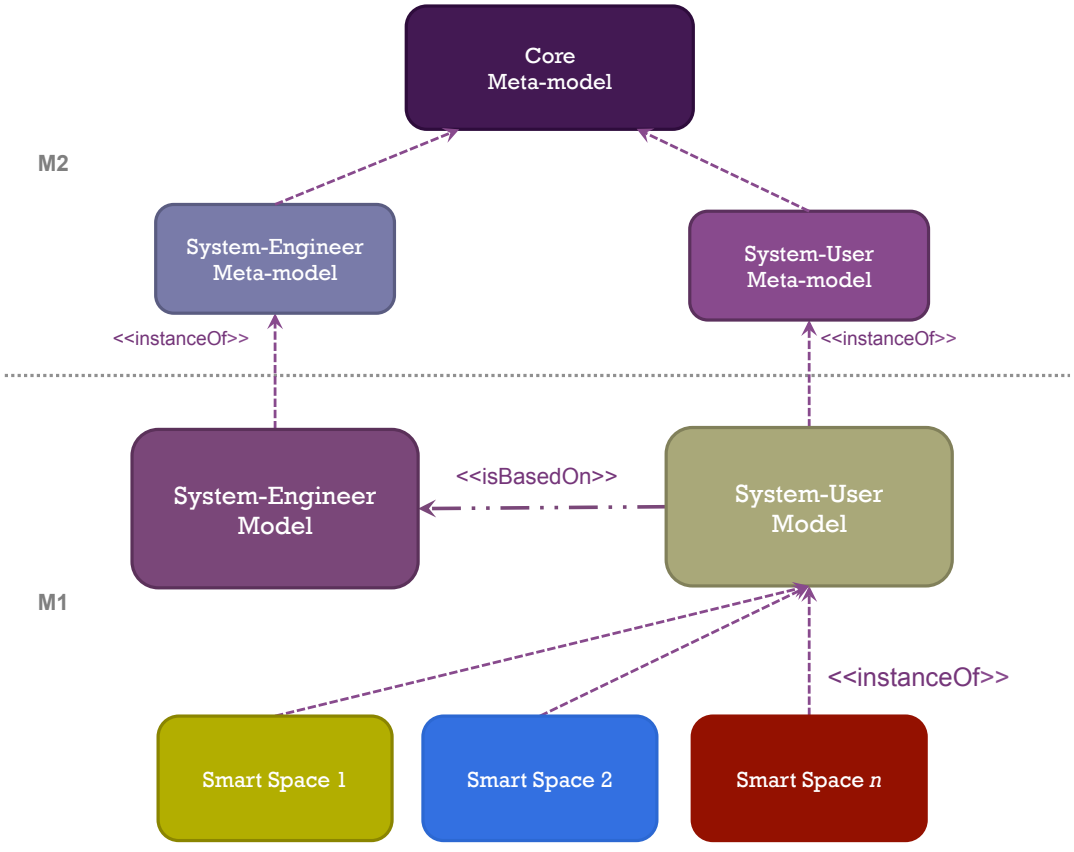



**Figura 3.1:** *Definição da linguagem da 2SML com base na arquitetura de metamodelagem*

Na camada M1 da arquitetura de metamodelagem da 2SML, temos os modelos desenvolvidos pelo engenheiro e pelo usuário do sistema, bem como suas instâncias. Esses modelos contêm a programação do espaço inteligente, que é feita através da definição de

tipos de usuários, *smart objects* e aplicações ubíquas, além de políticas que lidam com aspectos dinâmicos do ambiente, como o tratamento de mudanças de contexto. O modelo do usuário é baseado no modelo do engenheiro do sistema, uma vez que: *i)* deve obedecer as restrições de modelagem impostas pelo modelo do engenheiro; e *ii)* os elementos do modelo do usuário devem ser subtipos daqueles tipos definidos no modelo do engenheiro do sistema. Apenas o modelo do usuário é diretamente instanciável, haja visto que ele será o modelo em execução no espaço inteligente.

Por fim, temos as instâncias dos modelos do usuários, que contêm informações sobre as instâncias dos elementos do espaço inteligente que estão em execução. Dentre essas informações, temos: o usuário e seu conjunto de informações, os recursos dos objetos inteligentes, e meta-dados das aplicações, tais como nome, versão e plataforma.

## 3.3 Definição do Metamodelo da 2SML

De acordo com [125], a sintaxe abstrata e a semântica estática de uma linguagem de modelagem são formalizadas por meio de um metamodelo. O metamodelo da 2SML incorpora conhecimentos e requisitos que são específicos do domínio de espaços inteligentes e define os elementos que são necessários para a programação de ambientes desse tipo.

O metamodelo da 2SML possui metatipos que permitem ao engenheiro e ao usuário do sistema definir em tipos de recursos, tais como objetos inteligentes e aplicações, além de papéis de usuários e políticas. A criação do metamodelo da 2SML foi dividida em três partes: *Core-Metamodel*, *System-Engineer Metamodel* e *System-User Metamodel*.

### 3.3.1 Core-Metamodel

O *Core-Metamodel* é a base para os metamodelos do engenheiro e usuário do sistema. A Figura 3.2 apresenta o *Core-Metamodel*.

Os elementos presentes no *Core-Metamodel* são metatipos da linguagem de modelagem e permitem tanto ao engenheiro quanto ao usuário do sistema criarem seus próprios tipos. Os elementos presentes no metamodelo *Core-metamodel* são:

- `UserRole`: este metatipo permite aos engenheiros e usuários do sistema definirem os tipos de papéis de usuário que poderão participar do espaço inteligente e como atributo, temos o nome do papel do usuário. Alguns exemplos de papéis do usuário incluem: `professor` e `student`, para o cenário de sala de aula; e `curator`, `guide` e `guest`, para o cenário de um museu. O metatipo `UserRole` está associado ao metatipo `UbiquitousApplication` e, neste caso, indicamos que um

papel de usuário pode usar uma aplicação ubíqua. Esta associação, representada pelo metatipo `canUse` diz que um usuário em um dado momento, pode usar uma dentre as inúmeras aplicações existentes em seu objeto inteligente. Temos ainda dois possíveis tipos de associação entre o `UserRole` e `SmartObject`, em que o papel do usuário pode usar um objeto inteligente (`canUse`), ou é dono (`isOwnerOf`) dele. Quando a associação criada entre eles é do tipo `canUse`, estamos dizendo que o objeto inteligente pertence ao ambiente, ou seja, pode ser compartilhado entre todos os usuários do espaço inteligente. Quando a associação é do tipo `isOwnerOf`, temos que o objeto inteligente é de uso pessoal, ou seja, pertence exclusivamente ao usuário associado a ele;

- `UbiquitousApplication`: possibilita a criação de tipos de aplicações ubíquas. Por exemplo, podemos ter em um espaço inteligente de sala de aula, aplicações de *slides*, editores de texto, aplicações de compartilhamento de conteúdo, e assim por diante. Cada aplicação ubíqua tem um conjunto de comportamentos, que permitem que ela seja manipulada pela máquina de execução de modelos, entre eles: iniciar, pausar, continuar, salvar estado e finalizar aplicação. Estes comportamentos permitem que a máquina de execução de modelos manipule as aplicações ubíquas e tenha controle sobre as operações que nela podem ser realizadas no espaço inteligente. Cada aplicação ubíqua é associada ao elemento do modelo `UserRole`, e neste caso, este usuário pode utilizar a aplicação em seu objeto inteligente. Não há uma associação entre aplicação ubíqua e objeto inteligente, uma vez que todas as aplicações ubíquas podem ser executadas nos objetos inteligentes do usuário. Para isso, é necessário que as aplicações que executarão no ambiente de computação ubíqua implementem a interface ubíqua, composta por estas operações. Como atributos de uma aplicação ubíqua, temos:
  - `ubiAppName`: atributo correspondente ao nome da aplicação ubíqua;
  - `version`: versão da aplicação ubíqua;
  - `operatingSystem`: sistema operacional em que a aplicação ubíqua pode executar.
- `SmartObject`: este metatipo proporciona criar tipos de dispositivos que poderão participar do espaço inteligente. Ele é composto do metatipo `Feature`, que possibilita ao engenheiro ou usuário do sistema definir um conjunto de características para o dispositivo. Por exemplo, para o tipo `smartphone`, podemos ter como características o tipo do sistema operacional desejado, *display*, sensores, atuadores, entre outros;
- `Feature`: as *features* são metatipos que permitem criação de elementos específicos para os objetos inteligentes, em particular, elementos de *software* e *hardware*. De forma a definir como cada um dessas características são criadas, nós definimos os

seguintes elementos para cada uma delas:

- `featureName`: este atributo corresponde ao nome da característica;
- `featureType`: permite definir uma característica como `SOFTWARE` ou `HARDWARE`;
- `featureCategory`: possibilita definir uma característica como sendo um dispositivo de entrada e saída (`IO_RESOURCE`), dispositivos de monitoramento (`SENSOR`), e atuadores, (`ACTUATOR`);
- `featureDescription`: este atributo permite definir uma particularidade da característica. Por exemplo, para a `Feature` do tipo `display`, podemos definí-la como `retinaDisplay`, ou seja, uma característica bem particular desse objeto inteligente.

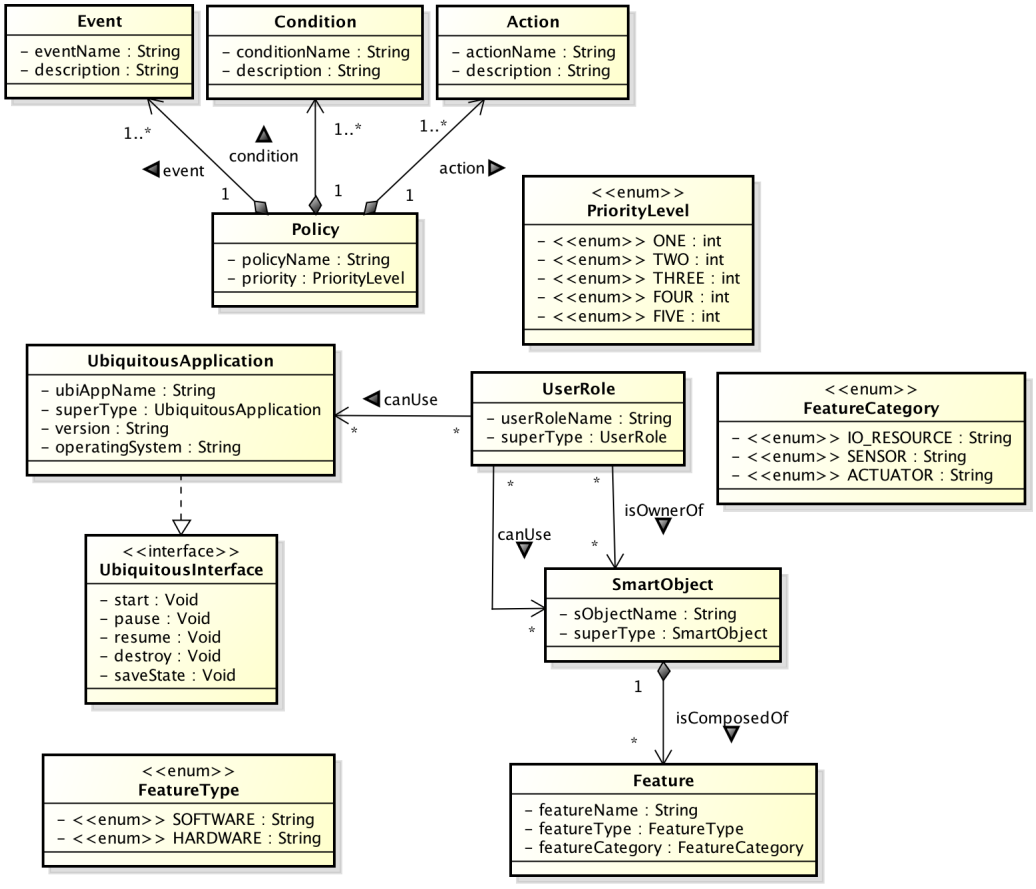



**Figura 3.2:** *Core-Metamodel da 2SML*

- `Policy`: este metatipo possibilita ao engenheiro ou usuário do sistema criar políticas que irão lidar com o comportamento dinâmico do espaço inteligente. Como atributos de `Policy`, temos o nome da política, e seu nível de prioridade, em que o nível 1 possui maior prioridade para execução da política, e o nível 5, menor prioridade. Para criar essas políticas, utilizamos regras *Event-Condition-Action*:

- `Event`: estes eventos são definidos pelo engenheiro ou usuário de sistema em seus modelos. Para criar cada um deles, é necessário que seja definido seu nome e uma descrição do tipo de evento. Essa descrição deve conter o tipo do evento, escrito na forma *substantivo* + *tipo de evento*, como por exemplo, *changeTemperature* ou na forma *tipo evento* + *substantivo*, como por exemplo, *smokePresent*;
- `Condition`: uma condição diz respeito a uma situação ou circunstância. Caso a condição seja avaliada como verdadeira, a ação será executada. Para descrever uma condição, usamos os tipos definidos no modelo, além de números inteiros ou de ponto flutuante. Por exemplo, para o evento de mudança de temperatura, temos: $temperature > 30$;
- `Action`: a ação é a execução de alguma operação no espaço inteligente. Estas ações são executadas pela máquina de execução de modelos a partir das aplicações ubíquas que estão em execução no espaço inteligente. Para fazer isso, nós definimos um conjunto de possíveis ações que podem ser executadas. O usuário, em seu modelo, seleciona alguma dessas ações que são engatilhadas a partir de um evento gerado e da avaliação da condição. Por exemplo, para o evento *"changeTemperature"*, condição *"temperature* $> 25$*"*, podemos ter a ação *"startApplication(climatizationApp, airConditioning)"*. Neste caso, a aplicação ubíqua de climatização é inicializada nos dispositivos do tipo ar-condicionado de tal forma a configurá-los para regular a temperatura do espaço inteligente.

### 3.3.2 System Engineer-Metamodel

A Figura 3.3 apresenta o metamodelo do engenheiro do sistema. Ele contém todos os elementos presentes no *Core-Metamodel*, e ainda o metatipo `ModelingRestrictions`. As restrições de modelagem, permitem ao engenheiro do sistema limitar o conjunto de possibilidades de modelagem para o usuário do sistema, de tal forma a atender ao conjunto de requisitos da organização. Portanto, uma organização estabelece o conjunto de restrições quanto ao acesso aos recursos do espaço inteligente, e o engenheiro do sistema implanta essas restrições a partir do metatipo `ModelingRestrictions`.

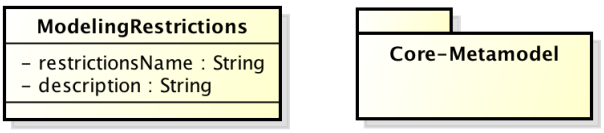



**Figura 3.3:** *Metamodelo do Engenheiro do Sistema*

É importante ressaltar que as restrições de modelagem são criadas no modelo do engenheiro, mas aplicadas apenas no modelo do usuário do sistema, impedindo que ele defina associações entre determinados elementos de seu modelo. As restrições de modelagem foram criadas com o intuito de atender aos requisitos das organizações que têm necessidade de controlar o acesso aos recursos do espaço inteligente.

Desta forma, elas impedem que o usuário do sistema crie associações entre determinados tipos de entidades, o que na prática, impossibilita que alguns usuários utilizem certos tipos de dispositivos e aplicações. Por exemplo, o engenheiro do sistema pode definir uma restrição de modelagem que impede o usuário do sistema de associar o papel do usuário `guest` ao dispositivo do ambiente do tipo `display`, em virtude de uma noma interna da organização. Por exemplo, o *display* é utilizado para exibir propagandas da organização para seus visitantes.

Essas restrições são definidas no modelo do engenheiro do sistema, encarregado pela organização de gerenciar o acesso aos recursos do espaço inteligente. Para criá-las, definimos um conjunto de conceitos, ilustrados na Figura 3.4.

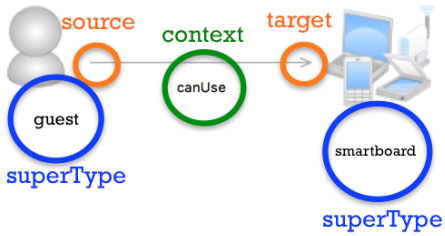



**Figura 3.4:** *Definição das Restrições de Modelagem*

As restrições de modelagem possuem um ou mais elementos do modelo denominados *source*, e um ou mais elementos denominados *target*. *Source* indica uma referência para o supertipo do elemento que é o ponto de partida da associação. O elemento *context* representa as associações entre *source* e *target*, entre elas `hosts`, `canUse`, `isOwnerOf`, `has`, e assim por diante; e em *target*, temos o supertipo do elemento que será o alvo da restrição. Por exemplo, para o engenheiro do sistema definir uma restrição de modelagem que impede que usuários do supertipo `guest` utilizem objetos inteligentes do supertipo `smartphone`, ele deve especificar a seguinte restrição:

**context (canUse):**

```
source.superType == "student"and target.superType == "smartboard"
```

As restrições de modelagem podem ser criadas entre os elementos do modelo que têm associações no metamodelo do engenheiro do sistema. Assim como foi dito anteriormente, o ponto de partida da associação é denominado *source* e o ponto de chegada, *target*. Como exemplo, podemos citar as associações entre papéis do usuário

e objetos inteligentes, que tem como *source*, `userRole`, como *target* `smartObject`, e como *context*, `canUse`.

### 3.3.3 System User-Metamodel

O metamodelo do usuário do sistema é ilustrado na Figura 3.5. Assim como o metamodelo do engenheiro, ele também possui todos os elementos pertencentes ao *Core-Metamodel*, adicionando o metatipo `SmartSpace`, assim como outros tipos de associações e atributos.

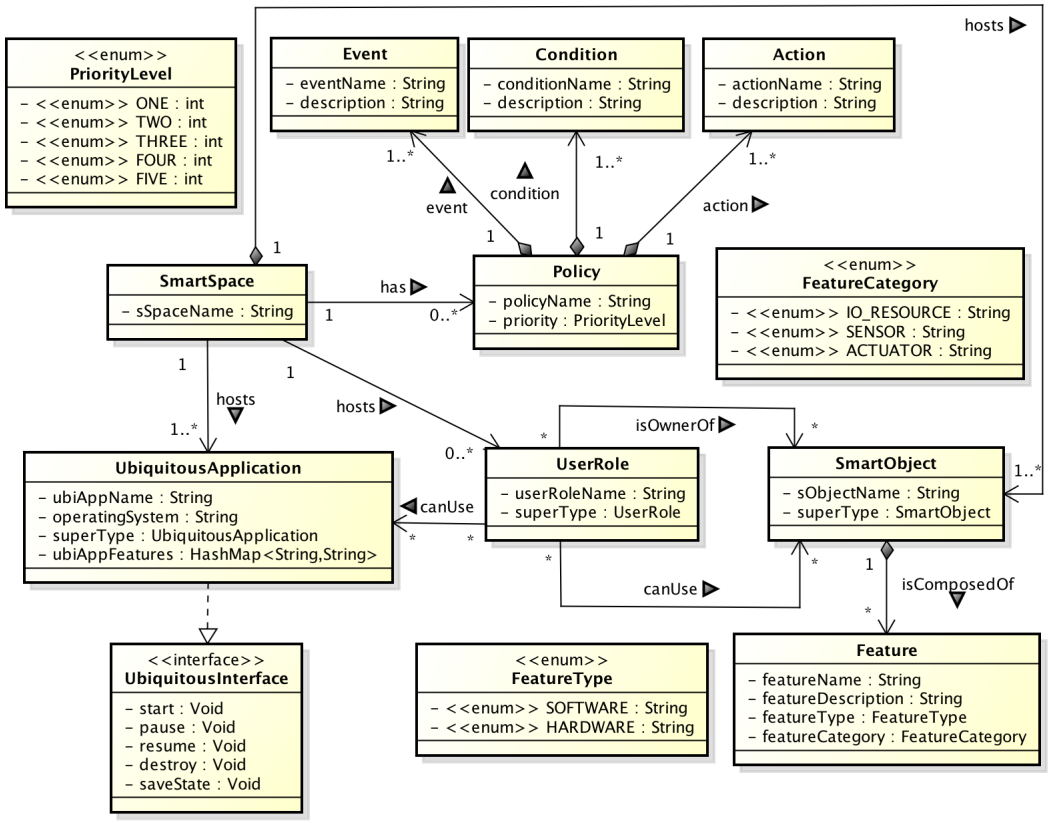



**Figura 3.5:** *Metamodelo do Usuário do Sistema*

O metatipo *SmartSpace* possui como atributo apenas o nome do espaço inteligente, e tem como função agrupar todos os elementos que farão parte do ambiente de computação ubíqua. Para cada um desses elementos, `UserRole`, `SmartObject` e `UbiquitousApplication`, devemos criar uma associação do tipo `hosts`, assinalando que `SmartSpace` hospeda zero ou mais desses elementos. O motivo de existir essas associações com os elementos do espaço inteligente é permitir que, futuramente, a máquina possa processar dois ou mais espaços inteligentes com recursos e usuários compartilhados entre eles. Para o metatipo `Policy`, temos a associação `has`, indicando que o espaço inteligente possui zero ou mais políticas.

## 3.4 Políticas

Políticas têm o objetivo de dirigir o comportamento dinâmico dos elementos do espaço inteligente. Elas são modeladas através de regras *event-condition-action* e devem ser criadas apenas com os elementos pertencentes à parte estrutural do modelo, haja visto que elas se referem apenas aos elementos que poderão fazer parte do espaço inteligente.

Por meio de políticas, engenheiro e usuário do sistema conseguem tratar todos os eventos de interesse que acontecem no espaço inteligente, desde aqueles ocorridos em nível de hardware, quanto software. A Tabela 3.1 apresenta alguns exemplos de tipos de eventos que podem ser tratados pela 2SVM.

**Tabela 3.1:** *Tabela de possíveis Eventos*

| **Tipo do Evento** | **Parâmetros** | **Descricão** |
|---|---|---|
| `changeLocation` | `userRole`, `userID`, `smartObjectType`, `smartObjectID`, `location` | Notifica sobre a mudança de localização do usuário/dispositivo no ambiente. |
| `changeBatteryLevel` | `smartObjectType` | Notifica sobre o nível de bateria de determinado tipo de *smart object* |
| `smokePresence` | - | Notifica sobre a presença de fumaça no ambiente. |
| `gasPresence` | - | Notifica sobre a presença de gás no ambiente. |
| `changeNoiseLevel` | - | notifica sobre o nível de ruído do ambiente |
| `changeUserTemperature` | `userRole`, `userID` | Notifica sobre a temperatura corporal do usuário. |
| `changeAmbientTemperature` | `smartObjectType`, `smartObjectID` | Notifica sobre a temperatura do ambiente. |
| `changeUserPressureLevel` | `userRole`, `userID` | Notifica sobre a pressão arterial do usuário. |
| `changeDate` | `userRoleType`, `smartObject`, `date` | Notifica sobre determinada data, incluindo horário, dia, mês e ano. |
| `entryIntoSmartSpace` | `userRoleType`, `smartObject` | Notifica sobre entrada de usuário/objeto inteligente no ambiente de computação ubíqua. |

Em relação às condições, elas podem conter uma ou mais expressões, sendo que, quando houver mais de uma, elas devem ser conectadas através de operadores lógicos do tipo `and`, `or`, `xor` e `not`. Todas elas devem fazer referência aos elementos do modelo e/ou às instâncias dos elementos do espaço inteligente, usuário, objetos inteligentes ou aplicações ubíquas, para que sejam consideradas válidas. Por exemplo, para avaliar o

nível de bateria de um determinado objeto inteligente utilizado por um tipo de papel do usuário, devemos escrever a condição da seguinte forma:

**Evento:** `changeBatteryLevel(<smartObjectType>)`

**Condição:** `if (changeBatteryLevel(<smartObjectType>) < 0.1 ) and (<userRole> == "student")`

Por fim, as ações indicam o procedimento que deve ser realizado caso o resultado da avaliação da condição seja positivo. A Tabela 3.2 ilustra possíveis ações que podem ser utilizadas pelo engenheiro e usuário do sistema em seus modelos. Estas ações foram definidas na linguagem e para que outras ações possam ser executadas no ambiente, elas também devem ser especificadas.

**Tabela 3.2:** *Tabela de possíveis Ações*

| Tipo da Ação | Parâmetros | Descricão |
|---|---|---|
| `moveUbiApp` | `ubiAppType`, `ubiAppID`, `smartObjectType`, `smartObjectID` | move uma aplicação ubíqua de um *smart object* para outro |
| `notifyUser` | `userRole`, `userRoleID` | notifica com uma mensagem de alerta um usuário de determinado tipo de papel |
| `startApplication` | `ubiAppType`, `ubiAppID`, `smartObjectType`, `smartObjectID` | inicia a execução de uma aplicação ubíqua de determinado tipo em determinado objeto inteligente |
| `pauseApplication` | `ubiAppType`, `ubiAppID`, `smartObjectType`, `smartObjectID` | suspende por determinado tempo uma aplicação ubíqua de determinado tipo em determinado objeto inteligente |
| `resumeApplication` | `ubiAppType`, `ubiAppID`, `smartObjectType`, `smartObjectID` | recupera o estado de execução uma aplicação ubíqua de determinado tipo, após ter sido suspensa |
| `destroyApplication` | `ubiAppType`, `ubiAppID`, `smartObjectType`, `smartObjectID` | finaliza uma aplicação ubíqua de determinado tipo em determinado objeto inteligente |

## 3.5 Exemplos de Modelos em 2SML

Para ilustrar o uso da linguagem de modelagem 2SML, esta seção apresenta um modelo para o cenário da sala de aula. A Figura 3.6 ilustra este cenário.

*Descrição do cenário de sala de aula*: Ao iniciar a aula, os *slides* do professor são carregados de seu *smartphone* para a lousa inteligente, no momento em que ele se aproxima dela. O professor apresenta o conteúdo da aula aos alunos por meio da aplicação

de *slides* que está executando na lousa. Os *slides* são copiados da lousa inteligente para cada um dos dispositivos dos alunos por meio de uma aplicação de compartilhamento, para que eles possam acompanhar a explicação do professor, bem como fazer suas anotações. Após a apresentação, os alunos iniciam as atividades que incluem perguntas sobre o conteúdo ministrado. Para respondê-las, eles utilizam uma aplicação de editor de textos, e ao final da aula, alguns alunos são selecionados para apresentar suas respostas para os demais alunos na lousa inteligente.

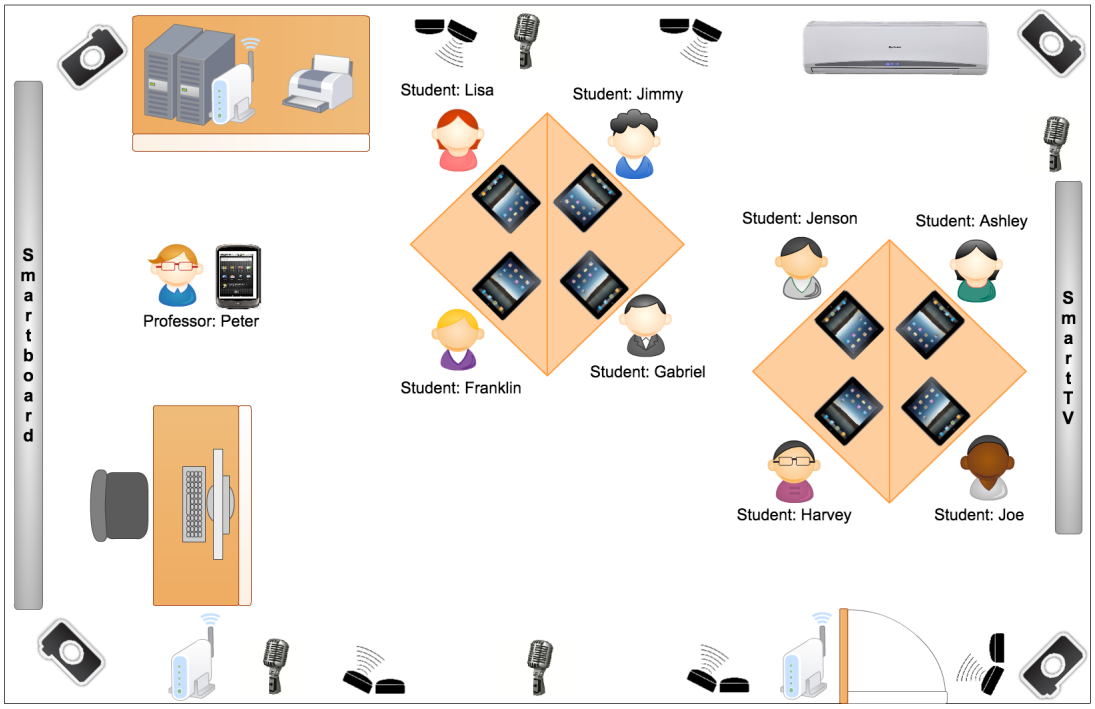



**Figura 3.6:** *Cenário de Sala de Aula*

A Figura 3.7 apresenta o modelo do engenheiro do sistema para este ambiente da sala. Nele, o engenheiro definiu um conjunto de elementos que serão utilizados como base para a criação do modelo do usuário do sistema. É importante ressaltar que esta sala poderá ter uma diversidade de comportamentos distintos, de acordo com o modelo do usuário do sistema. Para este exemplo, o usuário modelou o ambiente para se comportar como uma sala de aula.

Os tipos de papéis do usuário definidos no modelo do engenheiro do sistema para o espaço inteligente da sala são `professorUser`, `studentUser`, `guestUser` e `administrativeTechnicianUser`. Para os objetos inteligentes, temos `smartphoneDevice`, `smartTVDevice`, `airConditioningDevice`, `printerDevice`, `smartboardDevice`, `rfidDevice`, `lightSensor`, `cameraDevice`, `tabletDevice`, `smartWatchDevice`, `notebookDevice`, `printerDevice` e `microphoneDevice`.

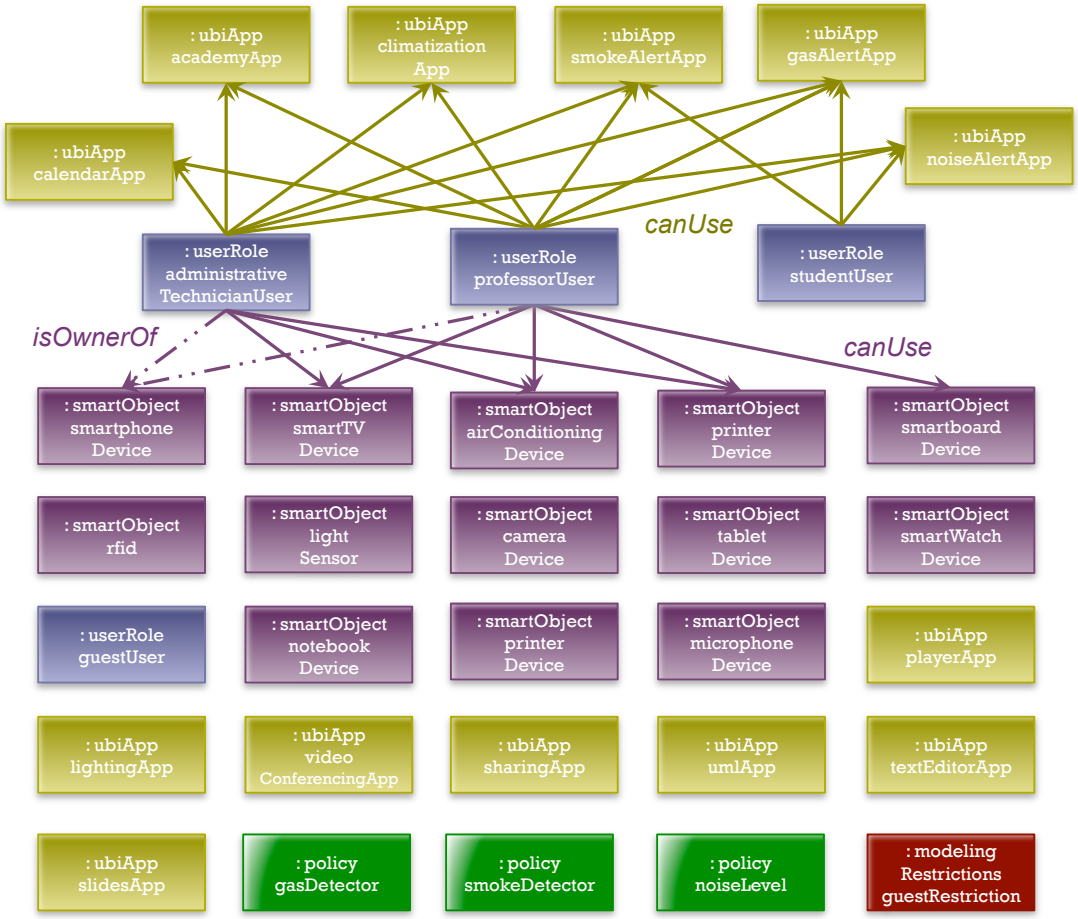



**Figura 3.7:** *Modelo do Engenheiro do Sistema para o ambiente da sala*

Cada um dos objetos inteligentes modelados possui uma ou mais características, definidas por meio do metatipo `Feature`. Elas foram omitidas da Figura 3.7 para não sobrecarregá-la. Entretanto, a Figura 3.8 ilustra a modelagem do tipo de objeto inteligente `tablet` com suas respectivas características e cada uma delas possui a seguinte modelagem:

- **Feature**: `accelerometer`
  - `featureType`: `HARDWARE`;
  - `featureCategory`: `SENSOR`;
  - `featureDescription`: `mplAccelerometer`, em que a sigla "mpl" significa *Motion Processing Library* (MPL).
- **Feature**: `operatingSystem`
  - `featureType`: `SOFTWARE`
  - `featureCategory`: `IO_RESOURCE`;
  - `featureDescription`: `androidSDK`.
- **Feature**: `display`

  - `featureType`: `HARDWARE`;
  - `featureCategory`: `IO_RESOURCE`;
  - `featureDescription`: `amoledDisplay`.

- **Feature**: `proximity`
  - `featureType`: `HARDWARE`;
  - `featureCategory`: `SENSOR`;
  - `featureDescription`: `gp2aProximitySensor`.

- **Feature**: `gyroscope`
  - `featureType`: `HARDWARE`;
  - `featureCategory`: `SENSOR`;
  - `featureDescription`: `mplGyroscope`.

- **Feature**: `orientation`
  - `featureType`: `HARDWARE`;
  - `featureCategory`: `SENSOR`;
  - `featureDescription`: `mplOrientation`.

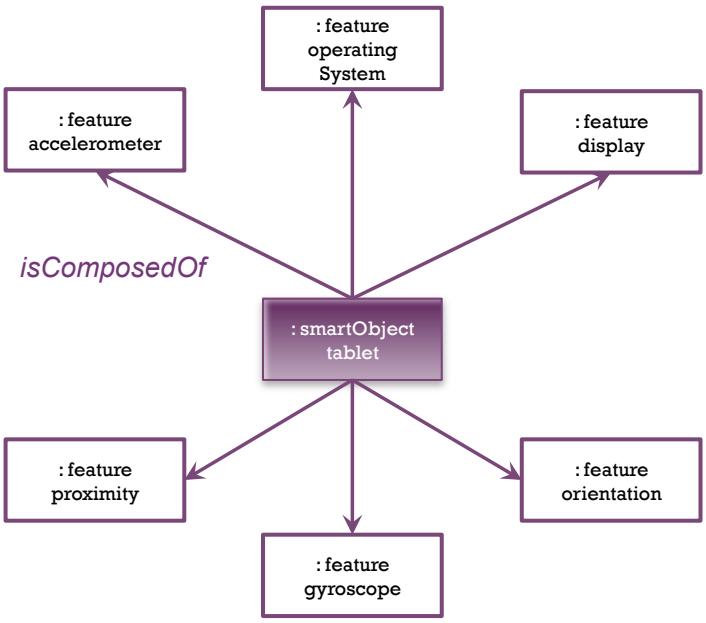



**Figura 3.8:** *Modelagem do objeto inteligente do tipo* `tablet`

Como aplicações ubíquas temos `calendarApp`, `academyApp`, `climatizationApp`, `videoConferencingApp`, `sharingApp`, `umlApp`, `playerApp`, `textEditorApp` e `slidesApp`. As aplicações ubíquas tem as seguintes funcionalidades no espaço inteligente:

- `calendarApp`: permite aos usuários do espaço inteligente registrarem seus compromissos e receberem notificações em determinados datas (dia, mês e ano) e horários, nos dispositivos na forma de eventos;
- `academyApp`: aplicação de sistema acadêmico que permite o lançamento de presenças dos alunos e notas;
- `climatizationApp`: configura os dispositivos do tipo `airConditioningDevice` com a temperatura adequada para o ambiente;
- `videoConferencingApp`: permite a transmissão de áudio e vídeo nos objetos inteligentes para usuários de diferentes localizações;
- `sharingApp`: permite o compartilhamento de conteúdo, como por exemplo, arquivos de texto, áudio e vídeo, entre os mais diversos dispositivos do espaço inteligente;
- `umlApp`: aplicação ubíqua que permite a modelagem de sistemas através de diagramas;
- `playerApp`: executa os mais diversos arquivos de áudio e vídeo nos objetos inteligentes do ambiente;
- `textEditorApp`: aplicação que permite aos usuários realizarem suas anotações e criação de textos;
- `slidesApp`: permite que as apresentações sejam exibidas nos dispositivos do ambiente de computação ubíqua.
- `smokeAlertApp`: notifica os usuários acerca de presença de fumaça no ambiente;
- `gasAlertApp`: notifica os usuários acerca de presença de gás no ambiente;
- `noiseAlertApp`: notifica os usuários acerca de ruído acima do permitido no ambiente;
- `lightingApp`: aplicação que configura a luminosidade no ambiente.

No modelo do engenheiro do sistema temos ainda uma restrição de modelagem, definida como `guestRestriction`, que impede que papéis do usuário do tipo `guest` tenham acesso aos objetos inteligentes do ambiente do tipo `microphone`, `camera` e `airConditioning`, e também às aplicações ubíquas do tipo `climatization` e `audioVideoRecord`. O objetivo desta restrição de modelagem é impedir que os usuários convidados, que não pertencem ao quadro de funcionários, utilizem estes dispositivos do ambiente e estas aplicações para gravarem as reuniões e demais atividades nestes ambientes.

Por fim, no modelo do engenheiro do sistema temos três políticas, que foram criadas para atender aos requisitos da organização em que se encontra o espaço inteligente: *i) gasDetector*: detecta a presença de algum tipo de gás no ambiente, como GLP, metano e propano; *ii) smokeDetector*: detecta a presença de fumaça no ambiente, em particular, aquelas relacionadas a cigarro; *iii) noiseLevel:* identifica ruídos no espaço inteligente,

emitidos acima de 85 decibéis (dB). É importante ressaltar que as políticas definidas no modelo do ES não tem o intuito de expressar a finalidade ou o comportamento do ambiente. As políticas do modelo do ES expressam apenas comportamentos gerais para o ambiente físico, haja visto que a finalidade do ambiente é realizada apenas no modelo do US. Portanto, apenas no modelo do US temos a definição de um espaço inteligente.

As regras ECA para as políticas *gasDetector*, *smokeDetector* e *noiseLevel* são apresentadas abaixo:

- **gasDetector**:
  - *event*: presença de gás no ambiente;
  - *condition*: sem condição;
  - *action*: emitir mensagem de alerta no ambiente através da inicialização da aplicação ubíqua *gasAlertApp*.
- **smokeDetector**:
  - *event*: presença de fumaça no ambiente;
  - *condition*: sem condição;
  - *action*: emitir mensagem de alerta no ambiente através da inicialização da aplicação ubíqua *smokeAlertApp*.
- **noiseLevel**:
  - **event**: alteração no nível de ruído do ambiente;
  - **condition**: se (ruído > 85dB)
  - **action**: emitir mensagem de alerta no ambiente através da inicialização da aplicação ubíqua *noiseAlertApp*.

As aplicações ubíquas tem papel essencial nas políticas, pois são elas que executam as ações relacionadas a cada uma dessas políticas. Por exemplo, quando a presença de um gás é detectada no ambiente físico da sala, a aplicação ubíqua do tipo `gasAlertApp` é inicializada nos dispositivos dos usuários do espaço inteligente a fim de exibir uma mensagem de alerta informando sobre a necessidade desses usuários se retirarem da sala. Isto equivale também para as aplicações ubíquas do tipo `smokeAlertApp` e `noiseAlertApp`. Em nosso trabalho, não manipulamos aspectos da lógica das aplicações ubíquas. Portanto, assumimos que estas aplicações, quando inicializadas, executam as devidas operações para as quais foram configuradas. Por exemplo, as aplicações do tipo `smokeAlertApp` e `noiseAlertApp`, quando inicializadas, emitem alertas relativos à detecção de fumaça no ambiente e ruído fora dos parâmetros desejáveis, respectivamente.

Os elementos do modelo do engenheiro servem como base para o usuário do sistema criar o seu modelo, portanto o US deve obedecer as restrições de modelagem impostas no modelo do ES e utilizar os blocos de construção básicos definidos para criar

os elementos de seu modelo. Estes blocos de construção básicos são tipo pré-definidos no modelo do ES que ajudam o US na criação de seus modelos. Vale ressaltar que esses blocos de construção podem ser utilizados no modelo do US a fim serem especializados, ou seja, os elementos definidos pelo usuário devem ser subtipos daqueles definidos no modelo do engenheiro. A Figura 3.9 ilustra o modelo do US desenvolvido a partir do modelo do ES para o cenário da sala de aula. Neste modelo, temos a finalidade do espaço inteligente.

O elemento raiz do modelo do usuário é representado pelo metatipo `SmartSpace`. Nele, estão concentrados todos os elementos do espaço inteligente e esses foram omitidos da Figura 3.9 de maneira a facilitar a visualização de todos os elementos e não deixá-la sobrecarregada. As setas na cor laranja indicam que estas associações foram definidas no modelo do ES, e portanto, não precisam ser novamente inseridas no modelo do US.

Logo, temos que o espaço inteligente do tipo `classroom` hospeda (`host`) tipos de papéis do usuário, de objetos inteligentes e aplicações ubíquas, além de ter políticas que guiam o comportamento dinâmico do ambiente. Neste modelo desenvolvido pelo US, temos os tipos de papéis do usuário `professor` e `student`.

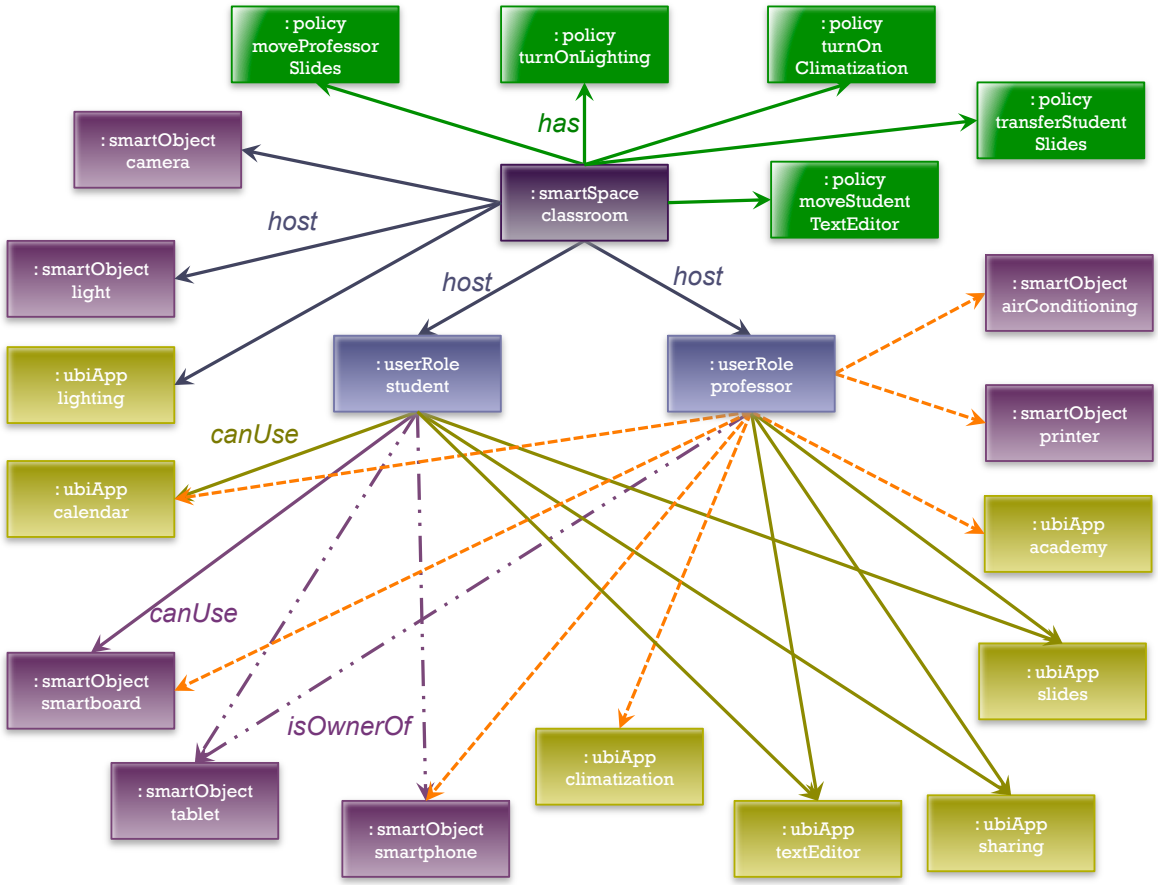



**Figura 3.9:** *Modelo do Usuário do Sistema para Cenário da Sala de Aula*

O tipo `professor` pode usar objetos inteligentes do ambiente `smartboard`, `printer` e `airConditioning`, além de seus dispositivos pessoais definidos pela associação `isOwnerOf`, `smartphone` e `tablet`. Ele também pode utilizar as aplicações ubíquas `slides`, `calendar`, `textEditor`, `sharing`, `academy` e `climatization`, indicadas pela associação do tipo `canUse`.

Já o papel do usuário `student`, pode usar o objeto inteligente do ambiente `smartboard`, indicado pela associação do tipo `canUse`, e tem como objetos inteligentes pessoais `smartphone` e o `tablet`, indicados pela associação do tipo `isOwnerOf`. Como aplicações ubíquas, ele pode utilizar em seus dispositivos as aplicações do tipo `calendar`, `textEditor`, `sharing` e `slides`.

Por fim, temos cinco políticas responsáveis por lidar com a parte dinâmica do ambiente, além daquelas já definidas no modelo do engenheiro do sistema. As políticas do usuário do sistema foram criadas a partir das possíveis ações que podem ser desempenhadas no espaço inteligente:

- **moveProfessorSlides**:
  - *event*: detecção de mudança de localização do usuário/objeto inteligente. Neste evento, as informações referentes à localização atual do usuário/objeto inteligente é obtida a partir do evento gerado. Ela será utilizada para ser avaliada na condição que foi definida pelo usuário do sistema;
  - *condition*: se (tipo do objeto inteligente em que a aplicação está executando == "smartphone") AND (tipo de aplicação ubíqua == "slides") AND (papel do usuário == "professor") AND (localização atual igual a "lousa inteligente");
  - *action*: mover a aplicação *slides* do dispositivo *smartphone* para a lousa inteligente, através da ação *moveUbiApp*. Esta ação é definida pelo usuário do sistema em seu modelo. Para isso, o US seleciona esta ação a partir de um de ações que foram especificadas na linguagem de modelagem 2SML, apresentadas na Tabela tab:TabActions.
- **turnOnLighting**:
  - *event*: horário, dia, mês, ano da aula;
  - *condition*: se (horário, dia, mês e ano atual == horário, dia, mês e ano da aula no calendário) AND (papel do usuário == "professor") AND (professor == Peter);
  - *action*: iniciar a aplicação ubíqua do tipo "lighting".
- **turnOnClimatization**:
  - **event**: detecção de mudança de temperatura do ambiente;
  - **condition**: se (temperatura $>$ 25 graus celsius);

  - **action**: iniciar a aplicação ubíqua do tipo “climatization”, responsável por configurar o ar-condicionado com a temperatura adequada para o ambiente.

- **moveStudentTextEditor**:
  - *event*: detecção de mudança de localização do usuário/objeto inteligente;
  - *condition*: se (tipo de aplicação ubíqua == “textEditor”) AND (papel do usuário == “student”) AND (localização atual igual a “lousa inteligente”);
  - *action*: mover a aplicação *textEditor* para a lousa inteligente, através da ação *moveUbiApp*.

- **transferStudentSlides**:
  - **event**: horário, dia, mês, ano da aula;
  - **condition**: se (horário, dia, mês e ano atual == horário, dia, mês e ano da aula no calendário) AND (papel do usuário == “student”) AND (professor == Peter);
  - **action**: copiar os slides da lousa do professor para os dispositivos dos alunos, por meio da aplicação ubíqua do tipo `sharing`. Esta aplicação foi configurada previamente pelo professor para copiar os *slides* de seu dispositivo para aqueles pertencentes aos alunos.

A parte estrutural do modelo bem como a parte comportamental definem a finalidade de um ambiente, ou seja, a programação de um espaço inteligente. Os papéis do usuário, tipos de objetos inteligentes, aplicações ubíquas e os relacionamentos entre cada um deles demonstram como definimos a estrutura dos elementos de um modelo. Já as políticas criadas permitem demonstrar como definimos o comportamento para um espaço inteligente a partir de todos os elementos presentes no modelo.

## 3.6 Conclusão

Este capítulo apresentou a linguagem de modelagem para espaços inteligentes denominada 2SML. Esta linguagem tem por capacidade definir tipos de elementos que poderão fazer parte do ambiente de programação ubíqua, dentre eles, usuários, objetos inteligentes e aplicações ubíquas, além das políticas, que regem o comportamento dinâmico do ambiente.

Primeiramente, começamos nossa discussão dando uma visão geral dos conceitos que utilizamos na tese, em particular, descrevemos os elementos que compõem o espaço inteligente, bem como os usuários responsáveis por programar o ambiente, engenheiro e usuário do sistema. Após discutirmo sobre esses conceitos, apresentamos a definição da linguagem e após isso, do metamodelo dela, que foi divido em *Core-Metamodel*,

*System Engineer-Metamodel* e *System User-Metamodel*. Por fim, finalizamos o capítulo descrevendo as políticas, que são baseadas nas regras ECA, e apresentando exemplos de modelagem com a linguagem 2SML, para o cenário de sala de aula inteligente.

CAPÍTULO 4

# Arquitetura da 2SVM

A 2SVM é uma plataforma de *middleware* dirigida por modelos para espaços inteligentes. Ela integra características desse domínio, tais como mobilidade de usuários e dispositivos, além de tratar a alta volatilidade das interações entre estes mesmos usuários, dispositivos e aplicações.

A 2SVM interpreta modelos criados pelo engenheiro e pelo usuário do sistema, o que resulta na execução de ações sobre os dispositivos e aplicações do espaço inteligente. Os modelos são mantidos em tempo de execução, o que permite adaptar o comportamento do ambiente de computação ubíqua em tempo de execução, em virtude de mudanças de contexto ou mudanças no próprio modelo do usuário.

A 2SVM também permite ao usuário do sistema definir diferentes comportamentos para um mesmo ambiente físico, o que possibilita o reaproveitamento dos recursos do ambiente. Isto significa que os mesmos recursos de um espaço físico podem ser utilizados para diferentes programações de um mesmo espaço inteligente. A essa mudança de comportamento do ambiente de computação ubíqua denominamos como reprogramação do espaço inteligente.

A 2SVM é formada por quatro camadas, e em cada uma das camadas temos o modelo em diferentes níveis de abstração. Nas duas primeiras camadas da máquina, que encontram-se no topo da pilha, temos o modelo em nível de tipos, na terceira camada temos o modelo em nível de tipos e de instâncias, e na última camada, o modelo em nível de instância.

O capítulo está organizado da seguinte forma: a Seção 4.1 apresenta uma visão geral da arquitetura da 2SVM, ilustrando as camadas da máquina de execução de modelos. A Seção 4.2 descreve a arquitetura de distribuição da 2SVM e a Seção 4.3 apresenta o modelo de interação inter-camadas da 2SVM. Na Seção 4.4, apresentamos uma definição formal para os modelos em tempo de execução utilizados pela 2SVM e na Seção 4.5, descrevemos o mecanismo utilizado pela 2SVM para configurar os dispositivos objetos inteligentes e as aplicações. Por fim, a Seção 4.6 descreve os componentes de cada uma das camadas da 2SVM.

## 4.1 Visão Geral da Arquitetura

A arquitetura da 2SVM foi baseada na arquitetura da máquina de execução de modelos para o domínio de comunicação, denominada *Communication Virtual Machine* (CVM) [35]. Isto implica que o conjunto de funcionalidades projetadas na máquina de execução de modelos 2SVM se aproxima daqueles definidos pela CVM, para o domínio de espaços inteligentes. Essa escolha foi feita pois a CVM apresenta uma maneira efetiva de processar modelos desenvolvidos pelo usuário e mantê-los em tempo de execução [35].

A Figura 4.1 ilustra a arquitetura geral da máquina de execução de modelos para espaços inteligentes e cada uma de suas camadas. Assim como a CVM, a 2SVM possui uma arquitetura estratificada em quatro camadas.

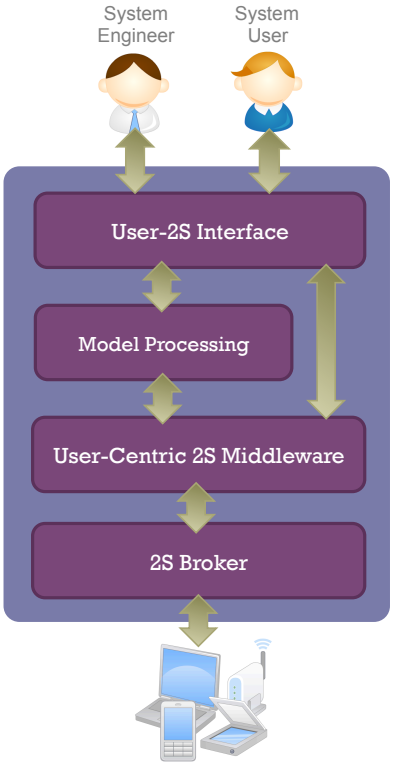



**Figura 4.1:** *Arquitetura da 2SVM*

Modelos são criados pelo engenheiro e pelo usuário do sistema na camada *User-2S Interface*, que também é responsável pela autenticação desses usuários. Em seguida, o modelo do usuário é processado na camada *Model Processing* que consiste de verificar se o modelo do usuário está em conformidade com o modelo do engenheiro e também de compará-lo com o modelo em execução no ambiente. Após isso, o modelo é interpretado e convertido em comandos na camada *User-Centric 2S Middleware*, para assim, serem executados na camada *2S Broker*. O conjunto desses comandos é denominado macro e existem diferentes macros para cada uma das operações que devem

ser executadas nos objetos inteligentes. Desta forma, temos macros para configuração dos objetos inteligentes, para as aplicações ubíquas, entre outras.

O fato de modelos passarem por diferentes níveis de abstração durante seu processamento motivou a utilização de uma arquitetura em camadas [19] para lidar com o problema de programação de espaços inteligentes. Neste estilo arquitetural, podemos decompor em um grupo de sub-tarefas todas as fases pelas quais passa o modelo: criação, processamento, interpretação e execução. Desta forma, os modelos criados pelo engenheiro e pelo usuário do sistema devem ser convertidos em comandos de mais baixo nível para serem executados em cada um dos dispositivos do espaço inteligente.

A camada *User-2S Interface* é responsável por fornecer uma linguagem de modelagem gráfica para o engenheiro e usuário do sistema criarem seus modelos de programação do espaço inteligente, bem como por prover uma interface de acesso à plataforma. A camada também armazena modelos reutilizáveis. A *User-2S Interface* gerencia o processo de autenticação do engenheiro e do usuário do sistema na máquina, além de permitir aos engenheiros cadastrarem usuários que poderão participar do espaço inteligente. A base de dados com as informações referentes a esses usuários encontra-se na camada *User-Centric 2S Middleware*, e é acessada por meio de comunicação direta com essa camada. A base de dados de usuários também é utilizada pela camada de *User-Centric 2S Middleware* para realizar operações relacionadas à execução do modelo. Esta camada recebe notificações das camadas inferiores relacionadas a problemas que podem ocorrer durante o processamento do modelo.

A camada *Model Processing* tem por objetivo validar os modelos criados pelo engenheiro e pelo usuário do sistema, por meio de checagens feitas pelos componentes internos da camada. Esta validação consiste em verificar se os modelos criados por estes usuários são passíveis de implantação no espaço inteligente. Para que que um modelo possa ser implantado no espaço inteligente, ele deve passar pelas seguintes checagens: *i)* Consistência interna; *ii)* Checagem de conformidade; e *iii)* Comparação entre o modelo submetido e o modelo em execução no espaço inteligente.

A camada de processamento de modelos recebe como entrada o modelo do engenheiro e do usuário do sistema para sua validação, e recebe notificações de possíveis exceções de cada um dos modelos. A primeira checagem é denominada consistência interna, em que a parte comportamental do modelo, representada pelas políticas, é avaliada. Para que uma política seja válida, cada um dos elementos utilizados em sua criação devem ter sido anteriormente definidos na parte estrutural do modelo, ou seja, todos os tipos de elementos utilizados nas políticas devem estar presentes na parte estrutural do modelo. Este mecanismo utilizado na camada de processamento checa apenas a validade da estrutura das políticas e não sua semântica.

Depois de realizada esta checagem, a camada de processamento de modelos ve-

rifica se o modelo do usuário é baseado no modelo do engenheiro do sistema. Esta verificação é denominada checagem de conformidade e tem por objetivo analisar se os elementos definidos no modelo do usuário são sub-tipos daqueles definidos no modelo do engenheiro. Por fim, a última funcionalidade da camada *Model Processing* é comparar o modelo submetido pelos usuários, com o modelo em execução no espaço inteligente. O modelo em execução no ambiente encontra-se executando na camada subjacente, e é consultado pela camada de processamento para que seja feito o cálculo da diferença entre eles. Desta forma, os novos elementos presentes no modelo submetido são adicionados ao modelo em execução, os elementos que não estão presentes no novo modelo são removidos e aqueles comuns a ambos os modelos permanecem no modelo em execução no espaço inteligente. Isto implica também que, nas camadas subjacentes, novos comandos são selecionados para efetuarem a configuração de objetos inteligentes e aplicações ubíquas.

A camada *User-Centric 2S Middleware*, ou simplesmente camada de *middleware*, tem por função manter e executar o modelo em tempo de execução. Este modelo contém informações de todos os elementos ativos no espaço inteligente, entre eles, usuários, *smart objects* e aplicações. Para cada um desses elementos temos informações de tipo e de instância, em que as informações de tipo contém as definições de cada um dos elementos, e as informações de instância, as características de cada elemento físico presente no espaço inteligente. Por exemplo, para um *smart object* do tipo *smart TV*, definido no modelo do usuário, podemos ter a instância *samsung TV*.

O modelo em tempo de execução está presente nesta camada, pois nela concentramos todas as operações relacionadas à execução do modelo em si, como por exemplo: *i)* Descoberta dos tipos de elementos que podem fazer parte do espaço inteligente, entre eles, usuários, *smart objects* e aplicações ubíquas; *ii)* Execução das políticas, responsáveis por lidar com a parte comportamental do espaço inteligente; e *iii)* seleção do conjunto de comandos que atuarão diretamente na configuração dos elementos do ambiente de computação ubíqua.

A camada de *middleware* recebe consultas acerca do modelo em execução no ambiente e chamadas relacionadas à atualização do modelo em tempo de execução em nível de tipos e de instâncias. Estas chamadas permitem adicionar e remover elementos do modelo em tempo de execução, a partir das comparações do modelo submetido com aquele em execução no ambiente.

Por fim, a camada *2S Broker*, ou simplesmente camada de *broker*, tem por objetivo interagir com o sistema subjacente à máquina em particular, com os recursos dos dispositivos, entre eles, sensores, atuadores, e aplicações. Esta camada ainda coleta informações sobre o usuário que está utilizando o objeto inteligente, identifica o conjunto de características destes dispositivos e recebe notificações acerca dos eventos locais detecta-

dos pelo próprio dispositivo. Estes eventos locais são aqueles gerados no próprio objeto inteligente, como por exemplo, mudança de localização do dispositivo, inicialização de uma aplicação, detecção de baixo nível da bateria, entre outros.

A camada de *Broker* lida com os recursos do objeto inteligente por meio de comandos que são encaminhados da camada de *middleware*. Estes comandos podem, por exemplo, iniciar uma aplicação, salvar seu estado para que seja enviada a outro *smart object*, recuperar o estado dessa aplicação, iniciar tratador para captura de eventos, entre outros.

É importante ressaltar que, em cada uma das camadas da 2SVM, temos o modelo em diferentes níveis de abstração. Nas camadas de *interface* e de processamento de modelos, temos o modelo em nível de tipos. Na camada de *middleware*, temos o modelo em nível de tipo e de instância, ou seja, o modelo em tempo de execução. Já na camada de *Broker*, temos apenas o modelo em nível de instância, que contém informações acerca dos elementos físicos presentes no espaço inteligente.

## 4.2 Arquitetura de Distribuição da 2SVM

A plataforma 2SVM é composta por dois tipos de máquinas de execução de modelos, uma para os dispositivos que estão presentes no espaço inteligente - denominada 2SVM-*Client* - e outra, pertencente à controladora do ambiente, denominada 2SVM-*Controller*. A Figura 4.2 apresenta a visão geral dessa arquitetura distribuída.

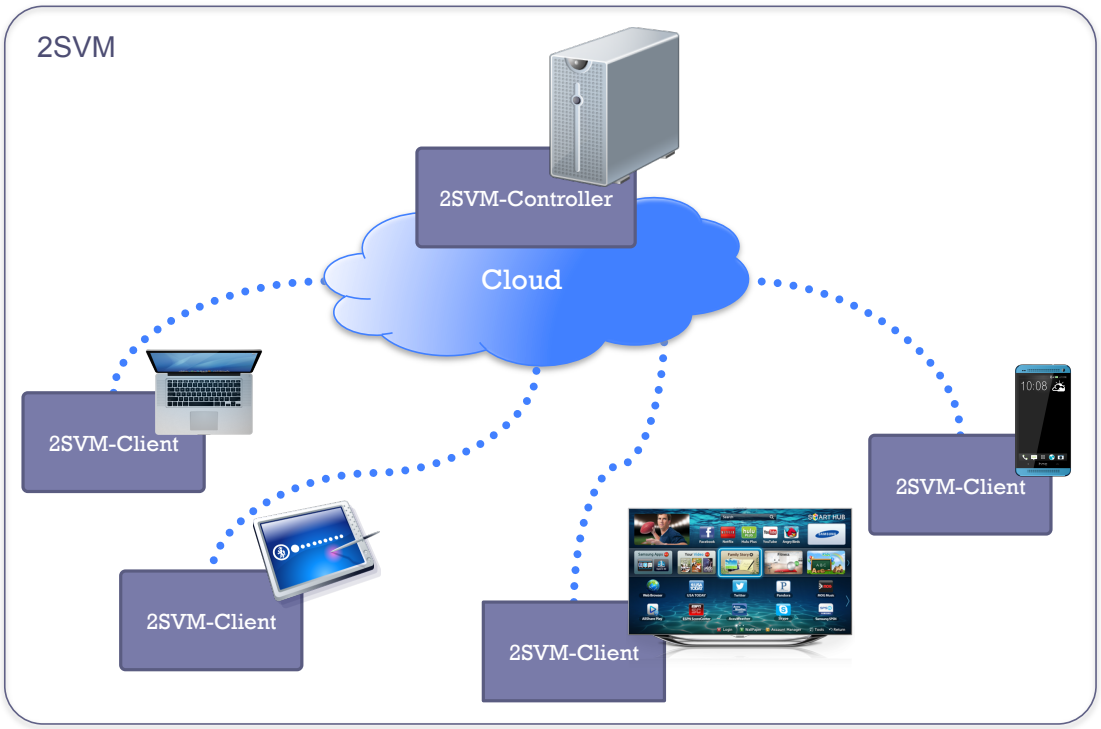



**Figura 4.2:** *Visão Geral da Plataforma 2SVM*

A máquina de execução de modelos, 2SVM-*Controller*, é responsável por coordenar as interações entre todos os elementos do espaço inteligente, usuários, objetos inteligentes e aplicações. Essas interações ocorrem sempre entre os usuários e seus dispositivos pessoais, ou entre usuários e os dispositivos pertencentes ao ambiente, por meio das aplicações. É importante ressaltar que as aplicações desempenham um papel fundamental no espaço inteligente, já que o conjunto delas caracteriza a finalidade do espaço inteligente. Como exemplo, podemos citar aplicações para os domínios de sala de aula, ambientes hospitalares, cenários militares, entre outros.

A 2SVM-*Controller* é responsável por manter o M@RT global do ambiente, que possui informações relativas a todos os elementos do espaço inteligente. Isso inclui todos os usuários com seus dispositivos pessoais e os dispositivos do ambiente, além das aplicações que estão em execução neles. Já a 2SVM-*Client* mantém e executa o modelo em tempo de execução local, que contém informações sobre os elementos em execução no objeto inteligente, além de permitir ao usuário interagir com os demais elementos do espaço inteligente, bem como com os recursos do próprio dispositivo na forma de sensores, atuadores, e demais interfaces de entrada e saída, tais como *displays*, teclados, entre outros.

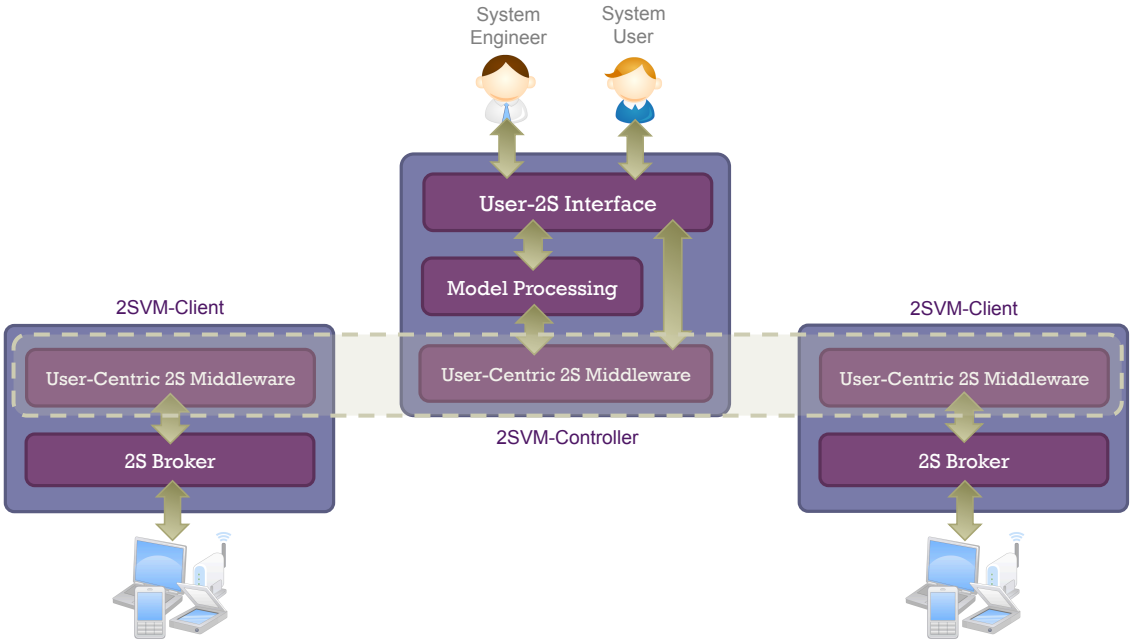



**Figura 4.3:** *Arquitetura Distribuída da 2SVM*

A Figura 4.3 apresenta a arquitetura distribuída da 2SVM, que pode ter uma instância da 2SVM-*Controller* e várias instâncias da 2SVM-*Client*. A 2SVM-*Controller* é estruturada em três camadas: *User-2S Interface*, *Model Processing* e *User-Centric 2S Middleware*. Esta última, está presente em ambas as configurações da máquina de execução de modelos, 2SVM-*Controller* e 2SVM-*Client*. Essas instâncias da camada

de *middleware* operam de forma coordenada para prover as funcionalidades da camada, como será visto na Seção 4.6.

Na 2SVM-*Client*, temos, além da camada de *middleware* , a camada *2S Broker*. Esta camada tem por objetivo lidar com os recursos do dispositivo, dentre eles: sensores; atuadores; interfaces de entrada e saída, tais como *display* e teclado, entre outros; e aplicações que podem ser executadas no dispositivo.

Com o objetivo de reduzir problemas relacionados às ações que os usuários realizam no ambiente, os espaços inteligentes são delimitados por áreas físicas, como por exemplo, uma sala, um quarto ou um ambiente qualquer. Para lidar com este desafio, temos um processo 2SVM-*Controller* para cada ambiente físico e, por conseguinte, para cada espaço inteligente.

## 4.3 Modelo de Interação das Inter-Camadas da 2SVM

O modelo de interação inter-camadas lida com a comunicação entre as camadas da 2SVM. A comunicação entre elas pode ser: *i)* Síncrona, em que uma camada interage com a outra através de chamadas bloqueantes; e *ii)* Assíncrona, em que a comunicação entre as camadas ocorre a partir da geração de eventos.

A comunicação entre a camada *User-2S Interface* e *User-Centric 2S Middleware* ocorre a partir de chamadas bloqueantes, sendo portanto, síncrona. A interação entre elas ocorre quando o engenheiro ou usuário do sistema tentam se autenticar na máquina de execução de modelos, ou quando o engenheiro realiza o cadastro de usuários na base de dados localizada na camada *User-Centric 2S Middleware*. Desta forma, os componentes internos da *User-2S Interface* comunicam com os componentes da *User-Centric 2S Middleware* apenas através das interfaces definidas por estas camadas.

As camadas *User-2S Interface* e *Model Processing* também se comunicam de forma síncrona. O engenheiro e o usuário do sistema criam seus modelos na camada de *interface*, e após isso, esse modelo é encaminhado à camada subjacente para ser processado. Como resposta a esse processamento, a camada *Model Processing* pode retornar um conjunto de exceções existentes nos modelos, o que exige do engenheiro ou usuário do sistema as devidas modificações em seus respectivos modelos. Caso nenhuma exceção seja detectada nos modelos, eles podem ser submetidos para serem executados na camada de *middleware*.

A comunicação entre as camadas *Model-Processing* e *User-Centric 2S Middleware*, assim como as anteriores, também é síncrona. Isso ocorre, pois a camada de processamento de modelos faz chamadas de consulta e/ou atualização ao modelo em nível de tipos presente na camada de *middleware*.

A Figura 4.4 apresenta as interfaces entre as camadas da máquina a partir de uma visão geral da arquitetura da 2SVM.

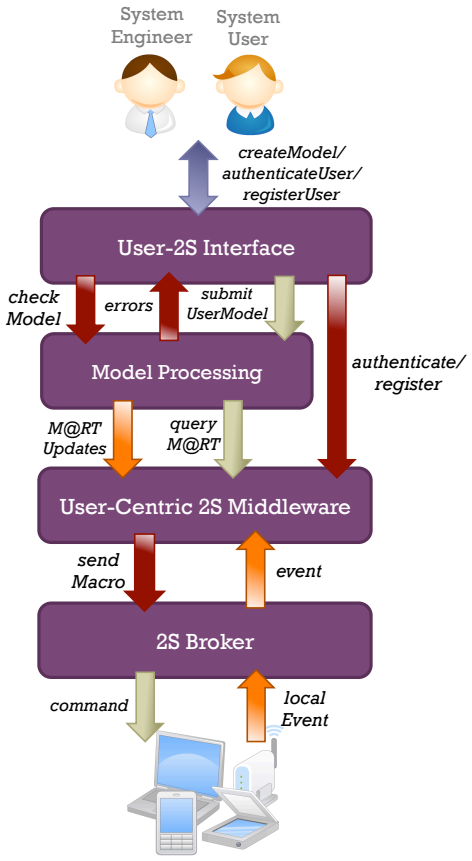



**Figura 4.4:** *Interfaces entre as camadas da 2SVM*

Por fim, temos a comunicação entre as camadas *User-Centric 2S Middleware* e *2S Broker*, que acontece de forma assíncrona. A camada de *middleware* envia um conjunto de comandos à camada de *Broker* apenas quando recebe eventos de interesse, que são encaminhados pela própria camada de *Broker*, ou seja, a camada de *Broker* recebe eventos locais, encaminhados pelo dispositivo ao qual a máquina está executando, e os repassa à camada de *middleware*. A camada *User-Centric 2S Middleware*, por sua vez, trata este evento de acordo com seu tipo, e envia os comandos adequados à *2S Broker*, a fim de que ela realize as operações relacionadas ao usuário, *smart object* e/ou aplicação ubíqua.

A Tabela 4.1 apresenta as interfaces entre cada uma das camadas da 2SVM.

**Tabela 4.1:** *Interfaces entre as camadas da 2SVM*

| Camada | Chamadas | *Callback* |
|---|---|---|
| *User-2S Interface* | checkModel(model)<br>submitUserModel()<br>authenticate(user)<br>register(user) | errors(listOfErrors) |
| *Model Processing* | M@RTUpdates(m@rt)<br>queryM@RT() | – |
| *User Centric-2S Middleware* | sendMacro(macro)<br>sendLocalM@RT(localM@RT) | event(eventInformation) |
| *2S Broker* | command() | localEvent(eventInformation) |

## 4.4 Modelos em Tempo de Execução

Modelos em tempo de execução são utilizados pela máquina para dirigir o comportamento de espaços inteligentes. Desta forma, eles armazenam informações relacionadas a todos os tipos de elementos que podem participar do espaço inteligente, bem como sobre aqueles elementos que estão efetivamente ativos no ambiente. Este modelo é atualizado pela 2SVM à medida que mudanças de contexto ocorrem no espaço inteligente a partir das mais diversas interações que podem ocorrer entre usuários, dispositivos e aplicações.

Nesta seção formalizamos os modelos em tempo de execução manipulados pela 2SVM-*Controller* e pela 2SVM-*Client*. Essa definição é apresentada por meio do modelo em tempo de execução global, na Seção 4.4.1, e pelo modelo em tempo de execução local, na Seção 4.4.2.

### 4.4.1 Modelo em Tempo de Execução Global

O modelo em tempo de execução global possui informações de todo o espaço inteligente. Essas informações dizem respeito aos usuários, dispositivos e aplicações ativos no espaço inteligente. A Figura 4.5 apresenta a uma definição desse modelo.

Do lado esquerdo, temos um modelo genérico criado pelo usuário do sistema, que apresenta a definição de um *smart space*, um *user role*, um *smart object* e uma *ubiquitous application*.

Já no lado direito da Figura 4.5, temos a ilustração do *snapshot* do espaço inteligente em um dado momento. Nele, temos as instâncias de cada um dos elementos do modelo, como por exemplo, usuários de determinado papel, tipos de dispositivos e aplicações. Suponhamos que para um determinado `userRole`, podemos ter instâncias em

um dado momento no ambiente, como `userX` e `userY`. Isso é aplicado para cada um dos elementos que podem pertencer ao espaço inteligente.

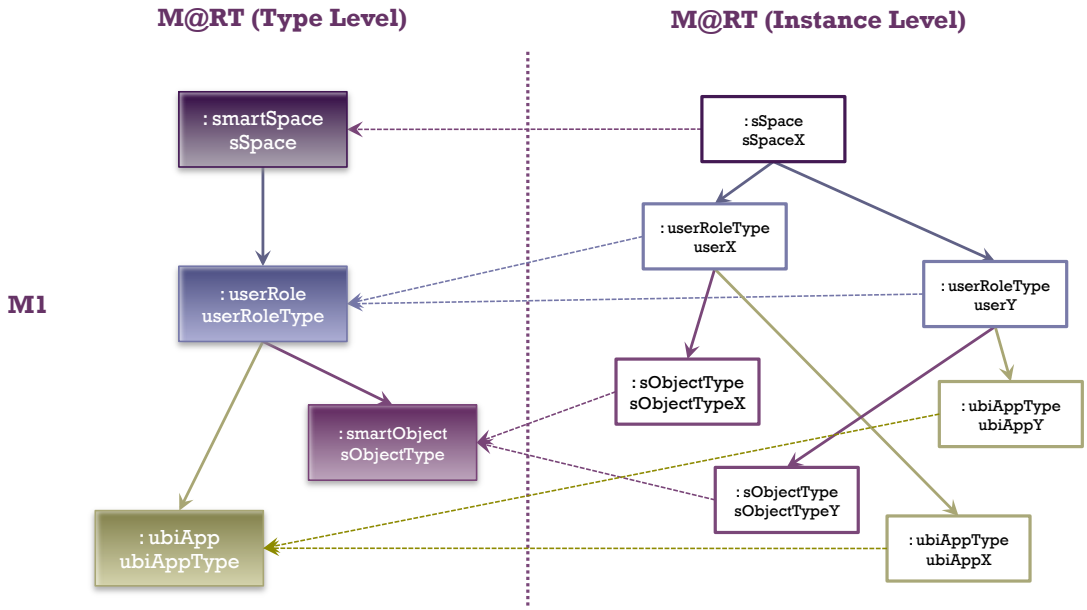



**Figura 4.5:** *Definição do Modelo em Tempo de Execução Global*

Portanto, o modelo em tempo de execução global, possui informações de todos os elementos ativos do ambiente. À medida que novos usuários e dispositivos entram ou saem do espaço inteligente, eles são adicionados ou removidos do modelo em tempo de execução global. Da mesma forma, isso acontece para as aplicações ubíquas, que são adicionadas ou removidas do modelo em tempo de execução global à medida que são iniciadas ou finalizadas nos dispositivos do espaço inteligente.

A Figura 4.6 ilustra um modelo em tempo de execução global para o cenário do espaço inteligente de sala de aula. No modelo, criado pelo usuário do sistema, temos dois papéis de usuário: `professor` e `student`. Como instâncias dos elementos do modelo do usuário, temos: *Peter* e *Harry*, do tipo `professor`; e *Lewis* e *Emily*, do tipo `student`.

Usuários com o papel `student` podem usar (`canUse`) o dispositivo do ambiente `smartboard`, e podem ter (*isOwnerOf*) dispositivos dos tipos `smartphone` e `tablet`. Estes usuários pode usar as aplicações ubíquas `slides` e `calendar`. Já usuários que desempenham o papel de usuário `professor` podem usar dispositivos do ambiente dos tipos `smartboard` e `airConditioning`, com as aplicações ubíquas `slides` e `calendar` para a lousa, e `climatization` para o ar-condicionado. Além disso, eles podem ter dispositivos pessoais dos tipos`smartphone` e `tablet`.

Os dispositivos dos tipos `smartphone` e `tablet` possuem instâncias no espaço inteligente, `smartphoneX`, `smartphoneY` e `smartphoneZ`, e `tabletX`. As aplicações ubíquas slides, calendar e climatization também possuem instâncias no ambiente: `slidesX`, `slidesY` e `slidesZ`; `calendarX` e `calendarY`; e `climatizationX`.

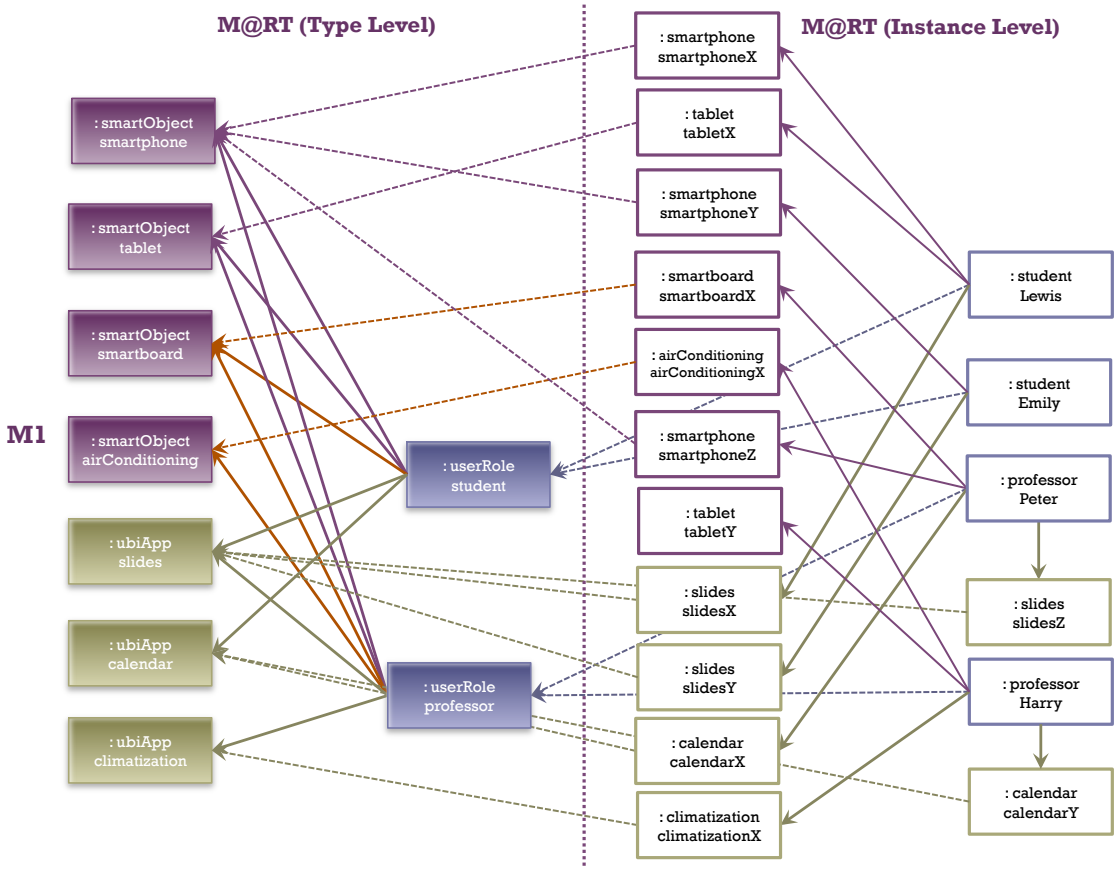



**Figura 4.6:** *Modelo em Tempo de Execução Global para cenário da sala de aula*

### 4.4.2 Modelo em Tempo de Execução Local

O modelo em tempo de execução local é criado pela 2SVM-*Controller* e mantido pela 2SVM-*Client*. Ele possui apenas informações referentes ao próprio objeto inteligente e sua definição é apresentada na Figura 4.7, ou seja, o modelo em tempo de execução local é uma parte do modelo global.

Do lado esquerdo da figura, o modelo contém o tipo do espaço inteligente, do papel do usuário, do dispositivo, e das aplicações que podem executar nele. Já no lado direito, temos as instâncias desses elementos em um dado momento, ou seja, um *snapshot*. O modelo em tempo de execução local possui apenas uma instância para `UserRole` e uma para `SmartObject`, já que elas se referem, respectivamente, ao usuário e ao dispositivo em questão. Em relação às aplicações ubíquas, podemos ter uma ou mais em execução no objeto inteligente.

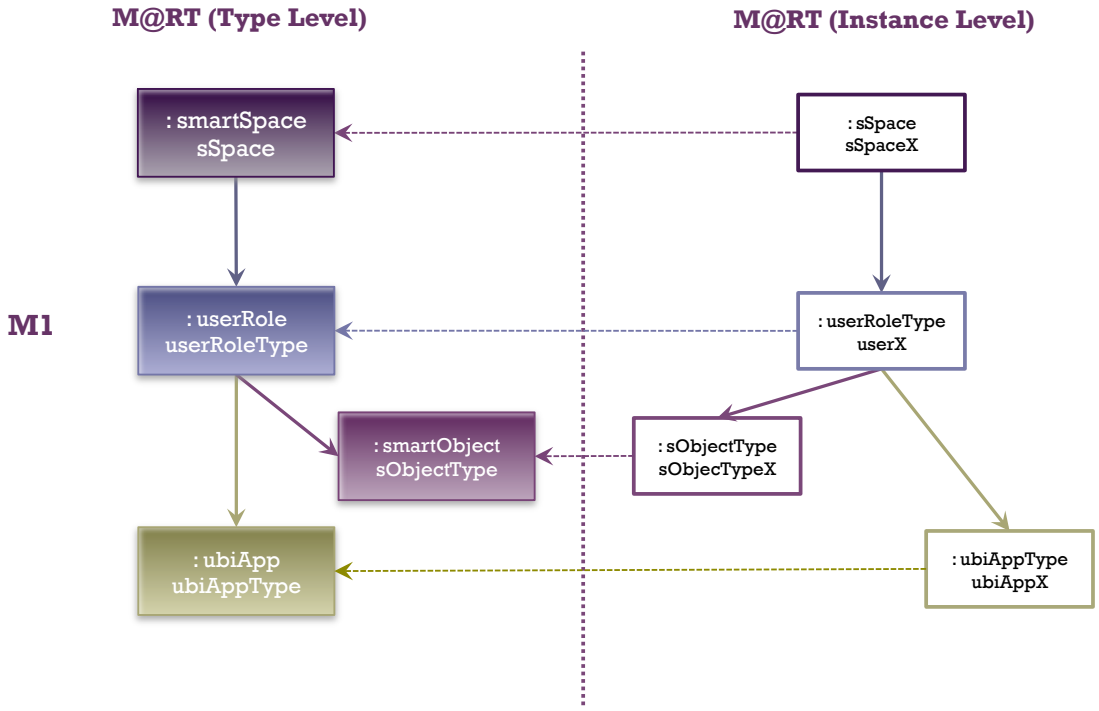



**Figura 4.7:** *Definição do Modelo em Tempo de Execução Local*

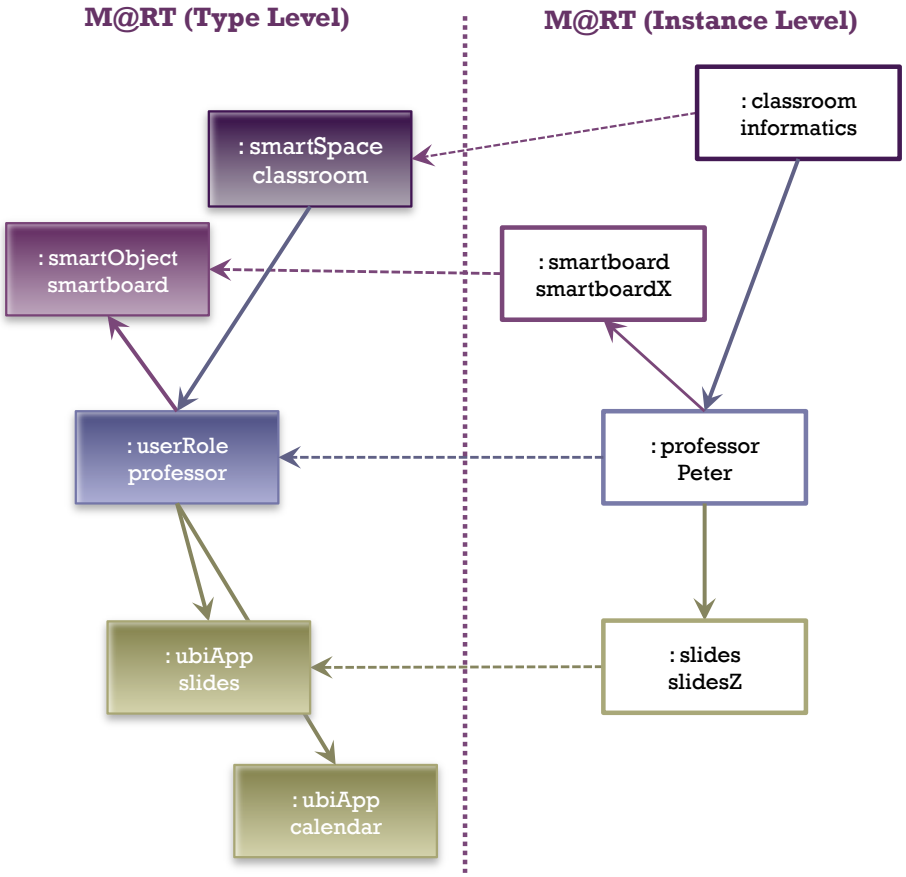



**Figura 4.8:** *Modelo em Tempo de Execução Local para cenário da sala de aula*

A Figura 4.8 exemplifica o modelo em tempo de execução local do dispositivo `smartboard` para o cenário de sala de aula. No modelo, temos a definição do tipo de dispositivo `smartboard`, do papel do usuário `professor`, que pode utilizar dispositivos desse tipo, e dos tipos de aplicações ubíquas que podem executar dispositivos desse tipo, `slides` e `calendar`. No *snapshot* da Figura 4.8, temos as informações da instância do usuário, do dispositivo e da aplicação em execução, neste caso, `slides`.

## 4.5 Macros

Para que modelos em tempo de execução sejam executados nos objetos inteligentes, eles devem ser convertidos em comandos de mais baixo nível, os quais são agrupados em macros. Elas são executadas na camada de *Broker* da máquina de execução de modelos da 2SVM-*Client* a fim de realizarem operações nos dispositivos do espaço inteligente. Elas são compostas por um conjunto de comandos e podem ser usadas para:

- adicionar os usuários no espaço inteligente;
- iniciar tratador de eventos relacionados ao conjunto de sensores do dispositivo. Estes sensores podem ser em nível de *hardware* ou *software*;
- iniciar tratador de eventos relacionados às aplicações. As operações realizadas pelas aplicações são recebidas pelos tratadores, que as repassam para o componente correspondente, responsável por realizar o devido tratamento;
- executar o conjunto de operações relacionadas às aplicações, entre elas, iniciar, pausar, continuar, e finalizar.

As macros são selecionadas a partir de cada um dos elementos do modelo que são identificados pela camada de *middleware*. Por exemplo, quando um novo objeto inteligente "entra" no ambiente, ele é descoberto e caso possa participar do espaço inteligente, uma macro é selecionada do repositório de macros para que possa ser encaminhada ao dispositivo e realize as devidas configurações nele. Isto vale também para os usuários e para as aplicações ubíquas.

A Tabela 4.2 ilustra as macros utilizadas pela 2SVM para executar operações nos dispositivos no espaço inteligente, com seus respectivos comandos. Elas são um conjunto fixo e predefinido, nas quais são parte integrante da 2SVM.

**Tabela 4.2:** *Macros utilizadas pela 2SVM, com seus respectivos comandos*

| **Nome da Macro** | **Lista de parâmetros** | **Comandos da Macro** | **Propósito da Macro** |
|---|---|---|---|
| addUser | userID, userName, password | activeUser | Adicionar o usuário ao espaço inteligente por meio de sua ativação no dispositivo. |
| addSmartObject | smartObjectID, smartObjectName, featureList | startListenerSmartObject, activeSmartObject | Adicionar o dispositivo ao espaço inteligente. Para isso, esta macro possui um conjunto de comandos que são responsáveis por iniciar tratadores do dispositivo, responsáveis por receber notificações de eventos de interesse. |
| addUbiApp | ubiAppList | startListenerApp | Permitir que as aplicações sejam manipuladas pelo *middleware*. Para isso, tratadores são inicializados por meio de comandos das macros na 2SVM-*Client* de maneira que os eventos gerados pelas aplicações sejam recebidos por eles. Estes eventos identificam quando uma aplicação é iniciada, parada, continuada ou finalizada. |
| actUbiApp | ubiAppID, ubiAppName | activeUbiApp | Ativar a aplicação no dispositivo. |
| moveUbiApp | userID, userName, password, ubiAppID | pauseApp, saveAppState, destroyApp | Iniciar o processo de mover a aplicação de um dispositivo para outro. |
| sendUbiApp | userID, userName, password, ubiAppID | restoreAppState, startApp | Mover a aplicação de um dispositivo para outro. |
| startUbiApp | ubiAppID | startApp | Iniciar a aplicação. |
| pauseUbiApp | ubiAppID | pauseApp | Colocar a aplicação em estado de espera. |
| resumeUbiApp | ubiAppID | resumeApp | Permitir à aplicação voltar ao estado de execução após ter sido colocada em estado de espera. |
| destroyUbiApp | ubiAppID | destroyApp | Finalizar a aplicação. |

## 4.6 Descrição das Camadas da 2SVM

A 2SVM-*Controller* é a configuração da máquina de execução de modelos que executa na controladora do espaço inteligente. Ela tem por objetivo coordenar as interações entre os elementos do espaço inteligente. O modelo criado pelo usuário é interpretado pela 2SVM-*Controller*, que gera um conjunto de comandos com o intuito de definir uma programação para o espaço inteligente. A 2SVM-*Controller* também mantém o M@RT global, com informações de todos os elementos ativos no espaço inteligente.

Já a 2SVM-*Client* é executada nos dispositivos dos usuários do espaço inteligente, e nos dispositivos pertencentes ao ambiente físico. O objetivo da 2SVM-*Client* é programar os dispositivos móveis e estáticos do ambiente, a partir de execução do modelo, por meio de comandos de mais baixo nível que são executados na camada de *Broker* da máquina. Ela possui apenas as camadas de *middleware* e *Broker* e, portanto, não fornece aos usuários do espaço inteligente um ambiente de desenvolvimento de modelos. Desta forma, para que um usuário possa criar seus modelos para o ambiente, é necessário que ele tenha credenciais de engenheiro ou usuário do sistema para acessar a 2SVM-*Controller*.

### 4.6.1 User-2S Interface

A Figura 4.9 ilustra a estrutura da camada *User-2S Interface*, que provê uma interface para utilização da plataforma. A camada permite ao engenheiro e usuário do sistema se autenticarem na máquina e criarem seus modelos. Apenas ao engenheiro do sistema é permitida a função de cadastrar usuários que poderão pertencer ao espaço inteligente. Essa camada é composta pelos seguintes componentes:

- *User Management Interface*: gerencia a autenticação do engenheiro e do usuário do sistema na máquina. Este componente também permite que o engenheiro cadastre usuários do espaço inteligente. Para cada tipo de usuário criado, o engenheiro define um papel de usuário, que o usuário assumirá quando entrar no espaço inteligente.
- *Smart Space Modeling Environment*: este componente corresponde ao ambiente de modelagem. Ele fornece dois editores gráficos, um para o engenheiro e outro para o usuário do sistema, que incorporaram as abstrações de cada um dos metamodelos. Como o metamodelo definido para o engenheiro é distinto daquele definido para o usuário do sistema, as construções empregadas por cada um deles para a criação dos modelos também são distintas;
- *Model Repository*: armazena e recupera modelos criados pelo engenheiro e usuário do sistema para espaços inteligentes. Os modelos armazenados nesse repositório podem ser utilizados a *posteriori* pelos usuários para serem implantados no ambiente. Por isso, temos neste caso um repositório de modelos reutilizáveis. Tanto o

engenheiro quanto o usuário do sistema podem realizar modificações nos modelos armazenados neste repositório, de maneira a atender as especificidades do cenário para o qual o modelo será implantado.

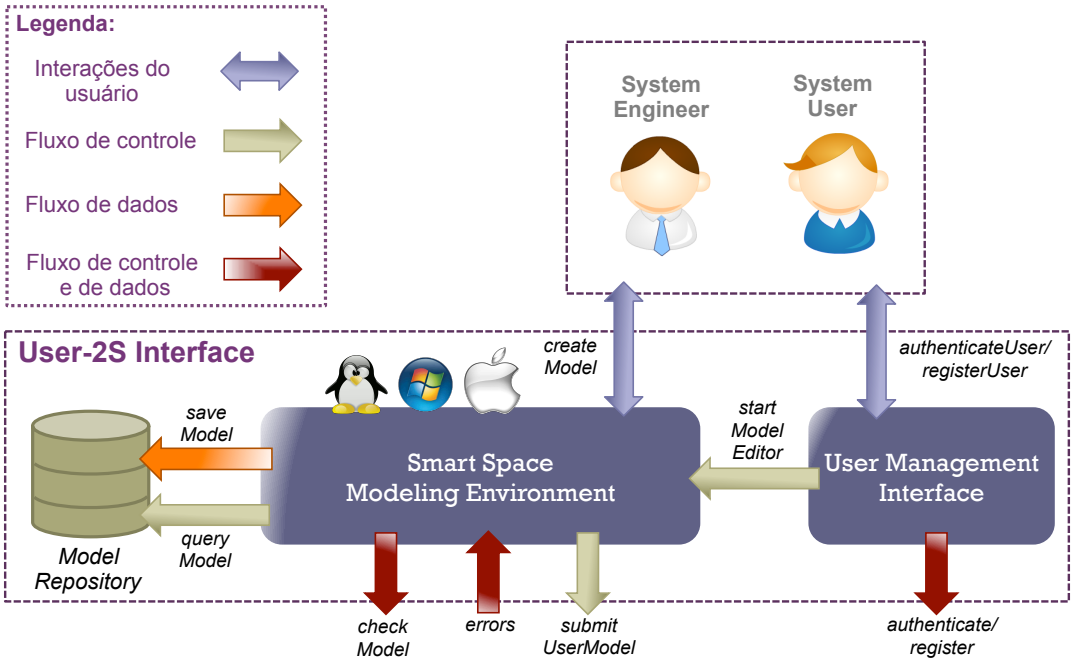



**Figura 4.9:** *Camada User-2S Interface da 2SVM-Controller*

### 4.6.2 Model Processing

A camada *Model Processing* está presente apenas na 2SVM-*Controller*. Este *design rationale* foi realizado com o intuito de concentrar no servidor todo o processamento do modelo, garantindo assim a visão global do espaço inteligente na controladora do ambiente. A Figura 4.10 ilustra os componentes internos desta camada.

- *Model Processing Manager*: o componente *Model Processing Manager* realiza a *interface* com a camada superior da 2SVM-*Controller*. A partir deste componente é possível fazer chamadas aos componentes internos da camada *Model Processing*. Podemos, por exemplo, solicitar a esta camada que faça as checagens relativas aos modelos do engenheiro e usuário do sistema, através da operação `checkModel`, bem como submeter o modelo para ser executado por meio da operação `submit`. Este componente ainda orquestra o processo de checagem dos modelos do engenheiro e usuário do sistema;
- *Model Parser*: este componente realiza o *parsing* dos elementos do modelo, e os agrega por metatipos. Desta forma, temos um conjunto de elementos do modelo para cada um dos metatipos `UserRole`, `SmartObject`, `UbiquitousApplication` e `Policy`. Após isso, estes elementos são encaminhados para o componente *Model*

*Cache*, onde ficarão armazenados para posterior acesso pelos componentes *Internal Consistency*, *Conformance Checking* e *Model Comparator*;

- *Model Cache*: este componente armazena após a operação de *parsing*, de forma temporária, os elementos do modelo do engenheiro e do usuário do sistema, para que possam ser acessados pelos demais componentes da camada. Nele, os elementos do modelo são agrupados por metatipos, ou seja, temos elementos do metatipo `UserRole`, `SmartObject`, `UbiquitousApplication` e `Policy`;
- *Internal Consistency*: verifica se os elementos utilizados para descrever a parte comportamental do modelo estão presentes na parte estrutural, criada pelo engenheiro ou usuário do sistema, ou seja, para que um elemento seja utilizado na criação das políticas, ele deve ter sido anteriormente definido na parte estrutural do modelo. Caso isso não ocorra, uma mensagem de erro é enviada para *Smart Space Modeling Environment* presente na camada *User 2S Interface*. O Algoritmo 4.1 corresponde ao pseudocódigo de checagem de consistência interna do modelo. Inicialmente, os elementos da parte estrutural do modelo são obtidos a partir do componente *Model Cache*, bem como a parte comportamental, através das políticas. Para esta checagem, comparamos cada uma das expressões de cada política com os elementos da parte estrutural. Caso algum elemento declarado na política não esteja presente no modelo, esta política é considerada inválida e deve ser reescrita novamente;

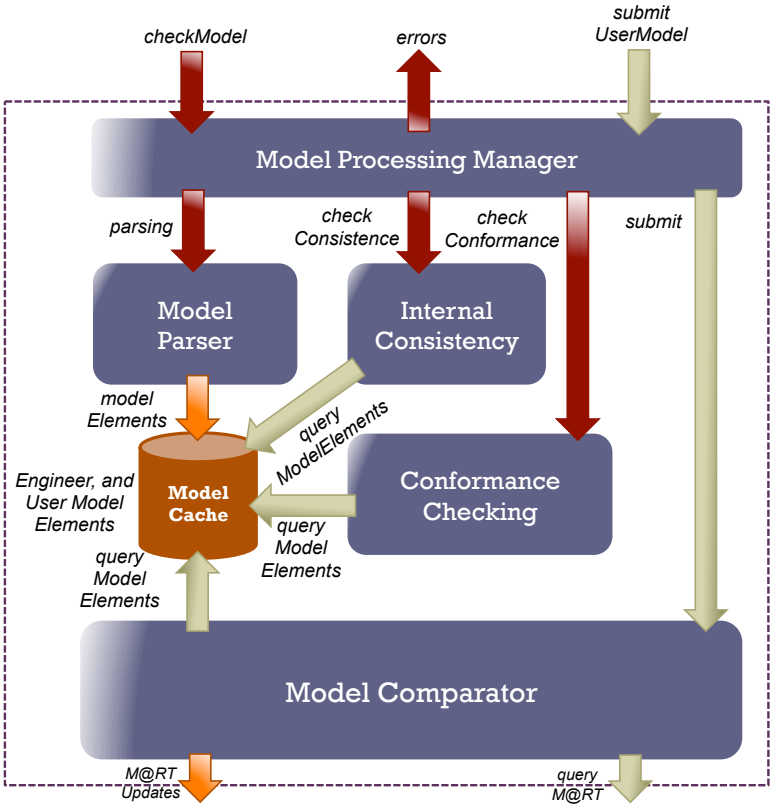



**Figura 4.10:** *Camada Model Processing da 2SVM-Controller*

**Algoritmo 4.1**: Internal Consistency

```
Result: Expressões incorretas nas políticas
Variable declaration:
elementoParteEstrutural[] ← hash de elementos do modelo referentes à parte
estrutural;
politica[] ← lista de condições das políticas do modelo;
expressaoPolitica[] ← lista de expressões das condições das políticas;
expressaoIncorreta[] ← lista de expressões escritas incorretamente;
begin
    elementoParteEstrutural[] ← obter de Model Cache os elementos do modelo
    relativos a sua parte estrutural. Estes elementos são armazenados em Model
    Cache de acordo com seu metatipo;
    politica[] ← obter de Model Cache a lista de condições das políticas do modelo;
    expressaoPolitica[] ← obter de politica[] cada expressão de cada uma de suas
    condições;
    while (expressaoPolitica[].size() > 0) do
        if (!expressaoPolitica[].equals.(key(elementoParteEstrutural[]))) then
            expressaoIncorreta[] ← expressaoPolitica[];
end
```

- *Conformance Checking*: este componente é executado apenas quando o usuário do sistema cria seu modelo, que deve estar em conformidade com o modelo do engenheiro do sistema. Para fazer a checagem de conformidade, o componente verifica se todos os elementos criados no modelo do usuário são subtipos daqueles definidos no modelo do engenheiro. Por exemplo, para o tipo *smartphoneDevice*, definido no modelo do engenheiro, o usuário do sistema poderá criar o subtipo *smartphone*. Desta forma, o usuário utiliza os blocos de construção básicos criados pelo engenheiro para definir os tipos dos elementos de seu modelo. A checagem de conformidade ainda tem como atribuição verificar se o modelo do usuário do sistema obedece as restrições de modelagem impostas pelo engenheiro em seu modelo. Essas restrições de modelagem dizem respeito às associações que podem ser criadas entre os elementos do modelo do usuário do sistema, o que, na prática, impõe limitações aos usuários do espaço inteligente quanto ao acesso a determinados recursos. Por exemplo, o engenheiro do sistema pode definir que apenas os funcionários da organização podem utilizar a aplicação ubíqua de compartilhamento de documentos encriptados, impedindo que qualquer convidado ou membro externo que esteja participando da reunião a utilize. O Algoritmo 4.2 apresenta o pseudocódigo correspondente a esse mecanismo de checagem de conformidade;
- *Model Comparator*: este componente tem por objetivo comparar os modelos submetidos pelos usuários do sistema com o modelo em execução no espaço inteligente. As comparações são sempre feitas pelos metatipos dos elementos, ou seja,

**Algoritmo 4.2**: Conformance Checking

```
Result: Elementos inconsistentes do modelo
Variable declaration:
elementoModeloES[] ← hash de elementos do modelo do engenheiro do sistema;
elementoModeloUS[] ← lista de elementos do modelo do usuário do sistema;
elementoInconsistente[] ← elementos inconsistentes do modelo do usuário do
sistema;
begin
    elementoModeloES[] ← obter de Model Cache os elementos do modelo do
    engenheiro do sistema. Estes elementos são armazenados em Model Cache de
    acordo com seu metatipo;
    elementoModeloUS[] ← obter de Model Cache os elementos do modelo do
    usuário do sistema;
    while (elementoModeloUS[].size() > 0) do
        if (!elementoMode-
        loUS[].superType().equals(key(elementoModeloES[].type())))
        then
            elementoInconsistente[] == elementoModeloUS[].type();
end
```

os tipos de elementos criados a partir do metatipo `UserRole` sempre são comparados com outros elementos do metatipo `UserRole`, e assim por diante. Usuários do sistema submetem os modelos, uma vez que seus modelos possuem a real intenção para o espaço inteligente. Os modelos submetidos podem ter sido criados a partir do "zero", ou podem ter sido criados a partir de modelos já existentes. Embora existam essas duas possibilidades, o componente *Model Comparator* utiliza o mesmo processo para a comparação. Para comparar o modelo submetido com o modelo em execução, este componente consulta os elementos armazenados em *Model Cache* com aqueles presentes na camada subjacente, neste caso, presentes em *Global M@RT*, e esta comparação ocorre sempre em nível de tipos e é feita em duas etapas: *i)* Comparar os elementos de *Model Cache* com aqueles presentes no componente Global M@RT. Os elementos que estão presentes em *Model Cache* e não estão presentes em Global M@RT são os novos elementos, que deverão ser adicionados a Global M@RT; e *ii)* Comparar os elementos de Global M@RT com aqueles presentes em *Model Cache*, de tal forma a obter os elementos a serem removidos. O Algoritmo 4.3 ilustra o pseudocódigo do mecanismo de comparação entre os modelos de *Model Comparator*.

**Algoritmo 4.3**: Model Comparator

**Result**: Lista de elementos que devem ser adicionados e removidos do modelo em tempo de execução

**Variable declaration:**

elementoModeloUS[] ← lista de elementos do modelo do usuário do sistema;
elementoM@RT[] ← lista de elementos do modelo em tempo de execução em nível de tipos;
elementoAdicionado[] ← lista elementos a serem adicionados à Global M@RT;
elementoRemovido[] ← lista de elementos a serem removidos de Global M@RT;
elementoMantido[] ← *hash* de elementos a serem mantidos em Global M@RT;

**begin**

elementoModeloUS[] ← obter de *Model Cache* os elementos do modelo do usuário do sistema;
elementoM@RT[] ← obter de *Global M@RT* a lista dos elementos em nível de tipos;
elementoAdicionado[] ← (elementoModeloUS[] − (elementoModeloUS[] ⋂ elementoM@RT[]))
elementoRemovido[] ← (elementoM@RT[] − (elementoModeloUS[] ⋂ elementoM@RT[]))

**end**

### 4.6.3 User-Centric 2S Middleware

A camada *User-Centric 2S Middleware* de espaços inteligentes está presente na 2SVM-*Controller* e na 2SVM-*Client*. Suas funcionalidades e componentes se complementam, uma vez que algumas operações são realizadas na máquina controladora e outras nas máquinas cliente. A camada de *middleware* da 2SVM-*Controller* e da 2SVM-*Client* tem diferentes configurações e executam em diferentes dispositivos do espaço inteligente. Cada uma dessas máquinas de execução de modelos possui uma instância da camada de *middleware*.

A camada de *middleware* da 2SVM-*Controller* mantém o modelo em tempo de execução global do espaço inteligente. Ela ainda tem um mecanismo para identificar os papéis dos usuários que entram no espaço inteligente, além dos tipos dos dispositivos. Esta camada também lida com demais eventos gerados no ambiente de computação ubíqua, como por exemplo, mudança de localização de dispositivos e usuários, e demais mudanças contexto, como temperatura, umidade, luminosidade do ambiente, dentre outros. Estes eventos são gerados por quaisquer objetos inteligentes que estejam no espaço inteligente.

Na 2SVM-*Client*, a camada de *middleware* é responsável por manter o modelo em tempo de execução local do dispositivo, e encaminhar as macros recebidas da camada de *middleware* da controladora para serem executados na camada de *broker*. Além disso, ela recebe notificações de eventos gerados localmente no próprio dispositivo, ou seja, pelos sensores em nível de hardware e software presentes no objeto inteligente. Como

mencionamos anteriormente, estes eventos locais são chamados de eventos do ambiente assim que eles são propagados no espaço inteligente para a 2SVM-*Controller*.

A Figura 4.11 ilustra os componentes da camada *User-Centric 2S Middleware* na 2SVM-*Controller* e na 2SVM-*Client*.

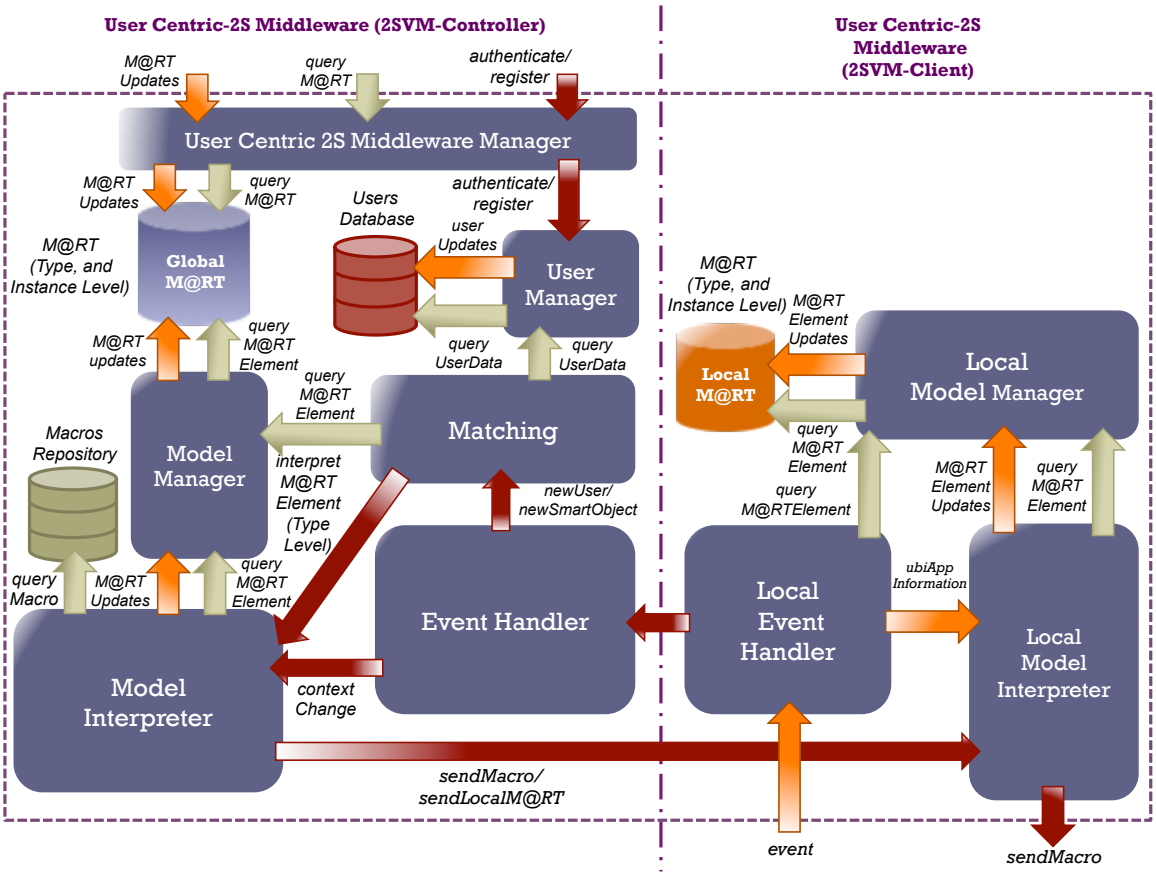



**Figura 4.11:** *Camada User-Centric 2S Middleware da 2SVM-Controller e da 2SVM-Client*

Os componentes da camada *User-Centric 2S Middleware* são:

- *User Manager*: gerencia a autenticação do engenheiro e do usuário do sistema na controladora. Ao engenheiro é permitido cadastrar os usuários que poderão pertencer ao espaço inteligente. Neste cadastro, é inserido o nome e o tipo de papel de usuário que ele terá naquele ambiente. Para realizar estas operações, o *User Manager* faz acesso à base de dados de usuários, *Users Database*;
- *Users Database*: base de dados de usuários. Cada um dos usuários que pode participar do espaço inteligente é previamente cadastrado no sistema pelo engenheiro e adicionado a essa base de dados, juntamente com os respectivos papéis que terão no ambiente;
- *Matching*: este componente é responsável por descobrir os tipos de elementos que entram no espaço inteligente. Para fazer isso, o componente utiliza o conjunto de

informações que esses elementos trazem consigo ao se anunciarem no ambiente. Por exemplo, os objetos inteligentes trazem consigo suas informações na forma de características, tais como sensores, atuadores e demais interfaces de entrada e saída. Já os usuários, trazem informações pessoais dele. Após o componente *Matching* receber essas informações, ele inicia o processo de descoberta do papel do usuário e o tipo do objeto inteligente. Para descobrir o papel de um usuário, o componente faz uma consulta à base de dados de usuários. Se o usuário estiver na base de dados, ela retornará seu papel; caso contrário, será atribuído ao usuário o papel *guest*. Este papel possui acesso apenas a determinados elementos do espaço inteligente, e varia de acordo com as regras definidas pela organização. O papel do usuário *default* é utilizado quando se deseja delimitar o acesso aos recursos do espaço inteligente, entre eles, objetos inteligentes e aplicações ubíquas. Após a descoberta do papel do usuário e do tipo do objeto inteligente, o componente *Matching* executa a chamada *interpretM@RTElement(modelElement)* em *Model Interpreter*, passando para ele o elemento do modelo obtido a partir do M@RT Global, acessado por meio do componente *Model Manager*;

- *Model Interpreter*:
  - interpreta os elementos do modelo para executar operações relativas a cada um deles. Interpretar um elemento do modelo é verificar suas associações com os demais elementos do modelo e, após isso, selecionar as macros adequadas do componente *Macros Repository*;
  - solicita ao componente *Model Manager* a atualização do M@RT em nível de instância;
  - realiza consultas ao repositório de macros para selecionar aquelas que serão executadas na camada de *Broker*. Esta seleção é feita de acordo com o elemento do modelo em questão, o que significa que, para cada elemento do modelo, temos macros correspondentes para adicionar usuários no ambiente, configurar os objetos inteligentes, realizar as operações das aplicações ubíquas, entre outros;
  - lida com mudanças de contexto ocorridas no ambiente e engatilha as políticas correspondentes. Para cada evento do ambiente podemos ter uma política associada. A finalidade destas políticas é dirigir o comportamento dinâmico do ambiente por meio de ações que são executadas no espaço. Essas ações, que ocorrem por meio de operações, são realizadas através de atuadores que podem estar em nível de hardware e software;
  - envia o M@RT local para o dispositivo. Este M@RT local contém informações referentes ao usuário, ao dispositivo e às aplicações que podem ser executadas no dispositivo;

- *Model Manager*: este componente acessa o modelo em tempo de execução global para consultar e atualizar suas informações;
- *Event Handler*: este componente recebe os eventos gerados no ambiente por meio de dois tratadores de eventos. A Figura 4.12 ilustra os subcomponentes internos deste componente.
  - *New User Handler*: tratador responsável por receber eventos do ambiente relacionados à entrada de novos usuários no espaço inteligente;
  - *New Smart Object Handler*: tratador responsável por receber eventos do ambiente relacionados à entrada de novos dispositivos no espaço inteligente;
  - *Context Change Handler*: tratador encarregado de receber eventos do ambiente relativos a demais mudanças de contexto no ambiente. Para fazer isso, este componente conta com um conjunto de tratadores que recebem eventos dos mais diversos tipos para tratá-los. Após receber estes eventos, o tratador descobre o tipo do evento a partir do conjunto de informações que nele estão contidas, enviadas a partir do tratador de eventos do objeto inteligente. Como exemplo, podemos citar a mudança de localização de usuários ou dispositivos, alertas sobre o baixo nível de bateria do dispositivo, mudanças de temperatura, umidade, luminosidade do ambiente, data e hora, entre outros;
  - *Ubiquitous Application Handler*: tratador responsável por receber eventos do ambiente relacionados às operações realizadas pelas aplicações nos dispositivos. Para fazer isso, temos um conjunto de tratadores que recebem eventos relacionados aos estados da aplicação, iniciar, pausar, continuar e finalizar;

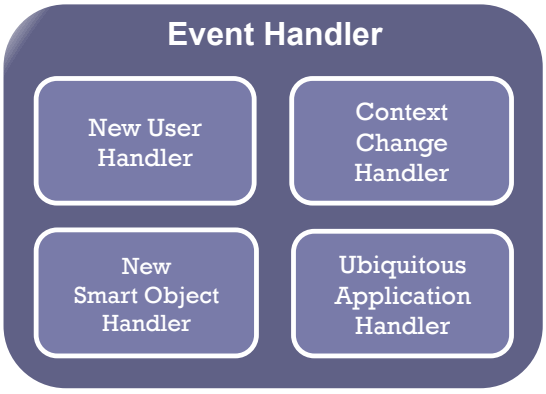



**Figura 4.12:** *Sub-componentes do componente Event Handler*

- *Macros Repository*: armazena as macros que serão executadas na camada de *Broker* da 2SVM-*Client*. Este repositório possui macros que são agrupadas por tipo, como por exemplo, macros para usuários, para objetos inteligentes e para aplicações ubíquas;

- *Global M@RT (Type, and Instance Level)*: modelo em tempo de execução global do espaço inteligente em nível de tipo e de instância. Ele possui informações de todos os elementos ativos do espaço inteligente, referente aos tipos definidos pelo usuário do sistema em seu modelo, e referente às instâncias.

A camada de *middleware* da 2SVM-*Client* é responsável por processar os eventos locais gerados no dispositivo, encaminhar para a camada de *Broker* as macros recebidas da 2SVM-*Controller*, e por executar e manter o M@RT local do dispositivo. Os componentes que constituem esta camada são:

- *Local Model Interpreter*:
  - Recebe as macros encaminhadas da controladora, e as encaminha para serem executadas na camada de *Broker*, na forma de comandos;
  - Recebe o M@RT local do dispositivo em nível de tipo e de instância, encaminhado pela 2SVM-*Controller* e solitica ao componente *Local Model Manager* a atualização do modelo em tempo de execução;
  - Atualiza o M@RT local em nível de tipo e de instância através do componente *Local Model Manager*;
  - Recebe notificações do componente *Local Event Handler* sobre eventos relacionados à aplicação ubíqua em execução no dispositivo. Por exemplo, quando o usuário inicia uma aplicação, o componente *Local Model Interpreter* é notificado deste evento. Ele faz uma consulta ao componente *Local Model Manager* a fim de verificar se existe uma aplicação ubíqua daquele tipo no M@RT local. Caso sim, ele solicita ao *Local Model Manager* a atualização do M@RT local em nível de instância;
- *Local Model Manager*: acessa o modelo em tempo de execução local para consultar e atualizar suas informações;
- *Local Event Handler*: este componente possui os mesmos subcomponentes do tratador de eventos da 2SVM-*Controller*. Ele lida apenas com eventos locais e adiciona informações relativas ao M@RT local antes de enviá-los ao tratador da 2SVM-*Controller*:
  - *New User Handler*: tratador responsável por receber eventos relacionados à entrada de usuários no espaço inteligente. Quando o dispositivo detecta que um usuário entrou em um espaço inteligente, a camada de *Broker* recebe este evento, gerado pelo próprio dispositivo, e encaminha para *New User Handler* as informações relativas ao usuário. Estas informações são obtidas a partir do sistema operacional do objeto inteligente, por meio de um mecanismo presente na camada de *Broker*. Após isso, este tratador de evento gera um evento para Event Handler da 2SVM-*Controller*;

  - *New Smart Object Handler*: tratador responsável por receber eventos relacionados à entrada de dispositivos em um espaço inteligente. A camada de *broker* da 2SVM-*Client* recebe este evento, adiciona a ele o conjunto de informações do dispositivo, tais como suas capacidades de *hardware* e *software*, e encaminha para *New Smart Object Handler*. Este tratador de eventos relacionado então gera um evento para o *New Smart Object Handler* da 2SVM-*Controller*;
  - *Context Change Handler*: tratador encarregado de receber eventos relativos a demais mudanças de contexto detectadas pelo dispositivo. A camada de *Broker* da 2SVM-*Client* recebe eventos deste tipo, e encaminha para *Context Change Handler*. Após isso, ele adiciona ao evento detectado informações relacionadas ao M@RT local, como o informações do usuário, do dispositivo e das aplicações em execução. Após isso, ele encaminha ao tratador de eventos da controladora do espaço inteligente;
  - *Ubiquitous Application Handler*: tratador responsável por receber eventos do ambiente relacionados aos estados da aplicação no dispositivo, tais como, iniciar, pausar, resumir e finalizar. Para cada uma dessas operações, temos um tratador de eventos responsável por lidar com eles.

- *Local M@RT (Type, and Instance Level)*: o modelo em tempo de execução local armazena informações de tipos e instância referentes ao usuário, dispositivo, aplicações e políticas que estão executando. Para o usuário, temos seu papel e as informações do usuário coletadas a partir do dispositivo que está utilizando, como por exemplo, seu nome, *login* e *e-mail*. Estas informações denominamos informações em nível de instância. Para os objetos inteligentes, temos seu tipo e o conjunto de informações da instância, como por exemplo, sensores, atuadores e interfaces de entrada e saída. Por fim, temos as aplicações ubíquas, em que armazenamos informações sobre seu tipo e sobre a instância que está executando. A instância da aplicação contém o nome da aplicação e a linguagem de programação em que foi implementada, além do estado em que ela se encontra, iniciada, pausada, continuada ou finalizada.

### 4.6.4 2S Broker

A camada *2S Broker* da 2SVM-*Client* tem por objetivo lidar com o conjunto dos recursos de *hardware* e *software* do objeto inteligente que poderão ser usados no espaço inteligente. Também é responsável por receber os eventos gerados pelo dispositivo, como por exemplo, mudança de localização, detecção de baixo nível do índice de bateria, inicialização e finalização de aplicações, entre outros. Os componentes desta camada são ilustrados na Figura 4.13 e descritos a seguir.

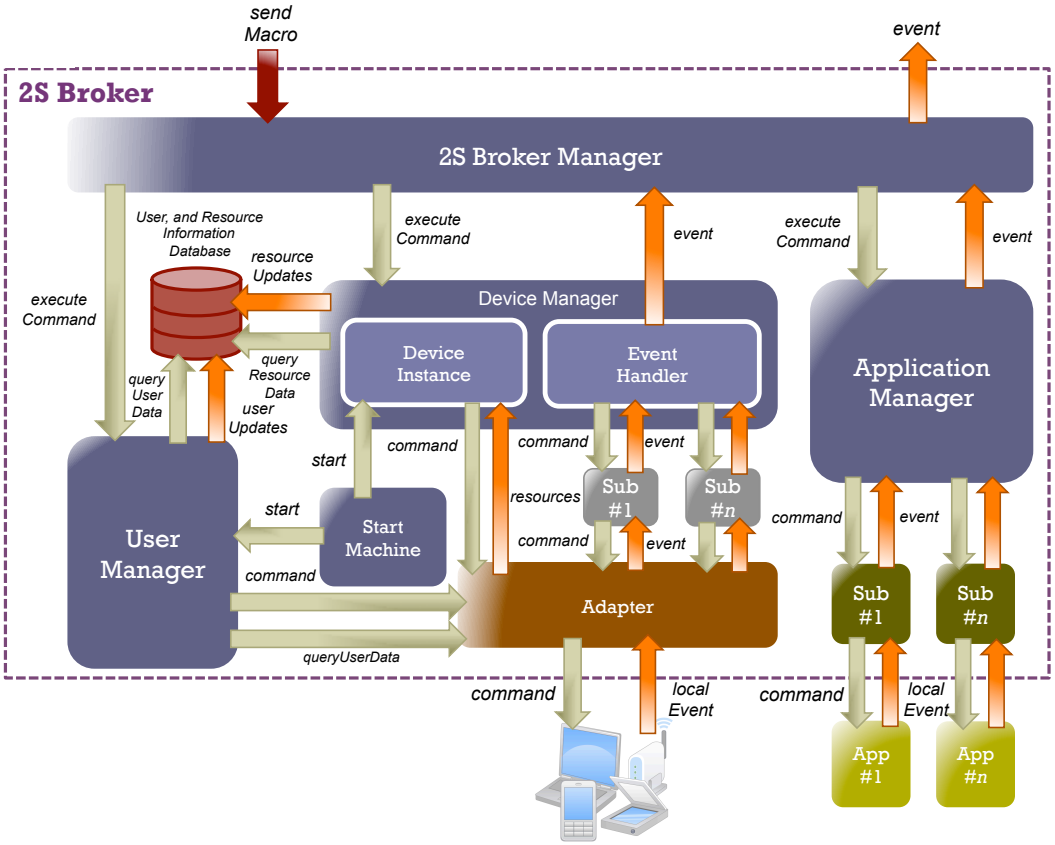



**Figura 4.13:** *Camada 2S Broker da 2SVM-Client*

- *2S Broker Manager*: este componente realiza a *interface* com a camada superior da 2SVM-*Client*. Ele recebe macros encaminhadas pela camada *User-Centric 2S Middleware*, realiza o *parsing* de seus elementos e executa cada um dos comandos pertencentes às macros, no componente correspondente, *User Manager*, *Device Manager* ou *Application Manager*. O componente *2S Broker Manager* encaminha para *User Manager* os comandos referentes aos usuários, para *Device Manager*, os comandos pertinentes ao objeto inteligente e os comandos relativos à aplicação ubíqua são encaminhados para o componente *Application Manager*;
- *User Manager*: este componente acessa o conjunto de informações do usuário e as armazena na base de dados de usuários e recursos. Estas informações são coletadas a partir do sistema operacional. Além disso, este componente tem como atribuição autenticar o usuário no espaço inteligente. Esta autenticação ocorre quando a macro *addUser* é executada no dispositivo;
- *Device Manager*: lida com os recursos do dispositivo. Para que esses recursos sejam usados pela 2SVM-*Client*, é necessária a criação de adaptadores específicos, que variam de acordo com os sistemas operacionais que executam nos dispositivos. Estes adaptadores interagem com os recursos de hardware e software do objeto inteligente e devem ser implementados pelo engenheiro do sistema, que possui

conhecimentos mais específicos de programação. Este componente é formado por dois sub-componentes:

  - *Device Instance*: este componente acessa os recursos do dispositivo e identifica seu conjunto de características, extraídas a partir do sistema operacional do objeto inteligente. Elas são armazenadas na base de dados de informações de usuários e recursos para que, posteriormente, sejam acessadas por outros componentes;
  - *Event Handler*: recebe eventos de interesse do dispositivo. Após isso, esses eventos são encaminhados para o tratador de eventos correspondente na camada de *middleware*, que pode ser o componente *New User Handler*, *New Smart Object Handler*, *Context Change Handler* e *Ubiquitous Application Handler*.

- *Application Manager*: este componente é encarregado de gerenciar as instâncias das aplicações ativas no dispositivo - por meio das operações iniciar e finalizar - além de salvar e restaurar o estado das aplicações;
- *Instance Information Database*: armazena informações relativas ao conjunto de características do dispositivo, bem como de seu proprietário. Estas informações são acessadas pela camada superior, a fim de serem encaminhadas à 2SVM-*Controller* do espaço inteligente no momento do anúncio do dispositivo;
- *Start Machine*: este componente é executado assim que a 2SVM-*Client* é iniciada. Seu objetivo é identificar as informações pertinentes ao usuário, proprietário do dispositivo e o conjunto de características dos recursos, tais como, sensores, atuadores, e demais interfaces de entrada e saída;
- *Adapter*: permite aos componentes da camada de *Broker* comunicarem com os recursos do dispositivo. Para cada tipo de dispositivo existe um determinado adaptador, que trata de maneira específica os recursos de cada dispositivo. Para este componente, utilizamos o padrão de projeto *Adapter* [49].

## 4.7 Conclusão

Neste capítulo apresentamos a arquitetura da máquina de execução de modelos 2SVM. Esta máquina tem por objetivo executar os modelos desenvolvidos pelo engenheiro e usuário do sistema e implantá-los nos objetos inteligentes do ambiente de computação ubíqua.

O capítulo inicia com uma visão geral da arquitetura da máquina, que foi baseada na *Communication Virtual Machine* [35]. Nesta seção apresentamos as quatro camadas da máquina em seus diferentes níveis de abstração. Na seção seguinte, discutimos sobre

a arquitetura de distribuição, que basicamente é composta por dois tipos de máquinas de execução de modelos, uma para os dispositivos que estão presentes no espaço inteligente, denominada 2SVM-*Client*, e outra, pertencente à controladora do ambiente, denominada 2SVM-*Controller*.

Após isso, apresentamos o modelo de interação entre as camadas da 2SVM. A comunicação entre as camadas da 2SVM pode ser síncrona, em que uma camada interage com a outra através de chamadas bloqueantes, e assíncrona, em que a comunicação entre as camadas ocorre a partir da geração de eventos.

A 2SVM lida com modelos criados pelos usuários que são tratados pela máquina como modelos em tempo de execução. Na 2SVM-*Controller*, temos o modelo em tempo de execução global, que contém as informações de todos os elementos do espaço inteligente, entre eles, usuários, objetos inteligentes e aplicações ubíquas em execução. Já na 2SVM-*Client*, temos o modelo em tempo de execução local, que contém informações relativas apenas a esse objeto inteligente em si. Por exemplo, temos informações do dispositivo, do usuário que está usando ele e de suas aplicações ubíquas.

Depois de apresentarmos estes modelos em tempo de execução, discutimos sobre as macros, que são formadas por comandos de mais baixo nível. Elas são armazenadas em um repositório e agrupadas de acordo com os metatipos dos elementos definidos no metamodelo da linguagem, ou seja, `UserRole`, `SmartObject`, `UbiquitousApplication` e `Policy`, e selecionadas para serem executadas na camada de *Broker* da máquina. Desta forma, temos macros para papéis do usuário, para objetos inteligentes, para aplicações ubíquas e para políticas.

Por fim, descrevemos cada uma das camadas da máquina com seus respectivos componentes internos. Nós apresentamos em detalhes a interação entre as camadas por meio das interfaces e o funcionamento dos componentes internos de cada uma delas. No Capítulo 3 apresentamos a linguagem de modelagem de espaços inteligentes e neste capítulo, como a máquina de execução processa e executa estes modelos. No capítulo seguinte, discutiremos como validamos nossos conceitos criados acerca da linguagem e da máquina, por meio da implementação.

A Figura 4.14 apresenta uma visão geral de todas as camadas da 2SVM-*Controller* e 2SVM-*Client*, bem como de seus respectivos componentes internos. Nela, podemos observar de uma maneira geral a interação entre cada uma das camadas da 2SVM, em particular, a presença de *n* instâncias da camada *middleware*, bem como os componentes internos de cada uma dessas camadas.

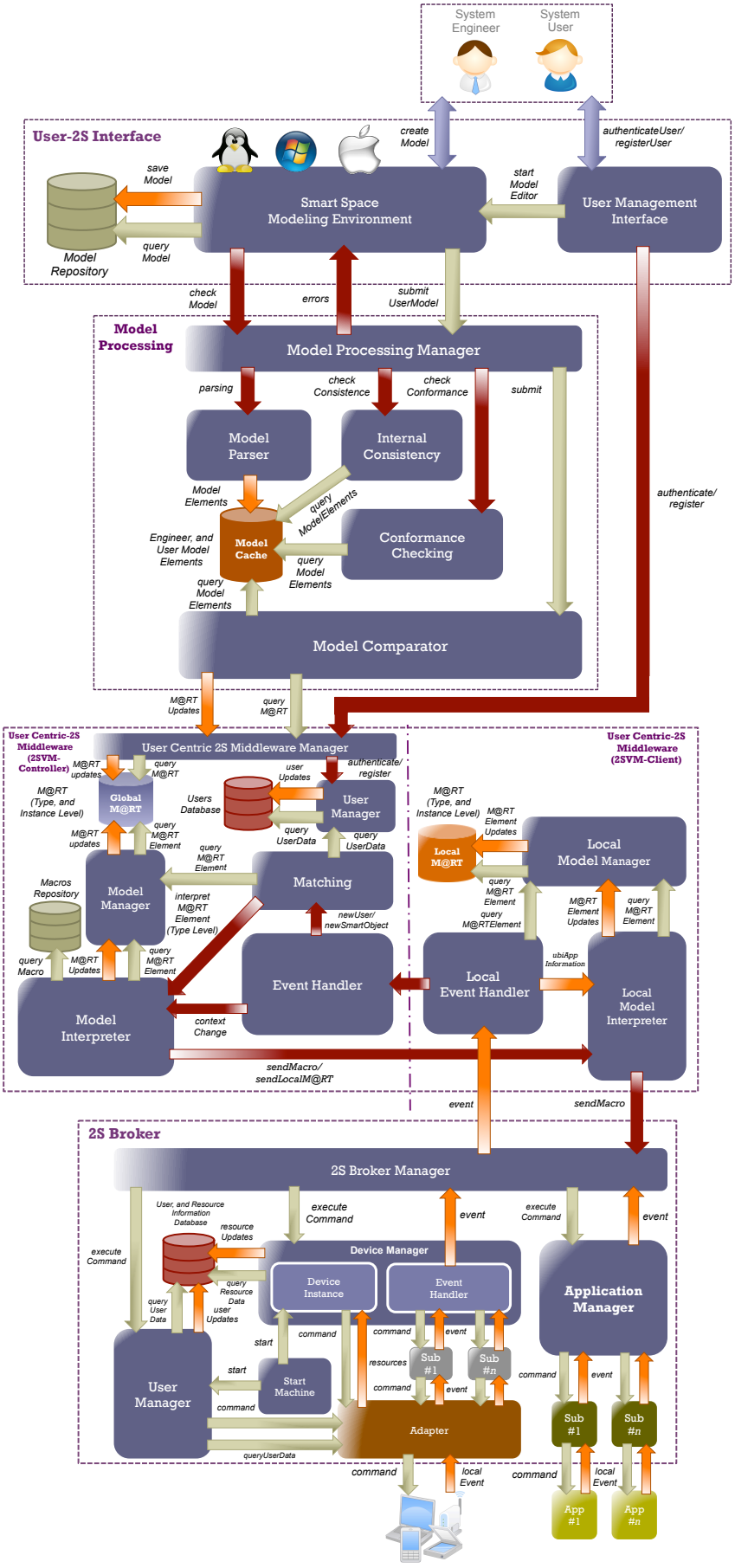



**Figura 4.14:** *Visão Geral das camadas da 2SVM-Controller e 2SVM-Client*

CAPÍTULO 5

# Implementação da 2SML e da 2SVM

A linguagem de modelagem de espaços inteligentes 2SML, permite aos programadores desenvolver modelos que definem a estrutura e comportamento dos elementos de ambientes de computação ubíqua. Entretanto, para que estes modelos sejam executados, é necessário que haja um mecanismo para implantá-los nos dispositivos destes ambientes. Para isso, desenvolvemos a máquina de execução de modelos 2SVM.

Este capítulo tem por objetivo apresentar a implementação da linguagem 2SML e da máquina 2SVM. Nele, expomos as ferramentas utilizadas e os mecanismos criados para realizar a arquitetura proposta. O restante do capítulo está organizado da seguinte forma: A Seção 5.1 apresenta a implementação da linguagem de modelagem 2SML. A Seção 5.2 apresenta a máquina de execução de modelos, bem como os detalhes de sua implementação.

## 5.1 Linguagem de Modelagem 2SML

### 5.1.1 Editor Gráfico de Modelagem de Espaços Inteligentes

A criação de modelos de programação para espaços inteligentes é realizada a partir de um editor gráfico de modelagem, implementado utilizando o *Eclipse Graphical Modeling Framework* (GMF) [121, 75]. Além deste *framework*, utilizamos as linguagens da família *Epsilon* [73] e sua construção teve como apoio a ferramenta *EuGENia*, que também faz parte desta família. A Figura 5.1 ilustra o editor gráfico de modelagem da 2SML.

A janela da ferramenta é dividida basicamente em duas partes:

1. Área de criação dos modelos do espaço inteligente: neste local, o engenheiro e o usuário do sistema definem os elementos que poderão participar do ambiente. Em relação à parte estrutural dos modelos, podem ser criados papéis de usuário, tipos de *smart objects*, tipos de aplicações, bem como associaçõe entre esses tipos. Quanto à parte comportamental, eles criam políticas com seus respectivos eventos, condições e ações;

2. Paleta de elementos de modelagem, na qual temos duas divisões:

    1 Elementos que chamamos de objetos ou nós, que são os metatipos utilizados para definir os tipos de elementos do espaço inteligente. Sendo assim, temos `User Role`, `Smart Object`, `Feature Ubiquitous Application`, `Policy`, `Event`, `Condition` e `Action`;

    2 Conexões, que permitem estabelecer relacionamentos entre os elementos definidos no modelo, como `CanUse`, `IsOwnerOf`, `Hosts`, `Has` e `IsComposedOf`.

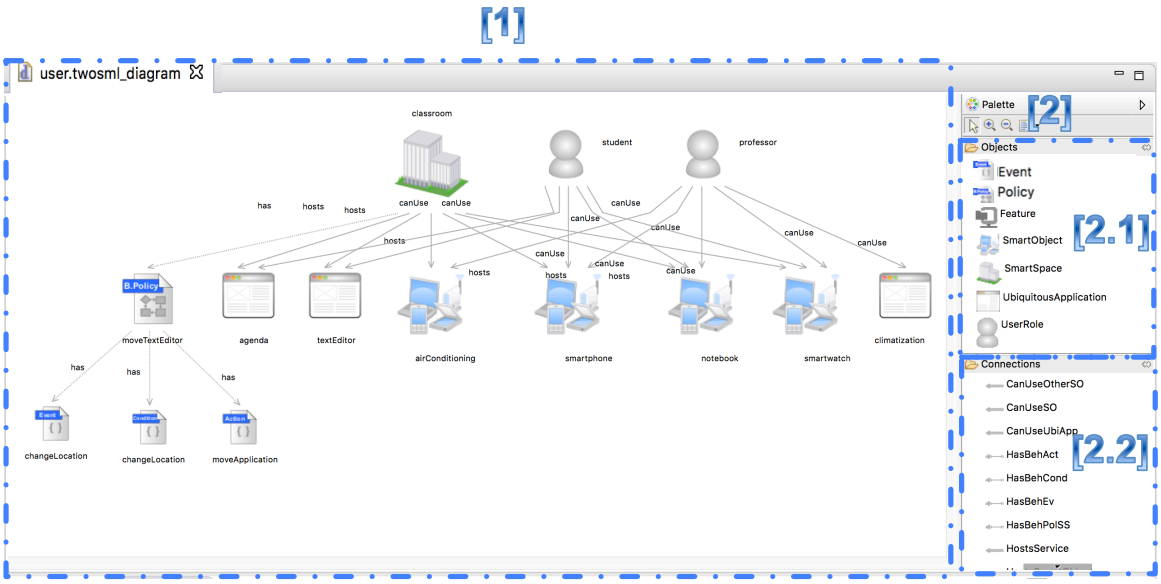


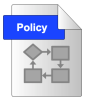
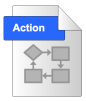

**Figura 5.1:** *Editor Gráfico da Linguagem 2SML*

Os modelos criados a partir da ferramenta de modelagem são arquivos XML salvos com as extensões `.twosml` e `.twosml_diagram`. O arquivo com `.twosml` é uma instância do metamodelo 2SML com os elementos definidos pelo engenheiro ou usuário do sistema. Este arquivo é processado pela máquina de execução, já que contém a definição de todos os elementos do modelo que será implantado no espaço inteligente. Já o arquivo `.twosml_diagram`, permite à ferramenta mapear os elementos e as associações entre eles de forma gráfica dentro do editor de modelagem. Portanto, a cada nova inicialização do editor gráfico, elementos gráficos são recarregados a partir deste arquivo.

Para criar a ferramenta de modelagem devemos criar o metamodelo da linguagem. Primeiramente, criamos um metamodelo Ecore, que possui todos os metatipos utilizados na ferramenta de modelagem, `UserRole`, `SmartObject`, `UbiquitousApplication` e `Policy`, além dos relacionamentos permitidos entre cada um dos elementos, `CanUse`, `IsOwnerOf`, `Hosts`, `Has` e `IsComposedOf`. Após isso, convertemos o arquivo Ecore (.ecore) em um arquivo Emfatic (.emf) [33] a partir do próprio GMF e geramos o editor gráfico de modelagem por meio da ferramenta *EuGENia*.

A Tabela 5.1 apresenta a sintaxe concreta da 2SML.

**Tabela 5.1:** *Sintaxe concreta da 2SML*

| **Elemento (Metatipo)** | **Representação Gráfica** |
|---|---|
| `SmartSpace` |  |
| `UserRole` |  |
| `SmartObject` |  |
| `Feature` |  |
| `UbiquitousApplication` |  |
| `Policy` |  |
| `Event` |  |
| `Condition` |  |
| `Action` |  |
| Associações: `hosts`, `canUse`, `isOwnerOf`, `isComposedOf` |  |
| Associações: `hasPolicy`, `hasEvent`, `hasCondition`, `hasAction` |  |

### 5.1.2 Restrições de Modelagem

As restrições de modelagem são regras de validação dos modelos utilizadas pelo engenheiro para definir restrições sobre os modelos do usuário do sistema. Estas restrições impedem, por exemplo, que o usuário do sistema defina associações entre determinados tipos de elementos do ambiente.

Para definir restrições de modelagem, utilizamos regras de validação criadas com a linguagem *Epsilon Validation Language* (EVL) [45]. A linguagem EVL tem por objetivo oferecer funcionalidades da família *Epsilon*, sendo portanto utilizada para especificar e avaliar restrições, também denominadas invariantes. Restrições EVL são similares a restrições *Object Contraint Language* (OCL) [54], embora a linguagem EVL também ofereça suporte para as dependências entre restrições, como por exemplo: se a `Restrição X` falhar, não avalie a `Restrição Y`. EVL permite ainda mensagens de erros customizadas e a especificação de *quick fixes*, utilizado para reparar as inconsistências.

As regras EVL foram utilizadas para representar as restrições de modelagem impostas pelo engenheiro do sistema ao modelo do usuário. Em seu modelo, o engenheiro pode definir restrições quanto às associações que o usuário do sistema pode criar em seus modelos. Por exemplo, a organização pode ter normativas internas que impedem que visitantes tenham acesso aos controles de regulagem do ar-condicionado. Para efetivar esta restrição, o engenheiro do sistema pode criar uma restrição em seu modelo que impeça que o usuário do sistema crie uma associação entre o papel do usuário *guestUser* e a aplicação ubíqua *climatizationApp*, e entre o papel do usuário *guestUser* e o objeto inteligente do tipo *airConditioningDevice*. O Código 5.1 apresenta a definição dessa restrição de modelagem. A palavra chave `context` faz referência ao relacionamento do metatipo `CanUseUbiApp` em que elementos do supertipo *guestUser* não podem ser associados à elementos do supertipo *climatizationApp*. Estes elementos são utilizados no modelo do usuário do sistema para definir seus tipos. A linha 9 do Código 5.1 permite declarar mensagens de alerta que são exibidas caso a associação não permitida seja efetuada. Neste exemplo, a mensagem emite um alerta com os elementos que não podem ser relacionados, *guestUser* e *climatizationApp*.

**Código 5.1:** *Restrição de modelagem que impede a associação entre o papel do usuário "guestUser"e a aplicação ubíqua "climatizationApp"*

```
1  context CanUseUbiApp {
2    constraint CannotAssociate {
3      check  {
4          if ( (self.source.superType = "guestUser") and (self.target.superType = "
               climatizationApp") ) {
5             return false;
6          }
7          return true;
8        }
9      message : "Cannot associate " +self.source.name+ " to " +self.target.superType
           + "!"
```

```
10      fix {
11        title : "Delete association!"
12        do {
13          delete self;
14        }
15
16      }
17    }
18  }
```

O Código 5.2 apresenta a restrição de modelagem criada pelo engenheiro do sistema a associação entre o papel do usuário `guestUser` e o objeto inteligente do tipo `airConditioningDevice`. Assim como no exemplo do Código 5.1, a palavra `context` faz referência ao relacionamento definido entre o papel do usuário e o objeto inteligente.

**Código 5.2:** *Restrição de modelagem que impede a associação entre o papel do usuário "guest"e o smart object "airConditioning"*

```
1   context CanUseSmartObject {
2     constraint CannotAssociate {
3       check  {
4           if ( (self.source.superType = "guestUser") and (self.target.superType = "
                airConditioningDevice") ) {
5              return false;
6           }
7           return true;
8         }
9       message : "Cannot associate " +self.source.name+ " to " +self.target.superType
              + "!"
10      fix {
11        title : "Delete association!"
12        do {
13          delete self;
14        }
15
16      }
17    }
18  }
```

As restrições de modelagem são criadas pelo engenheiro para que possam delimitar o conjunto de relacionamentos entre os elementos do modelo do usuário do sistema.

## 5.2 Máquina de Execução de Modelos 2SVM

A máquina de execução de modelos 2SVM foi implementada utilizando a linguagem de programação Java, Groovy e as plataformas de *middleware* de comunicação *CoreDX DDS* [66] e Java RMI [98, 38]. A implementação da máquina, juntamente com a linguagem de modelagem, contempla as quatro camadas da 2SVM. A Figura 5.2 apresenta o diagrama de classes da 2SVM-*Controller* e a Figura 5.3 o diagrama de classes da 2SVM-*Client*.

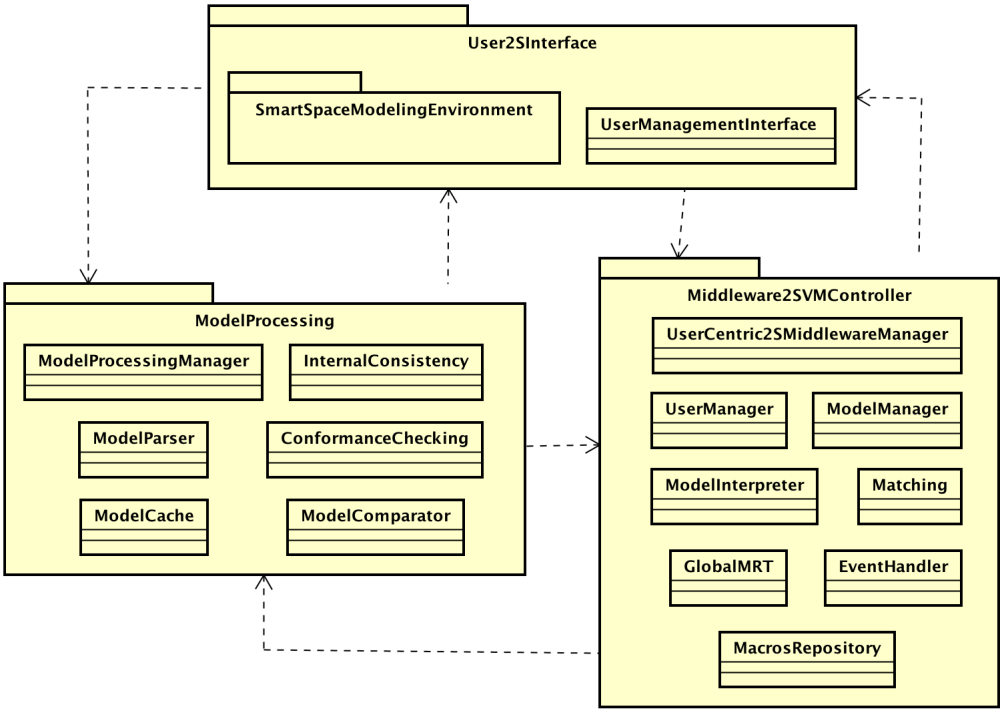



**Figura 5.2:** *Diagrama de classes da 2SVM-Controller*

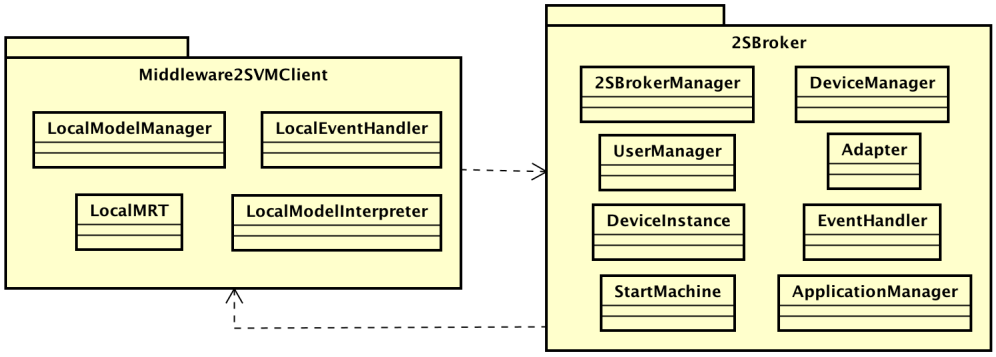



**Figura 5.3:** *Diagrama de classes da 2SVM-Client*

### 5.2.1 2SVM-*Controller*

#### User-2S Interface

A camada *User-2S Interface* possui três componentes: *i) Smart Space Modeling Environment*, que provê o editor gráfico de modelos; *ii) User Management Interface*, responsável pelo cadastro e autenticação de usuários; e *iii) Model Repository*.

As classes que compõem a 2SVM-*Controller* estão agrupadas em pacotes, sendo que cada um deles representa uma das camadas da máquina de execução de modelos. O pacote `User2SInterface` contém a classe `UserManagementInterface`, responsável por permitir ao engenheiro e ao usuário do sistema autenticarem-se na máquina para

criar seus modelos, bem como, permitir ao engenheiro cadastrar usuários. A classe `UserManagementInterface` foi desenvolvida utilizando a linguagem Java. Esta classe comunica diretamente com a camada de *User-Centric 2S Middleware*, em que se encontra a base de dados de usuários.

O pacote `SmartSpaceModelingEnvironment` contém todas as classes utilizadas na implementação da linguagem de modelagem de espaços inteligentes. Conforme apresentamos na Seção 5.1.1, o editor gráfico de modelagem foi construído utilizando o *framework* GMF. O repositório de modelos é acessado pelo próprio editor gráfico de modelagem. Para fazer isso, nós criamos um diretório onde armazenamos todos os modelos criados pelo engenheiro e usuário do sistema, nos formatos `.twosml` e `.twosml_diagram`.

### Model Processing

As classes que compõem esta camada estão agrupadas no pacote `ModelProcessing`. A classe `ModelProcessingManager` recebe os modelos vindos da camada superior no formato `.twosml` por meio da operação *checkModel*. Juntamente com o modelo temos a informação de quem o enviou, neste caso, podendo ser engenheiro ou usuário do sistema. Se o modelo enviado pertencer ao ES, ele deverá ser encaminhado ao componente *Model Parser* e *Internal Consistency*. Se pertencer ao US, ele passará pelos componentes *Model Parser*, *Internal Consistency* e *Conformance Checking*, para enfim, ser submetido para *Model Comparator*. O Código 5.3 apresenta o método *checkModel*.

**Código 5.3:** *Método checkModel*

```
1   /**
2    * Metodo checkModel
3    *
4    * @param model
5    * @param modelType: modelo do ES ou US
6    * @throws IOException
7    * @throws NotBoundException
8    * @throws InterruptedException
9    */
10  public void checkModel(Model model, String modelType)
11      throws IOException, NotBoundException, InterruptedException {
12
13    // Se o modelo do usuario do sistema estiver consistente, chame
14    // Conformance Checking
15    ArrayList<String> errors = new ArrayList<String>();
16    if (modelType.equals("engineerModel")) {
17      System.out.println("|----------------- Modelo do Engenheiro
            -----------------|");
18      // Realiza o parsing do modelo do ES
19      modelParser.parsingEngineerModel(model);
20      // Checa a consistencia interna do modelo
21      errors = internalConsistency.checkConsistency("modelprocessing/modelcache/"
            +modelType);
22      if (errors.size() > 0) {
23        System.out.println("Erros no Modelo do Engenheiro: " + errors.toString());
24      } else {
25        System.out.println("             Nenhum erro no Modelo do Engenheiro!");
26      }
27      System.out.println("
            |-------------------------------------------------------|\n");
```

```
28      } else {
29        if (modelType.equals("userModel")) {
30          System.out.println("|------------------- Modelo do Usuario
                -----------------|");
31          // Realiza o parsing do modelo do ES
32          modelParser.parsingUserModel(model);
33          errors = internalConsistency.checkConsistency("modelprocessing/modelcache/
                " +modelType);
34          if (errors.size() == 0) {
35            System.out.println("              Nenhum erro no Modelo do Usuario!");
36            conformanceChecking.checkConformance();
37          } else {
38            System.out.println("Erros no Modelo do Usuario: " + errors.toString());
39          }
40        }
41        System.out.println("
              |-----------------------------------------------------------|\n");
42      }
43    } // fim do metodo checkModel
```

A classe `ModelParser` implementa o *parsing* dos elementos do modelo do ES e US. Nesta etapa, separamos os tipos de elementos do modelo por metatipos (`UserRole`, `SmartObject`, `UbiquitousApplication` e `Policy`) e os agrupamos em conjuntos de elementos. Após estes conjuntos serem criados, eles são enviados para *Model Cache* armazenar estes elementos. Este componente é uma *cache* de elementos de modelos que é atualizada a cada envio de novos modelos do engenheiro e do usuário do sistema, ou seja, a cada modelo do ES ou US enviado para *Model Parser*, este componente remove todos os elementos lá contidos e os substitui por aqueles pertencentes ao novo modelo.

A classe `InternalConsistency` realiza a checagem de consistência interna dos modelos desenvolvidos. Para realizar esta operação, ela obtém os elementos dos modelos do ES ou US a partir de *Model Cache*, e inicia o processo de avaliação. Esta checagem consiste em verificar se os elementos utilizados para a criação das condições das políticas (políticas *event-condition-action*) estão presentes na parte estrutural do modelo. O Código 5.4 apresenta o método de checagem de consistência, presente na classe `InternalConsistency`. Inicialmente, extraímos as condições das políticas e as armazenamos em uma lista, na linha 12. Após isso, extraímos as expressões das políticas na linha 18. Nas linhas 22, 23 e 24, obtemos os elementos do modelo referente aos papéis do usuário, objetos inteligentes e aplicações ubíquas, respectivamente. Por fim, verificamos se os elementos pertencentes às políticas existem no modelo e caso isso não ocorra, uma mensagem de alerta é emitida ao usuário relatando o erro.

**Código 5.4:** *Método checkConsistency*

```
1   /**
2    * Metodo checkConsistency
3    *
4    * @param modelType: modelo do ES ou US
5    */
6   public ArrayList<String> checkConsistency(String modelType) {
7
8     ArrayList<String> errors = new ArrayList<String>();
9
```

```
10      ModelElementReaderBehPol modelElementReaderBehPol = new
            ModelElementReaderBehPol();
11      ArrayList<String> policyConditions = new ArrayList<String>();
12      policyConditions = modelElementReaderBehPol.getConditionList(modelType);
13
14      // extrai as expressoes da condicao
15      ArrayList<String> conditions = new ArrayList<String>();
16      int amountPolicies = policyConditions.size();
17      for (int index = 0; index < amountPolicies; index++) {
18        conditions = checkPolicy(policyConditions.get(index));
19      }
20
21      // extrai um hash de elementos do modelo a partir de ModelCache
22      getUserRoleHM(modelType);
23      getSmartObjectHM(modelType);
24      getUbiAppHM(modelType);
25
26      int amountCondition = conditions.size();
27      for (int index = 0; index < amountCondition; index++) {
28        if (!entitiesModelHM.containsKey(conditions.get(index))) {
29          errors.add(conditions.get(index));
30          System.out.println("O elemento " + conditions.get(index)
31              + " não existe no modelo!\n Corrija a política!");
32        }
33      }
34      return errors;
35
36    } // fim do metodo checkConsistency
```

O componente *Conformance Checking* verifica se o modelo do usuário do sistema está em conformidade com o modelo do engenheiro. Para isso, utilizamos novamente regras EVL, que executam a operação de validação, checando se cada um dos elementos definidos no modelo do usuário é subtipo de algum tipo definido no modelo do engenheiro. Para realizar esta operação, a classe `ConformanceCheking` gera automaticamente regras EVL a partir da leitura do modelo do engenheiro do sistema. Estas regras auxiliam a verificar se os elementos definidos nos modelos do US são subtipos daqueles definidos no modelo do ES.

As regras são criadas por meio de um arquivo .EVL, em que o editor gráfico de modelagem da linguagem as executa e verifica se há alguma inconsistência nos elementos criados no modelo. Para verificar inconsistências no modelo, o próprio usuário do sistema no editor gráfico de modelagem, abre o arquivo com extensão .twosml, que contém as definições de seu modelo, e clica em *Validate*. Se o modelo do US não estiver em conformidade com o modelo do ES, uma ou mais mensagens de alerta serão emitidas pelo editor gráfico, mostrando os elementos que contém erros.

Por fim, temos nesse pacote a classe `ModelComparator`, responsável por comparar os elementos do modelo submetido, presentes em *Model Cache*, com os elementos do modelo em execução, presentes na camada subjacente, em *Global M@RT*. Para acessar a camada de *middleware*, a classe `ModelComparator` utiliza a operação *queryM@RT*, que obtém todos os elementos do modelo em tempo de execução em nível de tipos. A comparação é realizada de tal forma a obter os elementos a serem adicionados e removidos de Global M@RT. A Figura 5.4 mostra um diagrama de *Venn* da comparação entre o modelo

submetido e o modelo em tempo de execução no espaço inteligente.

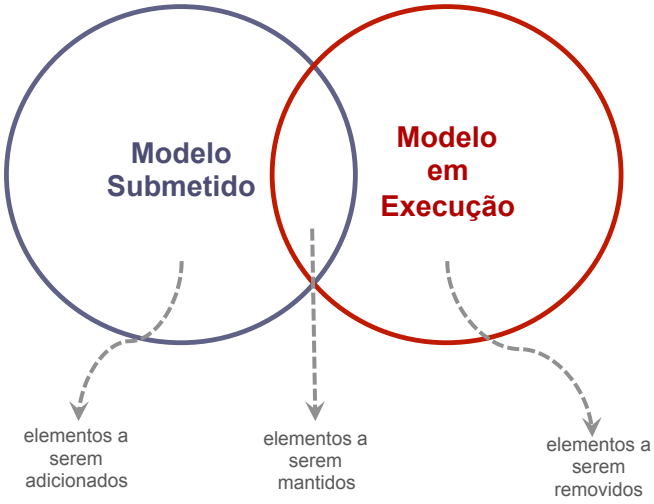



**Figura 5.4:** *Diagrama de Venn Venn demonstrando a comparação entre o modelo submetido e o modelo em execução*

Neste diagrama, temos: *i)* Elementos que devem ser adicionados ao M@RT em nível de tipos: (`ModeloSubmetido` − (`ModeloSubmetido` ⋂ `ModeloEmExecucao`)); *ii)* Elementos que devem ser removidos do M@RT em nível de tipos: (`ModeloEmExecucao` − (`ModeloSubmetido` ⋂ `ModeloEmExecucao`)). O Código 5.5 compara os elementos do modelo do usuário do sistema, em particular, aqueles pertencentes ao metatipo `Smart Object`. Para fazer isso, extraímos inicialmente os elementos de *Model Cache* e os adicionamos a uma coleção de elementos. Após isso, extraímos uma coleção de elementos de Global M@RT e repassamos ao método *getNewModel(sObjectCollection, newModelSObjects)*, presente na linha 31, para comparar os elementos. Como retorno a esse método, temos a um objeto que contém todos os elementos que devem ser adicionados e removidos de Global M@RT. Por fim, na linha 33 temos a atualização do modelo em tempo de execução.

**Código 5.5:** *Método compareSmartObjects*

```
1   /**
2    * Metodo para comparar elementos dos modelos de SmartObjects
3    *
4    * @throws IOException
5    * @throws NotBoundException
6    * @throws InterruptedException
7    */
8   public void compareSmartObjects() throws IOException, NotBoundException,
9       InterruptedException {
10
11    // lista de smartObjects
12    ArrayList<SmartObject> smartObjects = new ArrayList<SmartObject>();
13    smartObjects = modelCache
14        .queryModelElements("modelprocessing/modelcache/usermodel/smartobject");
15
16    // colecao de smartObjects
17    Collection<SmartObject> newModelSObjects = new ArrayList<SmartObject>();
```

```
18      int sObjetsSize = smartObjects.size();
19      for (int indexNew = 0; indexNew < sObjetsSize; indexNew++) {
20        newModelSObjects.add(smartObjects.get(indexNew)
21            .getSmartObjectType());
22      }
23
24      // colecao de SmartObjects de M@RT
25      Collection<SmartObject> sObjectCollection = new ArrayList<SmartObject>();
26      sObjectCollection = modelsAtRunTime.queryMRTSO();
27
28      // Elementos que devem ser adicionados e removidos do espaco
29      // inteligente
30      NewModelSO newModelSO = new NewModelSO();
31      newModelSO = getNewModelSO(sObjectCollection, newModelSObjects);
32
33      updateGlobalMRTSO(newModelSO);
34
35    } // fim do metodo comparatorSmartObjects
```

O Código 5.6 apresenta a comparação feita entre as coleções de elementos do tipo Smart Object. Como resultado desta comparação, obtemos uma lista contendo os elementos que devem ser adicionados e removidos do modelo em tempo de execução, presente na camada de *middleware*.

**Código 5.6:** *Método getNewModelSO*

```
1     /**
2      * Metodo para retornar um objeto contendo os elementos que devem ser
3      * adicionados e removidos do espaco inteligente
4      *
5      * @param oldModelCollection
6      * @param newModelCollection
7      * @return newModelSO
8      */
9     public NewModelSO getNewModelSO(Collection<SmartObject> oldModelCol,
10        Collection<SmartObject> newModelCol) {
11
12      // Lista dos elementos do modelo em execucao
13      ArrayList<SmartObject> oldElementsModel = new ArrayList<SmartObject>(
            oldModelCol);
14
15      // Lista dos novos elementos do modelo
16      ArrayList<SmartObject> newElementsModel = new ArrayList<SmartObject>(
            newModelCol);
17
18      // Elementos do modelo em tempo de execucao
19      oldElementsModel.removeAll(newModelCol);
20      int amountOldElements = oldElementsModel.size();
21
22      ArrayList<SmartObject> oldElements = new ArrayList<SmartObject>();
23      for (int index = 0; index < amountOldElements; index++) {
24        oldElements.add(oldElementsModel.get(index));
25      }
26
27      // Lista de elementos do modelo que se repetem
28      ArrayList<SmartObject> elementsToBeMaintained = new ArrayList<SmartObject>();
29      elementsToBeMaintained.addAll(newElementsModel);
30
31      // lista de elementos que deverao ser adicionados
32      // para esse lista
33      newElementsModel.removeAll(oldModelCol);
34      int amountNewElements = newElementsModel.size();
35      ArrayList<SmartObject> elementsToBeAdded = new ArrayList<SmartObject>();
36      for (int index = 0; index < amountNewElements; index++) {
37        elementsToBeAdded.add(newElementsModel.get(index));
38      }
39
40      // elementos do modelo que devem ser mantidos
```

```
41      elementsToBeMaintained.removeAll(newElementsModel);
42
43      // Lista dos elementos a serem removidos
44      ArrayList<SmartObject> elementsToBeRemoved = new ArrayList<SmartObject>(
45          oldElements);
46      elementsToBeRemoved.removeAll(elementsToBeAdded);
47
48      // Objeto com elementos que devem ser adicionados e removidos
49      // do espaco inteligente
50      NewModelSO newModelSO = new NewModelSO();
51      newModelSO.setElementsToBeAddeds(elementsToBeAdded);
52      newModelSO.setElementsToBeRemoveds(elementsToBeRemoved);
53
54      // novo modelo
55      return newModelSO;
56
57  } // fim do metodo getNewModelSO
```

### User-Centric 2S Middleware

A camada *User Centric-2S Middleware* possui duas configurações. Uma delas é executada na 2SVM-*Controller*, ou seja, na controladora do espaço inteligente, e a outra, na 2SVM-*Client*, ou seja, nos demais *smart objects* do ambiente. Temos, portanto, uma instância desta camada executando na controladora do espaço inteligente e *n* instâncias nos demais dispositivos do ambiente.

A camada de *middleware* da 2SVM-*Controller* possui componentes responsáveis pela execução do modelo. O componente *User Centric-2S Middleware Manager* é responsável por realizar operações relacionadas à camada de processamento de modelos. A implementação deste componente é realizada por meio da classe `UserCentric2SMiddlewareManager`, presente no pacote `Middleware2SVMController`. Esta classe recebe chamadas da camada de processamento com o objetivo de consultar e atualizar o M@RT Global, por meio das operações *queryMRTSO()* e *M@RTUpdates(modelElements)*; e da camada de *User-2S Interface*, a fim de permitir a autenticação do ES ou US na máquina e cadastro de usuários, por meio das operações *authenticate(user)* e *register(user)*. A classe `UserManager` faz acesso à base de dados de usuários atualizando e consultando informações acerca daqueles que podem participar do espaço inteligente, bem como, daqueles que podem se autenticar na máquina para criar seus modelos para o ambiente. A base de dados de usuários é um arquivo de texto que contém o nome do usuário, *login* e *e-mail*, e o papel do usuário que ele exerce naquele espaço inteligente.

O componente *Model Manager* é implementado pela classe de mesmo nome, `ModelManager`, que tem por função consultar e atualizar os elementos do modelo em tempo de execução. O Código 5.7 apresenta o método que consulta os elementos do modelo em tempo de execução e o Código 5.8, o método que atualiza.

**Código 5.7:** *Método queryMrtElementUR*

```
1   /**
2    * Metodo para retornar determinado M@RT de User
3    * @param userType
4    * @return
5    */
6   public UserMRT queryMrtElementUR(String userType) {
7
8     // MrtReader
9     MrtReader mrtReader = new MrtReader();
10
11    // Lista de M@RT do tipo app
12    ArrayList<UserMRT> userMRTs = new ArrayList<UserMRT>();
13
14    // M@RT do tipo app
15    UserMRT userMRT = new UserMRT();
16
17    // Cria lista
18    ArrayList<String> listAL = createListUser();
19
20    int size = listAL.size(); // verifica o tamanho da lista
21
22    for (int i = 0; i < size; i++) {
23      if ((!listAL.get(i).toString().equals(".DS_Store"))) {
24        userMRTs.add(mrtReader.readUserMRT(listAL.get(i)
25            .toString()));
26        int amountAppMRT = userMRTs.size();
27        for (int indexAppMrt = 0; indexAppMrt < amountAppMRT; indexAppMrt++) {
28          if (userMRTs.get(indexAppMrt).getUserType()
29              .equals(userType)) {
30            userMRT = mrtReader.readUserMRT(listAL.get(i)
31                .toString());
32          }
33        }
34      }
35    }
36    return userMRT;
37
38  } // fim do metodo queryMrtElementUR
```

**Código 5.8:** *Método mrtUpdatesUserRole*

```
1   /**
2    * Metodo para atualizar o M@RT de Usuario
3    * @param userMRT
4    */
5   public void mrtUpdatesUserRole(UserMRT userMRT) {
6     MrtRecorder mrtRecorder = new MrtRecorder();
7     mrtRecorder.recordMrtUser(userMRT);
8   } // fim do metodo mrtElementUpdatesUR
```

O modelo em tempo de execução da camada de *middleware* é um repositório de elementos do modelo que são separados por metatipos, `User Role`, `Smart Object`, `Ubiquitous Application` e `Policy`. Em cada uma das pastas que contém estes metatipos, existem tipos dos elementos definidos no modelo do US e instâncias desses tipos. Por exemplo, para o tipo *smartphone*, podemos ter as instâncias *iPhone6*, *galaxyNote*, *windowsphone*, e assim por diante. Isso equivale para todos os demais metatipos.

A classe *Matching* implementa o componente de mesmo nome e tem por objetivo descobrir os papéis dos usuários e tipos dos objetos inteligentes que entram no ambiente. Para descobrir o papel do usuário, é realizada uma chamada ao método *queryU-*

*serData(User user)* de *UserManager*, em que o objeto *user* contém o nome do usuário, *login* e *e-mail*. Como retorno a essa chamada de método, temos o papel daquele usuário no espaço inteligente.

Para descobrir o tipo do dispositivo, o componente *Matching* recebe o conjunto de características do dispositivo por meio do objeto *Resource*, que contém todas as informações do objeto inteligente separados por recursos de hardware e software. Os tipos de objeto inteligente definidos no modelo também contém essas informações de hardware e software, na forma de *features*. Para obter esses elementos, o componente *Matching* realiza uma chamada ao método *queryM@RTSmartObject()* do componente *Model Manager*, que retorna todos os tipos de objetos inteligentes definidos no modelo do US. Por fim, o componente *Matching* verifica qual elemento do modelo possui as mesmas características daquelas recebidas no objeto *Resource*, este é o tipo do dispositivo que entrou no espaço inteligente. Após isso, este elemento é enviado ao componente *Model Interpreter* por meio da chamada de método *interpretM@RTElement(modelElement)*.

A classe `ModelInterpreter` tem por objetivo:

- Interpretar os elementos do modelo recebidos da classe `Matching`: consiste de verificar as associações do elemento recebido com os demais elementos do modelo;
- Atualizar o M@RT global do espaço inteligente: por meio da chamada do método *mrtUpdates(modelElement, instance)* pertencente à classe `ModelManager`, informações de tipo e da instância do elemento são repassadas para que o M@RT global possa ser atualizado;
- Receber os eventos do ambiente de *Event Handler*: estes são considerados eventos do ambiente. Para interpretá-los, a classe `ModelInterpreter` realiza uma chamada de método a classe `ModelManager` de maneira a obter as políticas do modelo com o tipo de evento identificado, ou seja, o evento já vem com um tipo e a busca no componente *Model Manager* é realizada a partir dele. Após obter as políticas para o evento em questão, o componente *Model Interpreter* passa para a avaliação da condição da política. Para implementar o mecanismo de avaliação da condição, desenvolvemos um interpretador utilizando a linguagem Groovy [72], apresentado no Código 5.9. O objeto *contextChange* tem informações relativas à mudança de contexto ocorrida no ambiente, obtido a partir das demais informações do evento. O atributo *condition*, contém a condição definida no modelo do US.

**Código 5.9:** *Classe desenvolvida em Groovy [72], responsável por implementar o interpretador de condições da política*

```
1  class ContextChangeGroovy {
2    public String interpreterCtx(ContextChange cChange, String cond) {
3      ContextChange contextChange = new ContextChange()
4      contextChange = cChange
5      String condition = cond
6      return Eval.me("contextChange", contextChange, condition)
```

```
7     }
8   }
```

- Selecionar as macros de seu repositório: a seleção das macros é realizada por meio da chamada ao método *queryMacro(metatype)* e é feita por metatipo. Desta forma, temos macros relacionadas a `UserRole`, `SmartObject`, `UbiquitousApplication` e `Policy`; e
- Enviar o M@RT local e as macros para os objetos inteligentes correspondentes: o M@RT local e as macros são enviadas a partir de invocação de método remoto, com Java RMI. Para isso, especificamos a interface remota *RemoteCommunicationInterf*, apresentada no Código 5.10.

A comunicação entre as instâncias dessa camada ocorre de forma distribuída. Para isso, utilizamos duas plataformas de *middleware*: CoreDX DDS [66] [1], baseado na especificação DDS, que descreve um modelo de comunicação centrado nos dados para integração de aplicações distribuídas; e Java RMI, que utiliza chamada de métodos remotos. O *middleware* de comunicação CoreDX DDS é utilizado para a comunicação baseada em eventos entre os componentes da camada *User Centric-2S Middleware*.

**Código 5.10:** *Interface remota RemoteCommunicationInterf*

```
1  public interface RemoteCommunicationInterf extends Remote {
2
3    public void sendMacroUR(UserMacro userMacro) throws RemoteException,
4        IOException;
5
6    public void sendLocalMrtUR(UserMRT userMRT) throws RemoteException;
7
8    public void sendMacroSO(SmartObjectMacro smartObjectMacro)
9        throws RemoteException;
10
11   public void sendLocalMrtSO(SmartObjectMRT smartObjectMRT)
12       throws RemoteException;
13
14   public void sendMacroUA(ApplicationMacro appMacro) throws RemoteException,
15       ClassNotFoundException, InstantiationException,
16       IllegalAccessException, IOException;
17
18   public void sendMacroUAContextChange(ApplicationMacro applicationMacro, String
         destinationIP,
19       String contChangeDestinationIP) throws RemoteException,
20       ClassNotFoundException, InstantiationException,
21       IllegalAccessException, IOException;
22
23   public void sendMacroUAAppState(final ApplicationMacro appMacro,
24       final String destinationIP, final ApplicationState applicationState) throws
             RemoteException;
25
26   public void sendLocalMrtUA(ApplicationMRT appMRT) throws RemoteException,
27       ClassNotFoundException, InstantiationException,
28       IllegalAccessException;
29
30 }
```

[1] http://www.twinoakscomputing.com/coredx

Para o componente *Event Handler*, implementamos um conjunto de tratadores responsáveis por receber eventos do espaço inteligente, relacionadas a entrada de usuários e objetos inteligentes, mudanças de contextos gerais e demais eventos relacionados às operações nas aplicações ubíquas, iniciar, pausar, continuar e finalizar. Estes tratadores de eventos foram desenvolvidos utilizando a plataforma de *middleware* de comunicação CoreDX DDS, ou seja, existe um tratador para cada um destes elementos apresentados.

### 5.2.2 2SVM-*Client*

#### User-Centric 2S Middleware

A camada de *middleware* da 2SVM-*Client* possui componentes que se complementam em funcionalidades com aqueles pertencentes a camada de *middleware* da 2SVM-*Controller*. A classe `LocalModelInterpreter` tem por funcionalidade receber as macros encaminhadas da controladora por meio das interfaces remotas e as encaminhar para a camada inferior através da chamada ao método local *sendMacro(macro)*. Além disso, esta classe tem como funcionalidade receber da controladora o modelo em tempo de execução local e atualizá-lo por meio da chamada de método *m@rtElementUpdates(m@rt)*.

A classe `LocalEventHandler` recebe os eventos locais da camada de *Broker* por meio de tratadores de eventos locais e adiciona informações relativas ao M@RT local. Após isso, o evento é enviado para a controladora do ambiente por meio da plataforma de *middleware* Core DX DDS. Assim como na 2SVM-*Controller*, temos um tratador de evento específico para cada entrada de usuários e objetos inteligentes no ambiente, mudanças de contextos gerais e eventos relacionadas às aplicações ubíquas.

A classe *LocalModelManager* acessa e atualiza o modelo em tempo de execução local por meio das chamadas de métodos *queryM@RTElement(m@rt)* e *m@rtElementUpdates(m@rt)*, respectivamente. Por fim, o modelo em tempo de execução local da camada de *middleware* da 2SVM-*Client* é um repositório de elementos do modelo que são separados por metatipos, `User Role`, `Smart Object`, `Ubiquitous Application`. Em cada pasta que contém estes metatipos, temos o papel do usuário, o tipo do objeto inteligente e o tipo das aplicações ubíquas que podem executar no dispositivo.

#### 2S Broker

Por fim, temos a camada *2S Broker*, que executa apenas nos dispositivos cliente do ambiente, ou seja, na 2SVM-*Client*. A classe *2SBrokerManager* recebe comandos da camada de *middleware* na forma de macros, e os encaminha para serem executados pelas classes `UserManager`, `DeviceManager` e `ApplicationManager`. A classe

`StartMachine` é o primeira a ser executada quando a 2SVM-*Client* inicia. Ela inicializa os serviços responsáveis por receber as macros e o modelo em tempo de execução local da controladora, para obter o conjunto de características do dispositivo e as informações do usuário a partir do sistema operacional. O Código 5.11 exibe o método responsável por realizar estas operações.

**Código 5.11:** *Método startTwoSVMClient*

```
1   /**
2    * Metodo startTwoSVMClient
3    * @throws IOException
4    */
5   public void startTwoSVMClient() throws IOException {
6     System.out.println("|------- 2SVM-Client Iniciada ------|");
7     deviceInformation.startDeviceManager();
8     userManager.startUserManager();
9     startServicesMiddlewareLayer();
10  }
```

Para obter as informações destes dispositivos, bem como acessar seus recursos, desenvolvemos a classe Adapter, que abstrai alguns recursos do dispositivo. Ela permite à camada acessar informações sobre os recursos de *hardware* e *software* do dispositivo, por meio de comandos do sistema operacional do dispositivo. O Código 5.12 apresenta o método *getUserName()*, responsável por coletar o nome do usuário do sistema, e o método *getOperatingSystemInfo()*, que coleta informações do sistema operacional.

**Código 5.12:** *Métodos responsáveis por obter o nome do usuário do sistema e informações relativas ao sistema operacional*

```
1   /**
2    * Metodo para obter o nome do usuario do sistema
3    * @return
4    */
5   public String getUserName() {
6     return System.getProperty("user.name").toLowerCase();
7   }
8
9   /**
10   * Obtem informacao do Sistema Operacional
11   *
12   * @throws UnknownHostException
13   */
14  public String getOperatingSystemInfo() {
15    return System.getProperty("os.name", "generic").toLowerCase(
16        Locale.ENGLISH);
17  }
```

A classe `EventHandler` implementa alguns tratadores de eventos locais. Em nosso trabalho, implementamos um conjunto de tratadores, tais como tratadores relacionados à entrada do objeto inteligente em um espaço inteligente e mudança de localização do dispositivo. Outros tratadores de eventos foram implementados para as aplicações, relativos à inicialização e finalização de aplicações. Todos esses tratadores de eventos foram

desenvolvidos com a plataforma *middleware* CoreDX DDS e são *subscribers* responsáveis por consumir estes eventos.

Para que essas aplicações sejam manipuladas pela máquina de execução de modelos, a interface `UbiquitousApplication` deve ser implementada, de tal forma que a 2SVM tenha controle sobre as operações iniciar, pausar, continuar, finalizar, salvar e restaurar estado da aplicação. O Código 5.13 apresenta a interface `UbiquitousApplicationInterface`.

**Código 5.13:** *Interface UbiquitousApplicationInterface*

```
1  public interface UbiquitousApplicationInterface extends Serializable {
2
3    public abstract void startApplication();
4
5    public abstract void pauseApplication();
6
7    public abstract void resumeApplication();
8
9    public abstract void destroyApplication();
10
11   public abstract ApplicationState saveApplicationState();
12
13   public abstract void restoreApplicationState(ApplicationState applicationState);
14
15 }
```

A classe `ApplicationState`, pertencente ao método *public abstract void restoreApplicationState(ApplicationState applicationState)*, da interface, varia de aplicação para aplicação, pois cada uma delas possui um conjunto de atributos e métodos específicos do domínio da própria aplicação. Desta forma, o desenvolvedor destas aplicações será o responsável por implementar a classe responsável por salvar o estado da aplicação ubíqua.

## 5.3 Conclusão

Este capítulo teve por objetivo demonstrar a implementação da linguagem de modelagem de espaços inteligentes, 2SML, e da máquina de execução de modelos, 2SVM. A implementação utilizou como ferramentas principais a linguagem de programação Java e Groovy, o *middleware* de comunicação Java RMI e Core DX DDS. Apesar da escolha destas plataformas de *middleware* de comunicação, a 2SVM poderia ser implementada utilizando outras plataformas que orientadas a eventos.

No capítulo também discutimos a implementação das duas máquinas de execução de modelos, 2SVM-*Controller* e 2SVM-*Client*, bem como detalhamos o desenvolvimento de cada uma das camadas e de seus respectivos componentes. O capítulo seguinte apresenta a avaliação que realizamos acerca da linguagem de modelagem e da máquina de execução de modelos. Para avaliar a linguagem, utilizamos cenários que permitiram ins-

tanciar diferentes espaços inteligentes. Já a avaliação da máquina teve por intuito medir o desempenho de algumas de suas funcionalidades, a partir dos cenários apresentados.

CAPÍTULO 6

# Estudo de Caso e Avaliação da 2SML e da 2SVM

O objetivo deste capítulo é demonstrar o uso da linguagem de modelagem 2SML e avaliar a máquina de execução de modelos para espaços inteligentes 2SVM. Para demostrar o uso da linguagem de modelagem, iremos utilizar dois cenários distintos, a fim de demonstrar sua expressividade. Já a avaliação da máquina de execução de modelos, será dividida por funcionalidades, com uma avaliação de desempenho de cada uma delas.

Este capítulo está organizado da seguinte forma: na Seção 6.1, demonstramos o uso da 2SML por meio de cenários de computação ubíqua. Na Seção 6.2, apresentamos a metodologia utilizada para avaliação da máquina de execução de modelos. Na Seção 6.3 exibimos os resultados obtidos com a avaliação e, por fim, na Seção 6.4, temos a conclusão do capítulo.

## 6.1 Demonstração do Uso da 2SML

Nesta demonstração, verificamos a capacidade da linguagem 2SML de modelar diferentes cenários de espaços inteligentes. Esta demonstração está alinhada com o objetivo de nosso trabalho, que é oferecer uma abordagem centrada no usuário para modelagem de espaços inteligentes. Para fazer isso, utilizamos dois cenários distintos: *i)* O primeiro deles, é o cenário da sala, dividido em sala de aula, apresentado na Seção 3.5, e sala de reuniões. Ambos cenários utilizaram o mesmo ambiente físico e nosso intuito com isso foi de demonstrar a capacidade da linguagem de definir comportamentos distintos para um mesmo espaço físico; *ii)* O segundo cenário escolhido, uma academia inteligente, foi apresentado no trabalho de *Corredor et al*. [30].

O cenário da sala de aula apresenta uma aula em que o professor utiliza *slides* como material didático. Estes *slides* são carregados automaticamente do dispositivo *smartphone* do professor para a lousa inteligente, após ele se aproximar dela. Os *slides* também são copiados da lousa inteligente para os dispositivos dos alunos, de tal forma que eles possam acompanhar a aula e realizarem suas anotações. Após a apresentação do

conteúdo os alunos iniciam a sessão de perguntas e também as atividades, sendo que para a última eles utilizam a aplicação de editor de textos.

Neste cenário definimos dois papéis do usuário modelados por `professor` e `student`, bem como os tipos de objetos inteligentes pertencentes ao ambiente como: `smartboard`, `airConditioning`, `printer`, `rfid`, `light`, `camera`; e aqueles pertencentes ao usuário, como `smartphone` e `tablet`. Os tipos de aplicações ubíquas definidas são `calendar`, `climatization`, `textEditor`, `sharing`, `slides` e `academy`. Este conjunto de papéis do usuário, objetos inteligentes e aplicações ubíquas definem a finalidade e comportamento do espaço inteligente. Para fins didáticos, resolvemos apresentar a parte estrutural e comportamental do modelo em figuras separadas, de maneira que ela não ficasse sobrecarregada. A Figura 6.1 apresenta alguns elementos do modelo do usuário do sistema para o ambiente de sala de aula no que se refere a parte estrutural, já a parte comportamental é apresentada na Figura 6.2.

O elemento raiz do modelo é do metatipo `SmartSpace` e foi modelado com o tipo `classroom`. Neste elemento, concentramos os demais elementos do espaço inteligente, papéis do usuário, objetos inteligentes, aplicações ubíquas e políticas. Para estes elementos, definimos uma associação do metatipo`hosts`. No modelo para sala de aulas podemos observar, por exemplo, que o usuário do tipo `professor` pode usar (`canUse`) o objeto inteligente do ambiente do tipo, `smartboard` e que ele tem como dispositivo pessoal (`isOwnerOf`) o objeto inteligente do tipo `smartphone`. Além disso, ele pode usar (`canUse`) as aplicações ubíquas do tipo `slides`, `academy` e `climatization`.

As políticas definidas para esse espaço inteligente permitem: *i)* `moveProfessorSlides`: mover os *slides* do dispositivo do professor (*smartphone*) para a lousa inteligente; *ii)* `turnOnLighing` regular a intensidade de luminosidade do espaço inteligente; *iii)* `turnOnClimatization` controlar a temperatura do ambiente; *iv)* `transferStudentSlides` transferir os *slides* da lousa para os dispositivos dos alunos e *v)* `moveStudentTextEditor` mover a aplicação de editor de textos dos dispositivos dos alunos para a lousa.

A política `moveProfessorSlides` move a aplicação ubíqua`slides` do dispositivo do tipo `smartphone` para o objeto inteligente do tipo `smartboard`. Ela foi definida da seguinte forma no modelo:

- *event*: changeLocation(`professor`, <*userID*>, `smartphone`, <*smartObjectID*>)
- *condition*: `if` ( (*smartObjectSmartSpace* == `smartphone`) && (*ubiApp* == `slides`) && (*userRole* == `professor`) && (*userLocation* == *smartboard*) )
- *action*: moveUbiApp(`slides`, <*ubiAppID*>, `smartboard`, <*smartboardID*>)

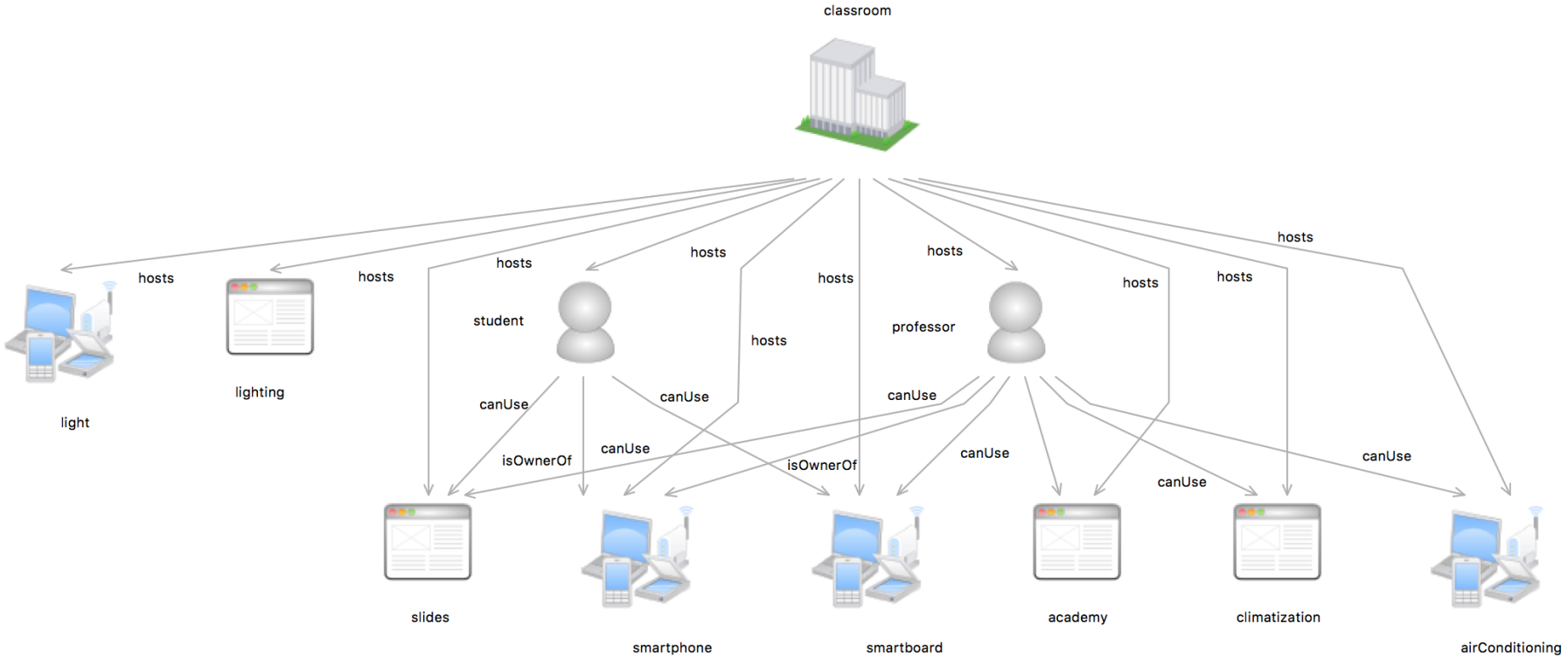



**Figura 6.1:** *Modelagem de elementos do Cenário da Sala de Aula - Parte estrutural do modelo*

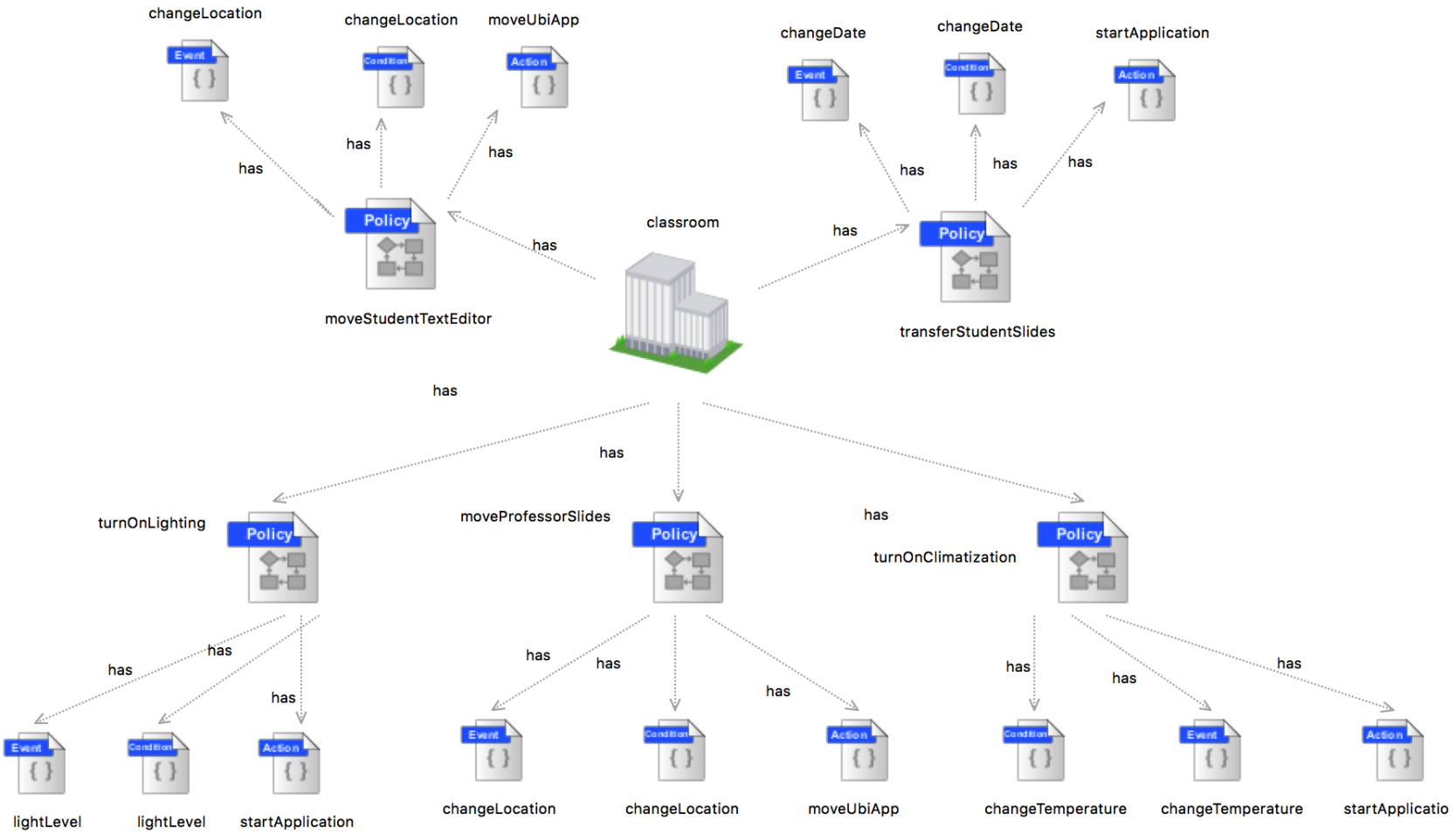



**Figura 6.2:** *Modelagem das políticas Cenário da Sala de Aula*

Como no modelo do usuário só temos informações referentes aos tipos definidos, o US define o evento apenas com o papel do usuário `professor` e o tipo do objeto inteligente `smartphone`. Já os elementos *<userID>* e *<smartObjectID>* são obtidos pela máquina em tempo de execução e são elementos em nível de instância. O evento *changeLocation* é gerado localmente no dispositivo do usuário, 2SVML-*Client* e traz consigo a informação da nova localização do objeto inteligente. Após isso, a máquina de execução de modelos acrescenta informações adicionais a este evento, relacionadas ao modelo em tempo de execução local, tais como papel do usuário; tipo do objeto inteligente; identificador do usuário e do objeto inteligente; bem como o tipo do evento. Em seguida, o evento é enviado da 2SVM-*Client* para a 2SVM-*Controller* na forma de evento do ambiente, que identifica o tipo de evento e inicia o processo de avaliação da condição. Se a condição avaliada for verdadeira, a ação mover a aplicação do tipo `slides` para o objeto inteligente do tipo `smartboard` será executada na forma de uma macro. O modelo em tempo de execução da 2SVM- *Controller* é então atualizado, e as macros, juntamente com o modelo em tempo de execução local dos dispositivos envolvidos na mudança de contexto, são enviadas para todos eles. Diante disso, a macro *moveUbiApp* é enviada para o dispositivo em que ocorreu a mudança de localização, nesse caso o *smartphone* e a macro *sendUbiApp* são enviados para o objeto inteligente que irá receber a aplicação de *slides*, assim sendo, a lousa inteligente.

É importante ressaltar que as políticas definidas no modelo do ES são aplicadas a todos os modelos do US. Assim, temos além das políticas definidas no modelo do US, a política para detecção de gás no ambiente, para detecção de fumaça e para identificação de ruído (como por exemplo, conversação alta) no ambiente.

*Descrição do cenário da sala de reuniões*: uma reunião foi agendada entre os professores e os funcionários administrativos de uma instituição. O ambiente está preparado para uma conferência com professores de outra instituição, que estabelecerão colaboração entre as entidades por meio do sistema de vídeo conferência. Nesta sala de reuniões temos os mais diversos objetos inteligentes, dentre eles, *tablets*, *smartphones*, sistema de vídeo conferência, câmeras, ar-condicionado e sensor de luminosidade. Durante a reunião, algumas aplicações ubíquas são utilizadas, entre elas um editor de texto para exibir a ata da reunião, calendário, controle de climatização e temperatura do ambiente, bem como a própria aplicação da vídeo conferência. A Figura 6.3 ilustra o cenário da sala de reuniões.

Os papéis do usuário definidos neste modelo são `professor` e `administrativeTechnician`. Como objetos inteligentes temos aqueles pertencentes ao ambiente como `videoConferencing`, `camera`, `airConditioning`, `smartTV`, `microphone` e `light`. Como pertencentes ao usuário temos `tablet`, `smartphone`, `notebook` e `smartwatch`. As aplicações ubíquas definidas para este espaço inteli-

gente foram o `textEditor`, `calendar`, `videoConferencingUbiApp`, `lighting` e `climatization`.

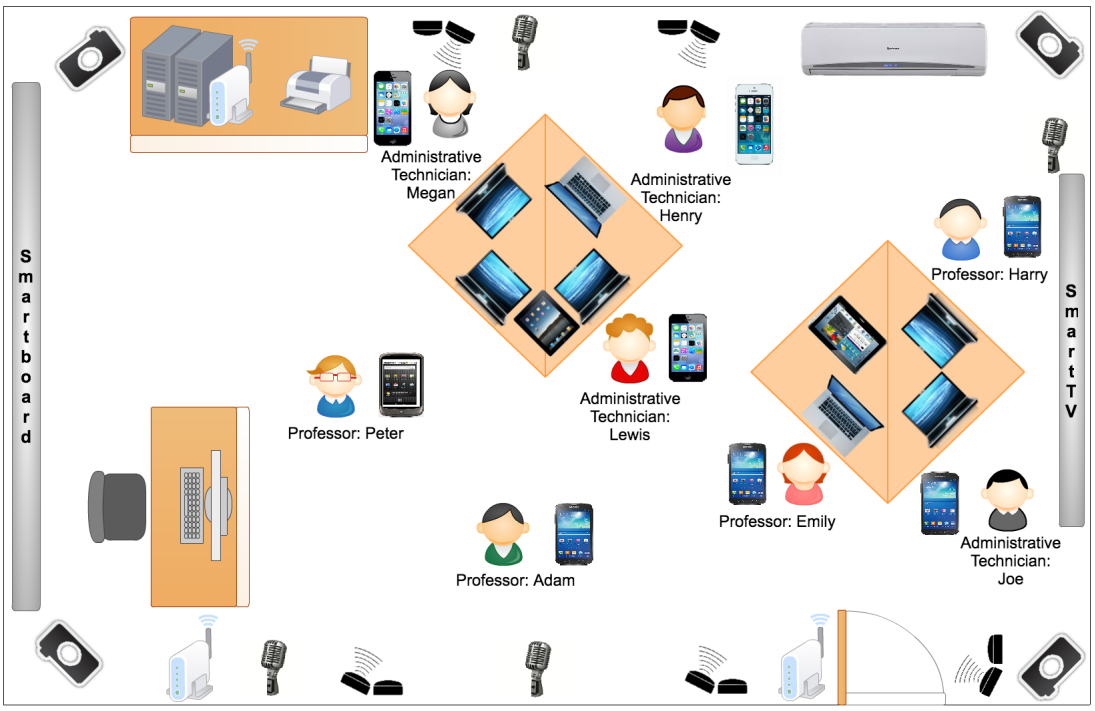



**Figura 6.3:** *Cenário da Sala de Reuniões*

Para este cenários, quatro políticas foram criadas, além daquelas definidas no modelo do engenheiro do sistema: *i)* `startVideoConfApp`: inicia a aplicação de vídeo-conferência assim que os usuários do espaço inteligente entram no ambiente; *ii)* `turnOnLighting`: configura a iluminação do ambiente para receber os usuários para a vídeo-conferência; *iii)* `agendaMeeting`: política que apresenta a ata da reunião a partir da aplicação de textos; e *iv)* `turnOnClimatization`: controla a temperatura da sala de reuniões.

Tal como no cenário da sala de aulas, nós apresentamos dois modelos, um referente a parte estrutural e outro a parte comportamental. A Figura 6.4, apresenta uma parte do modelo do usuário do sistema para o ambiente de sala de reuniões, no que se refere a estrutura. Na reunião, a conferência é exibida por meio da aplicação ubíqua do tipo `videoConferencingUbiApp`, através do objeto inteligente do tipo `videoConferencing`, que pertence ao ambiente físico e também nos dispositivos pessoais dos usuários do espaço inteligente.

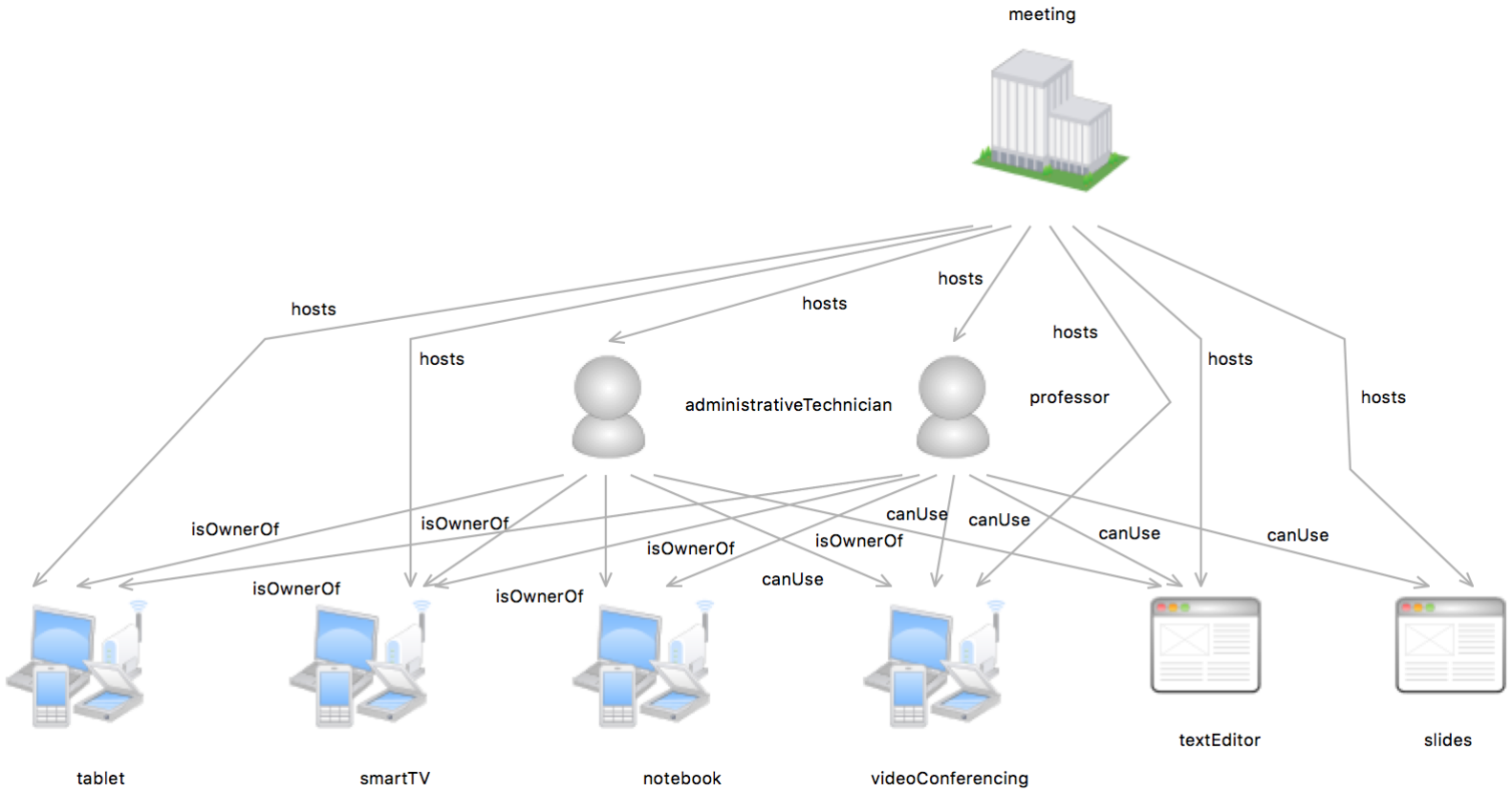



**Figura 6.4:** *Modelagem de elementos do Cenário da Sala de Reuniões - Parte estrutural*

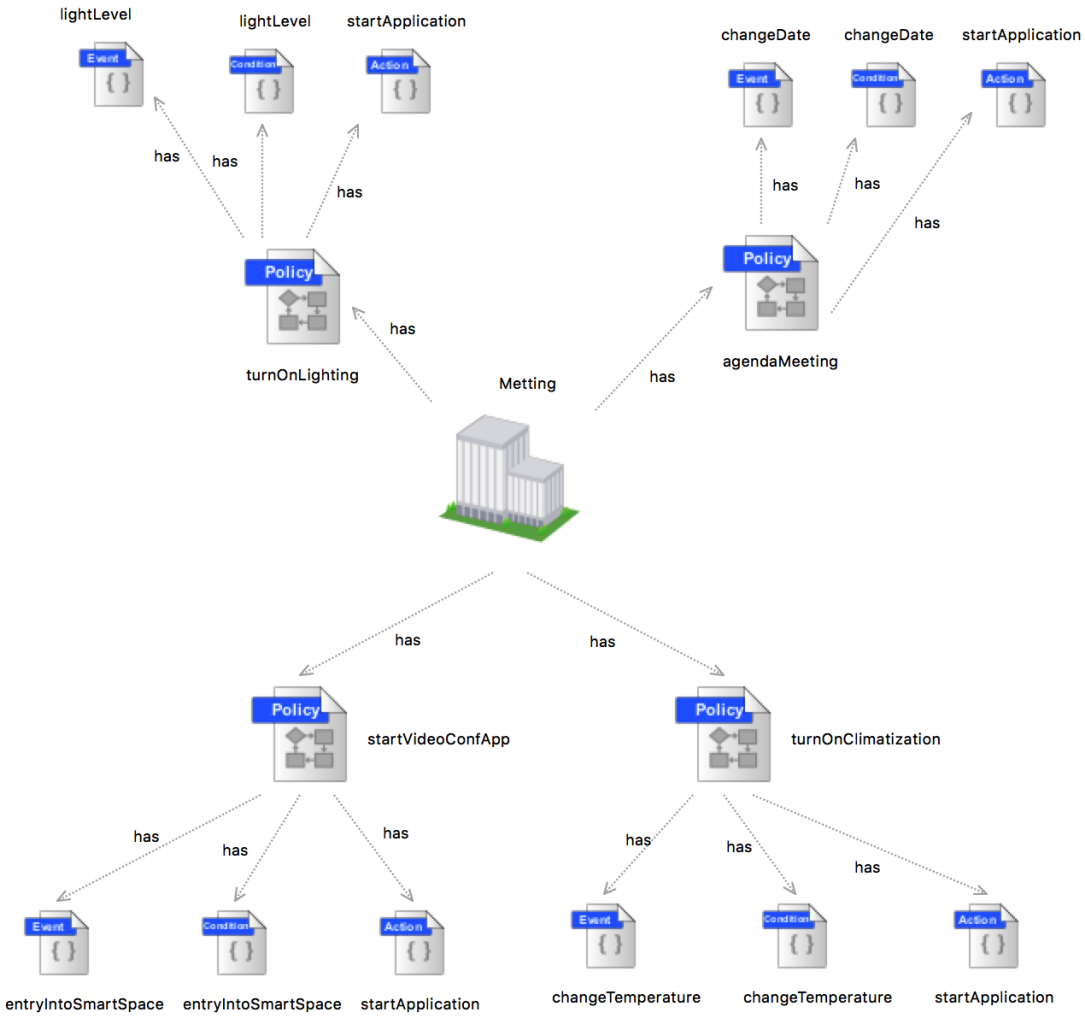



**Figura 6.5:** *Modelagem das políticas do Cenário da Sala de Reuniões*

Assim que os usuários entram no ambiente, a aplicação `videoConferencingUbiApp` é iniciada no objeto inteligente do tipo `videoConferencing`, bem como nos dispositivos do tipo `tablet` dos usuários. Para realizar esta ação, definimos a política `startVideoConfApp`. A Figura 6.5 apresenta a parte comportamental do modelo para o ambiente de sala de reuniões.

Como pôde ser visto, os cenários da sala de aula e sala de reuniões compartilham de alguns recursos do ambiente físico, como forma de reaproveitamento. Outros, são particulares a um cenário ou a outro. O objetivo de utilizar o mesmo ambiente físico foi de demonstrar a capacidade da linguagem 2SML de reaproveitamento dos elementos do modelo para ambiente distintos, e também a capacidade da máquina de execução de modelos de implantar diferentes comportamentos em um mesmo espaço físico.

Por fim, o terceiro cenário, academia inteligente (*SmartGym*), foi retirado do trabalho de *Corredor et al.* [30], em que eles propõem uma metodologia denominada *Resource-Oriented and Ontology-Driven Development* (ROOD) para a programação de espaços inteligentes. Esta metodologia baseia-se nos conceitos de ontologias e *Model-Driven Arquitecture* (MDA) [118]. Para demonstrá-la, os autores apresentaram o cenário de uma academia inteligente, que realiza o monitoramento da saúde de seus usuários por meio de objetos inteligentes e serviços. A Figura 6.6 apresenta o cenário da academia inteligente em que um usuário exercita-se em uma esteira.

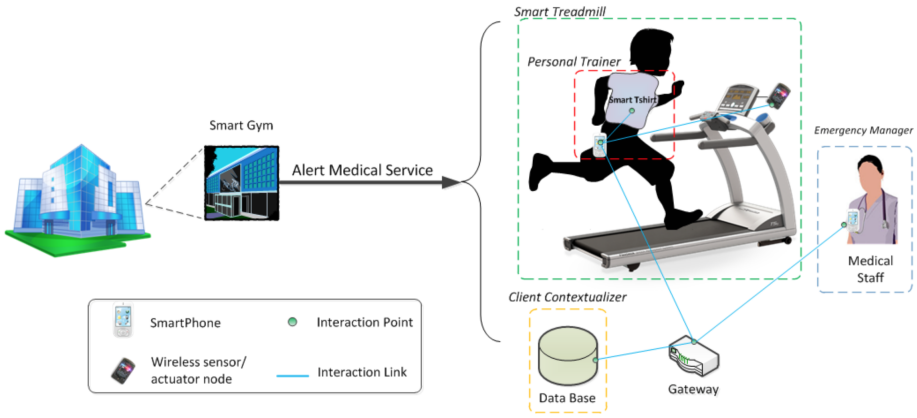



**Figura 6.6:** *Cenário da Academia Inteligente apresentado no trabalho de Corredor et al. [30]*

Entre os elementos pertencentes a este espaço inteligente, temos a camiseta inteligente, equipada com sensores de frequência cardíaca, temperatura corporal e movimento. Para extrair as informações desses sensores, eles utilizam uma aplicação denominada *personal trainer*. Esta aplicação possui uma lógica para processamento dos dados com o objetivo de determinar se uma emergência ocorreu a partir das informações coletadas do usuário. Também foi implementado um mecanismo de inferência das atividades

que combina o estado da saúde com o movimento do usuário na esteira, a fim de detectar quedas.

Para demonstrar o poder de expressividade da 2SML, modelamos alguns elementos deste cenário. A Figura 6.7 apresenta a modelagem de uma parte do cenário da academia inteligente. Neste modelo, temos o papel do usuário `smartGymUser` que utiliza a camiseta inteligente para monitoramento corporal. Esta camiseta foi modelada a partir do metatipo `SmartObject`, em que definimos o tipo `smartTshirt`, que possui as seguintes *features*:

- **Feature**: `heartRate`
  - `featureType`: `HARDWARE`;
  - `featureCategory`: `SENSOR`;
  - `featureDescription`: `mplHeartRate`.
- **Feature**: `bodyTemperature`
  - `featureType`: `HARDWARE`;
  - `featureCategory`: `SENSOR`;
  - `featureDescription`: `mplBodyTemperature`.
- **Feature**: `movementInertial`
  - `featureType`: `HARDWARE`;
  - `featureCategory`: `SENSOR`;
  - `featureDescription`: `mplMovementInertial`.

A aplicação que coleta informações do usuário da academia foi modelada na forma de uma aplicação ubíqua do tipo `personalTrainer`. Ela recebe as informações coletadas dos sensores presentes na camisa do usuário. Para iniciar esta aplicação, nós modelamos uma política. Assim que o usuário da academia aproxima-se da esteira, um evento é gerado localmente para objeto inteligente `smartTshirt` que processa a política e inicia a aplicação `personalTrainer`. A Figura 6.8 ilustra a política e sua modelagem é apresentada abaixo:

- *event*: changeLocation(`smartGymUSer`, <*userID*>, `smartTshirt`, <*smartObjectID*>)
- *condition*: `if` ( (*smartObjectSmartSpace* == `smartTshirt`) && (*userRole* == `smartGymUSer`) && (*userLocation* == *smartRunningMachine*) )
- *action*: startUbiApp(`personalTrainer`, <*ubiAppID*>, `smartTshirt`, <*smartObjectID*>)

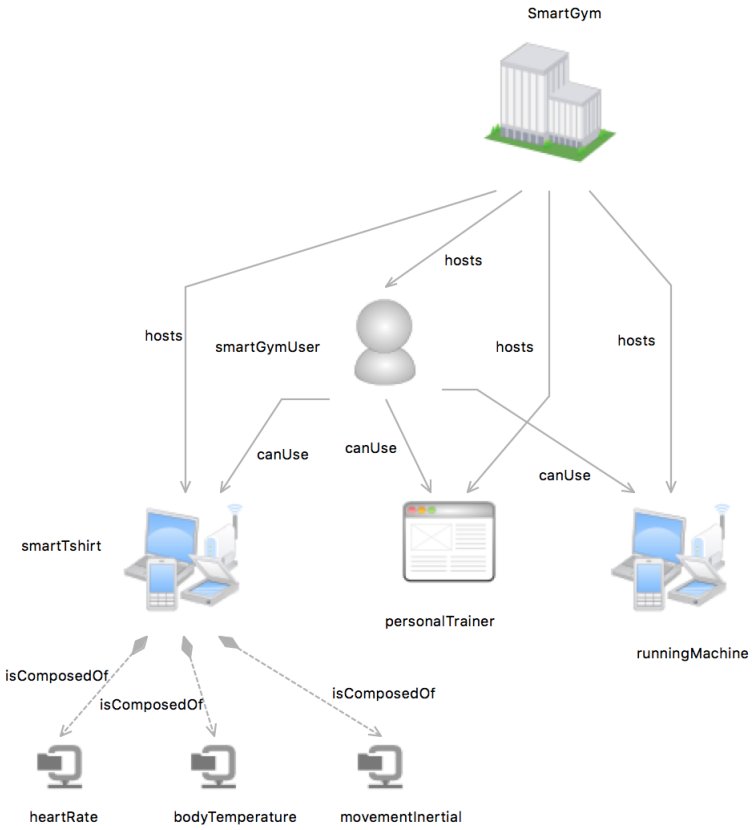



**Figura 6.7:** *Modelagem de elementos do Cenário da Academia Inteligente - Parte estrutural*

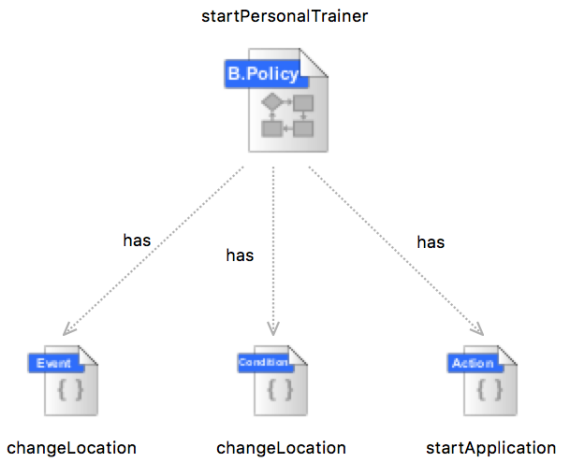



**Figura 6.8:** *Modelagem de uma política do Cenário da Academia Inteligente*

Os cenários apresentados demonstraram a capacidade da linguagem de modelar ambientes que têm comportamentos distintos. Em particular, os cenários da sala de aula e sala de reuniões foram criados no mesmo espaço físico, o que permitiu a reutilização dos objetos inteligentes deste ambiente. As aplicações ubíquas em todos os cenários foram

utilizadas de acordo com a finalidade de cada ambiente e, por isso, algumas aplicações foram utilizadas em um cenário e em outro, não.

## 6.2 Metodologia de Avaliação da Máquina de Execução de Modelos

Para a avaliação da máquina de execução de modelos, foi realizado um conjunto de experimentos, com o objetivo de medir o desempenho de algumas funcionalidades da 2SVM. Os experimentos elaborados para esta avaliação estão descritos abaixo:

- **Experimento 1**: nesse experimento avaliamos o tempo, em milissegundos, necessário para realizar a checagem de consistência interna dos modelos do usuário do sistema, utilizando para isso, o modelo para sala de aula. A consistência interna verifica se os elementos utilizados na parte comportamental do modelo, ou seja, os elementos das políticas, também estão presentes na parte estrutural. Neste experimento variamos a quantidade de elementos do modelo da parte estrutural, que foi de 5 a 35 elementos, e mantemos fixa a quantidade de políticas em 5. Inicialmente, definimos 5 políticas, cada qual para lidar com determinado comportamento do ambiente. Estas políticas foram apresentadas na Figura 6.2. Após isso, começamos criando 5 elementos para a parte estrutural e acrescentamos a cada avaliação, 5 novos elementos, até atingirmos 35 deles. Este experimento foi realizado em uma máquina com uma instância da 2SVM-*Controller*;
- **Experimento 2**: a checagem de conformidade consiste em verificar se os elementos definidos no modelo do usuário são sub-tipos daqueles presentes no modelo do engenheiro do sistema. Este experimento foi realizado após a submissão do modelo do engenheiro e mediu o tempo de processamento, em milissegundos, para realizar a checagem entre estes modelos. Este experimento, foi executado 7 vezes, em que mantemos fixa a quantidade de elementos do modelo do engenheiro, e variamos de 5 a 35 a quantidade de elementos do modelo do usuário do sistema. O experimento foi realizado em uma máquina com a 2SVM-*Controller*;
- **Experimento 3**: a comparação entre modelos tem por objetivo confrontar dois modelos submetidos pelo usuário do sistema. Como resultado desta comparação, temos a diferença entre os modelos e, portanto, o M@RT em nível de tipos atualizado, que será executado no ambiente. Neste experimento atualizamos o modelo do usuário para sala de aula por 7 vezes, e verificamos o tempo gasto, em milissegundos, para comparar estes modelos e atualizar o M@RT global do espaço inteligente. Ainda no mesmo experimento, em sua primeira comparação,

submetemos um modelo com 5 elementos. Este primeiro modelo foi comparado com um modelo "vazio", pois não havia um modelo em execução no momento. A partir daí, acrescentamos 5 elementos a cada novo modelo submetido, o que foi repetido por 7 vezes, ou seja, o primeiro modelo submetido tinha 5 e o último 35 elementos. Este experimento foi realizado em uma máquina que executou uma instância da 2SVM-*Controller*;

- **Experimento 4**: nesse experimento avaliamos o tempo gasto, em milissegundos, pela máquina para processar a entrada de usuários acompanhados de seus respectivos objetos inteligentes. O experimento 4 foi dividido em três etapas, em que medimos: *i)* O tempo de processamento para identificar o papel do usuário e o tipo do objeto inteligente; *ii)* O tempo necessário para atualizar o modelo em tempo de execução global; e *iii)* O tempo gasto para selecionar a macro correspondente a cada papel do usuário ou tipo de objeto inteligente. Para cada uma das etapas, o experimento foi executado 50 vezes a partir de um computador que tinha uma instância da 2SVM-*Client*. Em outro computador, tínhamos a 2SVM-*Controller*, responsável por receber os eventos de entrada dos usuários e seus objetos inteligentes. Para cada um destes experimentos, obtivemos o tempo médio de processamento, bem como o desvio padrão.

O objetivo dos experimentos 1, 2 e 3 foi verificar qual das funcionalidades da camada *Model Processing* consome mais tempo de processamento e o impacto de cada um deles para esta camada. Já o experimento 4 teve por objetivo verificar o tempo médio gasto para realizar às operações relacionadas a entrada de cada usuário acompanhado de seu objeto inteligente, e o impacto desse tempo na entrada de novos usuários com seus respectivos dispositivos.

Os experimentos foram realizados no laboratório do Instituto de Informática da UFG utilizando os seguintes equipamentos:

- Macbook Pro, processador i7 (2.6GHz), 512GB de SSD e 8GB de memória RAM:
  - Sistema Operacional OS X El Capitan 10.11.2;
  - Eclipse Modeling Tools, versão Luna 4.4.0: neste eclipse, desenvolvemos e executamos o editor gráfico de modelagem da 2SML;
  - Eclipse Java EE IDE, versão Luna 4.4.2: neste eclipse, desenvolvemos e executamos a máquina de execução de modelos 2SVM;
  - Máquina virtual Java versão 7;
  - Groovy versão 2.4.2;
  - Plataforma de *middleware* CoreDX DDS versão 3.6.28.
- Computador AMD Phenom, 320GB de HD e 4GB de memória RAM:
  - Sistema operacional Ubuntu versão 14.04 LTS;

  - Eclipse Java EE IDE, versão Luna 4.4.2;
  - Máquina virtual Java versão 7;
  - Plataforma de *middleware* CoreDX DDS versão 3.6.28.
- Notebook, processador i5 (2.26GHz), 6GB de memória RAM:
  - Sistema Operacional Ubuntu 14.04 LTS;
  - Eclipse Java EE IDE, versão Luna 4.4.2;
  - Máquina virtual Java versão 7;
  - Plataforma de *middleware* CoreDX DDS versão 3.6.28.
- Switch Ethernet 1Gbps;
- Ponto de acesso com especificação Wi-Fi 802.11ac.

O Macbook Pro executou uma instância da 2SVM-*Controller* e uma instância da 2SVM-*Client*. Já o computador e notebook executaram uma instância da 2SVM-*Client* cada. Os experimentos 1, 2 e 3, foram realizados exclusivamente no Macbook Pro, uma vez que as funcionalidades avaliadas encontram-se na 2SVM-*Controller*.

## 6.3 Resultados

**Experimento 1:**

A Figura 6.9 apresenta o experimento 1, de checagem de consistência interna do modelo, em que medimos o tempo necessário para realizar esta verificação. Para realizar este experimento, utilizamos o modelo do usuário do sistema para o ambiente da sala de aulas, e variamos de 5 a 35 a quantidade de elementos do modelo da parte estrutural, e mantemos fixa a quantidade de políticas em 5. A curva no gráfico mostra que medida que aumentamos a quantidade de elementos da parte estrutural do modelo, o tempo para checagem aumenta de maneira linear. Neste sentido, observamos que não teremos um crescimento exponencial ao longo do tempo, a medida que aumentamos a quantidade de elementos da parte estrutural, o que não prejudica o tempo de processamento da checagem de consistência interna dos elementos do modelo.

**Experimento 2:**

Neste experimento, avaliamos o tempo necessário para que a checagem de conformidade fosse realizada entre os elementos do modelos do usuário, para o cenário da sala de aula, e o modelo do engenheiro do sistema. A Figura 6.10 apresenta o experimento 2, de checagem de conformidade do modelo. Para este experimento, variamos a quantidade dos

elementos do modelo do usuário do sistema de 5 a 35, e mantemos fixa a quantidade daqueles pertencentes ao modelo do engenheiro, em 28 elementos. Como resultado disso, verificamos que a medida que aumentamos a quantidade dos elementos do modelo do usuário até 20 elementos, obtivemos um crescimento linear do gráfico. A partir desta quantidade, identificamos um crescimento maior no tempo de processamento da checagem de conformidade, o que pode indicar que este gráfico é uma parábola. Neste sentido, o tempo de processamento para esta funcionalidade pode ter um impacto prejudicial no tempo total de processamento do modelo do usuário do sistema, a medida que aumentamos ainda mais a quantidade de elementos presentes neste modelo.

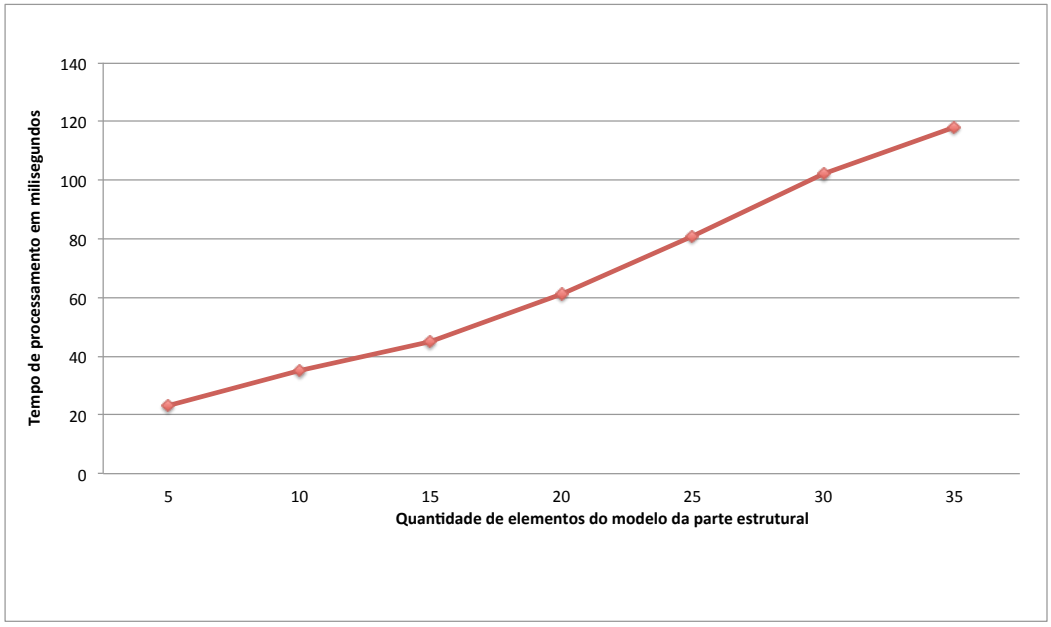



**Figura 6.9:** *Gráfico da Checagem da Consistência Interna*

**Experimento 3:**

A Figura 6.11 apresenta o experimento 3, que consistiu de comparar os elementos do modelo em execução com aqueles presentes no novo modelo submetido. Primeiramente, criamos um modelo a partir do zero e realizamos sua submissão. Neste caso, o tempo gasto para realizar esta operação foi apenas para checar se já havia um modelo executando. A partir disso, as próximas comparações levaram em consideração o modelo em execução no espaço inteligente e variamos de 5 em 5 a quantidade de novos elementos adicionados ao modelo, até chegar a um modelo com 35 elementos. A curva gerada no gráfico se comporta de maneira linear a medida que aumentamos a quantidade de elementos do modelo. Entretanto, o tempo de processamento para a comparação entre o modelo em execução e o modelo submetido consome a maior parte do processamento dos modelos do usuário do sistema, sendo portanto o gargalo da camada *Model Processing*.

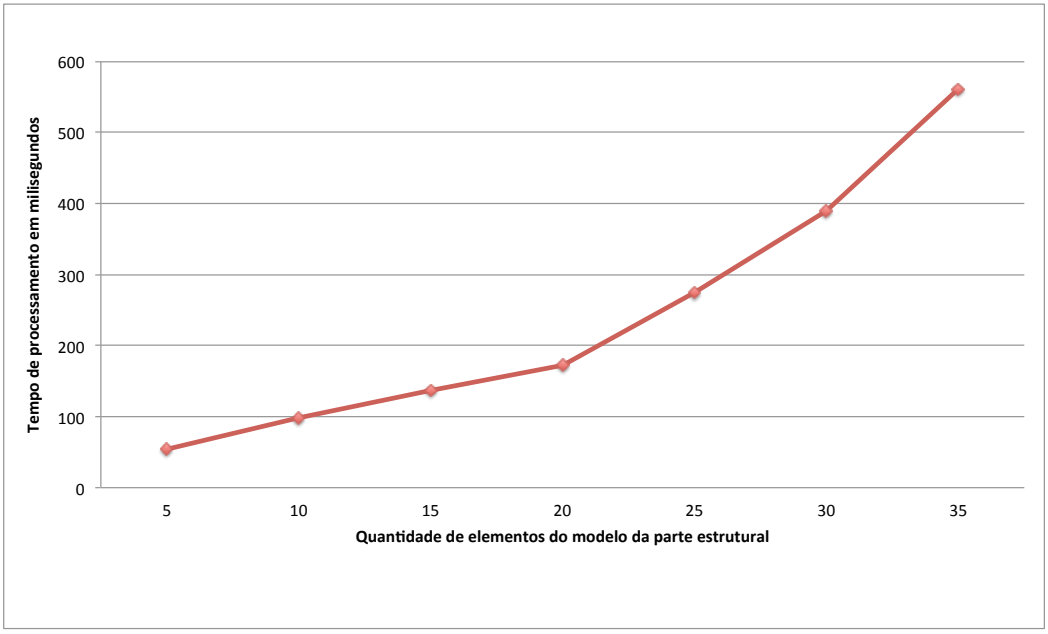



**Figura 6.10:** *Gráfico da Checagem de Conformidade do Modelo*

Para um modelo com 35 elementos, pertencentes a parte estrutural, e 5 políticas, o tempo médio de processamento para checagem de consistência interna do modelo, checagem de conformidade e comparação com um modelo em execução com 30 elementos, foi de 3336 milissegundos. O maior tempo de processamento gasto ocorreu no componente de comparação entre os modelos, que obteve 2658 milissegundos. Sendo assim, podemos concluir que o gargalo apresentado nesta camada está relacionado à comparação entre o modelo em execução e o novo modelo submetido pelo usuário do sistema.

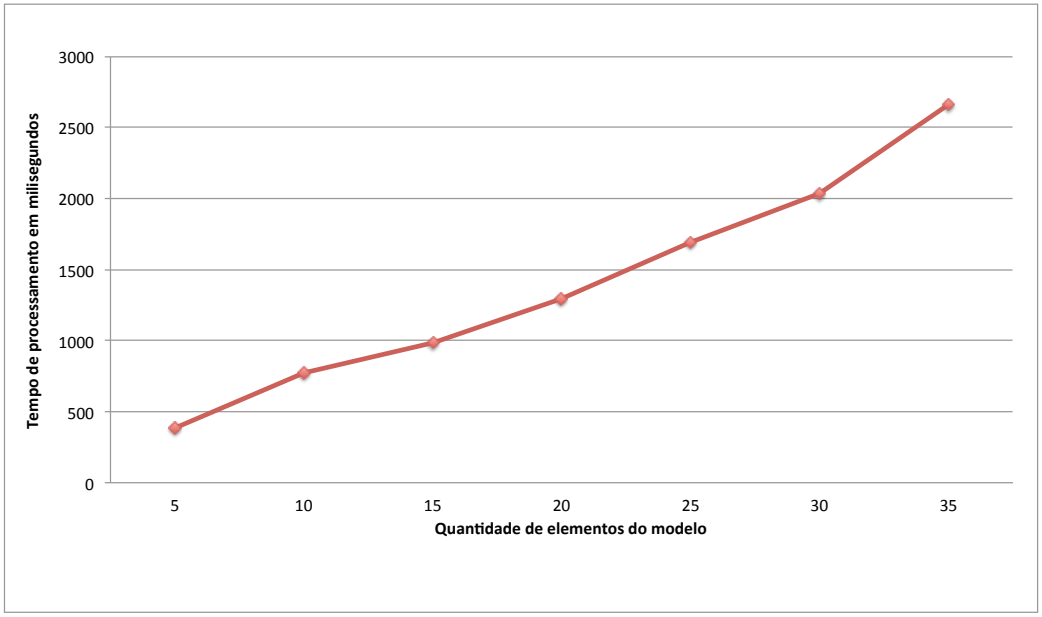



**Figura 6.11:** *Gráfico da Comparação entre os Modelos*

**Experimento 4:**

Por fim, nosso último experimento visou avaliar o tempo gasto para a 2SVM-*Controller* processar a entrada de usuários acompanhados de seus respectivos objetos inteligentes. Na base de dados de usuários da máquina, tínhamos 80 usuários cadastrados, e no modelo em tempo de execução global tínhamos 5 papéis do usuários definidos e 15 tipos de `SmartObjects`, cada qual com 5 características. Este experimento foi dividido em três etapas, em que medimos: *i)* Tempo de processamento para identificar o papel do usuário e o tipo do objeto inteligente; *ii)* Tempo necessário para atualizar o modelo em tempo de execução Global; e *iii)* Tempo gasto para selecionar a macro correspondente a cada papel do usuário ou tipo de objeto inteligente.

Para realizar este experimento, simulamos a entrada dos usuários a partir de outro computador em que tínhamos uma instância da 2SVM-*Client*. Estes eventos do ambiente foram gerados a cada segundo e a medição de tempo de processamento na 2SVM-*Controller* foi realizada a partir da chegada de cada um destes eventos, em que o evento é recebido pelo componente *Event Handler*, presente na camada de *middleware* da máquina.

O tempo para identificar o objeto inteligente foi superior aquele gasto para identificar o papel do usuário, haja visto que um dispositivo possui um conjunto de características que devem ser comparadas com aquelas definidas no modelo do usuário do sistema. O tempo médio de processamento para a identificação do papel do usuário é de 21 milissegundos, desvio padrão 20, e do objeto inteligente, é de 73 milissegundos, desvio padrão 38.

Já o tempo gasto para atualizar o modelo em tempo de execução global do espaço inteligente, assim como no processamento anterior, o objeto inteligente consome mais tempo, haja vista a quantidade de informações presentes neste elemento. O tempo médio de processamento para a atualização do modelo em tempo de execução global, relacionado ao papel do usuário é de 28 milissegundos, com desvio padrão 15. Em relação ao objeto inteligente, o tempo de processamento é de 98 milissegundos, com desvio padrão 27.

Por fim, medimos o tempo necessário para a 2SVM-*Controller* selecionar a macro correspondente ao elemento do modelo obtido a partir da identificação pelo componente *Matching*. O tempo gasto para a seleção das macros do repositório segue o mesmo padrão dos tempos de processamento para identificação e atualização do M@RT, em que os objetos inteligentes sempre consomem maior poder de processamento que aqueles gastos para os usuários. Se compararmos este experimento, com aquele referente à atualização do M@RT global, observamos que o tempo despendido para seleção das macros é um pouco inferior, uma vez que realizamos apenas uma consulta. O tempo

médio de processamento para seleção da macro relacionada ao papel do usuário é de 20 milissegundos, desvio padrão 15. Para seleção da macro relacionada ao objeto inteligente é de 75 milissegundos, desvio padrão 20.

O tempo médio de processamento para identificação do papel do usuário, atualização de seu modelo em tempo de execução e seleção de sua macro foi de 69 milissegundos. Já o tempo médio para realização destas mesmas operações em relação ao objeto inteligente foi de 241 milissegundos. Como somatório destes dois tempos, temos 310 milissegundos, ou seja, este é o tempo necessário para que outro usuário acompanhado de seu objeto inteligente deve esperar para passar por todo este processo de identificação, atualização do M@RT e seleção da respectiva macro. Caso novos usuários com seus objetos inteligentes entrem no ambiente de computação ubíqua, eles serão enfileirados e só poderão pertencer ao espaço inteligente após a conclusão deste processamento.

## 6.4 Conclusão

Este capítulo apresentou uma demonstração de uso da linguagem de modelagem de espaços inteligentes, bem como quatro experimentos em que foram medidos os tempos de processamento utilizados pela máquina de execução para realizar operações internas da 2SVM. A demonstração do uso da linguagem contou com a modelagem de dois cenários de espaços inteligentes, o da sala, dividido em sala de aulas e de reuniões, e o cenário da academia inteligente.

Já os experimentos relacionados à máquina de execução de modelos, medimos o tempo de processamento para as seguintes funcionalidades: *i)* Checagem da consistência interna do modelo; *ii)* Checagem de conformidade; *iii)* Comparação entre os modelos; *iv)* Identificação dos papéis do usuário e objetos inteligentes; *v)* Atualização do M@RT global; e *vi)* Seleção das macros em seu repositório.

Como trabalhos futuros, pretendemos avaliar a linguagem de modelagem 2SML com usuários reais, e compará-la com demais trabalhos que propõem outras abordagens de programação de espaços inteligentes. Esta comparação permitirá verificar se nossa linguagem facilita a programação destes ambientes, em relação aos trabalhos relacionados. Além disso, pretendemos avaliar a escalabilidade da máquina de execução de modelos diante da entrada de diversos usuários e seus objetos inteligentes em determinados intervalos de tempo.

CAPÍTULO 7

# Trabalhos Relacionados

Na literatura, existem diversas abordagens para lidar com a programação de espaços inteligentes. Alguns trabalhos lidam com este problema por meio de abordagens dirigidas por modelos ou mesmo *frameworks* e plataformas de *middleware* específicas para o domínio de programação de espaços inteligentes. Identificamos também plataformas de *middleware* propósito geral ou *middleware* adaptativos, que também permitem a programação dos espaços inteligentes. Por fim, apresentamos tecnologias e produtos desenvolvidos por empresas e universidades que auxiliam em tarefas diárias dos usuários dos espaços inteligentes, como por exemplo, alertar sobre condições de tráfego e climáticas.

Este capítulo apresenta uma análise dos trabalhos relacionados, comparando-os com nossa abordagem de modelos em tempo de execução. Nas Seções 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 7.10, 7.11, 7.12, apresentamos a revisão da literatura, com todos os trabalhos identificados, bem como uma comparação entre eles. Na Seção 7.13 apresentamos uma tabela com uma lista de desafios implementados em nosso trabalho na forma de funcionalidades e comparamos com os demais trabalhos relacionados. Por fim, na Seção 7.14, apresentamos a conclusão do capítulo.

## 7.1 Resource-Oriented and Ontology-Driven Development (ROOD)

O trabalho descrito por *Corredor et al.* [30] propõe uma metodologia denominada *Resource-Oriented and Ontology-Driven Development* (ROOD), baseada na abordagem de *Model-Driven Architecture* (MDA), para o desenvolvimento e implantação de espaços inteligentes. ROOD utiliza as tecnologias de Internet das Coisas (*Internet of Things* - IoT) e *Web* das Coisas (*Web of Things* - WoT) e as inclui no espaço inteligente que será modelado. A metodologia permite modelar entidades de um ambiente inteligente, como sensores, atuadores e interfaces de I/O. ROOD propõe uma linguagem de modelagem chamada *Smart Space Modeling Language* (SSML). De acordo com a arquitetura estrati-

ficada da OMG [93], em quatro níveis de abstração, a SSML encontra-se na camada M2 e estende o meta-modelo UML, para criar seus próprios perfis, que definem cada elemento do modelo como entidades, relacionamentos e interfaces. Assim como em nosso trabalho, a metodologia ROOD oferece programação centrada no usuário por meio da linguagem SSML, que possui dois tipos de modelos: *i) Smart Object Model* (SOM), que permite modelar os recursos do ambiente; e *ii) Environment Context Model* (ECM), que permite descrever comportamentos de alto nível e informações contextuais com o uso de ontologias. Nossa abordagem possui apenas uma linguagem, a 2SML, que permite a modelagem da parte estrutural e comportamental do espaço inteligente. No que se refere à implantação da programação do espaço inteligente nos dispositivos, a metodologia ROOD realiza esta tarefa em tempo de execução, por meio da geração automática de código. Em nosso trabalho, essa implantação é feita pela própria máquina de execução de modelos, a 2SVM. *Corredor et al*. não desenvolveram nenhum mecanismo para lidar com a re-configuração do estado de execução das aplicações em tempo de execução, ou seja, eles não permitem que as aplicações tenham seu estado reconfigurado, como é feito em nosso trabalho. Por fim, a metodologia ROOD também não oferece nenhum mecanismo para mudar a finalidade do espaço inteligente sem que seja necessária a reinicialização do sistema, como é feito pela 2SVM.

## 7.2 *Gilman et al.*

O trabalho apresentado por *Gilman et al*. [52] tem por objetivo gerenciar ambientes de computação ubíqua, por meio do estudo do controle de espaços inteligentes em meta-nível pode auxiliar na programação destes ambientes, devido à natureza volátil das interações que ocorrem entre recursos e usuários. Como contribuições, *Gilman et al*. apresentam um *framework* para controle de espaços inteligentes em meta-nível. Para fazer isso, ele separa as tarefas de adaptação de contexto do espaço inteligente, que ele denomina *object-level*, e a adaptação destas tarefas de adaptação, em meta-nível. Este *framework* é composto por: *i) Ground Level*: é formado por componentes de sensoriamento do ambiente e usuários, denominados perceptores, e por componentes que modificam o ambiente e entregam informações aos usuários, os atuadores; *ii) Object-Level*: contém mecanismos para compor serviços, localizar recursos e executar serviços de composição, implantação e execução e; *iii) Meta-Level*: controla e monitora a execução de tarefas do *object-level*. *Gilman et al*. oferecem um mecanismo que permite aos usuários programarem aspectos comportamentais por meio de uma linguagem centrada no usuário, que possui uma sintaxe própria. Em nosso trabalho, nós definimos a 2SML, que permite aos usuários que têm conhecimento específico do domínio de espaços inteligentes, programarem o ambiente por meio de um editor gráfico de modelagem. Ele oferece caixinhas e

associações que permitem definir os tipos de elementos que poderão fazer parte do ambiente, bem como o relacionamento entre cada um deles e seus respectivos comportamentos em tempo de execução. Entretanto, o trabalho de *Gilman et al.* obriga que esta implantação desta programação do espaço inteligente seja realizada de maneira manual em cada um dispositivos do ambiente, o que ocorre de maneira automática no nosso trabalho. Além disso, *Gilman et al.* não oferecem suporte a manipulação do estado de execução das aplicações, como é feito em nosso trabalho. No que se refere à mudança de finalidade dos espaços inteligentes, o *framework* proposto por *Gilman et al.* possui um mecanismo de controle que realiza esta operação em tempo de execução, assim como é feito em nosso trabalho.

## 7.3 *Soldatos et al.*

A proposta apresentada por *Soldatos et al.* [117] propõe um *framework* arquitetural e com um conjunto de componentes de *middleware* para facilitar a integração de elementos perceptuais (algoritmos de processamento de sinais, tipicamente para processamento de áudio e vídeo), sensores, atuadores, *scripts* de modelagem de contexto e aplicações de computação ubíqua em espaços inteligentes. A arquitetura permite também a descoberta e o gerenciamento de recursos. Este *framework* apresenta uma visão em três níveis, que auxiliam no gerenciamento das entidades de um *smart space*: *i) Sensors Tier*, responsável por realizar o monitoramento do ambiente, sendo que os sinais coletados pelos sensores podem ser posteriormente processados, para extrair o contexto do ambiente; *ii) Tier of Perceptual Components*, que possui algoritmos baseados em processamento de sinal, em particular, áudio e vídeo, sendo que estes componentes perceptuais extraem sugestões de contexto, principalmente relacionados à identificação e localização de pessoas e objetos; *iii) Tier of Agents*, responsável por modelar e rastrear situações contextuais. O trabalho proposto por *Soldatos et al.* não oferece a programação dos espaços inteligentes centrada no usuário, uma vez que é necessário que o usuário utilize as interfaces oferecidas pela plataforma de *middleware* desenvolvida para definir uma programação para o ambiente. Portanto, não há neste trabalho um mecanismo de mais alto nível que ofereça abstrações mais próximas do cotidiano do usuário, como por exemplo, abstrações que permitam definir tipos de usuários, dispositivos e aplicações que poderão fazer parte do espaço inteligente. Em nosso trabalho, nós definimos estas abstrações em nossa linguagem de modelagem de espaços inteligentes. O trabalho de *Soldatos et al.* não oferece mecanismos para que sejam realizadas mudanças de finalidade do ambiente em tempo de execução, sem a necessidade de reinicialização do sistema. Neste caso, é necessário que o sistema seja parado, a nova finalidade seja definida em cada um dos elementos do ambiente e, após isso, ele seja reiniciado. Quanto à implantação automática dos espaços

inteligentes em tempo de execução, eles oferecem esta funcionalidade de maneira parcial, apenas para as aplicações e serviços. Em nosso trabalho, esta implantação é feita em todos os elementos do espaços inteligente, incluindo os dispositivos, por meio da máquina de execução de modelos 2SVM. Por fim, a plataforma de *middleware* proposta por *Soldatos et al*. oferece mecanismos para que o estado das aplicações seja manipulado.

## 7.4 *Gouin-Vallerand et al.*

O trabalho descrito por *Gouin-Vallerand et al*. [53] tem por objetivo apresentar uma plataforma de *middleware* que permite implantar aplicações e serviços em espaços inteligentes através de mecanismos de auto-configuração, em particular, computação pervasiva autonômica. De acordo com este trabalho, o processo de auto-configuração pode simplificar a complexidade e reduzir o custo de programação e implantação de dispositivos, serviços e aplicações em um espaço inteligente. O mecanismo de auto-configuração é inspirado na computação autonômica e computação pervasiva que devem ser capazes de automatizar os elementos de um ambiente inteligente. Como demais objetivos, *Gouin et al*. propõem: *i)* implantação e programação da aplicação no espaço inteligente; *ii)* configuração e integração da aplicação em um espaço inteligente. Para tratar a propriedade de contexto, o trabalho utiliza ontologias definidas em *Ontology Web Language* (OWL), que permitem descrever situações de contexto do ambiente e selecionar os melhores recursos. O *middleware*, por sua vez, possui os seguintes componentes: *i) Managing Tool*: aplicação GUI para gerenciar e configurar o *smart space*; *ii) Environment Gateway Device*: componente por meio do qual os dispositivos recebem as requisições de gerenciamento do *Managing Tool*, analisam e executam estas requisições; *iii) Ontology Device*: dispositivo que gerencia a ontologia do espaço inteligente e oferece serviços de ontologia para os demais dispositivos do ambiente; *iv) Environment Device*: parte do software que é instalada em cada dispositivo do ambiente. Ele oferece os serviços para gerenciar a implantação de aplicações nestes mesmos dispositivos. *Gouin et al*. oferece uma aplicação GUI para gerenciar e configurar o espaço inteligente. Embora não seja uma linguagem centrada no usuário, ela facilita a programação do ambiente de computação ubíqua. Quanto à implantação da programação no espaço inteligente, ela é realizada de maneira parcial, pois é feita apenas para aplicações e serviços por meio da plataforma de *middleware* oferecida. Neste trabalho não há nenhum mecanismo para reconfiguração do estado de execução das aplicações em tempo de execução, assim como é feito pela 2SVM. Por fim, o trabalho descrito também não oferece nenhum mecanismo responsável por definir uma nova finalidade no ambiente em tempo de execução. Portanto, é necessário que o sistema seja parado, a nova finalidade seja definida e, após isso, o sistema seja novamente iniciado.

## 7.5 UbiXML

O *UbiXML* [3] é um sistema programável de gerenciamento de recursos de computação ubíqua. As aplicações são estruturadas como documentos *XML*. A arquitetura do *UbiXML* é composta por duas camadas de *middleware*: *i) Element Management-Level* (EML): contém operações de gerenciamento dos recursos, como câmeras, microfones, roteadores, *switches*, sensores de monitoramento do corpo humano, reconhecimento facial e serviços de *text-to-speech*; *ii) Application-Level XML*: compreende operações responsáveis pelo gerenciamento de aplicações ubíquas. O UbiXML permite construir aplicações que gerenciam o espaço inteligente e, portanto, manipula seus estados de execução assim como ocorre na 2SVM. O UbiXML não possui uma linguagem de alto nível que permite os usuários programarem o ambiente de computação de maneira centrada no usuário, uma vez que sua plataforma de *middleware* oferece apenas interface que auxiliam no gerenciamento dos recursos destes ambientes. A implantação deste sistema programado também não ocorre de maneira automática como na 2SVM e a mudança de finalidade do espaço inteligente requer a reinicialização do sistema.

## 7.6 *Feeney et al.*

O trabalho proposto por *Feeney et al*. [41] explora a sobreposição dos conceitos de computação ubíqua e computação autonômica para a programação de espaços inteligentes. Para fazer isso ele utiliza o *framework* OSGi e o *Toolkit* Autonômico da IBM para explorar esta sobreposição. No geral, este trabalho concentrou seu maior esforço em questões técnicas para a programação de espaços inteligentes. *Feeney et al*. desenvolveram uma arquitetura autonômica para espaços inteligentes e realizaram testes em um ambiente residencial. A arquitetura permite controlar serviços de autenticação e serviços de controle ambiental, como por exemplo, aquecedor e ar-condicionado. Para facilitar o gerenciamento do ambiente, foi desenvolvida no trabalho uma aplicação GUI para controlar a configuração da casa inteligente. Assim como o trabalho proposto por *Gouin et al*., *Feeney et al*. desenvolveu uma aplicação gráfica que permite configurar o espaço inteligente, e embora não tenha uma linguagem que permita a programação centrada no usuário, consideramos que esta funcionalidade foi realizada de maneira parcial, se comparada a nosso trabalho, que possui a linguagem de modelagem 2SML. As demais funcionalidades apresentadas neste trabalho, implantação automática de espaços inteligentes em tempo de execução, reconfiguração do estado das aplicações em tempo de execução e re-programação do espaço inteligente em tempo de execução sem a necessidade de reinicialização do sistema, não são realizadas.

## 7.7 *Van Der Meer et al.*

O trabalho descrito por *Van Der Meer et al.* [126] apresenta uma abordagem para gerenciar a integração de redes, serviços e aplicações em espaços inteligentes. Como objetivos principais, ele visa tratar a composição, escalabilidade, confiabilidade e robustez, por meio de um mecanismo de auto-adaptação. Seu foco principal está no *middleware* de gerenciamento, para controlar e utilizar os recursos distribuídos. Para atingir estes objetivos, ele fornece um modelo conceitual que é a base para a especificação da arquitetura. Este modelo apresenta regras para componentes de uma arquitetura específica e descreve os relacionamentos entre os componentes. As definições criadas são empregadas para descrever aspectos particulares da arquitetura: *i) Application Plane*: focado no projeto e implementação de aplicações; *ii) Object Plane*: concentra na modelagem de objetos distribuídos; *iii) Service Plane*: modela uma coleção de interfaces e objetos (serviços) que fornecem funções básicas para as aplicações; *iv) Technology Plane*: responsável pelas tecnologias, tais como SNMP, CORBA, Jini, entre outras. Nenhuma das funcionalidades desse *middleware* de gerenciamento de espaços inteligentes lidam com os desafios que enumeramos em nossa tese, relacionadas à programação de ambientes de computação ubíqua.

## 7.8 *O'Sullivan & Wade*

A proposta descrita por *O'Sullivan & Wade* [92] trata a programação de espaços inteligentes sob o ponto de vista da composição dinâmica e adaptação dos ambientes ubíquos. Sua abordagem permite que usuários comuns realizem estas tarefas, não obrigando que eles tenham conhecimentos técnicos sobre como compor e utilizar serviços. Para resolver isso, o trabalho propõe um *framework* juntamente com uma arquitetura e uma metodologia de desenvolvimento. O *framework*, denominado *Smart Space Management Framework* (SSMF), é estruturado em quatro partes: *i) Logical Architecture*: descreve os conceitos estruturais do *framework* e seus relacionamentos, independente de tecnologia de implementação; *ii) Development Methodology*: fornece processos e notações necessárias para desenvolver componentes gerenciáveis; *iii) Technology Architecture*: define como os conceitos expressos na *Logical Architecture* podem ser implementados usando um conjunto de tecnologias; *iv) Reusable Elements*: repositório para componentes reusáveis. Embora o trabalho proposto por *Declan et al.* não implemente a funcionalidade de programação centrada no usuário, consideramos que esta funcionalidade foi desenvolvida de maneira parcial, haja visto que o *framework* permite descrever os elementos por meio de conceitos estruturais. As demais funcionalidades, referentes à implantação automática de espaços inteligentes em tempo de execução, reconfiguração do estado de execução

das aplicações em tempo de execução e mudança de finalidade do espaço inteligente em tempo de execução sem a necessidade de reinicialização do sistema, não são realizadas por ele.

## 7.9 *Helal et al.*

O trabalho desenvolvido por *Helal et al.* [63] apresenta uma casa inteligente que tem por objetivo atender às necessidades de pessoas idosas e com deficiência. Os autores utilizaram um conjunto de objetos inteligentes existentes à época, tais como forno micro-ondas, piso inteligente, *displays*, tomadas elétricas e espelhos, além de objetos inteligentes que estavam em desenvolvimento, como por exemplo cama, máquina de lavar roupas e mesa de jantar. A arquitetura do *middleware Gator Tech Smart House* contém as seguintes camadas: *i) Physical layer*: lida com os dispositivos e aparelhos em gerais que os ocupantes da casa estão usando; *ii) Sensor platform layer*: lida com os sensores e atuadores do espaço inteligente; *iii) Service layer*: lida com os serviços; *iv) Knowledge layer*: camada de conhecimento, que possui ontologias para lidar com cada um dos serviços oferecidos; *v) Context management layer*: lida com o contexto do ambiente; e *vi) Application layer:* lida com as aplicações. O trabalho desenvolvido por *Sumi et al.* desenvolveu apenas a funcionalidade de reconfiguração do estado das aplicações em tempo de execução. Para isso, foi desenvolvida uma camada de aplicação que lida diretamente com elas e permite manipular aspectos relacionados à lógica da aplicação. Em nosso trabalho, nós manipulamos apenas os estados de execução definidos, iniciar, pausar, continuar e finalizar. As demais funcionalidades implementadas pela 2SVM como programação centrada no usuário, implantação automática dos espaços inteligentes, mudança de finalidade do espaço inteligente em tempo de execução sem a necessidade de reinicialização do sistema, não foram realizadas por ele.

## 7.10 *Roalter et al.*

*Roalter et al.* [106] propõem o *Robotic Operating System* (ROS), um *middleware* com a capacidade de integrar sensores, atuadores, aplicações e serviços em ambientes de computação ubíqua. O objetivo desse trabalho foi explorar o potencial do *middleware* ROS no contexto de caso de uso de um ambiente de um escritório inteligente. Para isso, eles conectaram um conjunto de dispositivos heterogêneos ao *middleware* e implementaram serviços que permitiram adicioná-los ao espaço inteligente. Dentre os serviços oferecidos temos aqueles que lidam com sensores e atuadores, serviço de contexto e serviços voltados para o usuário final que utilizam dados contextuais para oferecer informações aos usuários, tais como localização dos colegas no ambiente, entre outros. O *middleware*

*Robotic Operating System* não lida com nenhum daqueles desafios apresentados e tratados por nosso trabalho.

## 7.11 *García-Herranz et al.*

O trabalho descrito por *García-Herranz et al.* [50] apresenta um mecanismo de agentes baseado em regras para programar ambientes de computação ubíqua. Para fazer isso, eles proporam uma linguagem base que permite ao usuário final definir comportamentos para o ambiente através de determinados eventos, condições, que são especificadas pelo próprio usuário; e por fim, as ações que serão executadas no ambiente a partir dos eventos engatilhados e também a partir da avaliação das condições. *Garcia et al.* implementaram a funcionalidade de programação centrada no usuário através de uma linguagem que lida com regras contextuais baseadas na estrutura de evento, condição e ação. Desta forma, cada comportamento do ambiente é definido por meio dessas regras, que controla as interações entre os usuários, as aplicações e os dispositivos. Em nosso trabalho, também definimos regras evento, condição e ação, que também lidam com o comportamento dinâmico do ambiente. Entretanto, para definir o conjunto de recursos que podem fazer parte do ambiente, nós oferecemos na linguagem mecanismos que auxiliam na definição de tipos de usuários, aplicações e dispositivos. As demais funcionalidades de nosso trabalho não são contempladas por *Garcia et al.*

## 7.12 open Home Automation Bus (openHAB)

O *open Home Automation Bus* (openHAB) [91] é uma plataforma de software que permite a integração de sistemas e tecnologias no âmbito da automação residencial. Esta plataforma é baseada na especificação OSGi, tendo sido desenvolvida em Java. Sua arquitetura é modular, o que permite a inserção e remoção de funcionalidades em tempo de execução, não tendo a necessidade de parar o sistema. Como o openHAB integra diferentes tecnologias de hardware e protocolos, ele é considerado um sistema de sistemas. Ele possui um conjunto de interfaces de usuário, tais como *Classic UI*, *GreenT* e *Comet Visu*, tendo ainda clientes para Android (HABDroid), e para iOS (openHAB). O *open Home Automation Bus* não implementa nenhuma das funcionalidades oferecidas por nosso trabalho, entre elas a programação centrada no usuário, implantação automática de espaços inteligentes, reconfiguração do estado de execução das aplicações e mudança de finalidade do ambiente sem necessidade de reinicialização do sistema.

## 7.13 Comparação entre os Trabalhos Relacionados

A Tabela 7.1 resume todas as funcionalidades que foram implementadas em nosso trabalho, e as compara com os trabalhos relacionados identificados. Algumas dessas funcionalidades podem ter sido realizadas de maneira total, parcial ou podem não ter sido implementadas.

As plataformas de *middleware* mais gerais ou adaptativos, além dos demais trabalhos que apresentam produtos relacionados à programação, gerenciamento ou configuração de espaços inteligentes não apresentam as funcionalidades oferecidas pela 2SVM e, portanto, foram omitidos da tabela comparativa.

## 7.14 Conclusão

A Tabela 7.1 apresenta as funcionalidades implementadas em nosso trabalho, tendo em vista os desafios enunciados na Seção 1.2. A partir disso, constatamos que nenhum dos trabalhos relacionados consegue realizar todas elas. Isto ocorre principalmente em virtude da abordagem de modelos em tempo de execução que utilizamos para resolver o problema da programação dos espaços inteligentes, que se apresenta bem apropriada para lidar com a natureza dinâmica e volátil destes ambientes. Além disso, em nosso trabalho, identificamos problemas relacionadas a área de programação de espaços inteligentes que não receberam a devida atenção nos demais trabalhos, como por exemplo, a reconfiguração do estado das aplicações em tempo de execução e a re-programação do espaço inteligente em tempo de execução sem a necessidade de reinicialização do sistema.

Os trabalhos que utilizam abordagens dirigidas por modelos [30, 52] não exploraram seus benefícios, em particular, a funcionalidade de re-programação do espaço inteligente em tempo de execução sem a necessidade de reinicialização do sistema. Já demais abordagens [117, 53, 3, 41, 126, 92, 63, 106, 50, 91], resolvem os demais desafios relacionados à programação de espaços inteligentes através de soluções que foram implementadas de forma parcial ou mesmo não foram realizadas.

Nossa abordagem contribui para a programação de espaços inteligentes uma vez que oferece uma linguagem de modelagem que lida diretamente com os elementos utilizados no seu dia a dia do usuário, como dispositivos e aplicações. Estes elementos são apresentados por meio de um editor gráfico, em que o próprio usuário define quem poderá fazer parte do espaço inteligente, bem como como suas possíveis interações com os demais elementos do ambiente e ações de controle para ele. Uma máquina de execução de modelos também é oferecida, sendo responsável por implantar a programação definida e por gerenciar as interações ocorridas entre todos os elementos do espaço inteligente.

**Tabela 7.1:** *Tabela comparativa entre os Trabalhos Relacionados*

| Trabalho | Programação Centrada no Usuário | Implantação automática de espaços inteligentes em tempo de execução | Reconfiguração do estado das aplicações em tempo de execução | Mudança de finalidade do espaço inteligente em tempo de execução sem a necessidade de reinicialização do sistema |
|---|---|---|---|---|
| Corredor *et al*. [30] | Sim | Sim | Não | Não |
| Gilman *et al*. [52] | Sim | Não | Não | Sim |
| Soldatos *et al*. [117] | Não | Parcial | Sim | Não |
| Gouin-Vallerand *et al*. [53] | Parcial | Parcial | Não | Não |
| UbiXML [3] | Não | Não | Sim | Não |
| Feeney *et al*. [41] | Parcial | Não | Não | Não |
| Van Der Meer *et al*. [126] | Não | Não | Não | Não |
| O'Sullivan & Wade [92] | Parcial | Não | Não | Não |
| Helal *et al*. [63] | Não | Não | Sim | Não |
| Rolater *et al*. [106] | Não | Não | Não | Não |
| García-Herranz *et al*. [50] | Sim | Não | Não | Não |
| openHAB [91] | Não | Não | Não | Não |
| 2SVM | Sim | Sim | Sim | Sim |

CAPÍTULO 8

# Conclusão

A Computação Ubíqua tomou forma graças ao trabalho de *Mark Weiser* [133], que vislumbrou a possibilidade de integrar a computação às atividades do dia a dia, tornando seu uso invisível para o usuário. A pesquisa na área de computação ubíqua tem por objetivo criar um ambiente inteligente com dispositivos computacionais embarcados e com conectividade em rede, fornecendo aos usuários acesso transparente aos serviços [105].

Os ambientes de computação ubíqua são extremamente voláteis, uma vez que são altamente dinâmicos e mudam de maneira imprevisível. Os dispositivos desses ambientes possuem conectividade volátil, uma vez que a maior parte deles estão conectados por redes sem fio, mais suscetíveis a desconexão. Outra característica desses ambientes é a interação espontânea, ou seja, as associações entre os elementos de um espaços inteligente, sejam eles usuários, objetos inteligentes e aplicações, são configuradas a medida que novos elementos entram no espaço inteligente.

A fim de reduzir os problemas relacionados às ações que os usuários realizam em um ambiente físico, como por exemplo, utilizar dispositivos e aplicações que não têm permissão, uma vez que não é possível prever que tipo de operações cada um irá desempenhar, a redução do escopo de um ambiente de computação ubíqua a áreas delimitadas permite restringir e limitar os usuários a comportamentos particulares. Espaços inteligentes estendem a computação para os ambientes físicos e permitem que diferentes dispositivos neles presentes ofereçam suporte coordenado aos usuários, com base em suas preferências e na situação atual do ambiente físico [116].

Programar um espaço inteligente é definir uma finalidade/intenção para ele. Para isto, é necessário determinar a estrutura e o comportamento para os elementos do ambiente. Na estrutura, criamos os tipos dos elementos que farão parte do espaço inteligente e definimos como eles se relacionarão entre si. No comportamento, definimos as ações que serão realizadas pelos elementos do ambiente diante das mudanças de contexto que neles ocorrerem, ou por meio dos mais diversos eventos detectados pelos dispositivos presentes no ambiente de computação ubíqua.

## 8.1 Contribuições

Nesta tese nós investigamos o problema da programação dos espaços inteligentes, e em particular, tratamos os seguintes desafios:

- Programação de espaços inteligentes centrada no usuário;
- Implantação automática de espaços inteligentes no ambiente em tempo de execução;
- Reconfiguração do estado das aplicações em tempo de execução; e
- Re-programação do espaço inteligente em tempo de execução sem a necessidade de reinicialização do sistema.

Para resolvê-los, propomos uma linguagem de modelagem de espaços inteligentes bem como uma máquina para executar modelos do espaço inteligente. As contribuições desta tese são:

- Uma nova abordagem para programação de espaços inteligentes através dos modelos em tempo de execução: a abordagem de modelos em tempo de execução permite representar os elementos de um espaço inteligente e modificar a estrutura e comportamento do ambiente em tempo de execução;
- A definição de uma linguagem de modelagem específica de domínio, para espaços inteligentes: esta linguagem permite aos usuários programadores de espaços inteligentes definirem diferentes finalidades para um espaço físico, a partir de sua estrutura e de seu comportamento;
- A especificação de uma máquina de execução de modelos para espaços inteligentes: a máquina de execução tem por objetivo processar os modelos desenvolvidos pelo usuário e implantá-los nos objetos inteligentes do ambiente. A máquina lida com a natureza dinâmica do ambiente de computação ubíqua, reconfigurando os elementos nele presentes, como dispositivos e aplicações de acordo com os mais diversos eventos que podem ocorrer no espaço inteligente.

A definição de uma arquitetura de sistema para materializar estas contribuições deu origem à seguinte publicação:

- FREITAS, L. A.; COSTA, F. M.; ROCHA, R. C.; ALLEN, A. **An architecture for a smart spaces virtual machine**. In: *Proceedings of the 9th Workshop on Middleware for Next Generation Internet Computing*, p. 7. ACM, 2014.

## 8.2 Trabalhos Futuros

Como trabalhos futuros, pretendemos estender o metamodelo da linguagem para dar suporte a modelagem de múltiplos espaços inteligentes em um mesmo espaço físico.

Isso permitiria, por exemplo, que em um mesmo ambiente tivéssemos diferentes espaços inteligentes compartilhando os mesmos recursos do ambiente, entre eles, aplicações ubíquas e objetos inteligentes. A extensão do metamodelo implica também na extensão dos componentes da máquina de execução de modelos, que terá de lidar com a concorrência pelos recursos do ambiente. Desta forma, temos como implicação a extensão da máquina de execução de modelos para lidar com o acesso concorrente aos objetos inteligentes e aplicações ubíquas do ambiente por meio dos usuários.

Além disso, outro tópico importante a ser tratado é o mecanismo de tolerância a falhas para a 2SVM-*Controller*. Em nosso trabalho, existe apenas um ponto do processamento do modelo e por consequência, de manutenção do modelo em tempo de execução global. Com duas ou mais máquinas de execução de modelos controladoras do ambiente, é possível manter os espaços inteligentes em execução sem que haja interrupção, em caso de falha de uma delas.

Por fim, e também como trabalho futuro, pretendemos examinar a possibilidade de manipular aspectos da lógica de programação das aplicações ubíquas. Em nosso trabalho, não manipulamos aspectos específicos das aplicações, mas apenas aqueles que definimos como estados: iniciar, pausar, continuar e finalizar. Permitir que os usuários controlem demais aspectos das aplicações, intrínsecos a sua lógica, possibilitará explorar ainda os benefícios dos espaços inteligentes.

# Referências Bibliográficas


[1] ABOWD, G.; ATKESON, C.; ESSA, I. **Ubiquitous smart spaces**. *A white paper submitted to DARPA*, 1998.

[2] ABOWD, G. D.; DEY, A. K.; BROWN, P. J.; DAVIES, N.; SMITH, M.; STEGGLES, P. **Towards a better understanding of context and context-awareness**. In: *International Symposium on Handheld and Ubiquitous Computing*, p. 304–307. Springer, 1999.

[3] ALEXOPOULOS, D.; SOLDATOS, J.; KORMENTZAS, G.; SKIANIS, C. **UbiXML: programmable management of ubiquitous computing resources**. *International Journal of Network Management*, 17(6):415–435, 2007.

[4] ALFERES, J. J.; BANTI, F.; BROGI, A. **An event-condition-action logic programming language**. In: *Logics in Artificial Intelligence*, p. 29–42. Springer, 2006.

[5] ALISSON, M. **A Generic Model of Execution for Synthesizing Domain-Specic Models**. PhD thesis, Florida International University, 2014.

[6] ALLEN, A. A. **Abstractions to support dynamic adaptation of communication frameworks for user-centric communication**. PhD thesis, Florida International University, 2011.

[7] ALLISON, M. **A generic model of execution for synthesizing domain-specific models**. PhD thesis, Florida International University, 2014.

[8] ALLISON, M.; MORRIS, K. A.; YANG, Z.; CLARKE, P. J.; COSTA, F. M. **Towards reliable smart microgrid behavior using runtime model synthesis.** In: *HASE*, p. 185–192. IEEE Computer Society, 2012.

[9] ATZORI, L.; IERA, A.; MORABITO, G. **The internet of things: A survey**. *Computer Networks*, 54(15):2787–2805, 2010.

[10] BAHETI, R.; GILL, H. **Cyber-physical systems**. *The impact of control technology*, 12:161–166, 2011.

[11] BAILEY, J.; PAPAMARKOS, G.; POULOVASSILIS, A.; WOOD, P. T. **An event-condition-action language for xml**. In: *Web Dynamics*, p. 223–248. Springer, 2004.

[12] BARDRAM, J. E.; BOSSEN, C. **Moving to get ahead: Local mobility and collaborative work.** In: Kuutti, K.; Karsten, E. H.; Fitzpatrick, G.; Dourish, P.; Schmidt, K., editors, *ECSCW*, p. 355–374. Springer, 2003.

[13] BELLOTTI, V.; BLY, S. **Walking away from the desktop computer: distributed collaboration and mobility in a product design team**. In: *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, CSCW '96, p. 209–218, New York, NY, USA, 1996. ACM.

[14] BENCOMO, N.; WHITTLE, J.; SAWYER, P.; FINKELSTEIN, A.; LETIER, E. **Requirements reflection: requirements as runtime entities**. In: *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, p. 199–202. IEEE, 2010.

[15] BÉZIVIN, J. **Model driven engineering: An emerging technical space**. In: *Generative and transformational techniques in software engineering*, p. 36–64. Springer, 2006.

[16] BLAIR, G.; BENCOMO, N.; FRANCE, R. B. **Models@Run.Time**. *Computer*, 42(10):22–27, 2009.

[17] BOHN, J.; COROAMĂ, V.; LANGHEINRICH, M.; MATTERN, F.; ROHS, M. **Social, economic, and ethical implications of ambient intelligence and ubiquitous computing**. In: *Ambient intelligence*, p. 5–29. Springer, 2005.

[18] BRYANT, B. R.; GRAY, J.; MERNIK, M.; CLARKE, P. J.; FRANCE, R. B.; KARSAI, G. **Challenges and directions in formalizing the semantics of modeling languages**. *Computer Science and Information Systems/ComSIS*, 8(2):225–253, 2011.

[19] BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. **Pattern-oriented Software Architecture, Volume 1**. John wiley & sons, 1996.

[20] CARIOU, E.; LE GOAER, O.; BARBIER, F.; PIERRE, S. **Characterization of Adaptable Interpreted-DSL**. 2013.

[21] CARIOU, E.; LE GOAER, O.; BARBIER, F.; PIERRE, S. **Characterization of adaptable interpreted-dsml**. In: *European Conference on Modelling Foundations and Applications*, p. 37–53. Springer, 2013.

[22] CASTRO, P.; MUNZ, R. **Managing context data for smart spaces**. *IEEE Personal Communications*, 7(5):44–46, 2000.

[23] CHEN, H.; FININ, T.; JOSHI, A. **An ontology for context-aware pervasive computing environments**. *The knowledge engineering review*, 18(03):197–207, 2003.

[24] CHEN, H.; FININ, T.; JOSHI, A.; KAGAL, L.; PERICH, F.; CHAKRABORTY, D. **Intelligent agents meet the semantic web in smart spaces**. *IEEE Internet Computing*, 8(6):69–79, 2004.

[25] CHEN, K.; SZTIPANOVITS, J.; NEEMA, S. **Toward a semantic anchoring infrastructure for domain-specific modeling languages**. In: *Proceedings of the 5th ACM international conference on Embedded software*, p. 35–43. ACM, 2005.

[26] CLARK, T.; SAMMUT, P.; WILLANS, J. **Applied metamodelling: a foundation for language driven development**. 2008.

[27] CLARKE, M.; BLAIR, G. S.; COULSON, G.; PARLAVANTZAS, N. **An efficient component model for the construction of adaptive middleware**. In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, p. 160–178. Springer, 2001.

[28] CLARKE, P. J.; WU, Y.; ALLEN, A. A.; HERNANDEZ, F.; ALLISON, M.; FRANCE, R. **Towards Dynamic Semantics for Synthesizing Interpreted DSMLs**. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, p. 242, 2012.

[29] COOK, D. J.; DAS, S. K. **How smart are our environments? an updated look at the state of the art**. *Pervasive and mobile computing*, 3(2):53–73, 2007.

[30] CORREDOR, I.; BERNARDOS, A. M.; IGLESIAS, J.; CASAR, J. R. **Model-driven methodology for rapid deployment of smart spaces based on resource-oriented architectures**. *Sensors*, 12(7):9286–9335, 2012.

[31] COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. **Distributed Systems: Concepts and Design**. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[32] COULOURIS, G BLAIR, G. F.; DOLLIMORE, J.; KINDBERG, T. **Distributed systems: concepts and design**. Peearson Education, 2012.

[33] DALY, C. **Emfatic language reference**, 2004.

[34] DE ARAUJO, R. B. **Computação ubíqua: Princípios, tecnologias e desafios**. In: *XXI Simpósio Brasileiro de Redes de Computadores*, volume 8, p. 11–13, 2003.

[35] DENG, Y.; SADJADI, S. M.; CLARKE, P. J.; HRISTIDIS, V.; RANGASWAMI, R.; WANG, Y. **Cvm - a communication virtual machine.** *Journal of Systems and Software*, 81(10):1640–1662, 2008.

[36] DERTOUZOS, M. L. **The Future of Computing, Scientific American**. http://www.scientificamerican.com/article.cfm?id=the-future-of-computing-1999, March 2009. [Online; accessed 19-October-2013].

[37] DEY, A. K. **Understanding and Using Context**. *Personal Ubiquitous Comput.*, 5(1):4–7, Jan. 2001.

[38] DOWNING, T. B. **Java RMI: remote method invocation**. IDG Books Worldwide, Inc., 1998.

[39] ENDRES, C.; BUTZ, A.; MACWILLIAMS, A. **A survey of software infrastructures and frameworks for ubiquitous computing**. *Mobile Information Systems*, 1(1):41–80, 2005.

[40] FAVRE, J.-M. **Towards a basic theory to model model driven engineering**. In: *3rd Workshop in Software Model Engineering, WiSME*, p. 262–271. Citeseer, 2004.

[41] FEENEY, M.; FRISBY, R. **Autonomic management of smart spaces**. 2006.

[42] FERREIRA FILHO, J. B. **Leveraging model-based product lines for systems engineering**. PhD thesis, Université Rennes 1, 2014.

[43] FLOCH, J.; HALLSTEINSEN, S.; STAV, E.; ELIASSEN, F.; LUND, K.; GJORVEN, E. **Using architecture models for runtime adaptability**. *IEEE software*, 23(2):62–70, 2006.

[44] FORTINO, G.; GUERRIERI, A.; RUSSO, W.; SAVAGLIO, C. **Middlewares for smart objects and smart environments: Overview and comparison**. In: *Internet of Things Based on Smart Objects*, p. 1–27. Springer, 2014.

[45] FOUNDATION, T. E. **Epsilon Validation Language (EVL)**. https://www.eclipse.org/epsilon/doc/evl/, 2015. [Online; accessed 03-April-2015].

[46] FOWLER, M. **Domain-specific languages**. Pearson Education, 2010.

[47] FRANCE, R.; RUMPE, B. **Model-driven Development of Complex Software: A Research Roadmap**. In: *2007 Future of Software Engineering*, FOSE '07, p. 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[48] FRITZSCHE, M.; JOHANNES, J. **Putting performance engineering into model-driven engineering: Model-driven performance engineering**. In: *International Conference on Model Driven Engineering Languages and Systems*, p. 164–175. Springer, 2007.

[49] GAMMA, E. **Design patterns: elements of reusable object-oriented software**. Pearson Education India, 1995.

[50] GARCÍA-HERRANZ, M.; HAYA, P. A.; ALAMÁN, X. **Towards a ubiquitous end-user programming system for smart spaces.** *J. UCS*, 16(12):1633–1649, 2010.

[51] GARGANTINI, A.; RICCOBENE, E.; SCANDURRA, P. **A semantic framework for metamodel-based languages**. *Automated software engineering*, 16(3-4):415–454, 2009.

[52] GILMAN, E.; RIEKKI, J. **Is there meta-level in smart spaces?** In: *PerCom Workshops*, p. 88–93. IEEE, 2012.

[53] GOUIN-VALLERAND, C.; ABDULRAZAK, B.; GIROUX, S.; MOKHTARI, M. **A self-configuration middleware for smart spaces**. *International Journal of Smart Home*, 3(1), 2009.

[54] GROUP, O. M. **Object Constraint Language (OCL)**. http://www.omg.org/spec/OCL/, 2015. [Online; accessed 03-April-2015].

[55] GUERRA, C. A. N. **Um modelo conceitual para ambientes inteligentes baseado em interações formais em espaços físicos**. PhD thesis, Universidade de São Paulo, 2012.

[56] GUINARD, D.; TRIFA, V. **Towards the web of things: Web mashups for embedded devices**. In: *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, volume 15, 2009.

[57] GUINARD, D.; TRIFA, V.; MATTERN, F.; WILDE, E. **From the internet of things to the web of things: Resource-oriented architecture and best practices**. In: *Architecting the Internet of things*, p. 97–129. Springer, 2011.

[58] GUINARD, D.; TRIFA, V.; PHAM, T.; LIECHTI, O. **Towards physical mashups in the web of things**. In: *Networked Sensing Systems (INSS), 2009 Sixth International Conference on*, p. 1–4. IEEE, 2009.

[59] HAN, D.-M.; LIM, J.-H. **Design and implementation of smart home energy management systems based on zigbee**. *IEEE Transactions on Consumer Electronics*, 56(3), 2010.

[60] HAUCK, F.; BECKER, U.; MEIER, E.; RASTOFER, U.; STECKERMEIER, M.; ASPECT-ORIENTED, A. A.; HAUCK, F. J.; BECKER, U.; GEIER, M.; MEIER, E.; OTHERS. **Aspectix-an aspect-oriented and corba-compliant orb architecture**. 1998.

[61] HAYTON, R.; HERBERT, A.; DONALDSON, D. **Flexinet: a flexible component oriented middleware system**. In: *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, p. 17–24. ACM, 1998.

[62] HELAL, S. **Programming pervasive spaces**. *IEEE Pervasive Computing*, 4(1):84–87, 2005.

[63] HELAL, S.; MANN, W.; EL-ZABADANI, H.; KING, J.; KADDOURA, Y.; JANSEN, E. **The Gator Tech Smart House: A Programmable Pervasive Space**. *Computer*, 38(3):50–60, 2005.

[64] HELAL, S.; TARKOMA, S. **Smart spaces [guest editors' introduction]**. *IEEE Pervasive Computing*, 14(2):22–23, 2015.

[65] HORN, P. **Autonomic computing: Ibm\'s perspective on the state of information technology**. 2001.

[66] INC., T. O. C. **CoreDX DDS Data Distribution Service Middleware Twin Oaks Computing**. http://www.twinoakscomputing.com/coredx/, 2016. [Online; accessed 01-November-2016].

[67] JOHNSON, D. B.; MALTZ, D. **Mobile computing**. 1996.

[68] KELLY, S.; TOLVANEN, J.-P. **Domain-specific modeling: enabling full code generation**. Wiley.com, 2008.

[69] KENT, S. **Model-Driven Engineering**. In: *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, p. 286–298, London, UK, UK, 2002. Springer-Verlag.

[70] KEPHART, J. O.; CHESS, D. M. **The vision of autonomic computing**. *Computer*, 36(1):41–50, 2003.

[71] KINDBERG, T.; FOX, A. **System software for ubiquitous computing**. *IEEE pervasive computing*, 1(1):70–81, 2002.

[72] KOENIG, D.; GLOVER, A.; KÖNIG, D. **Groovy in action**, volume 1. Manning, 2007.

[73] KOLOVOS, D.; ROSE, L.; PAIGE, R.; GARCIA-DOMINGUEZ, A. **The epsilon book**. *Structure*, 178:1–10, 2010.

[74] KOLOVOS, D. S.; PAIGE, R. F.; KELLY, T.; POLACK, F. A. **Requirements for domain-specific languages**. In: *Proc. of ECOOP Workshop on Domain-Specific Program Development (DSPD)*, volume 2006, 2006.

[75] KOLOVOS, D. S.; ROSE, L. M.; ABID, S. B.; PAIGE, R. F.; POLACK, F. A.; BOTTERWECK, G. **Taming emf and gmf using model transformation**. In: *International Conference on Model Driven Engineering Languages and Systems*, p. 211–225. Springer, 2010.

[76] KON, F.; COSTA, F.; BLAIR, G.; CAMPBELL, R. H. **The case for reflective middleware**. *Communications of the ACM*, 45(6):33–38, 2002.

[77] KOPETZ, H. **Internet of things**. In: *Real-time systems*, p. 307–323. Springer, 2011.

[78] KORZUN, D. G.; BALANDIN, S. I.; GURTOV, A. V. **Deployment of smart spaces in internet of things: Overview of the design challenges**. In: *Internet of Things, Smart Spaces, and Next Generation Networking*, p. 48–59. Springer, 2013.

[79] KUBITZA, T.; SCHMIDT, A. **Towards a toolkit for the rapid creation of smart environments**. In: *International Symposium on End User Development*, p. 230–235. Springer, 2015.

[80] KUMAR, S. **Challenges for ubiquitous computing**. In: *Networking and Services, 2009. ICNS'09. Fifth International Conference on*, p. 526–535. IEEE, 2009.

[81] LASSERRE, P.; KAN, D. **User-centric interactions beyond communications**. *Alcatel telecommunications review*, (1):67–72, 2005.

[82] LEDOUX, T. **Opencorba: A reflective open broker**. In: *International Conference on Metalevel Architectures and Reflection*, p. 197–214. Springer, 1999.

[83] LEHMANN, G.; BLUMENDORF, M.; TROLLMANN, F.; ALBAYRAK, S. **Meta-modeling runtime models**. In: *International Conference on Model Driven Engineering Languages and Systems*, p. 209–223. Springer, 2010.

[84] LOPES, F. A. D. S. **Uma plataforma de integração de middleware para computação ubíqua**. 2011.

[85] LUPIANA, D.; O'DRISCOLL, C.; MTENZI, F. **Defining smart space in the context of ubiquitous computing**. *Ubiquitous Computing and Communication Journal*, 4(3):516–524, 2009.

[86] LUPIANA, D.; O'DRISCOLL, C.; MTENZI, F. **Taxonomy for ubiquitous computing environments**. In: *2009 First International Conference on Networked Digital Technologies*, p. 469–475. IEEE, 2009.

[87] MAES, P. **Concepts and experiments in computational reflection**. In: *ACM Sigplan Notices*, volume 22, p. 147–155. ACM, 1987.

[88] MAES, P. **Concepts and experiments in computational reflection**. In: *ACM Sigplan Notices*, volume 22, p. 147–155. ACM, 1987.

[89] MALECHA, G.; CHLIPALA, A.; BRAIBANT, T. **Compositional computational reflection**. In: *International Conference on Interactive Theorem Proving*, p. 374–389. Springer, 2014.

[90] MERNIK, M.; HEERING, J.; SLOANE, A. M. **When and how to develop domain-specific languages**. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[91] OPENHAB. **openHAB - Empowering the Smart Home**. http://www.openhab.org/, 2015. [Online; accessed 09-August-2015].

[92] O'SULLIVAN, D.; WADE, V. **A smart space management framework**. *Computer Science Technical Report, TCD-CS-2002-23, Trinity College Dublin*, 2002.

[93] OVERBEEK, J. **Meta Object Facility (MOF): investigation of the state of the art**. 2006.

[94] OVERBEEK, J. **Meta object facility (mof): investigation of the state of the art**. Master's thesis, University of Twente, 2011.

[95] PAPAMARKOS, G.; POULOVASSILIS, A.; WOOD, P. T. **Event-condition-action rule languages for the semantic web**. In: *Proceedings of the First International Conference on Semantic Web and Databases*, p. 294–312. CEUR-WS. org, 2003.

[96] PENHAKER, M.; ČERNỲ, M.; MARTINAK, L.; SPIŠÁK, J.; VALKOVA, A. **Homecare-smart embedded biotelemetry system**. In: *World Congress on Medical Physics and Biomedical Engineering 2006*, p. 711–714. Springer, 2007.

[97] PIRES, P. F.; CAVALCANTE, E.; BARROS, T.; DELICATO, F. C.; BATISTA, T.; COSTA, B. **A platform for integrating physical devices in the internet of things**. In: *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, p. 234–241. IEEE, 2014.

[98] PITT, E.; MCNIFF, K. **Java RMI: The Remote Method Invocation Guide**. Addison-Wesley Longman Publishing Co., Inc., 2001.

[99] PLOTKIN, G. D. **A structural approach to operational semantics**. 1981.

[100] POSLAD, S. **Ubiquitous computing: smart devices, environments and interactions**. John Wiley & Sons, 2011.

[101] POSLAD, S. **Ubiquitous computing: smart devices, environments and interactions**. John Wiley & Sons, 2011.

[102] PREKOP, P.; BURNETT, M. **Activities, context and ubiquitous computing**. *Comput. Commun.*, 26(11):1168–1176, July 2003.

[103] QIN, W.; SHI, Y.; SUO, Y. **Ontology-based context-aware middleware for smart spaces**. *Tsinghua Science & Technology*, 12(6):707–713, 2007.

[104] RAJKUMAR, R. R.; LEE, I.; SHA, L.; STANKOVIC, J. **Cyber-physical systems: the next computing revolution**. In: *Proceedings of the 47th Design Automation Conference*, p. 731–736. ACM, 2010.

[105] RAYCHOUDHURY, V.; CAO, J.; KUMAR, M.; ZHANG, D. **Middleware for pervasive computing: A survey**. *Pervasive Mob. Comput.*, 9(2):177–200, Apr. 2013.

[106] ROALTER, L.; KRANZ, M.; MÖLLER, A. **A middleware for intelligent environments and the internet of things**. In: *International Conference on Ubiquitous Intelligence and Computing*, p. 267–281. Springer, 2010.

[107] ROMÁN, M.; HESS, C.; CERQUEIRA, R.; RANGANATHAN, A.; CAMPBELL, R. H.; NAHRSTEDT, K. **A middleware infrastructure for active spaces**. *Pervasive Computing, IEEE*, 1(4):74–83, 2002.

[108] RORIZ, M.; MASSARANI, L.; FREITAS, L. A.; COUTO ANTUNES DA ROCHA, R.; MOREIRA COSTA, F. **C3S: A content sharing middleware for smart spaces**. In: *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, p. 163–168. IEEE, 2013.

[109] SÁNCHEZ, P.; MOREIRA, A.; FUENTES, L.; ARAÚJO, J.; MAGNO, J. **Model-driven development for early aspects**. *Information and Software Technology*, 52(3):249–273, 2010.

[110] SATYANARAYANAN, M. **Pervasive computing: Vision and challenges**. *IEEE Personal communications*, 8(4):10–17, 2001.

[111] SCHMIDT, A. **Programming ubiquitous computing environments**. In: *International Symposium on End User Development*, p. 3–6. Springer, 2015.

[112] SCHMIDT, D. C. **Model-Driven Engineering**. *IEEE Computer*, 39(2), February 2006.

[113] SCHMIDT, D. C.; OTHERS. **Tao, the ace orb**, 2007.

[114] SEIDEWITZ, E. **What models mean**. *Software, IEEE*, 20(5):26–32, 2003.

[115] SINGH, R.; BHARGAVA, P.; KAIN, S. **State of the art smart spaces: application models and software infrastructure**. *Ubiquity*, 2006(September):7, 2006.

[116] SMIRNOV, A.; KASHEVNIK, A.; SHILOV, N.; TESLYA, N. **Context-based access control model for smart space**. In: *Cyber Conflict (CyCon), 2013 5th International Conference on*, p. 1–15. IEEE, 2013.

[117] SOLDATOS, J.; DIMAKIS, N.; STAMATIS, K.; POLYMENAKOS, L. **A breadboard architecture for pervasive context-aware services in smart spaces: middleware components and prototype applications**. *Personal Ubiquitous Comput.*, 11(3):193–212, Feb. 2007.

[118] SOLEY, R.; OTHERS. **Model driven architecture**. *OMG white paper*, 308(308):5, 2000.

[119] SPRINKLE, J.; MERNIK, M.; TOLVANEN, J.-P.; SPINELLIS, D. **Guest editors' introduction: What kinds of nails need a domain-specific hammer?** *Software, IEEE*, 26(4):15–18, 2009.

[120] STAHL, T.; VOELTER, M.; CZARNECKI, K. **Model-Driven Software Development: Technology, Engineering, Management**. John Wiley & Sons, 2006.

[121] STEINBERG, D.; BUDINSKY, F.; MERKS, E.; PATERNOSTRO, M. **EMF: Eclipse Modeling Framework**. Pearson Education, 2008.

[122] STOJANOVIC, D. **Context-aware mobile and ubiquitous computing for enhanced usability: adaptive technologies and applications**. Information Science Reference-Imprint of: IGI Publishing, 2009.

[123] STOJANOVIC, D. **Context-Aware Mobile and Ubiquitous Computing for Enhanced Usability: Adaptive Technologies and Applications: Adaptive Technologies and Applications**. IGI Global, 2009.

[124] STRÖMBERG, H.; PIRTTILÄ, V.; IKONEN, V. **Interactive scenarios?building ubiquitous computing concepts in the spirit of participatory design**. *Personal and Ubiquitous Computing*, 8(3-4):200–207, 2004.

[125] THOMAS, MARKUS VOLTER, J. B.; HELSEN, S. **Model-Driven Software Development**. Wiley, 2005.

[126] VAN DER MEER, S.; O'CONNOR, R.; DAVY, A. **Ubiquitous Smart Space Management**. In: *1st International Workshop on Management of Ubiquitous Communications and Services (MUCS), Waterford, Ireland*, 2003.

[127] VAN DEURSEN, A.; KLINT, P. **Little languages: Little maintenance?** *Journal of software maintenance*, 10(2):75–92, 1998.

[128] VAN DEURSEN, A.; KLINT, P. **Domain-specific language design requires feature descriptions**. *CIT. Journal of computing and information technology*, 10(1):1–17, 2002.

[129] VAN DEURSEN, A.; KLINT, P.; VISSER, J. **Domain-Specific Languages: An Annotated Bibliography**. *Sigplan Notices*, 35(6):26–36, 2000.

[130] VAN DEURSEN, A.; KLINT, P.; VISSER, J.; OTHERS. **Domain-specific languages: An annotated bibliography.** *Sigplan Notices*, 35(6):26–36, 2000.

[131] WANG, X.; DONG, J. S.; CHIN, C.; HETTIARACHCHI, S. R.; ZHANG, D. **Semantic space: An infrastructure for smart spaces**. *Computing*, 1(2):67–74, 2002.

[132] WEBER, R. H.; WEBER, R. **Internet of things**, volume 12. Springer, 2010.

[133] WEISER, M. **The computer for the 21st century**. *Scientific American*, 265(3):66–75, Sept. 1991.

[134] WEISER, M. **Some computer science issues in ubiquitous computing**. *Communications of the ACM*, 36(7):75–84, 1993.

[135] WEISER, M. **Some computer science issues in ubiquitous computing**. *Communications of the ACM*, 36(7):75–84, 1993.

[136] WHITTLE, J.; CLARK, T.; KÜHNE, T. **Model driven engineering languages and systems**. In: *14th International Conference, MODELS*, p. 16–21. Springer, 2011.

[137] WU, Y. **A domain specific modeling approach for coordinating user-centric communication services**. PhD thesis, Florida International University, 2011.

[138] XIA, F.; YANG, L. T.; WANG, L.; VINEL, A. **Internet of things**. *International Journal of Communication Systems*, 25(9):1101, 2012.