ANDRÉ LUIS MENESES SILVA

FRAMEWORK FORMAL PARA COMPOSIÇÃO
AUTOMÁTICA DE SERVIÇOS EM SISTEMAS DE
INTERNET DAS COISAS

São Paulo
2018

ANDRÉ LUIS MENESES SILVA

**FRAMEWORK FORMAL PARA COMPOSIÇÃO
AUTOMÁTICA DE SERVIÇOS EM SISTEMAS DE
INTERNET DAS COISAS**

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para obtenção
do Título de Doutor em Ciências.

São Paulo
2018

ANDRÉ LUIS MENESES SILVA

**FRAMEWORK FORMAL PARA COMPOSIÇÃO
AUTOMÁTICA DE SERVIÇOS EM SISTEMAS DE
INTERNET DAS COISAS**

Tese apresentada à Escola Politécnica da
Universidade de São Paulo para obtenção
do Título de Doutor em Ciências.

Área de Concentração:

Sistemas Eletrônicos

Orientador:

Sergio Takeo Kofuji

São Paulo
2018

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 02 de maio de 2018

Assinatura do autor: _____

Assinatura do orientador: _____

Catálogo-na-publicação

Silva, André Luis Meneses

Framework Formal para Composição Automática de Serviços em Sistemas de Internet das Coisas / A. L. M. Silva -- versão corr. -- São Paulo, 2018.

163 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1. Internet das Coisas 2. Sistemas Inteligentes 3. Especificação Formal 4. Engenharia de Conhecimento I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos II. t.

Dedicatória

Aos meus pais José e Vilma

A minha tia Lili

Ao meu irmão Eric

AGRADECIMENTOS

Em primeiro lugar, gostaria agradecer ao meu orientador, o Prof. Dr. Sergio Takeo Kofuji, pela oportunidade de fazer parte deste excelente programa de pós-graduação, pelas valiosas lições e conhecimento compartilhado durante o período de minha orientação. Também gostaria de agradecer ao meu co-orientador, o Prof. Dr. José Jesus Pérez-Alcázar, pelos valiosos conselhos, tanto no âmbito profissional, acadêmico e pessoal, pela extrema paciência e zelo em minha co-orientação. Gostaria de agradecer também a toda a equipe do Grupo de Sistemas Pervasivos e de Alto Desempenho (PAD), pela ajuda direta e indireta na conclusão desta tese, em especial aos seguintes membros: João Neto, José Castillo, Roberto Kenji Hiramatsu, Stelvio Barbosa, Juan Carlos Zuñiga. Um agradecimento especial aos parceiros da EACH, ao Prof. Dr. Fábio Nakano e aos alunos que tive oportunidade de colaborar nessa unidade: Cinthia, Victor, Felipe e Thyago. Um agradecimento especial aos meus pais José da Silva e Vilma de Meneses Silva por todos os conselhos e ensinamentos. Ao meu irmão Eric e minha tia Maria Gildete, pelo ânimo dado nos momentos difíceis. A minha tia Gardênia, Luis e Enia, pela motivação, apoio e ótima recepção nos retornos a minha cidade, Aracaju. Agradecimento especial ao meu núcleo familiar de São Paulo: tia Maria, tio João, tia Margarida, tio Paulo (*em memória*) e aos meus primos e amigos: Thiago, Cristiane, Lucitelma, Jackson, Ilma, Joelma, Ricardo, Neia, Paola, Lourdinha e família, por proporcionarem tantos bons momentos durante minha estadia aí na cidade. Também agradeço a Cinthia e família pelo apoio, conversas, inspiração e bons momentos proporcionados, ora vendo *Netflix*, ora no PS4, ora compartilhando da culinária japonesa. Um agradecimento especial ao meu grande amigo e irmão James Diego e família, pelas boas conversas e pelo nosso bom e velho café das 16:00 horas de domingo.

Muito obrigado a todos vocês. Cada um, a sua maneira, deu uma contribuição significativa para conclusão deste trabalho.

*"Cada sonho que você deixa para trás,
é um futuro que deixa de existir"*

-Steve Jobs-

RESUMO

É cada vez mais notável o desenvolvimento da indústria micro-eletrônica. A criação de dispositivos eletrônicos menores, que apresentam maior autonomia de energia, aliados ao aumento do poder de processamento, armazenamento e comunicação sem fio de alta velocidade favoreceram o surgimento e disseminação de novas tecnologias e paradigmas, dentre elas a Internet das Coisas (IoT). Do ponto de vista tecnológico, IoT é uma rede de objetos físicos que possuem tecnologia embarcada de sensoriamento e atuação. Agências de consultoria empresarial, tais como a McKinsey & Company, afirmam que IoT apresenta valor de mercado bilionário e poderá ultrapassar a casa dos trilhões antes de 2020. Dessa forma, o mercado de IoT vem se apresentando como um dos mercados mais promissores para os próximos anos. Alguns dos problemas que podem postergar este crescimento são os problemas decorrentes da dificuldade de integração e escalabilidade das aplicações de IoT. Em IoT, problemas de interoperabilidade são corriqueiros, seja pela alta diversidade de dispositivos empregados, seja pela incompatibilidade entre fabricantes. Em relação a escalabilidade, sistemas de IoT possuem uma demanda natural por alta escala, visto que buscam atender demandas comuns a vários setores, seja na indústria, transporte, doméstica, segurança pública, comércio, entre outros. Este trabalho apresenta uma solução para esses problemas através do SWoTPAD, um *framework* formal que auxilia o projetista no desenvolvimento de soluções para IoT. SWoTPAD oferece uma linguagem para especificar dispositivos e serviços, descrever o ambiente e realizar requisições. Adicionalmente, ele gera o módulo de descoberta, composição automática de serviços e execução. Aplicações SWoTPAD são facilmente integráveis, pois usam e estendem um mesmo conjunto de ontologias, o que garante a compatibilidade nos dados gerados e consumidos por essas aplicações. A escalabilidade advém da associação de anotações semânticas a cada um dos elementos que compõem a aplicação de IoT. Essas anotações permitem ao SWoTPAD descobrir, classificar, selecionar e compor automaticamente serviços do ambiente. Dessa forma, SWoTPAD pode procurar por soluções alternativas, quando o serviço original apto a atender uma determinada demanda se encontra sobrecarregado ou indisponível. Para validação do *framework*, foram adotados dois estudos de caso. O primeiro deles, o problema de implantação de serviços em um ambiente de nuvem, e o segundo, uma aplicação de segurança residencial. O estudo de caso demonstrou que é possível desenvolver aplicações completas de IoT no *framework* proposto. Adicionalmente, o mecanismo de composição automática gerado pelo *framework* para essas aplicações apresenta uma piora média de 45% de desempenho quando comparado à composição manual.

Palavras-Chave – Internet das Coisas, Sistemas Inteligentes, Especificações Formais, Engenharia do Conhecimento

ABSTRACT

The development of the micro-electronics industry is becoming more and more remarkable. The creation of smaller electronic devices, with higher degree of autonomy, processing, storage, and wireless communication favor the emergence and dissemination of new technologies and paradigms, such as the Internet of Things (IoT). From the technological point of view, IoT is a network of physical objects that have embedded technology of sensing and actuation. McKinsey & Company says the IoT market is already reaching billionaire numbers and may exceed the trillions by 2020. Thus, the IoT market is proving to be one of the most promising markets in the next years. Problems that can delay this growth come from the difficulty of integration and scalability of IoT applications. In IoT, interoperability problems are common, either because of the high diversity of devices used, or because of the incompatibility between manufacturers. Regarding scalability, IoT systems have a natural demand for high scale, since they seek to meet common demands in various sectors, be it in industry, transportation, home automation, public safety, commerce, among others. This work solves these problems through SWoTPAD, a formal framework that assists the designer in developing solutions for IoT. SWoTPAD provides a language for specifying devices and services, describing the environment, and performing requests. Additionally, it generates the discovery, automatic service composition, and execution module. SWoTPAD applications are easily integrable, since they use and extend the same set of ontologies, which guarantees compatibility in the data generated and consumed by these applications. Scalability comes from the association of semantic annotations to each of the elements that compose the IoT application. These annotations allow SWoTPAD to discover, rank, select, and automatically compose services. In this way, SWoTPAD can search for alternative solutions, when the original service able to meet a particular demand is overloaded or unavailable. Two case studies were developed for validation of the framework. The first one, the problem of deploying services in a cloud environment, and the second, a home security system. The case study demonstrated that it is possible to develop complete IoT applications in the proposed framework. Also, the automatic service composition module generated by SWoTPAD for these applications has a mean worsening of 45 % of performance when compared to the manual composition.

Keywords – Internet of Things, Intelligent Systems, Formal Specifications, Knowledge Engineering

LISTA DE FIGURAS

1	Arquitetura Genérica de um Sistema para Composição Automática de Ser- viços.	46
2	Sistema de Planejamento	48
3	Exemplo de Decomposição de Tarefas.	53
4	Decomposição de tarefas em planejamento hierárquico	55
5	Esquema de Classe CartaoDeCredito	57
6	Esquema de Classe CartaoEspecial	59
7	Esquema de Classe CartoesDeCredito	60
8	Modelo Conceitual	63
9	Modelo de classe Object-Z usada para formalização do Modelo Conceitual .	65
10	Esquema de Classe EntidadeSWoTPAD	65
11	Esquema de Classe Ambiente	67
12	Esquema de Classe Servico	69
13	Esquema de Classe Ator	71
14	Esquema de Classe Instrumento	72
15	Esquema de Classe Requisicao	73
16	Esquema de Classe Propriedade	73
17	Esquema de Classe Habilidade	74
18	Esquemas da Hierarquia de Classes Expressao	76
19	Esquema de Classe Capacidade	77
20	Classe SubCapacidade	78
21	Esquema de Classe Comportamento	78
22	Esquema de Classe Comando	79
23	Esquema de Classe EstadoIoT	81

24	Esquema de Classe ProblemaPlanejamentoIoT	83
25	Camadas do Framework SWoTPAD	86
26	Detalhamento da Camada Semântica	88
27	Exemplo de Ambiente - Sala de Impressão	97
28	Solução gerada pela ferramenta Gerador do Sistema	104
29	Interação entre Ator do Sistema de IoT e Solução SWoTPADL	105
30	Exemplo de taxonomia de veículos	108
31	Mapeamento dos Conceitos de HTN para os Conceitos do Modelo Conceitual	110
32	Possíveis decomposições de <i>Deslocar</i> e paralelização	112
33	Decomposição com não-determinismo	115
34	SWoTPAD Designer Menu Adicionar	117
35	SWoTPAD Designer Menu Gerar	117
36	SWoTPAD Designer Menu Executar	118
37	Aplicação de segurança residencial	121
38	Plano obtido para AlertMode	127
39	Máquina de Estados de aplicações em um ambiente de nuvem	129
40	Descrição básica do modelo de serviços e pacotes.	129
41	Plano obtido para execução do serviço Wodpress.	134
42	Avaliação de desempenho dos estudos de caso	137

LISTA DE TABELAS

1	Desempenho do Sistema de Segurança Residencial	135
2	Sistema de Segurança Residencial - Cenário de Estresse	135
3	Tempo de implantação dos serviços no estudo de caso de computação em nuvem	136

LISTAGENS DE CÓDIGO

1	Exemplo de definição de ontologia WSML	39
2	Exemplo de definição de um <i>webService</i> WSML	40
3	Exemplo de <i>choreography</i> WSML	41
4	Exemplo de definição de <i>goal</i> WSML	42
5	Sintaxe dos construtores <i>webService</i> , <i>goal</i> e <i>instance</i> em WSML	89
6	Sintaxe de Definição de Serviços em SWoTPADL	92
7	Exemplo de definição de Serviço	94
8	Serviço RentACar em WSML (a esquerda) e em SWoTPADL (a direita). .	95
9	Descrição do <i>Ambiente</i> Sala de Impressão em JSON	97
10	Ontologia do <i>service Search</i> , presente na Listagem de Código 7	99
11	Ontologia da descrição do <i>Ambiente SalaDeImpressao</i>	100
12	Código relativo ao <i>service SearchPrice</i>	101
13	Sintaxe de uma <i>request</i> SWoTPADL.	102
14	SWoTPADL Request	103
15	Algoritmo grauDeAptidao	106
16	Algoritmo de definição de graus de casamento de saídas. Adaptado de (PAOLUCCI et al., 2002)	107
17	Algoritmo avaliaPrecondicao	108
18	Algoritmo avaliaCapacidade	109
19	O problema de deslocamento: Capacidade e as Habilidades que a realizam	111
20	O algoritmo de planejamento MTHierarquicoPlan. Adaptado de (RUS- SELL, 2010)	113
21	Versão multithreading do algoritmo ND-Shop2. Adaptado de (KUTER, 2006)	114
22	ChamarUber não determinístico	115
23	Descrição do ambiente em JSON	122
24	O Serviço de detecção de intrusos	122
25	O serviço TurnOnLightService	123
26	O serviço PlaySirenService	123
27	O serviço TakeAPhotoService	123
28	O serviço ContactPoliceStationService	124

29	O serviço SecurityGoal	124
30	Taxonomia dos Serviços de acordo com as habilidades	125
31	Serviço Instalar	130
32	Serviço Executar	130
33	Serviço InstDepend	131
34	Serviço ExecEInst	131
35	Taxonomia dos Serviços para Cloud de acordo com as habilidades	132

LISTA DE SIGLAS

3G	third generation of wireless mobile telecommunications
4G	fourth generation of wireless mobile telecommunications
API	Application Programming Interface
CBF	Constrained Bellman-Ford
CSP	Constrained Shortest Path
CTL	Computation Tree Logic
CTL*	Computation Tree Logic Star
CSS	Calculus of Communicating Systems
DSL	Domain-Specific Language
GSM	Global System for Mobile Communication
HTN	Hierarchical Task Network
IoT	Internet of Things
IoE	Internet of Everything
IPv6	Internet Protocol Version 6
LOD	Linked Open Data
LOV	Linked Open Vocabularies
MCKP	Multiple-Choice Knapsack Problem
MDA	Model Driven Architecture
OBDDs	Ordered Binary Decision Diagrams
PDDL	Planning Domain Description Language
PDP	Problema de Definição de Planos
PLTL	Propositional Linear Temporal Logic
QoS	Quality of Service (Qualidade de Serviço)
REST	Representational State Transfer
RFID	Radio Frequency Identification
SHOP2	Simple Hierarchical Ordered Planner 2
SOA	Service-Oriented Architecture
SW	Serviços Web
SWoT	Semantic Web of things
TIC	Sistemas de Tecnologia da Informação e Comunicação
UMTS	Universal Mobile Telecommunication System
URI	Uniform Resource Identifier
WiFi	Wireless Fidelity
WoT	Web of things
WSML	Web Service Modeling Language
WSMO	Web Service Modeling Ontology
WSN	Wireless Sensor Network

SUMÁRIO

1	Introdução	17
1.1	Hipótese	18
1.2	Justificativa e Motivação	20
1.3	Objetivos	22
1.3.1	Objetivos específicos	22
1.4	Resultados Obtidos	22
1.5	Escopo e Limitações	24
1.6	Metodologia	25
1.6.1	Fundamentação e aquisição bibliográfica	25
1.6.2	Criação de linguagem para descrição de serviços e requisições do usuário.	25
1.6.3	Formalização da linguagem desenvolvida	25
1.6.4	Criação de modelo conceitual para Aplicações de IoT	26
1.6.5	Estratégia para Composição Automática de Serviços	26
1.6.6	Geração do Protótipo e Validação	27
1.6.7	Elaboração de Artigos Científicos	27
1.7	Organização da Tese	28
2	Referencial Teórico	30
2.1	Internet das Coisas (IoT)	30
2.2	Web Semântica das Coisas	33
2.2.1	Web Service Modeling Language	36
2.2.1.1	Ontologias	38
2.2.1.2	Serviços	39

2.2.1.3	Definição de Metas	42
2.3	Descoberta e Composição de Serviços	43
2.4	Planejamento	46
2.4.1	Técnicas de Planejamento	47
2.4.2	Planejamento Clássico	48
2.4.3	Planejamento Hierárquico	52
2.5	Modelagem Formal	55
2.5.1	Classes, esquemas e tipos	56
2.5.2	Objetos e Herança	58
2.5.3	Agregação de Objetos	59
2.6	Conclusões	60
3	Formalização do Problema	62
3.1	Modelo Conceitual para Sistemas de IoT	62
3.2	Especificação Formal em Object-Z	64
3.3	Conclusões	84
4	O Framework SwotPAD	85
4.1	Visão Geral	85
4.2	A Camada Semântica	87
4.2.1	Descrição dos Serviços	89
4.2.1.1	Definição de Serviços em SWoTPADL	92
4.2.2	Descrição dos Ambientes	96
4.2.3	Mapeamento e Tradução	98
4.2.4	Requisição e Resposta	101
4.2.5	Gerador do Sistema	103
4.3	A Camada Funcional	104
4.3.1	Provedor do Serviço	105

4.3.2	Compositor de Serviço	109
4.3.3	Invocador de Serviços	116
4.4	A Ferramenta SWoTPAD Designer	116
4.5	Conclusões	118
5	Estudos de Caso	120
5.1	Estudo de Caso 1: Aplicação de Segurança Residencial	120
5.1.1	Artefatos de Entrada	121
5.1.2	Planejamento e Composição Automática	125
5.2	Estudo de Caso 2 - Serviços na Nuvem	128
5.2.1	Artefatos de Entrada	130
5.2.2	Planejamento e Composição Automática	132
5.3	Resultados e Discussões	134
5.4	Conclusões	137
6	Trabalhos Relacionados	139
6.1	Composição Automática de Serviços	139
6.2	<i>Frameworks</i> para Internet das Coisas	143
6.3	Representações e formalismos	145
6.4	Conclusões	148
7	Conclusões	149
7.1	Limitações	150
7.2	Trabalhos Futuros	151
	Referências	152

1 INTRODUÇÃO

O desenvolvimento da indústria eletrônica tem proporcionado a criação de dispositivos eletrônicos cada vez menores, que apresentam maior autonomia de energia (YU et al., 2013). Essas características, aliadas ao aumento do poder de processamento, armazenamento e ao desenvolvimento de novas tecnologias de comunicação sem fio favoreceram o surgimento e disseminação da computação pervasiva. Nesse contexto, está cada vez mais próxima a concretização da visão de Mark Weiser, na qual *"as tecnologias mais profundas são aquelas que desaparecem. Elas se entrelaçam no tecido da vida cotidiana até que se tornem indistinguíveis"* (WEISER, 1999).

Computação pervasiva é a área responsável pelo desenvolvimento de sistemas de tecnologia de informação e comunicação que disponibilizam serviços e informações em qualquer lugar. Esses sistemas dão suporte ao uso intuitivo humano, de forma transparente (POS-LAD, 2011). Um típico sistema pervasivo é constituído por atuadores, sensores, *displays* e dispositivos computacionais. Esses elementos são embutidos em objetos do cotidiano, que se encontram interconectados continuamente por meio de redes. Um ambiente dotado de um ou mais desses sistemas é denominado ambiente pervasivo (YU et al., 2013).

Apesar dos avanços obtidos na área, o projeto desses sistemas apresenta alguns desafios. Em domótica, por exemplo, analistas sabem pouco sobre como projetar, localizar e integrar essas tecnologias para dar suporte a decisões dos moradores (MAKONIN; BARTRAM; POPOWICH, 2013). Adicionalmente, uma grande variedade de produtos pode ser adotada. Produtos como relógios, óculos inteligentes, câmeras vestíveis e dispositivos de rastreamento geram uma grande quantidade de dados e apresentam novos desafios do ponto de vista de integração e processamento desses recursos.

Com intuito de amenizar esses problemas, percebe-se uma tendência para o desenvolvimento de ambientes compostos por objetos inteligentes acessíveis remotamente, denominada Internet das Coisas (ou simplesmente IoT, acrônimo de *Internet of Things*). IoT é definido como uma infraestrutura de rede global e dinâmica com capacidade de autoconfiguração, baseada em padrões e protocolos de comunicação interoperáveis. Nesse

contexto, elementos virtuais e físicos possuem identidade, atributos, são capazes de usar interfaces inteligentes e são aptos a ser integrados em uma rede de informação (LI; XU; ZHAO, 2015).

Apesar dessas características, projetistas de aplicações para IoT lidam com problemas de integração, devido a diversidade de dispositivos empregados no desenvolvimento dessas aplicações. No âmbito de dados, sensores de um mesmo domínio, produzidos por fabricantes diferentes, costumam apresentar vocabulário próprio em relação aos dados gerados (GYRARD et al., 2016b). Adicionalmente, aplicações de IoT sofrem com problemas de escalabilidade, por se tratar de sistemas complexos, de larga escala, que demandam comunicação dinâmica, envolvendo diferentes tecnologias (MIORANDI et al., 2012; MIORI; RUSSO, 2014). Do ponto de vista do usuário, esse amplo ecossistema de aplicações e serviços pode parecer confuso. A grande diversidade de opções tende a trazer problemas, tanto para localizar os serviços mais adequados às suas demandas, bem como na forma de utilizar esses serviços.

Uma solução para esses problemas é a adoção de semântica. A aplicação de semântica consiste em associar metadados aos elementos de IoT e adotar ontologias de domínio, como forma de possibilitar a localização, execução, uso automático e composição (FENSEL et al., 2011) do conjunto de serviços ofertados por um sistema de IoT. A composição possui caráter especial, pois dá suporte ao desenvolvimento de novos serviços a partir de serviços preexistentes (MEDJAHED; BOUGUETTAYA, 2011). A composição pode ser manual, quando há intervenção humana, ou automática, quando é criada por ferramentas ou aplicações sem o auxílio do desenvolvedor.

Este trabalho tem como objetivo apresentar um conjunto de soluções que buscam fomentar a interoperabilidade e escalabilidade em aplicações para IoT. Essas soluções são disponibilizadas por meio de um *framework* semântico. O *framework* possibilita a descoberta, composição dinâmica e automática, e invocação de serviços presentes em aplicações de IoT. O *framework* é genérico para esse domínio e apresenta pontos de extensão, que permite sua aplicação a diferentes cenários de IoT.

1.1 Hipótese

Esta tese adota a hipótese de que o desenvolvimento de descrições semânticas dos ambientes e entidades que formam uma aplicação de IoT é característica essencial para resolução do problema da interoperabilidade de dados entre aplicações de IoT. Adicional-

mente, propomos uma solução para a escalabilidade desses sistemas, a partir da composição automática e dinâmica de serviços. Este tipo de composição permite buscar soluções alternativas quando algum serviço encontra-se sobrecarregado ou indisponível.

No processo de concepção e desenvolvimento de sistemas, o conjunto de funcionalidades representam demandas do cliente, geralmente identificadas no ciclo de análise de requisitos. Após essa etapa, casos de utilização são criados e adotados como uma espécie de guia, retratando como os requisitos serão atendidos pelo sistema, através da interação dos atores com os componentes do sistema.

Apesar da evolução dos processos de desenvolvimento, relatórios da Standish Group mostram que de 2011 a 2015, em média, apenas 29% dos projetos de software apresentaram sucesso (HOVORUSHCHENKO; POMOROVA, 2016). Uma das causas da baixa qualidade é o crescimento do número de componentes (subsistemas) e das interfaces entre eles, além do aumento da complexidade dos sistemas (HOVORUSHCHENKO; POMOROVA, 2016). Adicionalmente, é difícil identificar propriedades futuras do sistema desenvolvido, assim, é essencial que os sistemas apresentem formas de evoluir com menor necessidade de intervenção humana. Nesse cenário, as ideias apontadas pela Web semântica podem ser adotadas.

A Web semântica é uma extensão da Web no sentido de associar um significado bem definido aos dados, com o intuito de permitir o trabalho cooperativo entre pessoas e máquinas (BERNERS-LEE; HENDLER; LASSILA, 2001). A interoperabilidade e a escalabilidade são obtidas pelo compartilhamento de dados, cuja interpretação não apresenta ambiguidade (BARNAGHI et al., 2012), dando suporte a modelagem do conhecimento de um domínio.

É fato que muito dos avanços da Web semântica se restringem a área acadêmica. A maioria dos sistemas semânticos são protótipos feitos para validar hipóteses (ABANDA; TAH; KEIVANI, 2013). Ainda assim, trabalhos nessa área tem obtido bons resultados (GLIMM; STUCKENSCHMIDT, 2016). Logo, podemos afirmar que com a adesão da comunidade de desenvolvedores, teremos aplicações de IoT interoperáveis e mais escaláveis.

No entanto, para estimular a adesão dessa comunidade, se faz necessário que a barreira de entrada existente em Web semântica seja amenizada. Web semântica adota linguagens e formalismos complexos, nem sempre bem compreendidos por parte dos desenvolvedores. Dessa forma, com intuito de amenizar essa barreira, este trabalho apresenta um *framework* como um de seus resultados. A distribuição de nossas soluções em um *framework* visa estimular desenvolvedores a adotar semântica em seus projetos.

O desenvolvimento de descrições semânticas para elementos de IoT facilitam a criação de projetos complexos de maior escala. Serviços como busca, composição e execução automática de serviços podem fazer boa utilização dessas descrições (FENSEL et al., 2011). Neste trabalho, avaliamos esta hipótese, mesclando técnicas de inferência em bases de conhecimento (RUSSELL, 2010) com algoritmos de dedução de planos (GHALLAB; NAU; TRAVERSO, 2004) para produzir composição automática de serviços.

Defendemos que o momento ideal para aplicação destas ideias seja em tempo de projeto, pois permite que descrições semânticas evoluam concomitantemente ao desenvolvimento do projeto. Por isso, o *framework* apresenta uma API (*Application Programming Interface*) e uma linguagem para especificação de serviços e requisições (SWoTPADL) que podem ser adotadas durante o processo de desenvolvimento de uma aplicação de IoT.

1.2 Justificativa e Motivação

De acordo com (SHETH, 2016), estima-se que em 2015, havia cerca de 5 a 9 bilhões de dispositivos conectados, com o mercado de IoT avaliado em 700 bilhões de dólares. Acredita-se no crescimento desses números para a ordem de 25 a 50 bilhões de dispositivos conectados, com valor de mercado IoT superando um trilhão de dólares antes do fim desta década.

Apesar do número elevado de pesquisadores e companhias atuando em IoT, há algumas questões em aberto. De acordo com o instituto Mckinsey Global (MANYIKA et al., 2015), problemas de interoperabilidade entre sistemas de IoT representam um impacto negativo na ordem de 40% do valor potencial de mercado de IoT. Em ambientes empresariais, onde se acentua a necessidade por integração e análise de dados, esse percentual pode chegar em 60%. Se os gastos com interoperabilidade forem superados, teremos um impacto econômico na área de IoT de cerca de 4 trilhões de dólares por ano (MANYIKA et al., 2015).

Apesar da ausência de dados recentes, a severidade do problema é corroborada pela atenção que a comunidade de IoT tem dado ao tema nos últimos anos (BRÖRING et al., 2017; FORTINO et al., 2018; AL; DOSS; CHOWDHURY, 2017; AHMED; RAHMAN; HUSSAIN, 2017; SERRANO et al., 2017), inclusive com o desenvolvimento de eventos e livros específicos ao tema (GRAVINA et al., 2018; GYRARD et al., 2018).

Uma solução para amenizar gastos com a força de trabalho extra, necessária para atuar tanto na integração como na expansão de sistemas de IoT, é a adoção de um voca-

bulário comum entre estes sistemas. Com esse vocabulário, é possível empregar técnicas sofisticadas conscientes de contexto, que podem inferir o serviço mais apto para execução, bem como integrar e compor serviços dinamicamente.

Uma forma de compartilhar esse vocabulário é fazer com que diferentes sistemas de IoT adotem uma mesma base de *software*. Por conta disso, este trabalho apresenta um conjunto de soluções, que promovem interoperabilidade e escalabilidade em aplicações para IoT, e as distribui como um *framework*, ou seja, um conjunto de módulos de *software* que podem ser adequados a depender do propósito da solução em desenvolvimento.

Adicionalmente, para garantir a cooperação entre dispositivos para IoT, bem como promover a capacidade de se adaptar a mudanças de contexto, é necessário que os elementos que compõe a solução de IoT sejam autônomos, autogeridos e apresentem comportamento auto adaptativo. É possível prover essa característica pela criação de uma descrição detalhada do domínio. Desta forma, elementos passam a ter a habilidade de adaptar seu comportamento a depender das características atuais do ambiente.

Como exemplo, podemos citar uma aplicação para gerenciamento de tráfego. O bom desempenho desse tipo de aplicação depende da habilidade dos controladores de semáforo em se adaptar às mudanças das situações de tráfego. Uma descrição precisa desse domínio, bem como das situações de tráfego nessa região, fornecem meios para que os controladores de semáforo apresentem desempenho próximo do ideal.

Além da descrição precisa do domínio, a composição automática de serviços pode auxiliar na tarefa de criação de um sistema auto-adaptativo. A composição automática possibilita a criação de novos serviços a partir da combinação de serviços preexistentes. Além de introduzir novas funcionalidades, apresenta alternativas a serviços falhos, pois permite substituí-los por serviços similares compostos. Um módulo robusto de composição automática possibilita o desenvolvimento de uma aplicação que monitore os estados dos serviços presentes no ambiente e, ao identificar uma falha, procure por um serviço similar composto. Logo, vemos que composição automática pode ser uma das abordagens adotadas na tentativa de obtenção de uma infraestrutura de serviços autônomos.

A tarefa de composição não deve ficar a cargo do usuário. Em um mundo onde a computação embarcada é cada vez mais presente, a tarefa de composição se torna demasiadamente complexa. Assim, o projetista de uma aplicação de IoT deve identificar as necessidades do usuário, permitir a comunicação dessas necessidades por meio de abstrações elevadas e, a partir daí, estabelecer estratégias para atender essas necessidades. Essa é a abordagem comumente usada pelos mecanismos de composição automática de

serviços.

1.3 Objetivos

Este trabalho tem como objetivo apresentar um conjunto de soluções que fomentem a interoperabilidade e escalabilidade de novas aplicações para IoT. Com o intuito de promover essas soluções na comunidade de desenvolvedores de aplicativos para IoT, optamos por disponibiliza-las por meio de um *framework* formal.

1.3.1 Objetivos específicos

A seguir, são apresentados os objetivos específicos deste trabalho:

1. Criação de uma nova linguagem para especificação de serviços e requisições apropriadas ao domínio de IoT. A intenção dessa linguagem, denominada SWoTPADL, é simplificar a especificação de serviços e requisições. Ela foi inspirada na linguagem WSML (BRUIJN et al., 2008), porém simplifica algumas de suas construções. Adicionalmente, apresenta características que não estão presente no WSML, tais como o suporte a hierarquia de serviços e associação de comportamentos ao serviço. Também, faz uso de um conjunto de ontologias que promovem a interoperabilidade a nível de dados.
2. Adaptação de algoritmos para descoberta e composição automática de serviços.
3. Desenvolvimento de um modelo conceitual que possa ser associado a aplicações de IoT de diferentes domínios.
4. Disponibilização deste conjunto de soluções por meio de um *framework* extensível, denominado SWoTPAD.
5. Criação de um ambiente de desenvolvimento para aplicativos de IoT, denominada *SWoTPAD Designer*, responsável pela integração do *framework* ao aplicativo de IoT.

1.4 Resultados Obtidos

Sistemas de IoT lidam com heterogeneidade, uma vez que o ecossistema de IoT requer conexão e integração de recursos heterogêneos, dispositivos, objetos e sistemas, sendo estes

denominados como coisas ou objetos inteligentes (KORTUEM et al., 2010).

Jara *et al* (JARA et al., 2014) descrevem a evolução da área de IoT até sua total consolidação. A primeira fase consistiu em interconectar tudo a internet para, em seguida, definir o IPv6 como protocolo de comunicação entre as entidades heterogêneas. O próximo passo estabeleceu os protocolos e linguagens usados na Web (HTTP, HTML, etc) como padrão para a camada de aplicação. Essa evolução deu origem a Web das Coisas (WoT, acrônimo do inglês *Web of Things*) (GUINARD et al., 2011).

Ainda assim, sistemas de WoT apresentam apenas integração vertical, pois cada sistema costuma ser concebido de forma isolada um dos outros. O cenário ideal é o de integração horizontal, onde diferentes camadas do sistema podem ser integradas, mesclando as capacidades dos sistemas envolvidos, bem como promovendo um conjunto maior de serviços.

Logo, o desafio após a WoT é construir uma Web Semântica das Coisas (SWoT, acrônimo do inglês *Semantic Web of Things*) (GYRARD et al., 2016a) para garantir um entendimento comum. SWoT é, por um lado, a fusão das tendências de IoT para avançar para as tecnologias da Web com protocolos como a arquitetura CoAP (SHELBY; HARTKE; BORMANN, 2014), REST e o conceito WoT e, por outro lado, a evolução da Web com as tecnologias de Web Semântica.

Nesse contexto, este trabalho contribui com a área de SWoT através de um *framework* que pode ser adotado como base para o desenvolvimento de novos sistemas de IoT. Adicionalmente, fornecemos uma ferramenta para o desenvolvimento de soluções de IoT, que auxilia o desenvolvedor na etapa de implementação. A ferramenta dá suporte a modelagem do domínio, sendo este centrado no ambiente, serviços e requisições do usuário.

Nossa visão é baseada em uma solução que apresenta os seguintes recursos: (i) adoção de uma linguagem comum para a definição de elementos SWoT que permite (ii) o mapeamento automático através de serviços Web RESTful e suporta (iii) a composição automática de sensores, atuadores e serviços Web. Com isso, auxiliamos a construção de novos sistemas para IoT.

Adicionalmente, o *framework* promove integração e escalabilidade nesses sistemas. A integração ocorre pelo compartilhamento de dados, pois aplicações que adotam o *framework* apresentam mesmo vocabulário base. O aumento da escalabilidade é obtida pela associação de anotações semânticas a cada um dos elementos que compõem o sistema de IoT. Essas anotações permitem ao SWoTPAD descobrir, classificar, selecionar e compor automaticamente serviços do ambiente. Desta forma, SWoTPAD pode buscar so-

luções alternativas, quando o serviço original, apto a atender uma determinada requisição, encontra-se sobrecarregado ou indisponível.

Na literatura, poucos trabalhos lidam com a questão da composição automática de serviços no âmbito de ambientes pervasivos ou de IoT. Kaldeli *et al* (KALDELI et al., 2013) apresentam uma abordagem para composição automática em ambientes pervasivo que aplica problema de satisfação de restrições. Georgievski (GEORGIEVSKI, 2015) aplica planejamento hierárquico para o mesmo propósito. Diferente dos trabalhos citados, realizamos a composição automática de serviços, usando uma base de conhecimento semântico. Neste trabalho, também aplicamos algoritmos de planejamento hierárquico multitarefa e não determinístico.

Nossa abordagem também propõe um modelo conceitual, adequado às necessidades dos sistemas de IoT. Nosso modelo conceitual consiste na extensão de conhecimento sobre os seguintes elementos:

- I **Serviço**. Esta categoria compreende atuadores, sensores e serviços web.
- II **Ator**. Entidades capazes de realizar requisições, bem como manipular instrumentos.
- III **Instrumento**. Entidades capazes de realizar alguma ação quando manipulada por algum ator.
- IV **Ambiente**. Lugares onde as entidades (I), (II) e (III) estão posicionadas.
- V **Requisição**. Representam anseios dos atores.

1.5 Escopo e Limitações

Nosso trabalho visa promover a interoperabilidade e escalabilidade entre aplicações de IoT e pode ser usado em dois contextos: durante o projeto de uma nova aplicação de IoT e em aplicações de IoT já implantadas.

No primeiro cenário, nossa abordagem tem como objetivo auxiliar o projetista na especificação de serviços e requisições que irão compor a aplicação de IoT. Para isso, usa a linguagem SWoTPADL e o vocabulário dessa aplicação. A partir dessa especificação, nossa abordagem faz o mapeamento para serviços reais. O mapeamento permite simular o sistema de IoT em desenvolvimento.

Em sistemas de IoT já implantadas, a especificação em SWoTPADL permite agregar alguns recursos aos serviços do sistema de IoT, são eles: (1) descoberta a partir de re-

quisições, (2) composição automática e dinâmica de serviços e (3) invocação dos serviços selecionados. Esses recursos também estão disponíveis no primeiro cenário.

Para implantação dessas ideias, a abordagem usa algumas ontologias de IoT ((COMP-
TON et al., 2012), (BERMUDEZ-EDO et al., 2016), (HEPP, 2013), (HOBBS; PAN,
2006), (FERRIS; JACOBSON, 2010) e (BRICKLEY; MILLER, 2007)), aplicadas de
acordo com nosso modelo conceitual de um sistema de IoT, descrito no Capítulo 3. Por
meio delas, os serviços que compõem diferentes sistemas de IoT podem ser integrados
pelo módulo de composição dinâmica e automática de serviços.

1.6 Metodologia

Para desenvolvimento e conclusão deste trabalho, foram realizadas o conjunto de
atividades detalhadas nas próximas seções.

1.6.1 Fundamentação e aquisição bibliográfica

Para realização do trabalho foi necessário o estudo profundo e detalhado dos seguintes
tópicos de pesquisa: projetos de sistemas de IoT, semântica, estratégias para composição
automática de serviços, linguagens de descrição semântica, desenvolvimento de modelos
formais e inferência em bases de conhecimento.

1.6.2 Criação de linguagem para descrição de serviços e requisi- ções do usuário.

Apesar da literatura apresentar linguagens que permitem elaborar anotações semânti-
cas (SAWSDL(LAUSEN; FARRELL, 2007)), bem como especificar serviços e requisições,
tais como OWL-S(MARTIN et al., 2004) e WSML(BRUIJN et al., 2008), é comum que
especificações desenvolvidas nessas linguagens sejam longas, o que dificulta sua compre-
ensão e reprodução por parte dos desenvolvedores. Neste sentido, criamos a linguagem
SWoTPADL que simplifica o processo de descrição dos serviços e requisições por parte do
usuário. No Capítulo 4, apresentamos detalhes dessa linguagem.

1.6.3 Formalização da linguagem desenvolvida

Métodos formais são empregados no desenvolvimento de sistemas computacionais e
consistem de técnicas baseadas em formalismos matemáticos para descrever proprieda-

des do sistema. Em geral, proveem *frameworks* para que as pessoas possam especificar, desenvolver e verificar sistemas de maneira sistemática e não *ad hoc* (WING, 1990).

Um método formal apresenta uma base matemática sólida, tipicamente dada por uma linguagem de especificação formal. Esta base fornece meios de definir precisamente noções como consistência, integridade e corretude, concebendo especificações e implementações mais precisas (WING, 1990).

Para proporcionarmos essas vantagens aos desenvolvedores que adotarem nossa abordagem, a linguagem SWoTPADL foi formalizada utilizando a linguagem de especificação formal Object-Z (LANO, 2001). Escolhemos Object-Z pelos seguintes fatores:

- Presença de um framework (MALIK; UTTING, 2005) que permite fazer validações em especificações Z ou Object-Z.
- Possibilidade do uso de Object-Z para verificação de modelos, como sugerido pelos trabalhos (WINTER; DUKE, 2002) e (KASSEL; SMITH, 2001).

1.6.4 Criação de modelo conceitual para Aplicações de IoT

Em ambientes de IoT, dados são gerados, transferidos e compartilhados em tempo real. A integração da informação é a essência da interoperabilidade de dados, pois possui influência significativa na eficiência e efetividade das decisões (PRAJOGO; OLHAGER, 2012). Neste trabalho, para garantir a interoperabilidade de dados, propomos um modelo conceitual específico a aplicações de IoT. O modelo conceitual descrito no Capítulo 3, apresenta um vocabulário comum, que é utilizado pela linguagem SWoTPADL. Desta forma, aplicações desenvolvidas nessa linguagem, compartilham um mesmo vocabulário.

1.6.5 Estratégia para Composição Automática de Serviços

Foram avaliadas diferentes estratégias para composição automática de serviços, detalhadas no Capítulo 6. Adotamos a técnica de planejamento de inteligência artificial, pois, das técnicas avaliadas, consegue equilibrar desempenho com uma representação próxima da semântica. Neste trabalho, adaptamos dois diferentes algoritmos para realizar a composição automática de serviços, ambos baseados em planejamento hierárquico (HTN, acrônimo de *Hierarchical Task Network*) (GHALLAB; NAU; TRAVERSO, 2004). O primeiro deles consiste em uma implementação determinística paralela do algoritmo HTN de (RUSSELL, 2010), o segundo, uma implementação não determinística e paralela do

algoritmo HTN apresentado em (KUTER, 2006).

1.6.6 Geração do Protótipo e Validação

A validação foi feita através de dois estudos de caso que fazem uso do *framework*. Adicionalmente, criamos um *plugin* no Eclipse, denominado SwoTPAD Designer, que permite ao projetista aplicar o *framework* nos sistemas de IoT em desenvolvimento.

1.6.7 Elaboração de Artigos Científicos

Durante a elaboração deste trabalho, tivemos os seguintes trabalhos aceitos para publicação.

Artigo completo publicado em Conferência Nacional (com *referee*)

- Magami, F. M., Perez-Alcázar, J. J., Nakano, F., **Silva, A. L.M.**, Neto, J. S. de O., Kofuji, S. T. *Abordagem semantica para o desenvolvimento de sistemas de suporte a ambientes inteligentes inclusivos*. Anais do 9º Simpósio Brasileiro de Computação Ubíqua e Pervasiva, SBCUP, v. 1, p. 898–907, 2017.

Este trabalho foi desenvolvido no âmbito do trabalho de conclusão de curso de Magami, no qual atuei como co-orientador. O artigo propõe um sistema para auxílio a navegação de deficientes visuais no ambiente de *shopping center*. O auxílio a navegação é feito através de solicitações a pontos de interesses feito pelo usuário deficiente. Neste artigo foi empregado o módulo de descoberta de serviços do *framework* SWoTPAD.

Capítulo de Livro

- Neto, J. S. de O., **Silva, A. L. M.**, Nakano, F., Pérez-Álcazar, J. J., Kofuji, S. T. *When wearable computing meets smart cities: Assistive technology empowering persons with disabilities*. In: Examining Developments and Applications of Wearable Devices in Modern Society. [S.l.]: IGI Global, 2018. p. 58–85.

O artigo apresenta um conjunto de soluções que podem auxiliar no desenvolvimento de cidades inteligentes inclusivas. O artigo adotou parte do referencial teórico presente nesta tese. Adicionalmente, apresenta a visão de semântica adotada nesta tese.

Também foram produzidos os seguintes trabalhos, ainda em avaliação.

Artigos submetidos a Periódicos Internacionais (com *referee*)

- **Silva, A. L. M.**, Pérez-Álcazar, J.J., Kofuji, S. T. *Interoperability in Semantic web of things: design issues and solutions*, artigo submetido ao *INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS*.

Este artigo apresenta o mecanismo de composição automática de serviços e descreve o estudo de caso de segurança residencial e implantação de serviços no ambiente de nuvem, apresentado no Capítulo 5.

- **Silva, A. L. M.**, Pérez-Álcazar, J.J., Kofuji, S. T. *Decreasing interoperability issues in Internet of Things*, artigo submetido ao *WIRELESS PERSONAL COMMUNICATIONS*.

Este artigo aborda a linguagem SWoTPADL e apresenta sua formalização em Object-Z.

1.7 Organização da Tese

Esta tese apresenta a seguinte organização:

- O Capítulo 2 apresenta o referencial teórico necessário para o entendimento deste trabalho. Nele são detalhados conceitos relacionados a Internet das Coisas, linguagens de descrição semântica, descoberta e composição de serviços, planejamento de inteligência artificial e Object-Z.
- O Capítulo 3 apresenta nossa visão de um sistema de IoT. Essa visão é formalizada através de um modelo conceitual. Todos os modelos presentes nesse modelo são definidos e mapeados para classes Object-Z.
- O Capítulo 4 apresenta o *framework SWoTPAD* e a linguagem *SWoTPADL*, adotada para definição de serviços e requisições de IoT. Adicionalmente é apresentada a ferramenta *SWoTPAD Designer*.
- O Capítulo 5 apresenta dois estudos de caso adotados para validação do *framework*. No primeiro, avaliamos o *framework* no âmbito de um sistema de automação residencial. O segundo, adaptado de (GEORGIEVSKI, 2015), consiste na aplicação do *framework* para implantação de aplicações na nuvem.

- O Capítulo 6 apresenta os trabalhos relacionados. Nossa revisão bibliográfica foi feita de acordo com os seguintes temas: trabalhos para composição automática de serviços, frameworks de suporte a IoT e representações de apoio a projetos semânticos.
- O Capítulo 7 apresenta algumas conclusões, bem como sugestões de trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta a base teórica necessária para compreensão desse trabalho. Na Seção 2.1, destacamos conceitos relacionados a Internet das Coisas (IoT), suas principais características e demandas. A Seção 2.2 dedica-se a Web Semântica das Coisas (SWoT), bem como a apresentação da linguagem WSML, que permite desenvolver anotações semânticas para esse tipo de sistema. Em seguida, na Seção 2.3, apresentamos aspectos relevantes da descoberta e composição de serviços. A Seção 2.4 apresenta o conceito de planejamento, com foco na técnica de planejamento hierárquico. Por último, a Seção 2.5 dedica-se a apresentação de modelagem formal, através da linguagem de especificação Object-Z.

2.1 Internet das Coisas (IoT)

O termo Internet das Coisas foi utilizado pela primeira vez por Kevin Ashton durante uma apresentação sobre o gerenciamento de cadeia de suprimentos (ASHTON, 2011). Na ocasião, Ashton definiu que 'coisas' estão relacionados com a maneira como interagimos e vivemos no mundo físico que nos rodeia. É fato que essa interação vem sofrendo mudanças. O aumento do poder de processamento dos dispositivos computacionais, maior disponibilidade de banda para comunicação e o aumento da taxa de geração de dados, por parte dos sensores, têm proporcionado a criação de uma nova geração de aplicações até então impossíveis de serem produzidas.

Os trabalhos desenvolvidos por Ashton, como diretor executivo do Centro de Auto-ID do MIT, criaram as bases para a visão atual de *IoT* (BUYYA; DASTJERDI, 2016). Atualmente, o conceito de 'coisa' foi ampliado e se refere a um conjunto mais genérico de entidades, na qual se incluem dispositivos inteligentes, sensores, seres humanos, atuadores e qualquer outro elemento consciente de contexto, apto a se comunicar com outras entidades, acessível de qualquer lugar, a qualquer instante (BUYYA; DASTJERDI, 2016).

Acredita-se que todo o potencial de *IoT* ocorrerá quando pequenas redes de coi-

sas conectadas, desde carros até pequenas tags RFID (acrônimo para *Radio Frequency Identification*), combinem-se para formar uma grande rede de coisas conectadas que se estendam entre indústrias e organizações (WITCHALLS; CHAMBERS, 2013). Esse potencial só será consolidado com o aprimoramento de algumas tecnologias e conceitos, que apresentamos a seguir:

- **Tecnologias de identificação por radiofrequência (RFID):** auxiliam na identificação automática de elementos ao qual estão associados.
- **Redes de sensores sem fio (WSN, acrônimo de *Wireless Sensor Network*):** cada vez mais eficientes por conta dos avanços tecnológicos, as WSNs atuais podem agregar uma grande quantidade de sensores sem fio e dão suporte a coleta, processamento, análise e divulgação de dados (GUBBI et al., 2013).
- ***Middleware*:** camada de *software* responsável por fazer a mediação entre o *software/hardware* e demais aplicações do sistema de *IoT*. Tem como objetivo tornar o uso do sistema mais genérico, prover um meio de acesso comum ao sistema, facilitando o desenvolvimento de novas aplicações.
- **Esquemas de endereçamento:** capacidade de identificar de maneira exclusiva um objeto. Para isso, adota-se o protocolo *IPv6*, que permite acessar recursos de forma exclusiva e remota.

É interessante observar *IoT* do ponto de vista de sistema e serviço para evidenciar suas principais carências. Como sistema, *IoT* é uma rede altamente dinâmica e distribuída, formada por um número elevado de objetos inteligentes que produzem e consomem dados, cuja interface com o meio é feita através de sensores e atuadores. Nesse cenário, temos como principal preocupação a escalabilidade. Do ponto de vista de serviço, a principal demanda passa a ser a necessidade de integração ou composição de funcionalidades e recursos fornecidos pelos objetos inteligentes para os serviços (MIORANDI et al., 2012). Logo, interoperabilidade e escalabilidade são questões cruciais quando se tratam de sistemas de *IoT*.

Muitos desses problemas podem ser amenizados com a aplicação de semântica (FENSEL et al., 2011). Por meio dela, podemos aprimorar as tarefas de endereçamento, rastreamento e descoberta de objetos, bem como a representação, armazenamento e troca de informações. Semântica, portanto, está emergindo como um elemento de construção necessário para o futuro da *IoT*. Caso contrário, os bilhões de dispositivos conectados não irão conversar de forma significativa (RAJ; RAMAN, 2017).

Panda (PANDA, 2017) enumera oito características que sistemas de *IoT* devem dar suporte, são elas:

- I. **Heterogeneidade de dispositivos:** *IoT* é caracterizado pela grande heterogeneidade de dispositivos que participam do sistema, que costumam apresentar recursos muito diferentes do ponto de vista computacional e de comunicação. A gerência dessa heterogeneidade deve ser suportada em níveis arquitetônicos e protocolares.
- II. **Escalabilidade:** a medida que os objetos cotidianos se conectam a uma infraestrutura de informação global, os problemas de escalabilidade surgem em diferentes níveis, incluindo: (1) nomeação e endereçamento - devido ao tamanho do sistema resultante, (2) comunicação de dados e rede - devido ao alto nível de interconexão entre um grande número de entidades, (3) gerenciamento de informações e conhecimento - devido à possibilidade de construir uma contrapartida digital para qualquer entidade ou fenômeno no domínio físico e (4) provisionamento e gerenciamento de serviços - devido ao número de opções de serviços que podem estar disponíveis e a necessidade de lidar com recursos heterogêneos.
- III. **Troca de dados ubíqua através de tecnologias sem fio de proximidade:** em *IoT*, um papel proeminente será desempenhado pelas tecnologias de comunicação sem fio, que permitirá que objetos inteligentes formem redes. A ubiquidade do meio sem fio para a troca de dados pode representar problemas em termos de disponibilidade de espectro, promovendo a adoção de sistemas de rádio cognitivos ou dinâmicos.
- IV. **Soluções otimizadas para o consumo energético:** para uma variedade de entidades de *IoT*, minimizar o consumo a ser utilizado para fins de comunicação ou computação será uma restrição primária. Enquanto as técnicas relacionadas à coleta de energia (por meio, por exemplo, de materiais piezoelétricos ou painéis micro-solares) aliviarão os dispositivos das restrições impostas pelas operações da bateria, a energia sempre será um recurso escasso a ser manuseado com cuidado. Deste modo, a necessidade de conceber soluções que tendem otimizar o uso de energia, se tornará cada vez mais atraente.
- V. **Capacidade de localização e rastreamento:** como entidades de *IoT* podem ser identificadas e apresentam capacidade de comunicação sem fio de curto alcance, torna-se possível rastrear a localização (e o movimento) de objetos inteligentes no domínio físico.

- VI. **Capacidade de auto-organização:** a complexidade e a dinâmica de muitos cenários de *IoT* para a distribuição de inteligência no sistema, tornando os objetos inteligentes (ou um subconjunto) capazes de reagir de forma autônoma a uma ampla gama de situações diferentes, a fim de minimizar a intervenção humana.
- VII. **Interoperabilidade semântica e gerenciamento de dados:** *IoT* será sobre como trocar e analisar elevados volumes de dados. Para transformá-los em informações úteis e para garantir a interoperabilidade entre diferentes aplicativos, é necessário fornecer dados com formatos, modelos e descrição semântica adequados e padronizados (meta-dados), usando linguagens e formatos bem definidos.
- VIII. **Mecanismos integrados de segurança e preservação da privacidade:** a tecnologia de *IoT* deve ser segura e preservar a privacidade por projeto. Isso significa que a segurança deve ser considerada uma propriedade chave e deve ser levada em consideração no projeto de arquiteturas e métodos para soluções *IoT*.

As questões I, II, VI, VII podem ser endereçadas com auxílio de semântica. Por meio de semântica, podemos desenvolver descrições explícitas, simplificadas e estruturadas do significado dos dados para que, diferentes dispositivos de *IoT* distribuídos, possam entender e trabalhar coletivamente de forma significativa. Dados de *IoT*, tipicamente heterogêneos tornam-se homogêneos, pois o mesmo vocabulário é empregado. Adicionalmente, técnicas e mecanismos de raciocínio semântico podem ser empregados nesse contexto. A Seção 2.2 apresenta esses recursos, através da linguagem de descrição semântica WSML, utilizada em nosso trabalho.

2.2 Web Semântica das Coisas

A Web Semântica das Coisas (SWoT) (GYRARD et al., 2016b), mapeia tecnologia de *Web Semântica* para *IoT*, possibilitando a criação de uma infra-estrutura de conhecimento (HAUSWIRTH; DECKER, 2007), com dados significativos do mundo físico e cibernético. Assim, o conhecimento de alto nível pode ser questionado e inferido dos detalhes das observações sensoriais para obter mais interações entre humanos, máquina-máquina e homem-máquina (WU et al., 2017).

Análogo ao que ocorre com *IoT*, a Web também sofre de problemas de interoperabilidade e escalabilidade há anos. Isso ocorre porque o conteúdo da Web, mesmo sendo simples e amigável para os seres humanos, é de difícil compreensão para os dispositivos computacionais. Nesse contexto surgiu a Web semântica.

A Web semântica estende a Web tradicional de forma que a informação disponível seja acompanhada por significado. A semântica incluída permite que a informação possa ser processada e interpretada computacionalmente. Dessa forma, máquinas desempenham um papel mais ativo no consumo de dados e não apenas na transmissão de informação para consumo humano (FENSEL et al., 2011).

A Web semântica permite que as máquinas compreendam e satisfaçam as solicitações dos usuários, processando dados semânticos. Como consequência, temos uma melhora da Web atual, pois promove o aumento da automação e precisão em vários aspectos, tais como busca, extração e integração de informações (FENSEL et al., 2011). Conforme previsto por Tim Berners-Lee (BERNERS-LEE; HENDLER; LASSILA, 2001), *"a Web semântica é uma extensão da Web atual na qual a informação possui um significado bem definido, melhor permitindo que os computadores e as pessoas trabalhem em cooperação"*.

Os conceitos que guiam a Web semântica podem ser adotados naturalmente em *IoT*. A ideia de dispositivos computacionais que ofertam serviços na Web é cada vez mais comum. Assim, o uso de descrições semânticas que apresentam as capacidades de cada dispositivo, bem como a forma de utilizá-lo, é uma característica essencial para promover a interoperabilidade e escalabilidade em sistemas de *IoT*. Adicionalmente, esses recursos podem ser adotados para descrever o meio ambiente, usuários, perfis de usuários, entre outros elementos, agregando maior valor a esse conhecimento.

O conhecimento é expresso por meio de ontologias. Gruber (GRUBER, 1993) define ontologia como *"uma especificação explícita de uma conceitualização"*. Conceitualização é uma abstração do mundo que queremos representar. Portanto, uma ontologia em computação é um artefato que modela a estrutura da área do conhecimento relevante, com suas classes e relações extraídas pela observação e abstração do domínio de interesse, de forma que ele possa ser compreensível a máquina (GUARINO; OBERLE; STAAB, 2009).

Existem algumas linguagens para descrever serviços Web semânticos. Nesse trabalho abordaremos duas: WSMO e OWL-S. OWL-S (MARTIN et al., 2004) é uma ontologia para descrição de serviços desenvolvida sobre OWL (Web Ontology Language) (DEAN et al., 2004), uma linguagem adotada para modelar conhecimento. A ontologia OWL-S é composta por três partes: *perfil de serviço*, *modelo de processo* e *grounding*. Cada um deles, descrito da seguinte forma:

- **O Perfil de Serviço** descreve o propósito do serviço, desde a perspectiva do provedor do serviço ao cliente que faz a solicitação. O perfil de serviço foi projetado para atender aos requisitos de um agente de busca de serviços e permite determinar

se o serviço atende às demandas dos usuários.

- O **Modelo de Processo** descreve a forma como o cliente interage com o serviço. Apresenta o conjunto de entradas, saídas, precondições e resultados da execução do serviço.
- O ***Grounding*** especifica como as descrições abstratas do serviço são mapeadas para as descrição concretas do serviço real.

WSMO (acrônimo para Web Service Modeling Ontology) (BRUIJN et al., 2005) propõe um modelo conceitual para descrever ontologias, serviços Web, metas e mediadores. O objetivo da linguagem é criar uma ontologia que descreve vários aspectos relacionados aos serviços Web semânticos, com foco na resolução do problema de integração. WSMO também leva em consideração domínios de aplicação específicos para garantir a aplicabilidade da ontologia para esses campos. Além disso, os elementos definidos no padrão WSMO são fracamente acoplados, em contraste com o acoplamento mais forte presente em OWL-S (LARA et al., 2004).

Por exemplo, em OWL-S, o perfil do serviço descreve o serviço oferecido pelo provedor e os serviços necessários ao solicitante. Já WSMO, usa *metas e capacidades* para descrever essas duas perspectivas. Assim, WSMO apresenta uma divisão clara entre a visão do provedor do serviço e do cliente. WSMO também introduz o conceito de *mediadores* para contornar os problemas de heterogeneidade. Mediadores são serviços especiais que ligam componentes heterogêneos envolvidos na modelagem de um serviço Web. OWL-S não apresenta construção correspondente e deixa este problema nas mãos do projetista do serviço (LARA et al., 2004).

Embora ambas linguagens apresentem objetivos semelhantes, existem vantagens na escolha de WSMO sobre OWL-S. Em nossa abordagem, escolhemos WSMO pelas seguintes razões:

- I. WSMO apresenta total suporte para descrever todos os elementos de um serviço Web. Esse recurso promove interoperabilidade e escalabilidade, pois evita o uso de vários formalismos para descrição do serviço (LARA et al., 2004).
- II. WSMO suporta a representação de regras como parte das descrições e ontologias, uma vez que o mecanismo de inferência é orientado por regras. Em OWL-S, o mecanismo de inferência é orientado a classificação (com base na inferência de sub-sunção) (HORROCKS; PADGHAM; THOMSON, 1999). Portanto, regras não são suportadas nativamente (PANZIERA et al., 2010).

- III. WSML suporta a representação de predicados de aridade arbitrária. OWL-S suporta apenas a representação de relações binárias, que são representadas por propriedades (PANZIERA et al., 2010).
- IV. OWL-S adota o ponto de vista do serviço para descrever as atividades do serviço. WSML adota o ponto de vista do cliente para descrever as metas do cliente (KAMARUDDIN; SHEN; BEYDOUN, 2012). WSML apresenta apenas o suficiente para o cliente, abstraindo muito da complexidade. Independentemente do cliente ser um usuário ou um desenvolvedor, eles apenas visualizam os recursos necessários para usar o serviço.
- V. WSML visa criar uma ontologia para descrever vários aspectos relacionados ao serviço semântico, incluindo descoberta, composição e invocação. É principalmente focado na resolução do problema de interoperabilidade (BOUROS; PANEPISTIMIOPOULIS, 2005).
- VI. WSML adota a hipótese do mundo fechado ao contrário de OWL-S que adota a hipótese do mundo aberto. Imagine, por exemplo, a definição de um relacionamento para representar países que fazem fronteira. Na hipótese do mundo fechado, dois países só fazem fronteiras se este fato é declarado explicitamente. Caso contrário, os dois países não fazem fronteira. No entanto, isso não será verdade para a hipótese do mundo aberto, pois a ausência de uma informação na base de conhecimento não é suficiente para invalidá-la.

Na próxima seção, apresentaremos as principais construções de WSML, sua semântica e alguns exemplos de aplicação.

2.2.1 Web Service Modeling Language

WSML (FENSEL et al., 2011) (acrônimo de *Web Service Modeling Language*) corresponde a implementação concreta de WSMO e é uma linguagem formal para descrever serviços Web semânticos. Visa fornecer meios para descrever todos os elementos definidos em WSMO. WSML foi especificada em termos de uma sintaxe legível para humanos com palavras reservadas semelhantes aos elementos do modelo conceitual WSMO. A seguir, detalhamos os quatro elementos adotados em WSML para a descrição de serviços semânticos:

- **Ontologia:** define a semântica formal para as terminologias usadas por todos os

componentes WSMO. Dessa forma, é possível definir, sem ambiguidade, o vocabulário a ser compartilhado, facilitando a interoperabilidade e automatização dos serviços.

- **Serviços Web:** apresenta os elementos necessários para descrever todos os aspectos de um serviço Web. Um serviço WSML é definido a partir dos seguintes elementos:
 - **Funcionalidade:** descreve as capacidades do serviço. Essa descrição é feita pela definição dos seguintes itens:
 - * **Precondições:** especificam condições nos parâmetros de entrada do serviço.
 - * **Suposições:** descrevem o estado do mundo antes da execução do serviço. Caso as suposições não sejam atendidas, não é garantido êxito na prestação do serviço.
 - * **Pós-condições:** descrevem a relação entre a entrada e a saída do serviço.
 - * **Efeitos:** descrevem o estado do mundo após execução bem sucedida do serviço.
 - **Propriedades não funcionais:** compreendem itens como custo de invocação de serviço e qualidade de serviço.
 - **Interface:** descreve como interagir com um serviço do ponto de vista do solicitante e descreve como o serviço faz uso de outros serviços e metas a fim de fornecer sua funcionalidade.
- **Metas:** representam os objetivos dos usuários com relação ao serviço Web solicitado.
- **Mediadores:** são elementos utilizados para resolver problemas de interoperabilidade entre componentes, ou seja, são adotados para integrar componentes. São quatro os tipos de mediadores:
 - *ggMediator*: relaciona duas metas. Representa o refinamento de uma meta origem em uma meta alvo.
 - *ooMediator*: importa ontologias e resolve possíveis combinações entre estas.
 - *wgMediator*: relaciona serviços Web a metas. Esse mediador é adotado quando um serviço Web satisfaz (totalmente ou parcialmente) a meta com a qual ele é relacionado.
 - *wwMediator*: relaciona dois serviços Web.

A seguir, detalhamos alguns aspectos da linguagem WSML relevantes para essa tese.

2.2.1.1 Ontologias

Em WSML, uma ontologia consiste dos seguintes elementos: *concept*, *relation*, *instance*, *relationInstance* e *axiom*. Adicionalmente, uma ontologia dá suporte a anotações e importações de outras ontologias.

Concept, também conhecido como *classe*, desempenha papel principal em uma ontologia. É através dele que definimos a terminologia a ser adotada pelo serviço. *Concept* apresenta um conjunto de atributos que o descreve. Atributos podem ter seus tipos definidos através da cláusula *ofType*, ou inferidos através da cláusula *impliesType*. Adicionalmente, WSML permite associar vários tipos a um único atributo.

Relation introduz o conceito de relações em uma ontologia WSML. Elas podem ser organizadas em hierarquias através da cláusula *subRelationOf* e seus parâmetros podem ser associados a *Concept*, através da cláusula *ofType* e *impliesType*. *Relation* pode ser instanciado através da cláusula *RelationInstance*.

Instance representa instâncias de *Concept*, podendo estar associada a um ou mais *Concept*. De maneira análoga, temos a cláusula *relationInstance* para representar instâncias de *relation*. No entanto, não é obrigatório que um *Instance* esteja associado a um *Concept*. Adicionalmente, um *Instance*, que esteja associado a um *Concept* pode apresentar campos não presentes nesse *Concept*. Segundo os autores (FENSEL et al., 2011), esse relaxamento de restrições visa permitir a representação de dados semi-estruturados.

Axiom provê o mecanismo de adicionar expressões lógicas em ontologia. Tais expressões lógicas podem ser usadas para refinar conceitos ou definições de relações presentes em ontologias, bem como adicionar conhecimento ao domínio ou expressar restrições.

A Listagem de Código 1 apresenta um exemplo de ontologia. Nas linhas 4 a 10, vemos a criação dos *Concepts* *Person*, *HomeUser* e *Intruder*, sendo os dois últimos uma especialização do *Concept* *Person*. Nas linhas 11 a 16 é definido o *axiom system_user_definitions*, no qual é realizado um refinamento nos *Concepts* *Intruder* e *HomeUser*. Basicamente, se um *Instance* possui o atributo *isKnown* com valor *false*, essa instância é considerada um *Intruder*, caso contrário, um *HomeUser*. Adicionalmente, instâncias do tipo *HomeUser* possuem valor *true* para o atributo *isKnown*, como denotado pela linha 15. Nas linhas 17 a 22 são criadas duas instâncias: *Andre* e *Jose* de tipo *HomeUser*. Por

Listagem de Código 1: Exemplo de definição de ontologia WSML

```

1: wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
2: namespace { _"http://www.usp.lsi.br/wsmlpe/ontologies/" }
3: ontology HomeElementsContext
4:   concept Person
5:     isKnown ofType _boolean
6:     hasLocalization ofType PlaceOfAHouse
7:   concept HomeUser subConceptOf Person
8:     hasSmartphone ofType _string
9:     hasEmail ofType _string
10:  concept Intruder subConceptOf Person
11:  axiom system_user_definitions
12:    definedBy
13:      ?x[isKnown hasValue _boolean("false")]
14:      implies ?x memberOf Intruder.
15:      ?x[isKnown hasValue _boolean("true")]
16:      equivalence ?x memberOf HomeUser.
17:  instance Andre memberOf HomeUser
18:    hasSmartphone hasValue "+55(79)8188-7966"
19:    hasEmail hasValue "andrelumesi@usp.br"
20:  instance Jose memberOf HomeUser
21:    hasSmartphone hasValue "+55(79)9988-0966"
22:    hasEmail hasValue "pedroJo@gmail.com"
23:  relation rlTwoPersons (ofType Person, ofType Person)
24:  relationinstance isfather (Jose, Andre):rlTwoPersons.

```

Fonte: Próprio Autor.

fim, nas linhas 23 e 24, são definidos, respectivamente, o *relation* *rlTwoPersons* que relaciona dois atributos do tipo *Person* e o *relationInstance* *isfather* que relaciona as *instances* *Jose* e *Andre*.

2.2.1.2 Serviços

WSML apresenta um conjunto de construções que permite formalizar funcionalidades, comportamentos e outros aspectos relacionados a serviços Web. Uma descrição de serviço Web consiste da cláusula *webService* seguida da definição de *capability*, que descreve a funcionalidade do serviço; uma ou mais *interfaces*, que descrevem as formas de interação com o serviço e *nonFunctionalProperties*, que descrevem aspectos não funcionais dos serviços.

Capability define funcionalidades providas pelo serviço e restrições sobre eles. Ela é definida por meio de um conjunto de *axioms* através de quatro cláusulas diferentes, são elas *preconditions*, *assumptions*, *postconditions* e *effects*. *Preconditions* e *assumptions* descrevem o estado antes da execução do serviço Web. A diferença entre elas é que *precondition* descreve condições sobre as entradas, enquanto que *assumption* descreve condições sobre o estado do mundo, onde o serviço está inserido. *Postcondition*

descreve a relação entre as entrada e saídas, enquanto que *effect* descreve mudanças causadas pelo serviço, não ligadas às entradas e saídas.

A *Interface* descreve como interagir com um serviço do ponto de vista de quem requisita o serviço (coreografia) e como o serviço interage com outros serviços e metas que ele precisa alcançar para desempenhar seu comportamento (orquestração). WSMML não formaliza orquestração, porém, permite fazer referências a orquestrações externas. A Listagem de Código 2 apresenta um exemplo de definição serviço Web. Nela podemos

Listagem de Código 2: Exemplo de definição de um *webService* WSMML

```

1: wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
2: namespace {discovery _"http://wiki.wsmx.org/index.php?title=DiscoverOntology#"}
3: webService _"http://example.org/service/ClimaBr"
4:   capability _"http://example.org/service/ClimaBrCapability"
5:     sharedVariables {?location}
6:     precondition
7:       annotations
8:         dc#description hasValue "Entrada refere-se a uma localizacao no Brasil"
9:       endAnnotations
10:    definedBy
11:      ?location [hasCountry hasValue loc#br] memberOf loc#Location.
12:    postcondition
13:      annotations
14:        dc#description hasValue "O clima de acordo com a localizacao passada"
15:      endAnnotations
16:    definedBy
17:      ?forecast [hasLocation hasValue ?location] memberOf WeatherForecast.

```

Fonte: (FENSEL et al., 2011)

ver a definição do *webService ClimaBr* e de sua *capability climaBrCapability*. O *capability* apresenta a variável compartilhada *?location* (linha 5). Variáveis compartilhadas são aquelas que aparecem em mais de uma cláusula (*precondition*, *postcondition*, *assumption* ou *effect*). No exemplo, *?location* aparece nas cláusulas *precondition* e *postcondition*. Basicamente, o serviço descrito apresenta como pré-condição a presença de uma localização no Brasil (denotada por *loc#br*, linha 11) e como pós-condição, ele informa condições de tempo da respectiva localização (linha 17).

Para coreografia, WSMML define uma linguagem de descrição de coreografias denominada *WSML Choreography Language*. Uma descrição de coreografia é uma sequência de trocas de mensagens entre o solicitante do serviço e o provedor do serviço. Trocas de mensagens são regidas por meio de regras de transição, de forma que dado o estado atual, as regras de transição são responsáveis por determinar o próximo passo da conversação. As mensagens consistem de instâncias de informações presentes em ontologias, bem como o conhecimento obtido pela inferência em ontologias.

O conceito de estado é um dos elementos centrais de uma coreografia. Um estado

consiste de instâncias de conceitos válidos durante a execução do serviço. Transições de estado podem desencadear atualizações dos valores, inserções ou deleção de instâncias. A comunicação entre o solicitante e o serviço é feito por meio da marcação dos conceitos de ontologias como conceitos de entrada ou conceitos saída. Uma mensagem de entrada do solicitante ao serviço é feita por meio da criação de uma instância de um conceito de entrada. De forma análoga, uma mensagem de saída do serviço ao solicitante é feita por meio da criação de uma instância de um conceito de saída.

Uma conversação entre solicitante e serviço corresponde a execução da coreografia. Uma coreografia consiste de um estado inicial, uma sequência de estados intermediários e um estado final. Transições de estado são regidas por regras de transição, que, ao ter suas condições atendidas, disparam uma transição de estado.

A Listagem de Código 3 ilustra a coreografia denominada *choreography WeatherUSA-InterfaceChoreography*. Nela, vemos o campo *stateSignature*, responsável por de-

Listagem de Código 3: Exemplo de *choreography* WSML

```

1: choreography WeatherUSAInterfaceChoreography
2:   stateSignature WeatheerUSAInterfaceSignature
3:     importsOntology _"http://example.org/ontology/Weather"
4:     in
5:       wea#WeatherQuery withGrounding
6:       _"http://example.org/webservices/USWeatherForecast/Weather#wsdl
7:         .interfaceMessageReference(WeatherServicePortType/GetWeatherForCityState/In)",
8:     out
9:       wea#WeatherForecast withGrounding
10:      _"http://example.org/webservices/USWeatherForecast/Weather#wsdl
11:        .interfaceMessageReference(WeatherServicePortType/GetWeatherForCityState/Out)",
12:   transitionRules WeatherUSAInterfaceTransitions
13:     forall ?search
14:       with
15:         (?search[
16:           hasCity hasValue ?city, hasState hasValue ?state
17:         ] memberOf wea#WeatherQuery and exists?location (
18:           ?location memberOf loc#Location and
19:           ?location [hasCity hasValue ?city, hasState hasValue ?state]
20:         )
21:       )
22:     do
23:       add ?forecast[hasLocation hasValue ?location
24:         ] memberOf wea#WeatherForecast)
25:       delete(?search memberOf wea#WeatherQuery)
26:     endForall

```

Fonte: (FENSEL et al., 2011)

finir *concepts* ou *relations* usados na coreografia, o modo como eles são utilizados e a entidade responsável por seu mapeamento ao serviço. As linhas 4 a 6 apresentam o *concept* *wea#WeatherQuery*, cujo modo de utilização é de entrada, definido pela palavra reservada *in* (linha 4). Adicionalmente, o mapeamento de *wea#WeatherQuery* para o

serviço é feito através da especificação WSDL assinalada pelas linhas 6 e 7. De maneira similar, a coreografia usa o *concept wea#WeatherForecast* como saída do serviço, definida através da palavra reservada *out*. O mapeamento de *wea#WeatherForecast* para o serviço é feito pela especificação WSDL assinalada pelas linhas 10 e 11.

Após o bloco *stateSignature*, temos o bloco *transitionRules WeatherUSAInterfaceTransitions*. O bloco *transitionRules* declara as condições necessárias para desencadear uma regra de transição de estado. Se a condição de uma *transitionRule* é atendida, a regra é disparada. Após disparo da regra, o estado pode ser alterado através de adições, deleções ou atualizações de instâncias.

No exemplo da Listagem de Código 3, a condição para realização da *transitionRule WeatherUSAInterfaceTransitions* é apresentada nas linhas 15 a 21. Para realização dessa *transitionRule*, é necessário existir instâncias de consultas do tipo *wea#WeatherQuery* associadas a uma determinada cidade (*?city*) e estado (*?state*) que pertencem a uma mesma localização (*?location*). Caso a condição seja atendida, o efeito da regra é realizado (linhas 22 a 25). Nesse exemplo, o efeito da regra é a criação de instâncias do *concept wea#WeatherForecast*, que apresentam informações climáticas da localização (*?location*) associadas a consulta *wea#WeatherQuery*. Adicionalmente, as instâncias de *wea#WeatherQuery* usadas na regra são removidas do sistema.

2.2.1.3 Definição de Metas

Para representação de metas, WSMML define a cláusula *goal*, que representa a visão do solicitador sobre a funcionalidade desejada do serviço. Um *goal* é composto pelos

Listagem de Código 4: Exemplo de definição de *goal* WSMML

```

1: wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"
2: namespace {discovery _"http://wiki.wsmx.org/index.php?title=DiscoverOntology#"}
3: goal _"http://example.org/service/WeatherInnsbruck"
4:   capability
5:     importsOntology {_"http://example.org/ontology/Location",
6:       _"http://example.org/ontology/Weather"}
7:     postcondition
8:       definedBy
9:         ?forecast[hasLocation hasValue loc#Innsbruck] memberOf WeatherForecast.
```

Fonte: (FENSEL et al., 2011)

mesmos elementos que descrevem um *webService*, ou seja, *capability*, *interface* e *nonFunctionalProperty*. A diferença é o ponto de vista da descrição. Enquanto um *webService* adota o ponto de vista do serviço, um *goal* segue o ponto de vista do solicitante do serviço. Um pequeno exemplo de *goal* é apresentado na Listagem de

Código 4. Nela, vemos que o atendimento da *goal* resulta no fornecimento da previsão de tempo da localização *loc#Innsbruck*.

2.3 Descoberta e Composição de Serviços

Apesar de *IoT* apresentar muitas oportunidades para o setor industrial e acadêmico, as atuais propostas de arquitetura para a criação de ambientes de *IoT* não apresentam suporte a uma forma eficiente e padrão de descoberta, composição e integração de serviços de forma escalável (BUYA; DASTJERDI, 2016).

A ausência de um algoritmo de descoberta eficaz pode resultar em atrasos na execução, proporcionar uma pobre experiência ao usuário e falhas de tempo de execução. De acordo com (LIU et al., 2014), algoritmos eficientes podem ajudar a minimizar o consumo de energia, embora outros parâmetros, tais como a mobilidade e a latência, devem ser levados em conta para oferecer uma solução adequada para *IoT*.

A aplicação de semântica a *IoT*, ou seja, *SWoT* é uma possível solução, pois prevê um gerenciamento avançado de recursos e descoberta de serviços para *IoT* através da combinação de Web semântica com *IoT*. Para isso, os recursos e seus metadados são definidos e anotados, usando linguagens para definição de ontologias, tais como WSML ou OWL. Como consequência, a busca e manipulação desses metadados podem ser feitas através de linguagens de consulta como o SPARQL. Ideias que vem sendo adotadas de forma bem sucedida no contexto de serviços Web (FENSEL et al., 2011).

Gustavo *et al* (ALONSO et al., 2004) definem um serviço Web como uma aplicação de software identificada por uma URI (Uniform Resource Identifier), cujas interfaces e vínculos são capazes de serem definidos, descritos e descobertos como artefatos XML. Um serviço Web suporta interação direta com outros agentes de software, através do uso de mensagens baseadas em XML, por meio de protocolos da internet.

Um serviço Web é composto por três entidades (DUSTDAR; SCHREINER, 2005): o *provedor de serviço*, o *registrador de serviço* e o *consumidor de serviço*. O provedor de serviço é responsável por criar ou facilitar o uso de algum serviço Web. Ele descreve o serviço Web em um formato padrão e o publica em uma central de registro de serviços. O registrador de serviço possui informações adicionais sobre o provedor do serviço, tais como endereço e contato da companhia que provê o serviço. Adicionalmente, possui detalhes técnicos sobre o serviço. O consumidor de serviço recupera a informação de registro e a usa para vincular e invocar o serviço Web. Essa organização possibilita desenvolver

serviços Web com baixo grau de acoplamento. Aliado ao uso de linguagens e protocolos consagrados, tais como XML, HTTP, etc, proporcionou sua rápida adoção e crescimento (ALONSO et al., 2004).

As ideias apresentadas até aqui podem ser mapeadas para o contexto de *IoT*, visto que, em sua maioria, os componentes de um sistemas de IoT apresentam conectividade com a rede, e dessa forma, suas funcionalidades podem ser ofertadas por meio de um ou mais serviços Web (SPIESS et al., 2009). Num ambiente como esse, de múltiplos serviços, também podemos pensar na adoção de compositores de serviços.

A composição de serviços refere-se ao processo de combinar vários serviços com intuito de prover um novo serviço (MEDJAHED; BOUGUETTAYA, 2011). Com isso, novos serviços podem ser criados, fazendo o reuso de funcionalidades providas por serviços já existentes. Os fatores que vem estimulando a adoção dessa técnica são apresentados a seguir:

- O emprego de mensagens XML ou JSON por grande parte dos protocolos ubíquos bem estabelecidos. Essa característica tem permitido a comunicação e integração entre sistemas desenvolvidos para diferentes finalidades.
- Serviços Web, geralmente, expõem sua funcionalidade através de um modelo de mensagens baseado em documento ou como um conjunto de recursos endereçáveis. Tais modelos favorecem um relacionamento de fraco acoplamento entre aplicações. Dessa forma, fica fácil para corporações proverem serviços para clientes, sem necessidade de fornecer detalhes de codificação do serviço.

Do ponto de vista comercial, a composição de serviço oferece algumas vantagens (PAZOGLOU et al., 2008). A composição de serviços permite às organizações minimizarem a quantidade de trabalho necessário para desenvolver novas aplicações, assegurando redução do *time-to-market*. Adicionalmente, o desenvolvimento de aplicações baseados em serviços Web reduz riscos, visto que o reuso de serviços evita a introdução de erros. Outro aspecto importante é que a adoção da composição de serviços permite a redução de habilidades e esforços necessários para o desenvolvimento de aplicações. Por último, a possibilidade de terceirizar seus melhores serviços permite às companhias incrementar suas receitas.

De acordo com a literatura de composição de serviços (SHENG et al., 2014), existem três categorias relacionadas à composição, são elas:

1. **Composição estática vs Composição Dinâmica:** lidam com o instante em que os serviços Web são compostos, ou seja, se em tempo de projeto (composição estática) ou em tempo de execução (composição dinâmica).
2. **Composição manual vs Composição semi-automática vs Composição automática:** lidam com a forma como é feita a composição dos serviços. Na composição manual, o projetista deve criar um processo composto abstrato, utilizando alguma linguagem padrão de serviço da Web, como WSML ou OWL-S. Em seguida, o projetista vincula os serviços Web ao processo abstrato manualmente. Na composição automática, um aplicativo é responsável por escolher os serviços a serem compostos. Para isso, faz uso de ontologias, bem como mecanismos de inferência para selecionar e escolher os serviços adequados para composição. Na semi-automática, uma aplicação fornece pistas para o usuário sobre quais serviços devem participar da composição. No entanto, o usuário toma a decisão final sobre quais serão os serviços participantes.
3. **Composição orquestrada vs Composição coreografada:** descrevem a sequência de atividades a serem realizadas pelo serviço composto. Na orquestração, um processo único é responsável por coordenar a interação entre os diferentes serviços. Já na coreografada, não há o papel do coordenador: cada serviço participante da composição sabe o momento em que deve atuar, através de trocas públicas de mensagens, regras de interação e acordos entre dois ou mais processos.

Nosso trabalho visa o desenvolvimento de um *framework* que apresenta as seguintes características das abordagens de composição de serviços: **automática, dinâmica e orquestrada**.

Rao *et al* (RAO; SU, 2004) propõem um *framework* genérico para composição automática de serviços Web. A Figura 1 apresenta cada uma das entidades presentes nesse *framework*. Nela, podemos ver dois participantes: o provedor de serviço e o solicitador de serviço. O provedor de serviço disponibiliza serviços Web prontos para uso. O solicitador de serviço consome informações ou serviços oferecidos pelos provedores de serviço. Adicionalmente, o sistema também apresenta os seguintes elementos: tradutor, gerador de processo, avaliador, motor de execução e o repositório de serviços. O papel do tradutor é realizar conversões entre a linguagem externa, utilizada pelos participantes, e a linguagem interna, utilizada pelo gerador de processo. Para cada requisição, o gerador de processo tenta gerar um plano. Um plano compõe os serviços disponíveis no repositório de serviços, no intuito de atender a requisição passada. Se mais de um plano é encontrado para

uma mesma requisição, o avaliador inspeciona todos os planos e escolhe o melhor para execução. O motor de execução realiza o plano e devolve o resultado para o solicitador de serviço.

Figura 1: Arquitetura Genérica de um Sistema para Composição Automática de Serviços.

Fonte: Adaptado de (RAO; SU, 2004)

O gerador de processo é o módulo essencial para composição de serviços. O papel desempenhado pelo gerador de processo pode ser desenvolvido através da adoção de técnicas de *workflow* (RAO; SU, 2004), planejamento, algoritmos genéticos, verificação simbólica de modelos, entre outros. Em nosso trabalho adotamos planejamento, que será apresentado na próxima seção.

2.4 Planejamento

Planejamento é um processo abstrato de deliberação que escolhe e organiza ações através da antecipação de suas saídas esperadas. Essa deliberação tem como meta atingir objetivos preestabelecidos da melhor forma possível (GHALLAB; NAU; TRAVERSO, 2004).

Uma dada atividade requer deliberação quando ela remete novas situações, possui tarefas e objetivos complexos ou quando lida com ações menos familiares. Planejamento também é aplicado quando é necessário adaptar ações, por exemplo, em um ambiente crítico envolvendo riscos e custos elevados (GHALLAB; NAU; TRAVERSO, 2004).

A área tem progredido em eficiência e sofisticação dos seus algoritmos e seu potencial para aplicação a problemas reais. Hoje em dia, tais técnicas são usadas em logística e

gestão de crises, planejamento e programação de espaçonaves, configuração de equipamentos, planejamento de processos de fabricação, planejamento de evacuação, jogos de computador e robótica (GHALLAB; NAU; TRAVERSO, 2016). A seguir, detalhamos algumas técnicas de planejamento.

2.4.1 Técnicas de Planejamento

A seguir, apresentamos algumas técnicas para planejamento. Nessa seção, adotamos a formalização comumente adotada na área, desenvolvida por Ghallab *et al* (GHALLAB; NAU; TRAVERSO, 2004). Esta formalização adota algumas considerações sobre o ambiente na qual é aplicada a técnica de planejamento. A essas considerações, Ghallab *et al* atribui o nome de modelo conceitual restrito, que possui as seguintes características:

1. Possui um conjunto finito de estados.
2. É completamente observável, isto é, sabe-se completamente sobre o estado de transição.
3. É determinístico, isto é, se uma ação (ou evento) é aplicável a um estado, sua aplicação leva a um outro estado único.
4. É estático, isto é, o conjunto de eventos é vazio. O sistema não tem dinâmica interna.
5. O planejador manipula apenas objetivos restritos, que são especificados como um estado objetivo explícito ou conjunto de estados objetivo. O objetivo é qualquer sequência de ações que termina em um dos estados-objetivo.
6. Um plano para um problema de planejamento é uma sequência de ações finita linearmente ordenada.
7. Ações e eventos não têm duração. São transições de estado instantâneas.

O planejamento clássico é o tipo de planejamento que adota o modelo conceitual restrito. Caso uma ou mais dessas características sejam relaxadas, denominamos o planejamento como não clássico. Nas próximas seções, apresentamos a formalização de duas técnicas adotadas nesse trabalho: o planejamento clássico e planejamento hierárquico.

2.4.2 Planejamento Clássico

Planejamento clássico refere-se ao planejamento aplicado a sistemas de transição de estados que apresentam as propriedades do modelo conceitual restrito, apresentado na Seção 2.4.1. A importância dessa técnica de planejamento é que sua formalização serve como base para as demais técnicas.

A Figura 2 apresenta um sistema de planejamento. Ele é composto de três entidades:

Figura 2: Sistema de Planejamento

Fonte: Adaptado de (GHALLAB; NAU; TRAVERSO, 2004)

- O **Planejador**, que recebe como entrada o estado inicial, os objetivos e a descrição do sistema real (Σ). A partir dessas entradas, o planejador elabora o plano, que consiste em uma sequência de ações que permite atingir os objetivos.
- O **Controlador**, que é responsável por operar o sistema real. Ele realiza o controle através do plano fornecido pelo planejador. Adicionalmente, de tempos em tempos, observa dados do ambiente.
- O **Modelo do Sistema**, que recebe a sequência de ações do controlador e as executa.

Planejadores trabalham com auxílio de sistemas de transição de estado (HOPCROFT; MOTWANI; ULLMAN, 2006), que simulam o comportamento do sistema real. A definição 2.4.1 apresenta o conceito de um sistema de transição de estados determinístico, comumente adotado em planejamento clássico.

Definição 2.4.1 (Sistema de transição de estados determinístico). *Um sistema de transição de estados é representado por uma 3-tupla $\Sigma = \langle \mathcal{S}, \mathcal{A}, \delta \rangle$, onde:*

- $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ é um conjunto finito ou recursivamente enumerável de estados.
- $\mathcal{A} = \{a_1, a_2, \dots, a_n\}$ é um conjunto finito ou recursivamente enumerável de ações.
- $\delta: \mathcal{S} \times \mathcal{A} \rightarrow 2^{\mathcal{S}}$ é uma função de transição de estados tal que $|\delta(s, a)| \leq 1$.

No contexto de planeamento, estados representam possíveis configurações do ambiente. Ações representam acontecimentos que promovem mudança de estado. A Definição 2.4.2 apresenta o conceito de aplicabilidade de uma ação, que formaliza quando uma ação é aplicável a um determinado estado no planeamento determinístico.

Definição 2.4.2 (Aplicabilidade de uma ação em planeamento determinístico). *Uma ação a é aplicável a um estado s se e somente se $\delta(s, a) \neq \emptyset$. Se a é aplicável a um estado s , temos $\delta(s, a) = \{s'\}$, com $s' \in \mathcal{S}$.*

Informalmente, podemos dizer que dado como entrada um sistema de transição de estados, o estado inicial e um ou mais estados meta, podemos definir o problema de planeamento clássico. O *problema* consiste na descoberta de um *plano*, ou seja, uma sequência de ações que modificam o ambiente a ponto de atingirmos um estado meta. As definições 2.4.3 e 2.4.4 apresentam a formalização do conceito de problema de planeamento clássico e plano, respectivamente:

Definição 2.4.3 (Problema de planeamento clássico). *O problema de planeamento clássico é representado por uma 3-tupla $\mathcal{P} = \langle \Sigma, s_o, \mathcal{S}_g \rangle$, onde:*

- Σ é um sistema de transição de estados ou domínio do planeamento.
- s_o é um estado inicial.
- $\mathcal{S}_g$ é o conjunto de estados meta.

Definição 2.4.4 (Plano). *Plano é uma sequência de ações $\pi = \langle a_1, \dots, a_n \rangle$ que, se executadas a partir do estado inicial s_o , levam a um estado meta. Desta forma, temos: $\delta(s_o, a_1) = \{s_1\}, \delta(s_1, a_2) = \{s_2\}, \delta(s_2, a_3) = \{s_3\}, \dots, \delta(s_{n-1}, a_n) = \{s_n\}$, onde $s_n \in \mathcal{S}_g$.*

Os conceitos apresentados até o momento não esclarecem como é definido um estado. Existem várias formas de se fazer essa definição. Uma das mais adotadas é denominada **representação clássica** (GHALLAB; NAU; TRAVERSO, 2004), a qual trabalha com lógica de primeira ordem para representação dos estados. A definição 2.4.5 apresenta o conceito de estado na representação clássica.

Definição 2.4.5 (Estado na representação clássica). *A representação clássica de um estado é um conjunto de átomos totalmente instanciados, isto é, com termos compostos exclusivamente de constantes. Um átomo p é verdadeiro em um estado s se e somente se $p \in s$. Seja g um conjunto de literais, o estado s satisfaz g , ou $s \models g$, quando existe uma substituição σ tal que todo literal positivo de $\sigma(g) \in s$ e nenhum literal negativo de $\sigma(g) \in s$.*

De acordo com a definição, estados são descritos através de expressões em linguagem de primeira ordem $\mathcal{L}$. Cada estado do mundo é representado por um número finito de símbolos de predicado e constantes. As definições 2.4.6 e 2.4.7 apresentam o conceito de problema de planejamento adotando a representação clássica e o conceito de domínio de planejamento.

Definição 2.4.6 (Problema de planejamento com representação clássica). *Um problema de planejamento é uma 3-tupla $\mathcal{P} = \langle \Sigma, s_o, g \rangle$, onde:*

- Σ , é o sistema de transição de estados.
- s_o , estado inicial membro de $\mathcal{S}$.
- $g \subseteq \mathcal{L}$ é um conjunto de átomos que um estado deve satisfazer para ser um estado meta. O conjunto de estados meta é $\mathcal{S}_g = \{s \in \mathcal{S} \mid g \subseteq s\}$.

Definição 2.4.7 (Domínio do planejamento). *Seja $L = p_1, \dots, p_n$ um conjunto finito de átomos de lógica de primeira ordem, o domínio do planejamento é um sistema de transição de estados $\Sigma = \langle \mathcal{S}, \mathcal{A}, \delta \rangle$, tal que:*

- $\mathcal{S} \subseteq 2^L$, isto é, cada estado $s \in \mathcal{S}$ é composto por um sub-conjunto de L , onde dizemos que p é verdadeiro se e somente se $p \in s$, e se $p \notin s$ então p é falso no estado do mundo representado por s (hipótese do mundo fechado).
- Cada ação $a \in A$ é uma 3-tupla de sub-conjuntos de L , denotada por $a = \langle \text{precond}(a), \text{efeitos}^-(a), \text{efeitos}^+(a) \rangle$. $\text{precond}(a)$ representa o conjunto de pré-condições de a e os conjuntos $\text{efeitos}^-(a)$ e $\text{efeitos}^+(a)$ são chamados de efeitos de a . Estes dois conjuntos são disjuntos, isto é, $\text{efeitos}^-(a) \cap \text{efeitos}^+(a) = \emptyset$. A ação a é aplicável a um estado s se $\text{precond}(a) \subseteq s$.
- A função de transição de estados é $\delta(s, a) = \{(s - \text{efeitos}^-(a)) \cup \text{efeitos}^+(a)\}$, se $a \in A$ e é aplicável a $s \in \mathcal{S}$. Caso contrário, $\delta(s, a)$ é indefinida.

- S possui a propriedade de que se $s \in S$, então, para cada ação a aplicável a s , $\delta(s, a) \in 2^S$, isto é, a aplicação de uma ação a um estado sempre leva a um outro estado válido.

Na representação clássica, generalizamos o conceito de ação através do conceito de operador de planejamento. Na verdade, uma ação nada mais é do que uma instanciação de um operador de planejamento. As definições 2.4.8 e 2.4.9 formalizam os conceitos de operador de planejamento e ação de planejamento, respectivamente.

Definição 2.4.8 (Operador de planejamento clássico). *Um operador de planejamento é uma 3-tupla $o = \langle \text{nome}(o), \text{precond}(o), \text{efeitos}(o) \rangle$, cujos elementos são:*

- $\text{nome}(o)$, o nome do operador, é uma expressão sintática da forma $n(x_1, \dots, x_k)$, onde n é um símbolo de operador, $x_1, \dots, x_k$ são os símbolos variáveis que aparecem em qualquer parte de o e n é único (não existem dois operadores com o mesmo nome);
- $\text{precond}(o)$ e $\text{efeitos}(o)$ são as pré-condições e efeitos de o , respectivamente, e são generalizações de uma ação.

Definição 2.4.9 (Ação em planejamento clássico). *Uma ação é qualquer instanciação de um operador de planejamento. Se a é uma ação e s é um estado tal que $\text{precond}^+(a) \subseteq s$ e $\text{precond}^-(a) \cap s = \emptyset$, então a é aplicável a s , e o resultado da aplicação de a a s é o estado:*

$$\delta(s, a) = \{((s - \text{effects}^-(a)) \cup \text{effects}^+(a))\}. \quad (2.1)$$

Os conceitos apresentados podem ser adotados no âmbito de planejadores não determinísticos. Para isso, basta relaxarmos a nossa restrição de transição de estado de $|\delta(s, a)| \leq 1$ para $|\delta(s, a)| \leq |2^S|$. A seguir, apresentamos o conceito de domínio de planejamento não determinístico equivalente, definido por Kuter (KUTER, 2006).

Definição 2.4.10 (Domínio de planejamento não determinístico equivalente). *Considere $\Sigma = (S, \mathcal{A}, \delta)$ o domínio do planejamento clássico. Então o domínio do planejamento não determinístico equivalente $\Sigma' = (S', \mathcal{A}', \delta')$ é uma versão não-determinística de Σ , se os seguintes pontos são verdadeiros:*

- $S = S'$;

- Existe um mapeamento um a um denominado *det* de $\mathcal{A}'$ para $\mathcal{A}$, tal que os seguintes pontos são verdadeiros:

- Para cada estado $s \in \mathcal{S}$, se $\delta'(s, a') = \emptyset$ então $\delta(s, \mathbf{det}(a')) = \emptyset$
- caso contrário, $\delta(s, \mathbf{det}(a')) \in \delta'(s, a')$.

Dessa forma temos que no planejamento não determinístico, a aplicação de uma determinada ação a' , pode levar o sistema a n diferentes estados, onde $n \leq |2^{\mathcal{S}}|$. No entanto, o conceito de domínio de planejamento não determinístico equivalente sinaliza que esse mesmo comportamento, pode ser obtido através de um planejador determinístico que apresenta ações determinísticas $a_1, a_2, \dots, a_n$ que desenvolvem transições de estado equivalentes às de a' .

2.4.3 Planejamento Hierárquico

O planejamento de redes hierárquicas de tarefas, também conhecido como HTN (acrônimo de Hierarchical Task Network), é um tipo de planejamento que adota heurísticas específicas de domínio para reduzir o espaço de busca de planos (GHALLAB; NAU; TRAVERSO, 2004). Baseia-se na introdução de níveis de abstração com o intuito de reduzir a complexidade da tarefa de planejamento (RUSSELL, 2010). O plano inicial consiste em uma descrição de alto nível do que deve ser feito. Os planos são refinados por meio de decomposições de ações. Cada decomposição de ação, reduz uma ação de alto nível a um conjunto parcialmente ordenado de ações de nível mais baixo. O processo continua até restarem somente ações primitivas no plano.

A Figura 3 apresenta um exemplo de aplicação de HTN. Partindo-se do plano inicial *Construir Casa*, que possui como pré-condição a existência de um *Terreno* e como pós-condição uma *Casa*, podemos decompor o plano nas subtarefas *Obter Licença*, *Contratar construtor*, *Construção*, *Pagar construtor*. Adicionalmente, percebe-se uma mudança nas pré-condições.

O planejamento hierárquico é similar ao planejamento clássico, no qual o estado do mundo é representado por um conjunto de átomos e cada ação corresponde a uma transição de estados. No entanto, diferem-se do planejamento com relação ao que eles planejam e como eles planejam.

Em planejamento hierárquico, o objetivo não é alcançar um conjunto de metas, mas executar alguns conjuntos de tarefas. A entrada do sistema de planejamento hierárquico adota um conjunto de operadores, similares ao de planejamento clássico, e também um

Figura 3: Exemplo de Decomposição de Tarefas.

Fonte: Adaptado de (RUSSELL, 2010).

conjunto de métodos. Um método é uma receita de como decompor alguma tarefa em um conjunto de sub-tarefas. O planejamento prossegue decompondo tarefas não primitivas recursivamente em tarefas menores, até que todas as tarefas se tornem primitivas. Nesse ponto, as tarefas primitivas são realizadas por meio dos operadores de planejamento.

A principal razão por trás do sucesso da HTN é que suas redes de tarefas capturam conhecimento útil de controle de procedimentos, ou seja, conhecimento sobre como executar uma tarefa, descritos através de decomposições de subtarefas. Esse conhecimento de controle reduz significativamente o espaço de pesquisa de um plano e, ao mesmo tempo, garante que os planos sigam um dos cursos de ação estipulados (SOHRABI; MCILRAITH, 2008).

As definições de operadores, ações, planos e a função δ de transição de estados é a mesma de planejamento clássico. Os novos conceitos introduzidos pelo planejamento hierárquico são o de tarefa, redes de tarefas e métodos.

Definição 2.4.11 (Tarefa). *Uma tarefa é uma expressão da forma $t(r_1, \dots, r_k)$, tal que t é um símbolo de tarefa e $r_1, \dots, r_k$ são termos. Uma tarefa é primitiva quando t é um operador. Caso contrário, a tarefa é não primitiva. Uma tarefa é instanciada se todos os termos que compõem essa tarefa são instanciados, caso contrário a tarefa é não instanciada.*

Definição 2.4.12 (Realização de tarefa primitiva). *Uma ação $a = (\text{nome}(a), \text{precond}(a), \text{effects}(a))$ realiza uma tarefa primitiva t em um estado s se $\text{nome}(a) = t$ e a é aplicável a s .*

Definição 2.4.13 (Rede de tarefas). *Uma rede de tarefas é um dígrafo acíclico $w = (U, C)$, onde U é um conjunto de tarefas e C é um conjunto de restrições. Dizemos que a*

rede de tarefas w é instanciada quando formada apenas por tarefas instanciadas. Caso contrário, w não é instanciada. w é primitiva se todas as tarefas de w são primitivas, caso contrário w é não primitiva.

Uma restrição em C especifica um requisito que deve ser satisfeito por todo plano que é uma solução para um problema de planejamento (GHALLAB; NAU; TRAVERSO, 2004). São dois os tipos de restrições:

- **restrições de precedência:** restringem a ordem que as tarefas podem ser executadas.
- **restrições sobre estados:** restringem os estados que um determinado literal é verdadeiro.

Definição 2.4.14 (Método hierárquico). *Um método hierárquico é uma 4-tupla $m = (\text{nome}(m), \text{tarefa}(m), \text{subtarefas}(m), \text{constr}(m))$, em que os elementos são descritos como seguem:*

- $\text{nome}(m)$ é uma expressão da forma $n(x_1, \dots, x_k)$, onde n é um símbolo de método único, e $x_1, \dots, x_k$ são todas variáveis que ocorrem em algum lugar em m .
- $\text{tarefa}(m)$ é uma tarefa não primitiva.
- $(\text{subtarefas}(m), \text{constr}(m))$ representa uma rede de tarefas.

A seguir, apresentamos o conceito de decomposição de tarefas. A decomposição de tarefas é um dos conceitos fundamentais nessa técnica de planejamento, pois a partir dela, tarefas complexas são decompostas em tarefas mais simples até o ponto que encontramos a sequência de ações necessárias para realização do plano.

Definição 2.4.15 (Decomposição). *Seja $w = (U, C)$ uma rede de tarefas, $u \in U$ é um nó de tarefa, t_u é uma tarefa, m é uma instância de um método em M e $\text{tarefa}(m) = t_u$. Então m decompõe u em $\text{subtarefas}(m')$, obtendo a rede de tarefas: $\delta(w, u, m) = ((U - \{u\}) \cup \text{subtasks}(m'), C' \cup \text{constr}(m'))$, resultante de uma ordenação topológica de acordo com as restrições de C .*

A Figura 4 ilustra o funcionamento do planejamento hierárquico. Dado a rede de tarefa inicial, representada por tarefa_1 , são realizados uma série de decomposições, representadas pelas setas em azul, até o momento que o sistema apresente apenas tarefas primitivas (ações). Em cada decomposição, há a presença de uma seta horizontal tracejada.

Ela indica uma restrição de ordem existente entre as tarefas oriundas da decomposição. Os nós folhas da árvore representam o plano, ou seja, a sequência de ações necessária para resolução do problema.

Figura 4: Decomposição de tarefas em planejamento hierárquico

Fonte: Próprio autor

A seguir apresentamos o conceito de domínio e problema de planejamento no âmbito de planejamento hierárquico.

Definição 2.4.16 (Domínio de Planejamento hierárquico). *Um domínio de planejamento hierárquico é um par $\mathcal{D} = (O, M)$, onde O é um conjunto de operadores e M é um conjunto de métodos HTN.*

Definição 2.4.17 (Problema de Planejamento hierárquico). *Um problema de planejamento hierárquico é uma 4-tupla $\mathcal{P} = (s_o, w, O, M)$, onde s_o é o estado inicial, w é a rede de tarefas inicial, O é um conjunto de operadores e M é o conjunto de métodos HTN.*

2.5 Modelagem Formal

Especificações formais usam notação matemática para descrever com precisão as propriedades que um sistema de informação deve ter. Elas não interferem na forma como essas propriedades são alcançadas. Especificações formais descrevem o que o sistema deve fazer sem dizer como deve ser feito. Essa abstração torna as especificações formais úteis para o desenvolvimento de aplicações, pois permitem responder com confiança questões

sobre a funcionalidade do sistema, sendo desnecessário coletar informações do código fonte ou de modelos informais do sistema (SMITH, 2012).

Object-Z é uma linguagem usada para fazer esse tipo de descrição. Object-Z surgiu em 1988 e teve como motivação o desejo de investigar a integração de técnicas formais com a metodologia de orientação a objetos (SMITH, 2012). A linguagem aprimora os elementos de estruturação da linguagem de especificação Z (SPIVEY, 1998). Nas próximas seções, apresentamos alguns conceitos necessários para o entendimento da especificação formal proposta nesse trabalho.

2.5.1 Classes, esquemas e tipos

Para apresentar os principais conceitos da linguagem Object-Z, iremos adotar o exemplo de cartão de crédito presente no livro de Duke *et al* (DUKE; ROSE, 2000).

Object-Z apresenta uma notação que adota caixas para representar diferentes conceitos de um sistema. Caixas são conhecidas como esquemas e são usadas para descrever os aspectos estáticos e dinâmicos do sistema. Em Object-Z, uma classe é um esquema rotulado, cujo rótulo designa seu nome. A Figura 5 apresenta um exemplo de definição de classe em Object-Z.

A classe *CartaoDeCredito* representa um exemplo de definição de um cartão de crédito bancário e suas possíveis operações. Ela é composta pela *lista de visibilidade*, designada pelo operador $\vdash$. Ela especifica que a constante *limite*, a variável de estado *saldo*, o esquema de estado *INIT* e as operações *Saque*, *Deposito* e *SaqueDisponivel* estão na interface da classe. A lista de visibilidade de uma classe Object-Z possui os seguintes elementos: constantes, variáveis de estado, o esquema de estado *INIT* e as operações da classe. Caso a lista de visibilidade seja omitida, todos os elementos que compõe a classe estão em sua interface.

A definição de uma constante é feita através de uma caixa de esquema aberta, denominada **definição axiomática**. Em uma definição axiomática, acima da linha, é apresentada uma lista de declarações e abaixo, uma lista de fórmulas expressas em lógica de primeira ordem. A Figura 5 apresenta a definição da constante *limite* e a declara como sendo um número natural. Em seguida, define o domínio da constante *limite* através do conjunto $\{1000, 2000, 5000\}$. No âmbito dessa classe, *limite* indica que instâncias de *CartaoDeCredito* podem apresentar diferentes limites. No entanto, o *limite* de uma instância de *CartaoDeCredito* não pode ser modificado e seu valor é 1000, 2000 ou 5000.

Figura 5: Esquema de Classe CartaoDeCredito

Fonte: Adaptado de (DUKE; ROSE, 2000).

O segundo esquema define o **esquema de estado** e é representado por uma caixa não rotulada. O esquema de estado apresenta o mesmo padrão da definição de constante, ou seja, elementos acima da linha representam declarações e abaixo, predicados. No esquema de estado, os elementos declarados correspondem a variáveis de classe e o predicado apresenta restrições a essas variáveis. No exemplo da Figura 5, definimos a variável *saldo* e a declaramos como um número inteiro. Adicionalmente, declaramos a restrição de que a soma entre o *saldo* e o *limite* seja maior ou igual a zero. Dessa forma, restringimos que *saldo* seja mais negativo que o *limite* do cartão do crédito.

O terceiro esquema, refere-se ao **esquema INIT**. Diferente de outros esquemas, ele apresenta apenas a definição de predicados. O esquema *INIT* representa o estado inicial da classe, ou seja, define os valores que as variáveis de estado devem assumir quando um objeto dessa classe é criado. O exemplo da Figura 5 sinaliza que todas as contas associadas a um cartão de crédito possuem inicialmente *saldo* igual a 0.

Os demais esquemas são operações. **Esquemas de operação** definem os mecanismos que permitem um objeto mudar de estado. O esquema começa com a lista Δ que mostra todas as variáveis de estado que são alteradas pelo esquema. Quaisquer variáveis não listadas permanecerão inalteradas quando um objeto realizar a operação. Após a

declaração da lista Δ , temos a declaração das variáveis de comunicação. Uma variável de comunicação é definida através de um rótulo e apresenta um tipo associado. Ela apresenta o sufixo *?* ou *!* para sinalizar que é uma variável de entrada ou saída, respectivamente. Por último, são definidas fórmulas no predicado para especificar restrições às variáveis de estado e aos parâmetros de comunicação antes e após a operação.

No exemplo da Figura 5, temos as operações *Saque*, *Deposito* e *SaqueDisponível*. As três operações modificam a variável *saldo*. A operação *Saque* apresenta a variável de comunicação de entrada *quantia?*. Um *Saque* só é realizado quando a soma entre *limite* e *saldo* é maior do que a *quantia* desejada. Caso a condição seja verdadeira, o novo *saldo* (representado por *saldo'*) é obtido pela subtração de *saldo* por *quantia?*. A operação *Deposito* é análoga a anterior, no entanto, o resultado dessa operação é a adição de *quantia?* a *saldo*. Por último, *SaqueDisponível* representa uma operação que realiza o saque total do valor disponível em conta. Ela possui uma variável de comunicação de saída *quantia!* que armazena o valor total debitado.

2.5.2 Objetos e Herança

Object-Z dá suporte a herança e a definição de objetos. O exemplo a seguir ilustra cada um desses recursos. A Figura 6 apresenta a classe *CartaoEspecial* que representa uma solução para ampliação de crédito. *CartaoEspecial* é uma especialização da classe *CartaoDeCredito*. O mecanismo de herança em Object-Z consiste em referenciar uma ou mais classes pai após a lista de visibilidade. A subclasse apresenta todos elementos da classe pai e pode modificar e adicionar novos elementos.

No esquema de estado, a classe *CartaoEspecial* apresenta a variável de estado *cartaoVirtual*, que é um objeto do tipo *CartaoDeCredito*. *CartaoEspecial* também apresenta uma *variável de estado secundária*, denominada *saldoTotal*. Variáveis secundárias são variáveis declaradas no esquema de estado, logo após o símbolo Δ . Elas recebem esse nome, pois são definidas em função das demais variáveis de estado, denominadas de *variáveis primárias*. Em *CartaoEspecial*, o valor da variável secundária *saldoTotal* é definido a partir da variável primária *saldo* e do atributo *saldo*, membro da variável primária *cartaoVirtual*.

Em seguida, é definido o esquema *INIT*. *CartaoEspecial* apresenta a mesma configuração inicial de *CartaoDeCredito* e adiciona o objeto *cartaoVirtual* em sua configuração inicial.

Figura 6: Esquema de Classe CartaoEspecial

Fonte: Adaptado de (DUKE; ROSE, 2000).

As próximas linhas são dedicadas as definições das operações *SaqueEspecial* e *DepositoVirtual*. Nesses exemplos, definimos ambas operações através de expressões. O símbolo $\hat{=}$ pode ser lido como "é definido como". Dessa forma, temos que a operação *DepositoVirtual* é definida como a operação *Deposito* do objeto *cartaoVirtual*. Já *SaqueEspecial* é definida a partir da operação *Saque* da classe *CartaoDeCredito* e da operação *Saque* do objeto *cartaoVirtual*. O operador $[]$, é denominado *operador de escolha*, e significa que *SaqueEspecial* é realizado a partir da escolha da operação *Saque* da classe *CartaoDeCredito* ou *Saque* do objeto *cartaoVirtual*. Se ambas operações estão aptas a serem aplicadas, o operador de escolha aplica apenas uma delas.

2.5.3 Agregação de Objetos

Quando se deseja modelar sistemas maiores com um número variável de objetos, é comum usar a notação de conjuntos. A Figura 7 apresenta um exemplo dessa notação. *CartoesDeCredito* é uma classe que apresenta uma variável de estado *cartoes*, cujo o tipo $\mathbb{P} \text{CartaoDeCredito}$ sinaliza que essa variável é um conjunto de objetos do tipo *CartaoDeCredito*. No esquema de estado, é apresentado uma restrição a esse conjunto que diz que para todo objeto *c* do conjunto *cartoes*, o atributo limite de *c* é igual ao valor da constante *limiteComum*.

Em seguida, é definido o esquema *INIT*. *INIT* indica que inicialmente, o valor da variável de estado *cartoes* é igual ao conjunto vazio. A classe *CartoesDeCredito* apresenta quatro operações: *Adicionar*, *Remover*, *Deposito* e *Saque*. As duas primeiras

Figura 7: Esquema de Classe CartoesDeCredito

Fonte: Adaptado de (DUKE; ROSE, 2000).

modificam a variável de estado *cartoes*. A operação *Adicionar* insere o objeto *cartao?* ao conjunto *cartoes*. Essa operação é aplicada caso o objeto *cartao?* não pertença ao conjunto *cartoes*. De maneira análoga, foi definida a operação *Remover*, que retira o objeto *cartao?* do conjunto *cartoes*. *cartao?* só é removido no caso dele não pertencer ao conjunto *cartoes*.

Por último, são definidos os operadores *Saque* e *Deposito*. O operador *Saque* recebe como entrada *cartao?* que deve pertencer ao conjunto *cartoes*. Em seguida, o operador *Saque* do objeto *cartao?* é aplicado. De maneira análoga, é descrita a operação *Deposito* que faz uso da operação *Deposito* do objeto *cartao?*.

2.6 Conclusões

Esse capítulo apresentou a fundamentação teórica necessária para compreensão dos demais capítulos dessa tese. A abordagem proposta nesse trabalho, visa resolver problemas relacionados a sistemas de IoT, tal como a interoperabilidade entre os componentes do sistema de IoT e a escalabilidade dos serviços presentes nesse sistema.

Em um primeiro momento, apresentamos conceitos inerentes a sistemas de IoT, e a evolução desses sistemas a uma IoT semântica, a qual atribui significado às coisas, de modo a proporcionar computacionalmente, meios para realizar inferência de conhecimento.

A atribuição de significado é feita através de linguagens de especificação semântica, que apresentam grande flexibilidade para atribuição de anotações. Nesse contexto, apresentamos WSML, uma linguagem de especificação semântica que se destaca por apresentar construções específicas para a modelagem de serviços e metas do usuário.

Em geral, um sistema de IoT apresenta grande diversidade de serviços, representados pelos *sensores*, *atuadores*, *middlewares* e demais componentes de software que o constitui. Dessa forma, chamamos a atenção para dois elementos úteis a esse tipo de sistemas. São eles, o módulo de descoberta de serviços e o módulo de composição automática de serviços.

O módulo de descoberta é responsável por buscar o serviço mais apto a atender a demanda do usuário. Já o módulo de composição automática pode ser usado para prover novos serviços, a partir da composição de outros serviços preexistentes. Uma das técnicas que podem ser empregadas para implementação desse segundo é o uso de planejamento.

Planejamento é uma técnica de inteligência artificial, que tem como objetivo a elaboração de planos a partir de um problema, denominado de problema de planejamento. Técnicas de planejamento se destacam por apresentar bom desempenho computacional e pela sua flexibilidade. Dessa forma, a escolhemos para solução de nosso problema de trabalho.

Por último apresentamos brevemente a linguagem de especificação formal Object-Z, que será adotada no próximo capítulo, para formalização do modelo conceitual adotado por nosso *framework*.

3 FORMALIZAÇÃO DO PROBLEMA

Neste capítulo, apresentamos o domínio adotado em nossa abordagem para representação de um sistema de *IoT*. O conjunto de conceitos desenvolvidos servem como base para o *framework SWoTPAD* e para a linguagem *SWoTPADL*, que dá suporte ao desenvolvimento de descrições de serviços e requisições semânticas.

As entidades adotadas em nosso trabalho são apresentadas através de um modelo conceitual. A técnica de modelagem conceitual é amplamente empregada na Ciência da Computação para o desenvolvimento de especificações precisas e de alta qualidade.

Esse capítulo apresenta a seguinte organização: na Seção 3.1 apresentamos uma visão geral do modelo conceitual adotado por esse trabalho para representar sistemas de *IoT*. Em seguida, na Seção 3.2, detalhamos esse modelo, através de sua especificação formal em Object-Z.

3.1 Modelo Conceitual para Sistemas de *IoT*

A literatura apresenta algumas propostas de modelos referências para internet das coisas. O modelo de referência RAMI (ADOLPHS et al., 2015) adiciona detalhes de fabricação e logística a *IoT*. O modelo de referência IIRA (LIN et al., 2015) tem um forte foco no setor industrial. O modelo *IoT-A* (BAUER et al., 2013) fornece uma visão detalhada dos aspectos da tecnologia da informação da *IoT*. Inspirada nessas propostas, desenvolvemos nosso modelo conceitual para sistemas de *IoT*.

O modelo conceitual descrito em UML na Figura 8, representa a visão de Sistemas de *IoT* adotada por nosso *framework*. Ele apresenta como elementos principais o Ambiente, o Serviço, o Ator, a Requisição e o Instrumento e um conjunto de conceitos acessórios que permitem descrever elementos comumente adotados em sistemas de *IoT*.

Figura 8: Modelo Conceitual - Sistema de IoT

Fonte: Próprio autor

O Ambiente é o conceito central, atuando como uma espécie de *container*. Em nosso modelo, um ambiente pode ser composto por outros ambientes e agrega todos os demais entidades presentes no modelo conceitual. Ao Ambiente, podemos associar um conjunto de propriedades temporais, espaciais, bem como um conjunto de preferências definidas pelo projetista do sistema.

Dispositivos e serviços Web representam os serviços fornecidos pelo ambiente. Podem possuir propriedades, que representam características não funcionais do serviço. Adicionalmente, apresentam habilidades, que correspondem à descrição semântica de suas precondições e pós-condições. Sua funcionalidade é descrita por meio do comportamento, que descreve de forma imperativa, como a habilidade é desenvolvida pelo serviço.

Através de requisições, uma entidade pode utilizar serviços. Uma requisição representa uma habilidade exigida por um Ator. O sistema prevê dois tipos de atores: agentes computacionais autônomos, representados pelo conceito de agente; humanos e robôs, representados pelo conceito de manipuladores de instrumento. Em nosso *framework*, um instrumento corresponde a qualquer objeto do ambiente que apresenta uma habilidade, mas não é disponível como serviço. Um extintor de incêndio, um martelo e um alicate

são exemplos de objetos. No momento que um Manipulador de Instrumento carrega um Objeto, ele herda as habilidades do objeto.

A Habilidade representa características associada a serviços, objetos e atores. Representa a capacidade de desencadear mudança no ambiente. Exemplos de habilidades são apresentadas a seguir: um instrumento extintor, tem a habilidade de apagar um incêndio; o serviço lampadas possui habilidade de acender luzes, etc. Habilidades são categorizadas por meio de Capacidades, que permite definir taxonomias das Habilidades presentes no sistema.

Reutilizamos ontologias para definir os elementos desse modelo conceitual. Nós encapsulamos ontologias por meio de propriedades. Uma propriedade é definida por conceitos, axiomas, instâncias e relações e podem ser associadas a cada uma das entidades do *framework*. Para dispositivos, adotamos os conceitos apresentados pelas ontologias SSN (COMPTON et al., 2012) e IoT-lite (BERMUDEZ-EDO et al., 2016). Em relação a Ambientes, utilizamos a Ontologia Accommodations (HEPP, 2013), que oferece um vasto conhecimento sobre acomodações internas. As propriedades e preferências temporais usam o W3C Time Ontology (HOBBS; PAN, 2006) e o RECommendations Ontology (FERRIS; JACOBSON, 2010), respectivamente. A definição dos Atores usam a ontologia FOAF (BRICKLEY; MILLER, 2007). Na próxima seção formalizamos cada um desses conceitos.

3.2 Especificação Formal em Object-Z

Nessa seção, apresentamos a especificação em Object-Z das entidades presentes no modelo conceitual da Seção 3.1. Para isso, adotamos o modelo de classe Object-Z ilustrado na Figura 9.

Cada entidade foi mapeada para uma classe Object-Z. Cada atributo de entidade do modelo conceitual foi mapeada para uma variável primária de estado. Os predicados foram utilizados para representar a semântica estática do modelo conceitual. A semântica dinâmica foi modelada através dos operadores. Também foram adicionadas variáveis de estado secundárias com o intuito de melhor estruturar os atributos da classe e os atributos das variáveis de estado primária.

Algumas novas classes e tipos foram introduzidos, com o intuito de agregar mais detalhes ao modelo conceitual, promover o reuso, bem como prover maior rigor formal. Adicionalmente, elementos simples ou que já apresentam formalização amplamente aceita

Figura 9: Modelo de classe Object-Z usada para formalização do Modelo Conceitual

Fonte: Próprio autor

na literatura, foram definidos como tipo básico. É o caso de ontologia (*ONTOLOGIA*), identificador (*ID*) e identificador de variável (*VID*), apresentados na Figura 10. Em Object-Z, a definição de um tipo básico é feita introduzindo o tipo básico entre colchetes. Os nomes assim definidos, se tornam parte do vocabulário global de tipos básicos e podem ser utilizados em qualquer ponto da especificação.

Figura 10: Esquema de Classe EntidadeSWoTPAD

Fonte: Próprio Autor.

A classe base de nosso modelo formal é *EntidadeSWoTPAD*. Todas as entidades presentes no modelo conceitual herdam de *EntidadeSWoTPAD* que define elementos comuns as classes do modelo conceitual. *EntidadeSWoTPAD* apresenta como atributos *nome* e *ontoImps*, que designa, respectivamente, o identificador da Entidade e o conjunto de ontologias importadas para a definição da entidade. Como variável secundária, *EntidadeSWoTPAD* apresenta *totalOntologias* que armazena todas as ontologias importadas, incluindo também as ontologias importadas pelas ontologias importadas por *EntidadeSWoTPAD*.

A seguir, apresentamos a definição formal dos elementos que aparecem no Diagrama da Figura 8.

Definição 3.2.1 (Ambiente). *Um ambiente é um conceito representado por uma 9-tupla (*nome*, *ServicoWeb*, *Dispositivo*, *Ator*, *Objeto*, *Propriedade*, *SubAmbiente*, *Requisição*, *localizacao*), onde:*

- *nome*: identificador do ambiente
- *ServicoWeb*: conjunto das instâncias de todos os serviços Web presentes no ambiente.
- *Dispositivo*: conjunto das instâncias de todos os dispositivos presentes no ambiente.
- *Ator*: conjunto das instâncias de todos os atores presentes no ambiente.
- *Instrumento*: conjunto das instâncias de todos os instrumentos presentes no ambiente.
- *Propriedade*: conjunto de todas as propriedades do ambiente.
- *SubAmbiente*: conjunto das instâncias de todos os sub ambientes que compõem o ambiente.
- *localizacao*: localização do Ambiente.

A formalização da entidade *Ambiente* é apresentada na Figura 11. A classe *Ambiente* mapeia todos os atributos presentes em seu modelo conceitual como variáveis de estado. Adicionalmente, apresenta as variáveis secundárias *totalAtores*, *totalDispositivos*, *totalServicosWeb*, *totalInstrumentos* e *totalAmbientes*. A variável secundária *totalAtores* representa o conjunto formado por todos os atores que estão no ambiente e nos subambientes. A restrição que descreve essa característica esta presente na primeira linha de predicados. Nela, vemos que *totalAtores* é definido a partir da união de *atores* com cada um dos *atores* presentes em *subAmbientes*. Observem o uso dos operadores de união $\cup$ e $\bigcup$. O operador $\cup$ é utilizado para fazer a união entre dois conjuntos. Já o operador $\bigcup$ é usado para fazer a união entre vários conjuntos. No exemplo, $\bigcup$ é usado para fazer a união de todos os atores presentes em cada objeto do tipo *Ambiente* que compõe o conjunto *subAmbientes*.

Adicionalmente, a classe *Ambiente* apresenta restrições aos elementos que compõem os atributos dos conjuntos *atores*, *instrumentos*, *subAmbientes*, *servicosWeb* e *dispositivos*. Como cada um desses elementos são de tipos que apresentam o atributo

Figura 11: Esquema de Classe Ambiente

Fonte: Próprio autor.

localização, a restrição indica que o *Ambiente* pertence a sua *localizacao*. Por exemplo, a declaração $\forall a : atores \bullet self \in a.localizacao$ indica que para todo *Ator* *a* do conjunto *atores*, o atributo *localizacao* do ator *a* terá *Ambiente* como membro. A referência ao *Ambiente* é feita através da palavra reservada *self*, que em Object-Z representa uma auto-referência ao objeto.

A classe *Ambiente* também é formada por um conjunto de Operações. *AdicionarDispositivo* é uma operação usada no momento em que um novo dispositivo é instalado no *Ambiente*. Ela apresenta uma variável de entrada *dispositivo?* que representa o novo dispositivo a ser inserido no ambiente. Como se trata de um novo dispositivo, inicialmente ele não pertence ao conjunto de *dispositivos*. Sua *localização* passa ser a do ambi-

ente, assinalado por *dispositivo?.localizacao=self* e o *dispositivo* é inicializado. Por último, ele é adicionado ao conjunto *dispositivos*. A operação *RemoverDispositivo* desinstala o dispositivo do ambiente. Como consequência o *disposito* é removido do conjunto *dispositivos* e sua localização é modificada para *null*. De maneira análoga, também são definidas as operações *AdicionarServicoWeb*, *RemoverServicoWeb*, *AdicionarAtor*, *RemoverAtor*, *AdicionarInstrumento*, *RemoverInstrumento*, *AdicionarPropriedade*, *RemoverPropriedade*, *AdicionarSubAmbiente*, *RemoverSubAmbiente*. Para manter o esquema conciso, não apresentamos essas operações na Figura 11.

Definição 3.2.2 (Serviço). *Serviço é um conceito que pode representar um dispositivo ou serviço Web. Ele é representado por uma 8-tupla (nome, tipo, OntID, Habilidade, Propriedade, comportamento, localização, cobertura), onde:*

- *nome*: é o identificador do serviço.
- *tipo*: representa o tipo de serviço que pode ser realizado por um device ou por um Web Service.
- *OntID*: é o conjunto dos identificadores das ontologias importadas.
- *Propriedade*: é o conjunto de propriedades do Serviço.
- *Habilidade*: representa o conjunto de habilidades do Serviço.
- *comportamento*: representa a forma adotada pelo serviço para
- *localização*: localização do Serviço
- *cobertura*: área de atuação do Serviço

A Figura 12 apresenta o esquema relativo a classe *Servico*. Conforme apresentado no modelo conceitual, um serviço apresenta os seguintes atributos: *propriedades*, *habilidades*, *comportamento*, *localização* e *cobertura*. Adicionalmente, apresenta o atributo *tipo* que indica se o serviço em questão é um serviço de software ou de hardware (*dispositivo*). Como restrição, a habilidade desenvolvida pelo serviço se associa ao serviço através do campo *usadoPor*.

A classe serviço também apresenta algumas operações. Na Figura 12, apresentamos três delas. A operação *AdicionarCobertura* adiciona um ambiente a cobertura de atuação do serviço. A operação recebe como variável de entrada o *ambiente?* e adiciona

Figura 12: Esquema de Classe Servico

$TipoServico ::= ServicoWeb \mid Dispositivo$

Fonte: Próprio Autor.

o ambiente ao conjunto cobertura. De maneira análoga, a operação *RemoverCobertura* recebe como variável de entrada o *ambiente?* e o remove do conjunto cobertura presente em *Servico*. Métodos similares são definidos para propriedade (*AdicionarPropriedade*, *RemoverPropriedade*).

Por último, temos *Executar*. A operação *Executar* tem o papel de realizar a habilidade apresentada pelo serviço. A operação *Executar*, utiliza em sua definição dois operadores. O operador de composição sequencial §, que indica a sequência de execução das operações e o operador de escolha [], que opta pela operação *habilidade.PreVerdadeira*, caso as precondições para a realização do serviço sejam atendidas ou a operação *habilidade.PreFalsa*, caso contrário. Caso as precondições sejam atendidas, é realizada a operação *Executar* da classe *Comportamento* em sua área de cobertura. Em seguida, as poscondições do serviço se tornam verdadeiras. Para ratificar esse comportamento, é chamada a operação *habilidade.PosVerdadeira*, que deve retornar *true*.

Definição 3.2.3 (Ator). *Um ator é um conceito representado por uma 7-tupla (nome, tipo, Propriedade, Habilidade, Objeto, localização, Requisicao)*

- *nome*: identificador do ator.
- *tipo*: tipo do ator que pode ser humano, robo, animal de estimação ou agente.

- *Propriedade*: conjunto de propriedades do ator.
- *Habilidade*: é o conjunto de habilidades do ator.
- *Instrumento*: é o conjunto de instrumentos em propriedade do ator em um dado momento. Quando um instrumento está em propriedade de um ator, ator herda as habilidades do instrumento.
- *localização*: localização do ator.
- *Requisicao*: é o conjunto de requisições do ator.

A Figura 13 apresenta a definição associada a entidade *Ator*. Em nosso modelo conceitual, apresentamos três tipos de atores: *Agente*, *Humano* e *Robo*, definidos em Object-Z através da definição do tipo *TipoAtor*. Adicionalmente, a classe *Ator* é composta pelas *propriedades* do *Ator*, suas *habilidades*, *instrumentos* que estão em seu poder, *requisicoes* que o ator pode fazer ao sistema de IoT e a *localizacao* do *Ator*. Como variável secundária, *Ator* apresenta a variável *totalHabilidades*, cuja restrição a define como um conjunto formado pela união das *habilidades* do *Ator* e as *habilidades* de cada *instrumento* que o *Ator* possui. Caso ator seja do tipo *Agente*, *totalHabilidades* apresenta apenas as habilidades do *Ator*, pois o conjunto *instrumentos* é sempre vazio. Por último, os campos *usadoPor* de cada *habilidade* e *instrumento* é associado ao *Ator*.

Na classe *Ator*, não apresentamos as operações comuns de adição e remoção, tais como *AdicionarPropriedade*, *RemoverPropriedade*, *AdicionarHabilidade*, *RemoverHabilidade*, *AdicionarInstrumento*, *RemoverInstrumento*, *AdicionarInstrução* e *RemoverInstrução*, pois apresentam implementação idêntica ao das demais classes apresentadas. Aqui, nos restringimos a apresentar as operações que se diferem das demais, que são as operações *Executar* e *Requisitar*.

Embora também presente na classe *Servico*, a operação *Executar* de *Ator* difere por não possuir um *Comportamento* associado. Isso acontece, pois em *Ator*, a variável *totalHabilidades* está relacionada a ações que o *Ator* pode desenvolver sozinho ou com auxílio de um *Instrumento* no *Ambiente*. Dessa forma, o comportamento desenvolvido não apresenta código fonte correspondente no sistema de IoT final. Apesar disso, a operação *Executar* devolve o nome da *Habilidade* almejada, caso suas precondições sejam atendidas. Assim, em um sistema real, a partir desse comportamento é possível, por exemplo, gerar uma mensagem ao *Ator* notificando a necessidade de se realizar alguma ação no ambiente.

Figura 13: Esquema de Classe Ator

$$TipoAtor ::= Agente \mid Humano \mid Robo$$

Fonte: Próprio Autor.

A operação *Requisitar* representa a solicitação de uma requisição por parte do *Ator*. Ele recebe como entrada uma requisição $?r$ que pertence ao conjunto de *requisicoes* do *Ator*. Caso a precondição da requisição $r?$ seja verdadeira, avaliamos se existe algum serviço que apresente capacidade igual a capacidade da requisição. Em caso de existência, o serviço é executado e a pós-condição do serviço torna-se verdadeira. Caso a precondição de $r?$ seja falsa, nada acontece.

Definição 3.2.4 (Instrumento). *Um instrumento é um conceito representado por uma 4-tupla (*nome*, *Habilidade*, *usadoPor*, *localizacao*)*

- *nome*: identificador do instrumento.
- *Habilidade*: conjunto de habilidades do instrumento.
- *usadoPor* axioma que define os tipos ou nomes de atores que podem usar o instrumento.
- *localizacao*: localização atual do instrumento.

A classe *Instrumento* (Figura 14) representa um objeto que pode ser manipulado por um *Ator* do tipo *Humano* ou um ator do tipo *Robo*. Em seu esquema de estado, são definidas suas *habilidades*, o ator que está de posse do instrumento e a localização, respectivamente, representados pelos atributos *habilidades*, *usadoPor* e *localizacao*.

Figura 14: Esquema de Classe Instrumento

Fonte: Próprio Autor.

Além da operação *AdicionarHabilidades*, *RemoverHabilidades* que possui especificação comum às demais operações de adição e remoção apresentadas em outras classes, *Instrumento* apresenta a operação *Executar*. A operação *Executar* é realizada perante a passagem de uma variável de entrada do tipo *Habilidade* que pertence ao conjunto de habilidades do instrumento. O ator deve estar de posse do instrumento, ilustrado pela condição $usadoPor \neq \emptyset$. Caso a condição seja satisfeita, o operador de escolha $[]$ irá avaliar as operações *h.PreVerdadeira* e *h.PreFalsa*. Caso *h.PreVerdadeira* apresente valor verdadeiro, *Executar* apresenta o nome da habilidade. Ao final, chama a operação *h.PosVerdadeira* que apresenta o valor verdadeiro, consequência da ação do *Instrumento* no *Ambiente*.

Definição 3.2.5 (Requisição). *Uma requisicao é um conceito representado por uma 4-tupla (nome, OntID, Propriedade, Habilidade), onde:*

- *nome*: identificador da requisição.
- *OntID*: conjunto dos identificadores das ontologias importadas.
- *Propriedade*: conjunto de propriedades da requisição.
- *Habilidade*: conjunto de habilidades desejadas pela requisição.

O esquema relativo a entidade requisição é apresentado na Figura 15. Ele apresenta como atributos a *capacidade* associada a essa requisição e a *habilidade* almejada.

Como operações, *Requisicao* apresenta as operações *PreVerdadeira* e *PreFalsa*, definidas, respectivamente, por meio das operações *PreVerdadeira* e *PreFalsa* do objeto *habilidade*.

Figura 15: Esquema de Classe Requisicao

Fonte: Próprio Autor.

Definição 3.2.6 (Propriedade). *Uma propriedade é uma 2-tupla (nome, valor)*

- *nome*: identificador da propriedade.
- *valor*: é uma instância ou um identificador de valor de dado.

Na Figura 16 é apresentada a classe que define *Propriedade*. O esquema *Propriedade* representa objetos do tipo nome/valor, representado pelas variáveis de estado nome e valor, respectivamente. *Nome* é do tipo *ID* e é um atributo herdado de *EntidadeSWoTPAD*. Já *valor* pode apresentar elementos do tipo *Variavel* ou do tipo *Valor*. *Valor* corresponde a um tipo definido pela união dos tipos *Literal*, que representa uma constante, *null*, *booleanos* e *EntidadeSWoTPAD*. Já *Variavel* é uma classe que denomina uma variável *SWoTPADL*. Ela apresenta um atributo *id* do tipo *IDV* (identificador de variável) e *val* do tipo *Valor*.

Figura 16: Esquema de Classe Propriedade

Fonte: Próprio Autor.

Definição 3.2.7 (Habilidade). Uma habilidade é uma 5-tupla (*nome*, *capacidade*, *SubCapacidades*, *precondicao*, *poscondicao*)

- *nome*: representa o nome da habilidade. É uma expressão sintática da forma $h(x_1, \dots, x_k)$, onde h é único e representa o identificador da habilidade e $x_1, \dots, x_k$ são variáveis.
- *capacidade*: representa a capacidade desenvolvida por essa habilidade. Habilidades diferentes podem ter uma mesma capacidade.
- *SubCapacidades*: conjunto de capacidades necessárias para realização dessa habilidade.
- *precondicao*: axioma que define as condições para realização da habilidade
- *poscondicao*: axioma que define as mudanças provocadas ao ambiente pela habilidade.

A Figura 17 apresenta a definição da entidade *Habilidade*, que representa a entidade do modelo conceitual responsável por definir as habilidades desenvolvidas por um *Servico*, *Ator* ou *Instrumento*. Ela é composta pelo atributo *variaveis* que representa o conjunto de parâmetros da *habilidade*, o atributo *capacidade*, que representa

Figura 17: Esquema de Classe Habilidade

Fonte: Próprio Autor.

a *capacidade* que essa *habilidade* realiza, *subCapacidades*, que representa o conjunto de subcapacidades que compõe essa *habilidade*. Por último, temos a *precondicao* e

poscondicao, expressões que definem a condição necessária para a *habilidade* ser realizada e as mudanças que esperam acontecer no *ambiente*, respectivamente. As restrições apresentadas por essa classe é que, dado duas habilidades diferentes *h* e *k*, implica que elas não apresentam o mesmo nome. A outra restrição associa *capacidade* a *habilidade*. Essa associação é feita através do atributo *desenvolvidoPor* de *capacidade*.

Por último, temos as operações. A classe *Habilidade* apresenta as operações *PreVerdadeira*, *PosVerdadeira* e *PreFalsa*, usadas para avaliar se a precondição é verdadeira, se a poscondição é verdadeira e se a precondição é falsa, respectivamente. A operação *PreVerdadeira* avalia se todas as expressões que compõem o conjunto precondição são verdadeiras. De maneira análoga, atua a operação *PosVerdadeira*, no âmbito da variável *poscondicao*. Por último é definida a operação *PreFalsa*, cuja condição necessária para *PreFalsa* ser verdadeira é a existência de ao menos uma expressão em precondição que apresente valor falso.

A seguir definimos a classe *Expressao*, usadas no âmbito de *Habilidade* para representar suas *precondicoes* e *poscondicoes*. Ela é composta por um conjunto de variáveis, termos e valores, denominados, respectivamente, como *variaveis*, *termos* e *v*. A variável *v* representa o valor de uma expressão e pode assumir os seguintes valores: *true*, *false* e *nil*. *Expressao* apresenta uma série de especializações. Na Figura 18, para tornar a apresentação mais concisa, apresentamos apenas *ExpressaoPossui*, *AndBinaria*, *ExpressaoParaTodo* e a *ExpressaoEMembro*.

ExpressaoPossui representam expressões do tipo *hasValue*, usada em nossa formalização para avaliar se o atributo de uma entidade é de um determinado tipo. *ExpressaoPossui* apresenta a forma *objeto[atributo hasValue val]*, e avalia se o "atributo" de "objeto" apresenta o valor "val". Ela devolve verdadeiro, caso "val" seja o valor do "atributo" e falso, caso contrário. Em nossa formalização, "objeto", "atributo" e "val" são representados, respectivamente, pelas variáveis *obj*, *atr* e *val*.

Outro aspecto desenvolvido na classe *ExpressaoPossui* são as restrições aos valores que seus atributos podem assumir. Como *obj* e *atr* pode ser uma variável ou uma instância de um objeto *EntidadeSWoTPAD*, o atributo *variaveis*, que representa todas as variáveis que aparecem na expressão é definido a partir da união de *obj*, *atr* e *val* com a interseção de todas as variáveis definidas no sistema. De maneira análoga, *termos* representa todos os termos que aparecem em *ExpressaoPossui* e é formado pela união de *obj* e *atr* com a interseção de todos os objetos *EntidadeSWoTPAD* definidos no sistema. Nesse caso, não usamos *val*, pois *val* refere-se ou a um *Valor* ou a uma *Variavel*.

Figura 18: Esquemas da Hierarquia de Classes Expressao

[OrBinaria] [ImplicaBinaria] [ImplicadoPorBinaria] [ExpressaoExisteAoMenos]

Fonte: Próprio Autor.

Também foram feitas restrições ao valor v que a expressão pode assumir. Caso obj , atr ou val apresentem valor igual a nil , o atributo v de *ExpressaoPossui* possuirá valor nil . Caso contrário, o valor v será definido a partir da avaliação se o atributo obj apresenta algum atributo *propriedades.nome* igual a atr e o atributo *propriedades.valor* igual a val .

A classe *AndBinaria* representa expressões formadas a partir de disjunções de expressões. Ela apresenta os atributos esq e dir que representam, respectivamente, as expressões localizadas a esquerda e a direita do operador de disjunção binária. Como restrições, apresenta que o atributo $termos$ de *AndBinaria* é formada a partir dos termos de suas componentes. De maneira análoga é definida o atributo $variaveis$. Caso a componente $esq.v$ ou $dir.v$ apresente valor nil , o atributo v de *AndBinaria* também apresentará valor nil . Caso $esq.v$ e $dir.v$ diferentes de nil , aplica-se a v o valor da disjunção entre $esq.v$ e $dir.v$. De maneira análoga são definidas classes para os operadores binários $\vee$ (*OrBinaria*), $\Rightarrow$ (*ImplicaBinaria*) e $\Leftarrow$ (*ImplicadoPorBinaria*).

A classe *ExpressaoParaTodo* é utilizada para representar expressões feitas a partir do quantificador universal. Ela é formada pela variável (atributo *var*) usada pelo quantificador universal e um operando (atributo *operando*), que representa uma expressão que usa essa variável. A classe *ExpressaoParaTodo* define restrições para o atributo *variaveis*, sinalizando que esse atributo é formado pelas mesmas variáveis que compõem *operando.variaveis*. De maneira análoga, é definida *termos*. O valor do atributo *v* de *ExpressaoParaTodo* é definido da seguinte forma. Se o atributo *var.v* apresenta um *Valor* válido, isso implica que a expressão de *operando* é verdadeira. Dessa forma, a variável *v* também será verdadeira. Caso contrário, *v* apresenta valor falso.

Definição 3.2.8 (Capacidade). *Uma capacidade possui a forma $c(r_1, \dots, r_k)$, onde c é único e representa o identificador da capacidade, e $r_1, \dots, r_k$ são variáveis.*

A Figura 19 representa a classe *Capacidade*, que atua em SWoTPADL de forma aná-

Figura 19: Esquema de Classe Capacidade

Fonte: Próprio Autor.

loga a uma interface que aglutina um conjunto de habilidades comuns. *Capacidade* apresenta como atributos o *nome* e o conjunto de *parâmetros*, denominados como *variaveis*. Adicionalmente, apresenta o campo *desenvolvidoPor*, que serve para aglutinar objetos do tipo *Habilidade* que realizam uma certa *Capacidade*. A única restrição associada a esse tipo é que o atributo *nome* de uma *Capacidade* é único. Dessa forma, não existem duas capacidades diferentes *i* e *j* com mesmo nome.

A Figura 20 apresenta a classe *SubCapacidade*. Entidade presente em SWoTPADL que apresenta comportamento similar ao de uma rede de tarefas no âmbito de planejamento. Um objeto *Subcapacidade* apresente um conjunto de *Capacidade*, denominado capacidades e um conjunto de restrições a essa capacidade, denominada *restricoes*. Uma restrição pode ser uma de precedência, descrita pela classe *RPrecedencia* ou, uma restrição de estados, descrita através de *REstados*.

Figura 20: Classe SubCapacidade

Fonte: Próprio Autor.

Definição 3.2.9 (Comportamento). *Um comportamento é 3-tupla (*nome*, *Parametro*, *Comando*)*

- *nome*: identificador da operação
- *Parametro*: conjunto de definições de parâmetros.
- *Comando*: conjunto de comandos.

A classe *Comportamento* (Figura 21) é uma entidade que descreve a sequência de ações desenvolvidas por objetos do tipo *Servico*. Ele apresenta como atributos um *nome*, um conjunto de variáveis *parâmetro* e um conjunto de objetos do tipo *Comando*. Apresenta também a operação *Executar* que é definida pela operação *Executar* de cada *Comando* *c* que compõe o conjunto comando.

Figura 21: Esquema de Classe Comportamento

Fonte: Próprio Autor.

Definição 3.2.10 (Comando). *Um comando pode ser:*

- *Um comando condicional na forma $(expLog, Comando)$.*
- *Uma chamada de função na forma $(nome, parametros, Comando)$*
- *Um comando de repetição na forma $(expLog, comando)$.*

A hierarquia da Classe *Comando* é apresentado na Figura 22. Ela é composta por três tipos de comando: o comando condicional (*CmCondicional*), o comando de repetição (*CmRepeticao*) e a chamada de métodos (*CmChamada*).

Figura 22: Esquema de Classe Comando

Fonte: Próprio Autor.

CmCondicional e *CmRepeticao* apresentam os mesmos atributos: uma variável booleana que representa a expressão condicional presente nesse tipo de comando, e o conjunto de comandos a ser executado. Os objetos do tipo *Comando* que compõem comandos são executados através da operação *Executar*. *Executar* em *CMCondicional* avalia se a variável *explog* apresenta valor verdadeiro. Caso positivo, é chamado o operador *Executar* de cada *Comando* *c* de comandos. A operação *Executar* de *CmRepeticao*, avalia *explog* e, caso a variável apresente valor verdadeiro, é chamado o operador *Executar* de cada *Comando* *c* de comando. Após isso, o comando *Executar* de *CmRepeticao* é chamado novamente. O comportamento segue, até o momento que *explog* passa a ter um valor falso.

Por último temos *CmChamada*, um *Comando* que representa chamadas de métodos. Ele apresenta como atributos o identificador do método e o conjunto de parâmetros, denominados *idComando* e *parametros*, respectivamente. A operação *Executar* apresenta uma variável de entrada *ambiente?* que representa o estado atual do *Ambiente*. *Executar* atua sobre o ambiente através da função *mudaProps*. A função *mudaProps* recebe como argumento o conjunto de parâmetros e *ambiente?* e os mapeia para um conjunto de *propriedades*. Esse conjunto de *propriedades* representam *propriedades* do *Ambiente* ou de suas componentes que foram modificadas. Esse conjunto de *propriedades* é armazenado na variável de saída *Propriedades*.

Os seguintes conceitos relacionados a planejamento hierárquico precisaram ser estendidos para o domínio de nosso trabalho. Os demais elementos não apresentados, estão em conformidade com o que foi apresentado na Seção 2.4.

Definição 3.2.11 (Estado de um sistema de IoT). *Seja S o conjunto de estados. Um estado $s \in S$ corresponde a uma 5-tupla formada pelos seguintes elementos $(\mathbb{I}Propriedadeambiente, \mathbb{I}PropriedadeAtor, \mathbb{I}PropriedadeInstrumento, \mathbb{I}PropriedadeServico, \mathbb{I}PropriedadeSubAmbientes)$, onde:*

- *$\mathbb{I}Propriedadeambiente$: conjunto de instâncias de propriedades do Ambiente.*
- *$\mathbb{I}PropriedadeAtor$: seja $ambiente_{Ator}$ o conjunto formado por todos os atores presentes em ambiente. Seja $\{a_1, a_2, \dots, a_n\}$ os atores do ambiente. $Propriedade_{Ator}$ representa o conjunto formado pela união de todas as instâncias de propriedades dos atores $a_1, a_2, \dots, a_n$.*
- *$\mathbb{I}PropriedadeInstrumento$: seja $ambiente_{instrumento}$ o conjunto formado por todos os instrumentos presentes em ambiente. Seja $\{o_1, o_2, \dots, o_n\}$ os instrumentos de ambiente, $Propriedade_{Instrumento}$ representa o conjunto formado pela união de todas as instâncias de propriedades dos instrumentos $o_1, o_2, \dots, o_n$.*
- *$\mathbb{I}PropriedadeServico$: seja $ambiente_{servico}$ o conjunto formado por todos os serviços presentes em ambiente. Seja $\{s_1, s_2, \dots, s_n\}$ os serviços de ambiente, $Propriedade_{servico}$ representa o conjunto formado pela união de todas as instâncias de propriedades dos serviços $s_1, s_2, \dots, s_n$.*
- *$\mathbb{I}PropriedadeSubAmbientes$: seja $ambiente_{subAmbientes}$ o conjunto formado por todos os subAmbientes presente em ambiente. Seja $\{t_1, t_2, \dots, t_n\}$ os sub-ambientes de ambiente, $IPropriedade_{subAmbientes}$ representa o conjunto formado pela união de todas as instâncias de propriedades dos subAmbientes $t_1, t_2, \dots, t_n$, bem como pela união*

das instâncias das propriedades dos atores, instrumentos e serviços que pertencem ao *subAmbiente*.

A Figura 23 representa um estado válido de um sistema de IoT. De acordo com nosso modelo conceitual, o estado de um sistema de IoT é definido em função do conjunto de propriedades que cada uma de suas componentes possui. A classe *EstadoIoT* representa cada uma dessas componentes pelos seguintes atributos: *propriedadesAmbiente*, *propriedadesAtor*, *propriedadesInstrumento*, *propriedadesServico*, *propriedadesSubAmbientes*, que representam, respectivamente, as propriedades associadas ao *Ambiente*, *Atores*, *Instrumentos*, *Serviços* e *SubAmbientes*. Adicionalmente, apresenta a variável secundária *totalPropriedades* que é definida a partir da união de todas as propriedades de *Ambiente* e das propriedades de suas componentes. Também são feitas restrições para cada uma dos atributos (variáveis primárias). O atributo *propriedadesAtor* é definido a partir das propriedades de todos os objetos do tipo *Ator*, obtidas por meio da sentença $\forall a : \text{Ator} \bullet a.\text{propriedades}$. As demais variáveis primárias são definidas de forma análoga.

Figura 23: Esquema de Classe EstadoIoT

Fonte: Próprio Autor.

A classe *EstadoIoT* apresenta a operação *Transicao*, usada para realizar mudanças

de estado. Ela recebe como entrada a variável *propriedade?* que representam um conjunto de propriedades do Ambiente ou de suas componentes que serão modificadas. A modificação dessas propriedades é obtida através da sentença $\forall p : totalPropriedades, q : propriedade? \bullet (p.nome = q.nome) \Rightarrow (p'.valor = q.valor)$, que avalia as propriedades que apresentam nome igual ao das propriedades passada na variável de entrada *propriedades?* e atualiza todas as propriedades de *EstadoIoT* que apresentou mudança.

As definições 3.2.12, 3.2.13 e 3.2.14 não apresetam formalização em Object-Z, pois se tratam de conceitos adotados em planejamento hierárquico que apresentam conceitos correspondente em nosso modelo conceitual. O conceitos de operador, ação e método correspondem ao de habilidade.

Definição 3.2.12 (Operador de um Sistema de IoT). *O conceito de operador é representado por uma 3-tupla $o = (nome(o), precond(o), efeitos(o))$.*

1. *$nome(o)$: o nome do operador é uma expressão sintática da forma $n(x_1, \dots, x_k)$, onde n é único e representa o identificador do operador e $x_1, \dots, x_k$ são variáveis.*
2. *$precond(o)$ e $efeitos(o)$: são generalizações de precondições e efeitos de uma ação.*

O operador representa uma tarefa primitiva. Em nosso contexto, o operador é representado por uma habilidade $h = (nome, capacidade, SubCapacidades, precondicao, poscondicao)$ cujo conjunto $SubCapacidades = \emptyset$ e $nome = capacidade$.

Definição 3.2.13 (Ação de um Sistema de IoT). *Uma ação representa uma instância de um operador e possui a forma $a = (precond(a), efeitos^-(a), efeitos^+(a))$, onde:*

- *$precond(a)$: conjunto que representa as precondições da ação a . Uma ação a só pode ser executada em um dado estado s se $precond^+(a) \subseteq s$ e $precond^-(a) \cap s = \emptyset$.*
- *$efeitos^+(a)$ e $efeitos^-(a)$: são denominados como o conjunto de efeitos de a . A aplicação da ação a a um estado s resulta em $s[a] = (s \cup efeitos^+(a)) \setminus efeitos^-(a) = s'$.*

Definição 3.2.14 (Método de um Sistema de IoT). *O conceito de método é comumente usado em planejamento hierárquico e consiste em uma 5-tupla $m = (nome(m), tarefa(m), precond, subtarefa(m), restricoes(m))$, onde:*

- *nome(m)*: é uma expressão da forma $n(x_1, \dots, x_k)$, onde n é um símbolo único de método, isto é, não há dois métodos em um domínio de planos que possam ter o mesmo símbolo e $x_1, \dots, x_k$ são todas as variáveis de símbolos que podem ocorrer em qualquer lugar em m .
- *tarefa(m)*: é uma tarefa não primitiva.
- *precond*: é o conjunto que representa as precondições do método m . Um método m é aplicável a um estado s se $\text{precond}^+(m) \subseteq s$ e $\text{precond}^-(m) \cap s = \emptyset$.
- *(subtarefa(m), restricoes(m))*: é uma rede de tarefas.

Em nosso contexto, um método é representado por uma habilidade $h = (\text{nome}, \text{capacidade}, \text{SubCapacidades}, \text{precondicao}, \text{poscondicao})$ cujo o conjunto $\text{SubCApacidades} \neq \emptyset$, onde $\text{tarefa}(m) = \text{capacidade}$ e $(\text{subtarefa}(m), \text{restricoes}(m)) = \text{SubCapacidades}$.

Definição 3.2.15 (Problema de planejamento hierárquico para Sistemas de IoT). Um problema de planejamento hierárquico é uma 8-tupla $\mathcal{P} = (\mathbb{I}\text{Propriedadeambiente}_o, \mathbb{I}\text{PropriedadeAtor}_o, \mathbb{I}\text{PropriedadeInstrumento}_o, \mathbb{I}\text{PropriedadeServico}_o, \mathbb{I}\text{PropriedadeSubAmbientes}_o, w, O, M)$, onde o conjunto de propriedades representa o estado inicial, w é a rede de tarefas inicial, O é um conjunto de operadores e M é o conjunto de métodos HTN.

A Figura 24 apresenta a classe que define o problema de planejamento para IoT. A classe em questão, *ProblemaPlanejamentoIoT* apresenta como atributos o *estadoInicial* do tipo *EstadoIoT* e as *subCapacidades*. Adicionalmente possui acesso a todas as ca-

Figura 24: Esquema de Classe ProblemaPlanejamentoIoT

Fonte: Próprio Autor.

pacidades e habilidades do sistema, através das variáveis *capacidades* e *habilidades*. Esse característica é obtida por meio da restrição definida no esquema de estado. Nela, para toda *Capacidade* c , temos que c pertence ao conjunto *capacidades*. De forma análoga foi definida *habilidades*.

3.3 Conclusões

Esse capítulo teve como objetivo apresentar o modelo conceitual desenvolvido em nosso trabalho, bem como a formalização adotada, em Object-Z, para atribuir maior rigor formal ao modelo desenvolvido.

As classes do modelo conceitual adotado representam entidades comuns a qualquer sistema de IoT. A importância desse modelo conceitual em nosso trabalho é o de fornecer um vocabulário comum a diferentes sistemas de IoT que optem por adotar nossas representações.

Escolhemos formalizar o modelo em Object-Z por se tratar de uma linguagem que fornece uma descrição consistente e inequívoca. Nela, especificamos um sistema, descrevendo os estados em que pode estar, juntamente com uma descrição de como o estado evolui ao longo do tempo (DERRICK; BOITEN, 2001).

Adicionalmente, sistemas descritos em Object-Z podem ser submetidos a um processo denominado refinamento. O propósito usual de refinamento é mostrar que uma implementação atende aos requisitos estabelecidos em uma especificação inicial. Por isso, o desenvolvimento do programa começa com uma especificação abstrata e prossegue através de uma série de etapas, cada passo produzindo um projeto um pouco mais detalhado que se mostra como um refinamento da especificação anterior (DERRICK; BOITEN, 2001).

Dessa forma, através do modelo formal apresentado, podemos descrever sistemas de IoT e, através de refinamento, obtermos código fonte cujo comportamento é igual ao especificado. Se bem empregado, tal característica possibilita a comunidade de desenvolvedores a criação de sistemas com menores índices de retrabalho.

4 O FRAMEWORK SWOTPAD

Esse capítulo apresenta o *framework* SWoTPAD, que tem como objetivo dar suporte semântico a projetos de *IoT*. Ele possui a seguinte organização. A Seção 4.1 apresenta a visão geral do *framework* e seu modelo em camadas, em seguida, a Seção 4.2 apresenta a camada semântica, que oferece um conjunto de soluções para facilitar a adoção de semântica no âmbito de projetos de sistemas de *IoT*. A Seção 4.3 apresenta a camada funcional, caracterizada pelas ferramentas de descoberta e composição automática de serviços. Finalmente, a Seção 4.4 apresenta a ferramenta SWoTPAD *Designer*, que implementa as ideias apresentadas neste capítulo.

4.1 Visão Geral

A abordagem proposta por esse trabalho visa o desenvolvimento de um *framework* para composição automática de serviços em ambientes de *IoT*. Segundo Fayad *et al* (FAYAD; SCHMIDT, 1997), *framework* é uma estrutura formada por blocos pré-fabricados de software que programadores podem usar, estender ou adaptar para uma solução específica.

Riehle (RIEHLE, 2000) define *framework* como uma abstração que provê funcionalidades genéricas, seletivamente modificadas ou estendidas, provendo um *software* de aplicação específica. Um *framework* de *software* é um ambiente de *software* universal, reutilizável, que provê funcionalidades particulares como parte de uma ampla plataforma de *software* para facilitar o desenvolvimento de aplicações.

A literatura apresenta vários exemplos de *frameworks* de *software*: Flask (GRINBERG, 2014) e Django (HOLOVATY; KAPLAN-MOSS, 2009), bastante adotados para o desenvolvimento de aplicações Web; scikit learn (PEDREGOSA et al., 2011) e Weka (HALL et al., 2009), aplicados no âmbito de aprendizado de máquina; Hadoop (WHITE, 2012) e Spark (KARAU et al., 2015), adotados em sistemas que lidam com questões de *Big Data*; etc.

Frameworks modelam um domínio específico ou um importante aspecto de um dado domínio. Eles representam o domínio como um projeto abstrato, consistindo de classes abstratas (ou interfaces). O projeto abstrato é mais do que um conjunto de classes, porque ele define como instâncias de classes colaboram umas com as outras em tempo de execução. Efetivamente, ele atua como o arcabouço que determina como objetos do *framework* se relacionam uns com os outros.

Um *framework* apresenta implementações reutilizáveis na forma de implementações de classes abstratas e concretas. Implementações abstratas são usadas para implementar partes da abstração do *framework*, porém, podem delegar às suas subclasses decisões sobre pontos cruciais do sistema. Dessa forma, projetos que usam *frameworks* podem realizar mudanças, com intuito de adequá-lo ao domínio da aplicação.

Existem dois tipos de clientes de um *framework*, classificados de acordo com sua relação de utilização. O cliente baseado em uso, apenas utiliza classes de um ou mais *frameworks* com o intuito de resolver seu problema. O segundo tipo, baseado em extensão, cria subclasses de classes do *framework* de forma a adaptá-las às suas necessidades.

Inúmeras são as vantagens no emprego de *frameworks*. A primeira delas é que eles favorecem a reutilização de projetos e código. Adicionalmente, sistemas baseado em *frameworks* são mais fáceis de manter, pois muito dos aspectos chaves do projeto e decisões de implementação estão localizadas em um único lugar: o *framework*. Do ponto de vista de negócios, as principais vantagens estão relacionadas ao reuso de código, e, consequentemente aumento da produtividade, proporcionando a redução do *time-to-market* de novas aplicações.

A Figura 25 apresenta as camadas do *framework* SWoTPAD. Ele é composto de

Figura 25: Camadas do Framework SWoTPAD

Fonte: Próprio autor

três camadas, cada uma dando suporte a diferentes etapas do projeto do sistema. A camada de Modelagem e Especificação apresenta um conjunto de modelos que podem ser adotados como base para a modelagem e especificação de novos projetos de IoT. Nessa camada, estão presentes o modelo conceitual e sua definição formal em Object-Z, detalhada no Capítulo 3. Um dos objetivos dessa camada é estimular o desenvolvimento de projetos adotando métodos formais. A vantagem de aplicar essa técnica é que através dela, podemos validar o sistema do ponto de vista da consistência, integridade e correteza. Como consequência, promove o desenvolvimento de especificações e implementações mais precisas.

A Camada Semântica tem como papel prover elementos para a criação de descrições do domínio onde o sistema de IoT vai atuar. Ela se baseia no modelo conceitual apresentado no Capítulo 3 e inclui a linguagem SWoTPADL que apresenta um conjunto de construtores que visam facilitar o desenvolvimento de especificações semânticas. Adicionalmente, fornece ferramentas responsáveis por traduzir as representações SWoTPADL em um conjunto de artefatos denominado *Solução SWoTPAD*.

Por último, temos a Camada Funcional, composta pela *Solução SWoTPAD* e apresenta os componentes de provimento, descoberta e composição automática. A seguir, apresentamos mais detalhes das camadas Semântica e Funcional. Não apresentamos a camada de Modelagem e Especificação, pois os artefatos constituintes dessa camada já foram descritos no Capítulo 3.

4.2 A Camada Semântica

A Camada Semântica apresenta recursos que mapeiam as entidades do modelo conceitual para o sistema de IoT em desenvolvimento. A Figura 26 apresenta os clientes dessa camada, que são o *Projetista de Sistemas de IoT* e o *Ator do Sistema de IoT*, e sua interação com os artefatos dessa camada. O Projetista de Sistemas de IoT é o indivíduo ou grupo de indivíduos responsáveis pela análise, projeto e implementação do sistema de IoT.

Do ponto de vista desse cliente, a camada semântica fornece uma linguagem de especificação e um conjunto de ferramentas, com intuito de dar suporte semântico ao projeto em desenvolvimento. Para obter esse suporte, o projetista deve fornecer um conjunto de *descrições dos serviços* e *descrições dos ambientes*. Os serviços devem ser descritos em SWoTPADL, uma linguagem elaborada no âmbito desse trabalho para especificar serviços

Figura 26: Detalhamento da Camada Semântica

Fonte: Próprio autor

e requisições semânticas. Os ambientes e demais entidades são descritas através de coleções que adotam *palavras chaves* que remetem os elementos apresentados em nosso modelo conceitual. Nas seções 4.2.1 e 4.2.2, apresentamos maiores detalhes dessas representações.

As descrições elaboradas pelo projetista são a entrada da *ferramenta de Mapeamento e Tradução*. Nela, o projetista deve associar as descrições dos serviços com as entidades que as realizam. Nesse momento, os elementos do modelo conceitual do tipo *Ator*, *Dispositivo* e *ServicoWeb*, cujas instâncias estão presentes na *descrição do ambiente*, passam a ter uma habilidade associada. A ferramenta gera como artefatos as ontologias dos ambientes, as ontologias dos serviços e serviços *Restful* para cada instância de dispositivo e serviços Web presentes no arquivo de *descrições dos ambientes*. As ontologias do ambiente, as ontologias dos serviços e os serviços *Restfull* são a entrada do *Gerador do Sistema*, que, ao final produz a *Solução SWoTPAD*. A *Solução SWoTPAD* é composta pelos seguintes elementos:

- **Provedor de serviços:** responsável pela descoberta dos serviços e registro de novos serviços.
- **Repositório de serviços:** representa o conjunto de serviços registrados, aptos a ser executado.
- **Compositor de serviço:** realiza a composição automática de serviços.
- **Invocador de serviço:** responsável pela invocação dos serviços.

Do ponto de vista do *Ator do Sistema de IoT*, ou seja, do usuário final que fará uso do sistema, a camada semântica apresenta suporte a criação de requisições semânticas.

Requisição semântica é a forma como os usuários interagem com o sistema de IoT.

É válido notar que o *Ator do Sistema de IoT* representa a entidade *Ator* de nosso modelo conceitual. Dessa forma, o *Projetista de Sistemas de IoT* pode descrever habilidades específicas a esse *Ator* e utilizar a ferramenta de mapeamento e tradução para associá-la ao *Ator*. Um exemplo de aplicação desse recurso é em ambientes industriais, onde temos a presença de técnicos com formações diferentes. Podemos associar a um determinado técnico a habilidade de dar manutenção a uma classe de equipamentos. Dessa forma, no momento que o sistema diagnostica a falha de algum equipamento, o técnico hábil é notificado. Nas próximas seções, detalharemos cada uma das entidades presentes na Figura 26.

4.2.1 Descrição dos Serviços

Na Seção 2.2, vimos que a literatura de Web Semântica possui algumas notações para o desenvolvimento de especificações de serviços, das quais, destacamos o WSMML. No entanto, existem algumas demandas em nossa abordagem que não são atendidas por WSMML. Por conta disso, usamos apenas um subconjunto do WSMML, destinado a representação do conhecimento (*ontologias*). Para especificação de serviços e metas do usuário, desenvolvemos um novo vocabulário denominado SWoTPADL. Nos próximos parágrafos, justificamos nossa decisão.

Um primeiro problema da linguagem WSMML é o excesso de elementos opcionais em sua sintaxe. Esse excesso pode levar a criação de descrições obscuras. Com intuito de tornar a linguagem mais flexível, os autores adotaram um elevado grau de relaxamento na sintaxe de seus construtores. A Listagem de Código 5 ilustra esse problema. Nela é apresentada a sintaxe dos construtores *webService*, *goal* e *instance*, adotados para a definição de Serviços Web, metas do usuário e instância de conceitos, respectivamente. De acordo com a sintaxe da linguagem, a palavra reservada *webService*, desacompanhada de qualquer elemento descritivo, é aceita como um serviço válido. O mesmo comportamento acontece com *goal* e *instance*.

Listagem de Código 5: Sintaxe dos construtores *webService*, *goal* e *instance* em WSMML

```

1: <webservice> ::= 'webService' <id>? <header>* <capability>? <interface>*
2: <goal> ::= 'goal' <id>? <header>* <capability>? <interface>*
3: <instance> ::= 'instance' <id>? <memberof>* <nfp>? <attributevalue>*

```

Fonte: Próprio Autor

Outro aspecto relevante é que WSMML não apresenta suporte a definição de *hierarquia de serviços*. Em WSMML, o construtor *webService* não permite definir um serviço como

especialização de outro serviço. A linguagem não apresenta uma cláusula para definição de sub-serviços. Dessa forma, não podemos trabalhar, ao menos de forma natural, com o conceito de hierarquia serviços, ou seja, a definição de uma taxonomia de serviços de acordo com sua funcionalidade.

O conceito de hierarquia é útil para resolução do problema de composição automática, visto que, o uso de hierarquia de serviços permite restringir o espaço de busca de serviços candidatos (SOHRABI; MCILRAITH, 2008). A redução ocorre, pois a busca pode se restringir a apenas uma categoria de serviços. Fensel *et al* (FENSEL et al., 2011) sugerem a adoção de *mediators* para esse propósito. Porém, não fica claro como é feita a aplicação desse conceito. Adicionalmente, a solução não é elegante, pois acaba introduzindo *overhead* desnecessário aos *mediators*, visto que do ponto de vista conceitual, essa construção é destinada a realizar tradução de vocabulário.

As definições *goal* e *webService* apresentam outros problemas. Além da sintaxe, em algumas situações é difícil diferenciar *preconditions* de *assumptions*, bem como *postconditions* de *effects*. Outro problema associado a *webService* é a dificuldade de descrever serviços compostos, ou seja, serviços formados a partir de outros serviços. A especificação não é clara a respeito desse tema, visto que o conceito de coreografia não apresenta uma especificação semântica. Mesmo a ferramenta WSMT (FENSEL et al., 2011), que consolida muito dos aspectos semânticos ausentes na documentação oficial de WSML (BRUIJN et al., 2008), não explora a semântica do construtor *choreography*.

Sharifi *et al.* (SHARIFI; BAYRAM, 2016) apresentam uma série de críticas a linguagem WSML. A seguir, destacamos quatro pontos em que concordamos com os autores:

- I. WSML não oferece uma especificação semântica, por exemplo, na forma de pré-condições e pós-condições, para métodos/operações de serviço na Web.
- II. A linguagem que define coreografia é muito complexa, e é apenas uma aproximação grosseira do formalismo de máquinas de estado abstrato (GUREVICH; ROSSMAN; SCHULTE, 2005).
- III. A natureza paralela das regras de coreografia não combina bem com o comportamento presente em serviços reais, geralmente sequenciais.
- IV. A interação entre coreografia, *grounding* e especificação lógica de que o serviço Web faz, foi negligenciado no WSML. Não há verificação semântica entre os conceitos, relações e variáveis usadas por essas especificações. Adicionalmente, em WSML,

não é clara as responsabilidades e a influência que cada uma dessas especificações apresentam sobre as outras.

Além dos problemas citados, é natural que a linguagem não apresente suporte a IoT. WSML foi criada para Web Semântica. Logo, questões mais pontuais, relativas a especificação de sistemas de IoT, não são contempladas. Por esses motivos, propomos uma nova linguagem para nosso *framework* denominada SWoTPADL. Ela apresenta um conjunto de novos construtores que simplificam especificações WSML, bem como introduz novos recursos. SWoTPADL resolve os problemas apresentados pela linguagem WSML através de:

- **Visão simplificada de serviço, com suporte a hierarquia e instâncias de serviço:** SWoTPADL mapeia automaticamente os serviços em conceitos. Desta forma, é possível gerar instâncias, bem como gerar novos serviços através de herança. Além disso, simplificamos a definição de serviços eliminando *assumptions* e *effects*. Simplificamos a linguagem, unificando *assumptions* e *preconditions* em *preconditions*. De maneira análoga, unificamos *effects* e *postconditions* em *postconditions*.
- **Desenvolvimento de serviços compostos:** no WSML, usamos coreografia para especificar serviços compostos, cuja sintaxe é de difícil entendimento e não apresenta semântica definida. Nossa abordagem cria o *compCapacities*, um novo construtor para definir serviços compostos. *compCapacities* usa o comando sequencial (*SEQ*) e o paralelo (*PAR*) para especificar os serviços que formam o serviço composto, bem como a sua ordem de execução.
- **Modelagem do comportamento de serviço:** SWoTPADL introduz o operador de comportamento *behavior*, que nos permite descrever o comportamento dos serviços, adotando uma representação próxima das linguagens de programação imperativas. Além disso, ele exibe como padrão a execução sequencial de comandos e suporta comandos condicionais (*if*) e de repetição (*while*, *for*, *do...until*).
- **Substituição de *goal* por *request*:** o SWoTPADL é uma linguagem orientado a requisições. Uma requisição é definida por suas pré e pós-condições. O mapeamento para o serviço ou conjunto de serviços mais apropriados é feito automaticamente, através das ferramentas fornecidas no *framework*.

Na próxima seção, apresentamos a sintaxe da linguagem SWoTPADL relativa a definição de serviços, acompanhado de um exemplo de utilização.

4.2.1.1 Definição de Serviços em SWoTPADL

Embora inspirada em WSMML, a definição de serviços em SWoTPADL apresenta pontos de simplificação. Em geral, uma mesma especificação feita em SWoTPADL possui menor quantidade de linhas, quando comparada a especificação correspondente em WSMML. A Listagem de Código 6 apresenta a sintaxe de definição de serviços em SWoTPADL.

Listagem de Código 6: Sintaxe de Definição de Serviços em SWoTPADL

```

1: <servicoSwotpadl> ::= <namespace> <impOntologies>? <defService> <behavior>?
2: <namespace> ::= URI
3: <impOntologies> ::= 'import' '{ ID URI (',' ID URI)* '}'
4: <defService> ::= 'service' ID 'extends' ID (',' ID)* <precondition> <postcondition> <cCapacities>?
5: <precondition> ::= 'precondition' <exp>*
6: <postcondition> ::= 'postcondition' <exp>*
7: <exp> ::= <exp> <binop> <exp>
8:         | <unarioop> <exp>
9:         | <quantifier> ID (',' ID)* '(' expr ')
10:        | ID
11:        | LITERAL
12:        | ...
13: <binop> ::= 'memberOf' | 'and' | 'or' | 'implies' | 'impliedBy' | '+' | '-' | '*' | '/' | 'hasValue'
14: <unaryop> ::= 'not' | '-'
15: <quantifier> ::= 'forall' | 'exists'
16: <cCapacities> ::= 'compCapacities' <compCapacities>
17: <compCapacities> ::= 'PAR' '(' <members> ')
18:         | 'SEQ' '(' <members> ')
19: <members> ::= ID ':' ID (',' <member>)*
20:         | <compCapacities> (',' <member>)*
21: <behavior> ::= 'behavior' 'procedure' ID '(' ID ':' ID (',' ID ID)* ')' <comando>*
22: <command> ::= 'if' <exp> 'then' <command>* ('else' <command>*)?
23:         | 'foreach' <exp> <command>*
24:         | 'while' <exp> <command>*
25:         | ID '(' ID ? (',' ID)* ')'
26:         | 'return' exp

```

Fonte: Próprio Autor.

De acordo com gramática, vemos que a definição de um serviço em SWoTPADL é composta por quatro elementos. O primeiro, dedicado a declaração do espaço de nomes (*<namespace>*) adotado pelo serviço. Em SWoTPADL, espaços de nome são declarados por meio de uma URI (*Uniform Resource Identifier*), que representa uma sequência de caracteres usada para identificar recursos na Internet. Em seguida, temos a seção de importação de ontologias (*<impOntologias>*), introduzida pela palavra reservada *import*. Cada ontologia importada deve vir precedida de um identificador. Em outros pontos do código, podemos usar esse identificador para fazer referência a ontologia de forma simplificada.

A terceira seção (*<defServico>*) é dedicada a declaração e definição semântica do serviço. Em SWoTPADL, um serviço é definido através da palavra reservada *service*,

seguido pelo seu identificador. Caso o serviço seja especialização de algum outro, usamos a palavra reservada *extends* seguida pelos nomes dos serviços pai. A semântica do serviço é definida por meio de *preconditions* e *postconditions*. A primeira, define o conjunto de expressões que descrevem as entradas do serviço e o estado do ambiente antes da execução dos serviços, já a segunda, descreve as saídas do serviço e o estado do ambiente após a execução do serviço.

As *precondition* e *postcondition* são definidas através de expressões. SWoT-PADL dá suporte a expressões lógicas (através dos operadores *and*, *or*, *not*, *implies*, *impliedBy*), expressões aritméticas (+, -, *, /), expressões com quantificadores universal (*forall*) e existencial (*exists*), expressão para teste de tipo (*memberOf*) e para teste de valor (*hasValue*), herdadas de WSMML (BRUIJN et al., 2008).

A definição de *service* termina com a definição de *compCapacities*. Tal construtor só é utilizado para definição de serviços compostos. O construtor *compCapacities* é formado por uma lista de serviços, sendo essa lista introduzida por meio dos operadores *PAR*, para composição paralela, e *SEQ*, para composição sequencial.

A quarta seção (*<behavior>*) introduz o comportamento do serviço. Esse comportamento é definido através de uma *procedure*, cuja sintaxe é similar a de uma função em programação imperativa. A *procedure* define a sequência de comandos usada por *service* durante sua execução.

A Listagem de código 7 apresenta um exemplo de definição de serviço em SWoTPADL. O serviço *SearchPrice* é uma especialização de *rental#rentalValue*, que descreve um serviço genérico para locação de objetos. Já *SearchPrice* é uma especialização que representa um serviço para locação de automóveis e apresenta o preço de locação de acordo com os parâmetros passados em sua entrada e definidos em sua *precondition*. Para realizar esse comportamento, a especialização *SearchPrice* faz uso de outros serviços de consulta de preço. *SearchPrice* importa as ontologias *car* (linha 2), uma ontologia que possui o vocabulário relativo a automóveis, e *rental*, uma ontologia que apresenta vocabulário relativo a serviços de locação (linha 3). Nas linhas 5 a 7 é definida a *precondition* de *SearchPrice*, que são o lugar (*?place*), data (*?date*), tipo de veículo (*?type*) e modelo do veículo (*?model*) a ser alugado. Como *postcondition* (linhas 8 e 9), *SearchPrice* apresenta o preço do aluguel (*?price*) do veículo solicitado.

O construtor *compCapacities* indica que *SearchValue* é composto pelos serviços *P1*, *P2* e *P3* do tipo *rental#rentalValue* (linha 11). A execução de *P1*, *P2* e *P3* pode ser feita em paralelo (indicada pelo operador *PAR*). Como comportamento, *SearchPrice*

chama os três serviços e associa os resultados às variáveis *price1*, *price2* e *price3*. Em seguida, ele retorna o menor dos três valores, usando a relação *min* (linhas 12 a 17).

Listagem de Código 7: Exemplo de definição de Serviço

```

1: _"http://pad.br.usp/travel#"
2: import { car _"http://www.wsmo.org/sws-challenge/cars#"
3:   rental _"http://www.wsmo.org/sws-challenge/rental#" }
4: service SearchPrice extends rental#rentalValue
5:   precondition
6:     ?place memberOf rental#Place and ?date memberOf rental#date
7:     and ?type memberOf car#type and ?model memberOf car#model
8:   postcondition
9:     ?price memberOf rental#price
10:  compCapacities:
11:    PAR(P1:rental#rentalValue, P2:rental#rentalValue, P3:rental#rentalValue)
12:  behavior
13:    procedure bestPrice()
14:      price1 = P1(?place, ?date, ?type, ?model)
15:      price2 = P2(?place, ?date, ?type, ?model)
16:      price3 = P3(?place, ?date, ?type, ?model)
17:      result min(price1, price2, price3)

```

Fonte: Próprio Autor

A Listagem de Código 8 apresenta uma mesma definição de serviço utilizando a sintaxe WSMML (a esquerda) e a sintaxe SWoTPADL (a direita). O código WSMML possui o seguinte comportamento. As linhas 1-2 mostram a variante WSMML adotada (*WSMML-rule*). Em seguida, as linhas 3-6 definem o *namespace*, que serve para criar o *URI* do serviço e também para definir rótulos para outros *namespaces* adotados nesse serviço. O *URI* usado para o serviço foi *http://pad.br.usp/travel*. Foram definidas os rótulos *car*, *rental* e *foaf* para referenciar as ontologias de carro, aluguel e pessoas, respectivamente. As linhas 7-40 são destinadas para descrição de serviço, introduzidas por meio do construtor *webService*. As ontologias necessárias para definir este serviço são importadas nas linhas 8-9. As linhas 10-22 definem uma *capability*, que corresponde a uma descrição do serviço por meio de axiomas. Nesse exemplo, foram associados apenas axiomas para os construtores *precondition* e *postcondition* do serviço.

No exemplo, o serviço apresenta como *precondition* o nome (*?nome*) do usuário, o local (*?place*), a data (*?date*), o tipo (*?type*) e o modelo (*?model*) do veículo, isto é, o conjunto de dados necessários para realizar o aluguel de um veículo. Como *postcondition*, o serviço apresenta a relação *rented*. Caso exista algum veículo para o dia e hora informados pelo solicitante, a relação *rented* é criada e associa os parâmetros fornecidos ao solicitante.

O comportamento do serviço é descrito por meio de uma *choreography* (linhas 23-40). A coreografia é definida através de uma *interface* (linha 23), que importa as

Listagem de Código 8: Serviço RentACar em WSMML (a esquerda) e em SWoTPADL (a direita).

<pre> 1: wsmlVariant _"http://www.wsmo.org/wsml/ 2: wsml-syntax/wsml-rule" 3: namespace { _"http://pad.br.usp/travel#", 4: car _"http://pad.br.usp/cars#" 5: re _"http://pad.br.usp/rental#", 6: foaf _"http://xmlns.com/foaf/spec/" } 7: webService RentACar 8: importsOntology{ car#CarOntology, 9: re#RentalOntology, foaf#Foaf} 10: capability RentACarCapability 11: sharedVariable?name, ?place, ?date, ?type, 12: ?model 13: precondition 14: definedBy 15: ?name memberOf foaf#name and ?place 16: memberOf re#Place and ?date 17: memberOf re#date and ?type 18: memberOf car#type and ?model memberOf 19: car#model 20: postcondition 21: definedBy 22: rented(?name, ?place, ?date, ?type, ?model) 23: interface RentACarInterface 24: choreography RentACarChoreography 25: importsOntology{ 26: car#CarOntology, 27: re#RentalOntology 28: } 29: transitionRules rentACarRule 30: forall ?place, ?date, ?type, ?model 31: with 32: (?place memberOf re#Place and ?date 33: memberOf re#date and ?type memberOf 34: car#type and ?model memberOf car#model 35: and re#carAvailable(?place, ?date, 36: ?type, ?model)) 37: do 38: add(_#1 rented (?name, ?type, ?model, 39: ?date, ?place)) 40: endForall </pre>	<pre> 1: _"http://pad.br.usp/travel#" 2: import { car _"http://pad.br.usp/cars#" 3: re _"http://pad.br.usp/rental#", 4: foaf _"http://xmlns.com/foaf/spec/" } 5: service RentACar extends re#rentalService 6: precondition 7: ?name memberOf foaf#name ?place 8: and memberOf re#Place and ?date 9: memberOf re#date and ?type 10: memberOf car#type and ?model 11: memberOf car#model 12: postcondition 13: rented(?name, ?place, ?date, ?type, 14: ?model) 15: behavior 16: procedure makeRental() 17: add("rented(?name, ?place, ?date, 18: ?type, ?model)") </pre>
---	---

Fonte: Próprio Autor

Fonte: Próprio Autor

ontologias necessárias através do construtor *importsOntology* (linhas 25-28). O construtor *TransitionRules* define coreografias por meio de regras que promovem mudanças de estado. No exemplo, o comando *forall* avalia a existência de um veículo que atenda às condições informadas (linhas 30-36). Caso exista, o relacionamento *rented* é adicionado à base de conhecimento com os dados passados (linhas 37-40).

A linguagem SWoTPADL cria novos construtores que simplificam essa definição. A linha 1 da especificação SWoTPADL (a direita) define o *namespace* do serviço. As linhas 2-4 importam as ontologias necessárias. Para criação de serviços, SWoTPADL usa a palavra reservada *service*, seguido opcionalmente por *extends* e um tipo de serviço. No exemplo, o serviço *RentACar* foi definido como uma especialização de *re#rentalService*. As linhas 6-11 descrevem as expressões associados a *precondition* e a *postcondition*.

Finalmente, as linhas 15-18 destinam-se a definir o comportamento do serviço. Isso é feito através do *procedure makeRental* (linha 56). O procedimento cria uma instância da relação *rented* se a *precondition* for atendida.

4.2.2 Descrição dos Ambientes

Conforme apresentado no Capítulo 3, *Ambiente* é a entidade central de nosso modelo conceitual. Todas as demais entidades do modelo se relacionam com o *Ambiente*. Nesse intuito, utilizamos uma notação simples para a descrição de ambientes, bem como dos elementos que o compõe.

Três fatores influenciaram nessa escolha: a necessidade de adotar uma representação cuja comunidade de desenvolvedores de software já estivesse familiarizado; representação compatível com ferramentas de desenho assistido por computador (CAD) e por último, a opção por uma notação leve e flexível, de fácil manipulação. A partir dessas premissas, adotamos uma notação baseada no uso de coleções, através da linguagem JSON (BRAY, 2017).

Para ilustrar um exemplo de descrição, considere a Figura 27, que representa uma sala de impressão de uma empresa. Tal sala é dotada de alguns elementos de sensoriamento (trava biométrica, sensor de presença), elementos suscetíveis a atuação (lâmpada) e um dispositivo provedor de serviço (Impressora). O ambiente apresenta um *Ator*, de nome Gilmar, funcionário da empresa.

O mapeamento desse ambiente para JSON é apresentada na Listagem de Código 9. Todas entidades do ambiente, incluindo o próprio ambiente, são mapeados para as

Figura 27: Exemplo de Ambiente - Sala de Impressão

Fonte: Próprio autor

respectivas entidades do modelo conceitual através do atributo *memberOf*.

As linhas 2-5 são dedicadas para a descrição do ambiente principal. O *Ambiente Empresa* representa todo o espaço físico da empresa. Ele tem como componentes o *subAmbiente SalaDeImpressao*, apresentado na Figura 27. Adicionalmente, definimos

Listagem de Código 9: Descrição do *Ambiente* Sala de Impressão em JSON

```

1: {
2:   "Empresa":{"subAmbiente": ["SalaDeImpressao", ...],
3:     "localizacao": [639, 524, -78, -63],
4:     "memberOf": "swotpad#Ambiente"
5:   },
6:   "SalaDeImpressao":{"dispositivos": ["sensorDePresencaSI", "impressoraSI", "lampadaSI", "travaBiome-
   tricaSI"],
7:     "memberOf": "swotpad#Ambiente",
8:     "localizacao": [444, 398, -10, -35],
9:     "propriedades": {'emUso' : 'true'}
10:    "atores": [Gilmar]
11:  },
12:  "lampadaSI": {"memberOf": "swotpad#Dispositivo",
13:    "propriedades": {'state' : 'on'}
14:  },
15:  "sensorDePresencaSI": {"memberOf": "swotpad#Dispositivo",
16:    "propriedades": {'detected' : 'true'}
17:  },
18:  "impressoraSI": {"memberOf": "swotpad#Dispositivo"},
19:    "habilidades": [imprimir],
20:  },
21:  "travaBiometricaSI": {"memberOf": "swotpad#Dispositivo"},
22:  "Gilmar": {"memberOf": "swotpad#Ator}
23: }
```

Fonte: Próprio Autor.

a localização e associamos o tipo de *Empresa*. As linhas 6-11 são dedicadas a descrição do *Ambiente SalaDeImpressao*. Nele podemos ver a definição dos dispositivos que estão associados a esse ambiente (*sensorDePresencaSI*, *impressoraSI*, *lampadaSI* e *travaBiométrica*) e o *Ator Gilmar*, que está fazendo uso do ambiente. Por conta disso,

a propriedade *emUso* de *SalaDeImpressao* apresenta valor *true*. Também foi definida a localização do ambiente.

As linhas 12-21 são dedicadas a definição de cada dispositivo presente no ambiente. Cada um deles foi associado a entidade *Dispositivo* do modelo conceitual através do conceito *swotpad#Dispositivo*. Adicionalmente, foram definidos valores para algumas propriedades, tais como a propriedade *state* de *lampadaSI* com valor *on* e a propriedade *detected* do *sensorDePresencaSI* com valor *true*. A última linha é dedicada a definição do *Ator Gilmar*.

4.2.3 Mapeamento e Tradução

A etapa de Mapeamento e Tradução trabalha com as descrições apresentadas nas seções 4.2.1 e 4.2.2 e, a partir delas, gera a ontologia do ambiente e dos serviços. Além das ontologias, a ferramenta também gera serviços *Restful* a partir das descrições de serviço SWoTPADL.

Todas as ontologias empregadas em nossa abordagem são definidas em WSMML. Para melhor entendimento do trabalho efetuado por essas ferramentas, iremos retomar os exemplos presentes nas listagens de código 7 e 9. A partir da especificação de *SearchPrice*, a ferramenta de Mapeamento e Tradução gera a ontologia apresentada na Listagem de Código 10.

Toda ontologia gerada a partir de um *service* pela ferramenta de Mapeamento e Tradução possui o prefixo *mapse* (linha 6), que indica que essa é uma ontologia oriunda de um *service*. Adicionalmente, seu sufixo será sempre o nome do *service* que deu origem a essa ontologia. A *precondition* e *postcondition* definidas em *SearchPrice* dão origem ao *axiom Precondition* e *PostCondition* na ontologia (linhas 10-16).

Em seguida, é definido o conceito *SearchPrice*. Todo *service* SwotPADL é mapeado para um conceito. Caso o *service* seja uma especialização de outro *service*, o conceito correspondente ao primeiro *service* deve ser um *subconceito* do segundo. A linha 17 apresenta a definição de *SearchPrice*, a qual representa um serviço que é uma especialização de *rental#rentalValue*. Os demais campos de *service* são mapeados por meio da instância de conceito *instanceSearchPrice* (linhas 18-26). Na instância, são definidos os valores de *compCapacities* (linhas 19-21), bem como o valor de *behavior* (linhas 23-26). Por último, é descrito um exemplo de associação de serviço a uma entidade. Uma instância de *ServicoWeb* denominada *wsComparador* (linha 28) é definida e

Listagem de Código 10: Ontologia do *service Search*, presente na Listagem de Código 7

```

1: wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
2: namespace { _"http://pad.br.usp/travel#",
3:   car _"http://www.wsmo.org/sws-challenge/cars#",
4:   rental _"http://www.wsmo.org/sws-challenge/rental#",
5:   swotpad _"http://pad.br.usp/swotpad#" }
6: ontology mapse_SearchPrice
7:   importsOntology{
8:     car#CarOntology, rental#RentalOntology,
9:     swotpad#SwoTPADOntology }
10: axiom Precondition
11:   definedBy
12:     ?place memberOf rental#Place and ?date memberOf rental#date
13:     and ?type memberOf car#type and ?model memberOf car#model.
14: axiom PostCondition
15:   definedBy
16:     ?price memberOf rental#price.
17: concept SearchPrice subConceptOf rental#rentalValue
18: instance instanceSearchPrice memberOf SearchPrice
19:   compCapacities hasValue "PAR(P1:rental#rentalValue,
20:     P2:rental#rentalValue,
21:     P3:rental#rentalValue)"
22:   behavior hasValue "procedure bestPrice() price1 =
23:     P1(?place, ?date, ?type, ?model)
24:     price2 = P2(?place, ?date, ?type, ?model)
25:     price3 = P3(?place, ?date, ?type, ?model)
26:     result min(price1, price2, price3)"
27:   usadoPor hasValue wsComparador
28: instance wsComparador memberOf swotpad#WebService

```

Fonte: Próprio Autor.

associada a *instanceSearchPrice* (linha 27), por meio do atributo *usadoPor*.

Um trabalho similar é feito com as descrições do ambiente em JSON. A Listagem de Código 11 apresenta a ontologia gerada a partir do arquivo de descrição do ambiente Sala de Impressão, apresentado na Listagem de Código 9. Cada objeto presente no primeiro nível da especificação JSON dá origem a uma instância na ontologia WSML. Os nomes associados a esse objeto correspondem ao nome da instância e cada valor, a atributos da instância.

O arquivo de ontologia gerado através do arquivo de descrições de ambiente são registrados no *namespace* _"http://pad.br.usp/Ambientes/<SuperAmbiente>#", onde <SuperAmbiente> corresponde ao *Ambiente* que contém os demais ambientes. Caso o arquivo apresente mais de um <SuperAmbiente>, é gerado uma ontologia para cada um deles. No exemplo da Listagem de Código 9, <SuperAmbiente> corresponde ao objeto Empresa. Dessa forma, o *namespace* tem a designação _"http://pad.br.usp/Ambientes/Empresa#" (linha 2).

As ontologias geradas a partir dos arquivos de descrição de ambiente apresentam o

nome composto pelo prefixo *mapamb*, seguido pelo nome do *<SuperAmbiente>* (linha 4). Elas importam a ontologia *SWoTPADOntology*, uma ontologia que apresenta todas as entidades definidas no modelo conceitual do Capítulo 3.

Listagem de Código 11: Ontologia da descrição do *Ambiente SalaDeImpressao*

```

1: wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
2: namespace { _"http://pad.br.usp/Ambientes/Empresa#",
3:   swotpad _"http://pad.br.usp/swotpad#" }
4: ontology mapamb_Empresa
5:   importsOntology{
6:     swotpad#SwoTPADOntology}
7:   instance Empresa memberOf swotpad#Ambiente
8:     subAmbientes hasValue {SalaDeImpressao}
9:     localizacao hasValue {639, 524, -78, -63}
10:  instance SalaDeImpressao memberOf swotpad#Ambiente
11:    localizacao hasValue {444, 398, -10, -35}
12:    dispositivos hasValue {sensorDePresencaSI, impressoraSI, lampadaSI, travaBiometricaSI}
13:    propriedades hasValue {"emUso", true}
14:    atores hasValue {Gilmar}
15:  instance lampadaSI memberOf swotpad#Dispositivo
16:    propriedades hasValue {"state", "on"}
17:  instance sensorDePresencaSI memberOf swotpad#Dispositivo
18:    propriedades hasValue {"detected", true}
19:  instance impressoraSI memberOf swotpad#Dispositivo
20:    habilidades hasValue {imprimir}
21:  instance sensorDePresencaSI memberOf swotpad#Dispositivo
22:    propriedades hasValue {"detected", true}
23:  instance impressoraSI memberOf swotpad#Dispositivo
24:    habilidades hasValue {imprimir}
25:  instance travaBiometricaSI memberOf swotpad#Dispositivo
26:  instance Gilmar memberOf swotpad#Ator

```

Fonte: Próprio Autor.

As demais linhas correspondem ao mapeamento dos objetos presentes no arquivo de descrição JSON. Ao todo, o arquivo define sete objetos: *Empresa*, *SalaDeImpressao*, *lampadaSI*, *SensorDePresencaSI*, *impressoraSI*, *travaBiometricaSI* e *Gilmar*. As linhas 7-14 são destinadas as definições das instâncias de *Ambiente Empresa* e *SalaDeImpressao*. A associação ao tipo *Ambiente* é feita por meio do valor do atributo *memberOf*, presente na especificação JSON. Os campos *subAmbiente* e *localizacao* de *Empresa* são mapeados para atributos da instância *Empresa*. Da mesma forma, os campos *dispositivos*, *localizacao*, *propriedades* e *atores* de *SalaDeImpressao* são mapeados para atributos da instância *SalaDeImpressao*. Analogamente, são definidos os demais objetos do arquivo JSON. As instâncias de *Dispositivo* nas linhas 15-25 e a instância de *Ator* na linha 26.

Por último, apresentamos o serviço criado a partir da definição SWoTPADL. A versão atual do *framework* apresenta suporte a geração de código fonte para a linguagem de programação Python. Os serviços gerados por meio da ferramenta de tradução faz

uso da biblioteca Flask (GRINBERG, 2014), comumente usada em Python para o desenvolvimento de serviços *RESTful*. Os códigos gerados pela ferramenta de mapeamento e tradução visa a geração de protótipos para validação do serviço, bem como servem como ponto de partida para a etapa de codificação do serviço.

A Listagem de código 12 apresenta o código relativo ao *service SearchPrice*, especificado na Listagem de Código 7. A seguir, explicaremos seu comportamento. As linhas 1 e 2 realizam as importações das bibliotecas necessárias ao serviço, são elas a biblioteca *Flask* e a biblioteca do *framework swotpad*. Na linha 3, criamos um objeto *Flask*. Por meio dele, criaremos o serviço. As linhas 4-7 definem *dr*. A variável *dr* armazena o resultado da função *deduce*. A função *deduce* é uma rotina presente na biblioteca *swotpad* que realiza consultas em ontologias. Para a consulta do exemplo, o resultado de *deduce* é uma coleção do tipo *chave/valor*, com um conjunto de valores associados às variáveis *?place*, *?date*, *?type* e *?model*. A próxima linha faz uso do decorator *route()*, que informa ao *Flask* que o sufixo de URL *SearchPrice* deve ser associada ao método *bestPrice*. As linhas 9 a 13 descrevem o método *bestPrice* que faz chamadas aos serviços *P1*, *P2* e *P3*. Ao final, *bestPrice* devolve o valor do serviço que apresentou menor valor. Na linha 14, chamamos o método *run* do objeto *app*. Nesse momento, o serviço entra em execução e está apto a responder solicitações.

Listagem de Código 12: Código relativo ao *service SearchPrice*

```

1: from flask import Flask
2: from swotpad import *
3: app = Flask( __name__ )
4: dr = deduce('?place memberOf rental#Place and
5:           ?date memberOf rental#date
6:           and ?type memberOf car#type and
7:           ?model memberOf car#model')
8: @app.route("/SearchPrice")
9: def bestPrice():
10:  price1 = P1(dr['place'], dr['date'], dr['type'], dr['model'])
11:  price2 = P2(dr['place'], dr['date'], dr['type'], dr['model'])
12:  price3 = P3(dr['place'], dr['date'], dr['type'], dr['model'])
13:  result deduce('min(' + 'price1' + ',' + 'price2' + ',' + 'price3' + ')')
14: app.run(host='http://pad.br.usp/travel', debug=TRUE, use_reloader=TRUE)

```

Fonte: Próprio Autor.

4.2.4 Requisição e Resposta

A interação dos atores com o sistema de IoT é feita através de requisições (*request*). A *request* é uma construção da linguagem SWoTPADL que permite aos atores descrever demandas por meio de precondições (*preconditions*) e pós-condições (*postconditions*). O papel delas em uma requisição é similar ao que elas apresentam em um serviço,

ou seja, descrever os dados de entrada e saída, bem como dados relativos ao estado do sistema antes e após o atendimento da requisição.

Em nosso *framework*, a *request* substitui a construção *goal* de WSML. Assim como em *service*, *request* elimina algumas construções de *goal*, no intuito de tornar mais simples as especificações de requisição.

A sintaxe de uma *request* é bastante similar a de um *service*. A Listagem de Código 13 apresenta a gramática que representa uma *request* SWoTPADL. A única diferença do construtor *request* para o construtor *service* é a ausência do construtor *behavior*.

Listagem de Código 13: Sintaxe de uma *request* SWoTPADL.

```

1: <reqSwotpadl> ::= <namespace> <impOntologies>? <defRequest>
2: <namespace> ::= URI
3: <impOntologies> ::= 'import' '{ ' ID URI (',' ID URI)* ' }'
4: <defRequest> ::= 'request' ID <precondition> <poscondition> <cCapacities>?
5: <precondition> ::= 'precondition' <exp>*
6: <postcondition> ::= 'postcondition' <exp>*
7: <exp> ::= <exp> <binop> <exp>
8:       | <unarioop> <exp>
9:       | <quantifier> ID (',' ID)* '(' expr ')'
10:      | ID
11:      | LITERAL
12:      | ...
13: <binop> ::= 'memberOf' | 'and' | 'or' | 'implies' | 'impliedBy' | '+' | '-' | '*' | '/' | 'hasValue'
14: <unaryop> ::= 'not' | '-'
15: <quantifier> ::= 'forall' | 'exists'
16: <cCapacities> ::= 'compCapacities' <compCapacities>
17: <compCapacities> ::= 'PAR' '(' <members> ')'
18:       | 'SEQ' '(' <members> ')'
19: <members> ::= ID ':' ID (',' <member>)*
20:       | <compCapacities> (',' <member>)*

```

Fonte: Próprio Autor.

De acordo com gramática, vemos que a definição de um requisição em SWoTPADL é composta por três elementos. O primeiro, dedicado a declaração do espaço de nomes (<*namespace*>) onde será definida a requisição. Em seguida, temos a seção de importação de ontologias (<*impOntologias*>), introduzida pela palavra reservada *import*.

A terceira seção (<*defRequest*>) é dedicada a declaração e definição semântica da requisição. Em SWoTPADL, uma requisição é definida através da palavra reservada *request*, seguida pelo seu identificador. Em seguida, definimos a semântica da requisição por meio de *preconditions* e *postconditions*.

A definição de *request* termina com o construtor *compCapacities*. Esse construtor é opcional, mas se adotado, é utilizado pelo componente de descoberta de servi-

ços, apresentado na Seção 4.3.1. Por meio dele, podemos especificar que a requisição seja atendida somente por serviços que apresentem capacidade igual às especificadas em *compCapacities*.

A Listagem de Código 14 apresenta um exemplo de definição de *request*. *TurnLightson* é uma *request* para ambientes que apresentam um serviço de iluminação automática. Sua operação consiste na detecção de uma pessoa (*?Person*) em algum lugar (*?Loc*) da residência (linha 5). Se a luz estiver desligada (linha 6), o serviço altera o estado para ligado (linha 9).

Listagem de Código 14: SWoTPADL Request

```

1: _"https://gitlab.com/andreluis-ms/IOTPADL-HSS/wikis/turnLightsOn#"
2: hec _"https://gitlab.com/andreluis-ms/IOTPADL-HSS/wikis/home#"
3: request turnLightsOn
4:   precondition
5:     ?person[hec#hasLocalization hasValue ?loc] memberOf hec#HomeUser and
6:     ?lamp[hec#hasLocalization hasValue ?loc, hec#hasState hasValue "off"] memberOf hec#Lamp.
7:   postcondition
8:     ?person[hec#hasLocalization hasValue ?loc] memberOf hec#HomeUser and
9:     ?lamp[hec#hasLocalization hasValue ?loc, hec#hasState hasValue "on"] memberOf hec#Lamp.

```

Fonte: Próprio Autor.

4.2.5 Gerador do Sistema

A ferramenta denominada Gerador do Sistema é responsável por gerar o artefato Solução SWoTPAD, que agrega os elementos providos em nosso framework, para dar suporte ao desenvolvimento de sistemas de IoT. A Figura 28 apresenta maiores detalhes dessa Solução.

A Solução SWoTPADL é composta por todas as classes que representam as entidades do modelo conceitual e permite ao usuário reutilizar, bem como refinar as entidades fornecidas. Adicionalmente, uma série de ontologias são mapeadas e fornecidas por meio de especializações da classe Propriedades. Para dispositivos, adotamos conceitos apresentados pelas ontologias SSN (COMPTON et al., 2012) e IoT-lite (BERMUDEZ-EDO et al., 2016) mapeadas em IoTProp. Para espaços, a ontologia de Acomodações (HEPP, 2013), mapeadas em *EspacialProp*, para propriedades temporais e preferências, a ontologia Time Ontology (HOBBS; PAN, 2006) e RECommendations Ontology (FERRIS; JACOBSON, 2010), respectivamente, através das classes *TemporalProp* e *PreferenciaProp*. As propriedades dos atores usam a ontologia FOAF (BRICKLEY; MILLER, 2007), mapeada em *AtorProp*.

A parte relativa ao provimento do Serviço é formada pelas classes relativa a descoberta

Figura 28: Solução gerada pela ferramenta Gerador do Sistema

Fonte: Próprio autor

de serviços através da interface *IDescoberta* e sua implementação *DescobertaSWoTPAD*. Adicionalmente, apresenta uma classe de gerência de entidades (*GerenteEntidades*) e a classe *SWoTReasoner*, que provê métodos de acesso ao sistema de inferência de regras IRIS (BISHOP; FISCHER, 2008).

O módulo de Composição de Serviços é formada pelos planejadores *MTHierarquicoPlan* e *MTShop2Plan*, ambos implementações da interface *IPlanejador*. Adicionalmente, pelas classes que representam as entidades que apresentam conceitos relacionado a planejamento: *Plano* e *ProblePlanejamento*.

Por último, temos o *IInvocador* e *InvocadorDeServico* que representam, respectivamente, uma interface para invocadores de serviço e a implementação concreta de um invocador simples de serviço.

4.3 A Camada Funcional

A Camada Funcional do *framework* SwoTPADL apresenta um conjunto de ferramentas que dão suporte ao sistema de IoT final. Ela é composta pelo módulo Provedor de Serviço, o módulo de Composição de Serviço, o Repositório de Serviço e o módulo de Invocação de Serviço. Todos esses módulos estão presentes no artefato de código que denominamos Solução SwoTPADL. O desenvolvedor pode estar usando, estendendo ou

modificando esses módulos, de acordo com suas necessidades.

A Figura 29 apresenta a interação do Ator do Sistema de IoT e os componentes da Solução SWoTPADL. Inicialmente, um ator faz uma requisição através de uma *request* SWoTPADL. Esta requisição é repassada para o provedor de serviços, que será responsá-

Figura 29: Interação entre Ator do Sistema de IoT e Solução SWoTPADL

Fonte: Próprio autor

vel por descobrir o serviço mais adequado para atender esta solicitação. Se o repositório de serviços apresentar um serviço adequado, o provedor de serviços o repassa para o invocador. Caso contrário, o provedor de serviços repassa a *request* para o compositor de serviços, bem como informações adicionais sobre o estado atual do ambiente. Esta descrição é chamada de Problema de Planejamento, e a partir dela, o compositor do serviço tentará encontrar um conjunto de serviços que, quando composto, atende a requisição do usuário. Quando a composição é viável, o compositor de serviço produz um plano e envia a descrição semântica do serviço composto para o repositório de serviços. Adicionalmente, o plano é dado ao Invocador, responsável pela invocação dos serviços. Se a invocação ocorrer sem problemas, o Invocador entrega uma resposta ao Ator do Sistema de IoT. Caso contrário, o Invocador elabora uma nova *request* SWoTPADL com base no pedido original e no estado atual do ambiente. A nova solicitação é composta pelas pós-condições e entradas da requisição original, além do estado atual do ambiente.

4.3.1 Provedor do Serviço

O provedor de Serviço é composto pelo módulo de registro de serviço e pelo módulo de descoberta de serviço. O módulo de registro de serviço realiza a gerência dos serviços descritos no *framework*. Ele apresenta funcionalidades de registro e desregistro de serviços. Adicionalmente, provê algumas funções de consulta ao modelo conceitual, ofertadas por

meio da classe GerenteEntidades.

Já o módulo de descoberta tem como objetivo encontrar o serviço mais adequado ao atendimento da requisição do *Ator*. O algoritmo de descoberta usado no *framework* é composto de três fases, são elas:

- Fase 1: Aplicação do algoritmo de casamento de entradas e saídas de Paolucci *et al* (PAOLUCCI et al., 2002). Nesse algoritmo, é feita uma avaliação semântica das entradas e saídas do serviço candidato e da requisição. A partir daí, define um grau de similaridade entre as respectivas entradas e saídas.
- Fase 2: Avaliação das precondições do serviço e da requisição. Precondições podem impor restrições aos valores assumidos pelas entradas do serviço. Assim, é feita uma nova avaliação com as entradas da requisição e do serviço, com o intuito de avaliar o impacto dessas restrições.
- Fase 3: Avaliação da Capacidade. Serviços SWoTPADL podem ser classificados por meio de uma hierarquia que envolve capacidades e habilidades. Requisições SWoTPADL podem especificar capacidades, através de *compCapacities*. Assim, avaliamos se a capacidade do serviço candidato é compatível com a da requisição.

Dessa forma, temos o algoritmo *grauDeAptidao*, apresentado na Listagem de Código 15. O módulo de descoberta o adota para avaliar o quão apto um serviço é para atender uma determinada requisição.

Listagem de Código 15: Algoritmo grauDeAptidao

```

1: def grauDeAptidao(request, service)
2:   duplasEnt, duplasSai                                     # dicionarios
3:   Para cada(entreq, entserv) de (request, service)
4:     encontrar duplas (entserv, entreq) de valor máximo, usando algPaolucci(entserv, entreq);
5:     duplasEnt.append({(entserv, entreq): algPaolucci(entserv, entreq)})
6:   Para cada (saireq, saiserv) de (request, service)
7:     encontrar dupla(saireq, saiserv) de valor máximo usando algPaolucci(saireq, saiserv);
8:     duplasSai.append({(saiserv, saireq): algPaolucci(saireq, saiserv)})
9:    $Fase_1 = \sum_{i=1}^n (duplasEnt[ent_{servi}, ent_{reqi}] / n + \sum_{i=1}^k (duplasSai[sai_{servi}, sai_{reqi}] / k)$ 
10:  Fase2 = avaliaPrecondicao(duplasEnt.keys(), request, service)
11:  Fase3 = avaliaCapacidade(request, service)
12:  return (Fase1*4/3 + Fase2*3/2 + Fase3 * 2) / 9

```

Fonte: Próprio Autor.

O algoritmo recebe como entrada uma requisição e um serviço e apresenta dois dicionários, *duplasEnt* e *duplasSai*. A variável *duplasEnt* é um dicionário cuja chave é uma dupla (*ent_{serv}*, *ent_{req}*), onde *ent_{serv}* representa uma das entradas do serviço e *ent_{req}* uma das entradas da requisição. Inicialmente, essa coleção é vazia e vai

sendo populada durante a execução do algoritmo. De maneira análoga, temos o dicionário *duplasSai* para as saídas dos serviços e requisições.

A Fase1 do algoritmo é apresentada nas linhas 3 a 9. Nas linhas 3 a 5, usamos o algoritmo *algPaolucci* para descobrirmos as duplas de entrada de requisição e entrada de serviço, com tipos semânticos mais próximos. Armazenamos essas duplas em *duplasEnt*, bem como o valor calculado por *algPaolucci*. Esse mesmo processo é realizado com as saídas (linhas 6-8). Ao final, atribuímos um resultado numérico a Fase1. O valor de Fase1 é a média aritmética do somatório dos valores calculados por *algPaolucci*, atribuídos a cada dupla de *duplasEnt* e *duplasSai*. Fase1 apresenta um valor no intervalo [0-3], onde 3 representa o grau de casamento máximo, de acordo com (PAOLUCCI et al., 2002).

A Fase2 do algoritmo é apresentada na linha 10. Ela consiste na chamada do algoritmo *avaliaPrecondicao* e avalia o impacto das precondições do serviço e da requisição sobre os valores que as entradas da requisição e do serviço podem assumir. O algoritmo *avaliaPrecondicao* devolve um valor no intervalo de [0-2], onde 2 representa o grau de casamento máximo.

Por último temos a Fase 3 do algoritmo, realizada através de *avaliaCapacidade*, que devolve o valor 1 quando a capacidade associada ao serviço e a requisição são iguais e 0, caso contrário. Ao final, aplicamos a fórmula da linha 12 para obtermos um valor $\mathbb{R}$ na escala [0-1].

A Listagem de Código 16 apresenta o algoritmo de Paolucci *et al* que define o grau de casamento entre duas entradas ou duas saídas. Apenas modificamos o algoritmo associando valores a cada um dos casamentos possíveis. Associamos três para *exact*, dois para *plugin*, um para *subsumes* e zero para *fail*.

Listagem de Código 16: Algoritmo de definição de graus de casamento de saídas. Adaptado de (PAOLUCCI et al., 2002)

```

1: degreeOfMatch(par1,par2):
2:   if par1=par2 then return 3           # exact
3:   if par2 subclassOf par1 then return 3 # exact
4:   if par1 subsumes par2 then return 2   # plugin
5:   if par2 subsumes par1 then return 1   # subsumes
6:   else 0                               # fail

```

Fonte: Próprio Autor.

O algoritmo *degreeOfMatch* tem como parâmetros *par1* e *par2*, que representam entradas ou saídas de serviços e requisições. A partir desses parâmetros, o algoritmo avalia seu grau de casamento. Para ilustrar melhor a ideia, utilizaremos a Figura 30 que apresenta uma taxonomia de veículos. Considerando o casamento entre saídas, podemos ter os seguintes cenários:

- *exact*, quando as saídas estão associadas a uma mesma classe ou quando a saída da requisição é uma subclasse direta da saída do serviço. Por exemplo, *outR* do tipo *Hatch* e *outA* do tipo *Carro*;
- *plugin*, quando a saída da requisição é uma subclasse indireta da saída do serviço. Por exemplo *outR* do tipo *Hatch* e *outA* do tipo *Veiculo*;
- *subsumes*, quando a saída do serviço é um subconjunto da saída da requisição. Por exemplo *outR* do tipo *Veiculo* e *outA* do tipo *Carro*;
- *fail*, quando não há relação entre a saída do serviço e da requisição é estabelecida.

Figura 30: Exemplo de taxonomia de veículos

Fonte: Próprio autor

O segundo algoritmo, *avaliaPrecondicao*, é apresentado na Listagem de Código 17. Ele apresenta como parâmetro *entradas*, uma coleção que apresenta duplas de entrada (*e1*, *e2*), onde *e1* representa uma entrada da requisição e *e2*, uma entrada do serviço.

Listagem de Código 17: Algoritmo *avaliaPrecondicao*

```

1: def avaliaPrecondicao(entradas, service, request):
2:   entradaRequest = deduce(request.precondition)
3:   entradaServico = deduce(service.precondition)
4:   Para (er, es) em entradas:
5:     Se entradaRequest[er]  $\subseteq$  entradaRequest[es]: score = score + 2
6:     Se entradaRequest[er]  $\supset$  entradaRequest[es]: score = score + 1
7:   return score / len(entradas)
  
```

Fonte: Próprio Autor.

Adicionalmente, recebe o serviço e a requisição. Com esses dados, o algoritmo faz inferência na base de conhecimento, utilizando a expressão presente na *precondition* da requisição e na *precondition* do serviço (linhas 2 e 3), através do método *deduce*. O método *deduce(request.precondition)* devolve uma coleção, onde a chave são as entradas da *request* e o valor, o conjunto de instâncias que podem ser associadas a cada uma dessas entradas. Isso é possível pois o sistema lógico de SWoTPAD adota a hipótese do mundo fechado, na qual o conhecimento que não está explicitamente descrito na

base é considerado falso. De maneira análoga, também fazemos essa inferência usando a *precondition* de *service*. Em seguida, nas linhas 4 a 6, comparamos o conjunto de instâncias associados a cada entrada de *request* (*er*) com o conjunto de instâncias associados a cada entrada de *service* (*es*), vinculando a relação subconjunto, superconjunto ou conjuntos disjuntos entre *er* e *es*.

A relação desejada é a de *subconjunto*, pois indica que os possíveis valores associados a entrada da requisição é um subconjunto dos possíveis valores associados a entrada do serviço. Nesse caso, *service* contempla qualquer possibilidade de instância fornecida por *request*. Assim, associamos o valor dois a essa relação. A relação de *superconjunto* não é a ideal, pois podem existir instância na entrada da *request* que não serão atendidas pelo serviço. Porém, existe a possibilidade do serviço atender parcialmente as requisições do usuário. Por conta disso, associamos o valor um a essa relação. Por último, no caso das entradas serem disjuntas, atribuímos o valor zero. Ao final, o algoritmo devolve a média aritmética associada a cada dupla de entrada presente em *entradas*.

O último algoritmo, o *avaliaCapacidade*, é apresentado na Listagem de Código 18. Trata-se de um algoritmo bastante simples. Ele recebe a categoria associada ao serviço e a requisição e devolve o valor um, caso pertençam a mesma categoria ou 0, caso contrário.

Listagem de Código 18: Algoritmo *avaliaCapacidade*

```

1: def avaliaCapacidade(service, request):
2:   Se (service.capacidade == request.capacidade): return 1
3:   Senão return 0

```

Fonte: Próprio Autor.

4.3.2 Compositor de Serviço

Para a composição automática de serviços, utilizamos técnicas de planejamento de inteligência artificial. Planejamento é um processo abstrato e explícito de deliberações que escolhe e organiza ações antecipando seus resultados esperados. Esta deliberação visa alcançar objetivos pré-definidos da melhor forma possível (GHALLAB; NAU; TRAVERSO, 2004).

Nossa abordagem adota planejamento hierárquico (HTN), definido na Seção 2.4.3. Em um planejador HTN, o objetivo é realizar algum conjunto de tarefas. A entrada para o sistema de planejamento é um conjunto de operadores e um conjunto de métodos, cada um dos quais é uma receita de como decompor alguma tarefa em algum conjunto de sub-tarefas. O planejamento passa pela decomposição de tarefas não primárias de forma recursiva em sub-tarefas menores, até que sejam alcançadas tarefas primitivas que podem

ser executadas diretamente usando os operadores de planejamento (GHALLAB; NAU; TRAVERSO, 2004).

Uma grande vantagem é que o conhecimento pode ser codificado, não apenas nas sequências de tarefas especificadas em cada decomposição, mas também nas condições prévias para as decomposições. Para alguns domínios, os planejadores HTN conseguem gerar planos grandes com muito pouca busca (RUSSELL, 2010).

A Figura 31 apresenta o mapeamento dos conceitos de planejamento hierárquico para os conceitos do Modelo Conceitual apresentado Capítulo 3. Tarefas foram mapeadas para *Capacidades*, métodos e ações para *Habilidades*. Uma habilidade é simples ou primitiva quando é realizada totalmente por uma entidade e composta, quando é realizada a partir de uma composição de capacidades e/ou habilidades primitivas. Em nosso modelo conceitual, as entidades que apresentam habilidade são os serviços e os atores.

Figura 31: Mapeamento dos Conceitos de HTN para os Conceitos do Modelo Conceitual

Fonte: Próprio Autor

A seguir, apresentamos o problema do deslocamento, no qual aplicamos esse mapeamento. Seja p , um indivíduo que deseja se deslocar de um lugar para outro. Chamaremos de *pontoOrigem* o local de partida e *pontoDestino*, o local onde o indivíduo p deseja chegar.

A Listagem de Código 19 apresenta a *Capacidade Deslocar* e as *Habilidades PasseioAPé*, *PasseioDeBicicleta* e *PasseioDeUber* que realizam *Deslocar*, utilizando a notação SWoTPADL.

As *Habilidades PasseioAPé*, *PasseioDeBicicleta* e *PasseioDeUber* apresentam

Listagem de Código 19: O problema de deslocamento: Capacidade e as Habilidades que a realizam

```

1: [Capacidade]
2:   Deslocar(?p, ?pontoOrigem, ?pontoDestino)
3: [Habilidades]
4:   PasseioAPé extends Deslocar
5:     precondition
6:       ?p memberOf Ator and ?pontoOrigem memberOf Lugar and em(?p, ?pontoOrigem)
7:     postcondition
8:       ?p memberOf Ator and ?pontoDestino memberOf Lugar and em(?p, ?pontoDestino)
9:   PasseioDeBicicleta extends Deslocar
10:    precondition
11:      ?p memberOf Ator and ?pontoOrigem memberOf Lugar and em(?p, ?pontoOrigem)
12:    postcondition
13:      ?p memberOf Ator and ?pontoDestino memberOf Lugar and em(?p, ?pontoDestino)
14:    subCapacities SEQ(P1: PegarBicicleta, P2: Pedalar)
15:   PasseioDeUber extends Deslocar
16:     precondition
17:       ?p memberOf Ator and ?pontoOrigem memberOf Lugar and em(?p, ?pontoOrigem)
18:     postcondition
19:       ?p memberOf Ator and ?pontoDestino memberOf Lugar and em(?p, ?pontoDestino)
20:     subCapacities SEQ(P1: ChamarUber, P2: AndarDeUber, P3: PagarUber)
21:   PegarBicicleta
22:     precondition
23:       ?p memberOf Ator and neg ?p[instrumento hasValue bicicleta]
24:     postcondition
25:       ?p [instrumento hasValue bicicleta] memberOf Ator
26:   Pedalar
27:     precondition
28:       ?p [instrumento hasValue bicicleta] memberOf Ator and ?pontoOrigem memberOf Lugar and
29:       em(?p, ?pontoOrigem)
30:     postCondition
31:       ?p [instrumento hasValue bicicleta] memberOf Ator and ?pontoDestino memberOf Lugar and
32:       em(?p, ?pontoDestino)
33:   ChamarUber
34:     precondition
35:       ?p[instrumento hasValue ?s] memberOf Passageiro and em(?x, ?pontoOrigem) and
36:       ?s[app hasValue uberApp] memberOf Smartphone
37:     postcondition
38:       ?x memberOf MotoristaUber and em(?x, ?pontoOrigem)
39:   AndarDeUber
40:     precondition
41:       ?pontoOrigem memberOf Lugar ?x memberOf MotoristaUber and em(?x, pontoOrigem) and ?p
42:       memberOf Passageiro and em(?x, ?pontoOrigem)
43:     postcondition
44:       ?pontoDestino memberOf Lugar ?x memberOf MotoristaUber and em(?x, pontoDestino) and ?p
45:       memberOf Passageiro and em(?x, ?pontoDestino)
46:   PagarUber
47:     precondition
48:       ?p [dinheiro hasValue ?m] memberOf Passageiro and em(?p, ?pontoDestino) and ?m > ?valorUber
49:     postcondition
50:       ?x memberOf MotoristaUber and ?p memberOf Passageiro and ?pontoDestino, ?local memberOf
51:       Lugar em(?p, ?pontoDestino) and em(?x, ?local) and lugar <> ?pontoDestino

```

Fonte: Próprio Autor.

como *precondition* um dado ator *?p* e um local de origem (*?pontoOrigem*) e sinalizam que o ator se encontra nesse local, através da relação *em*. Como *postcondition*, o mesmo

ator $?p$ agora se encontra no local de destino ($?pontoDestino$). As definições dessas Habilidades estão presentes nas linhas 4 a 20.

PasseioAPé trata-se de uma habilidade simples ou primitiva, ou seja, ela não depende de outras habilidades ou capacidades para realizar a capacidade deslocar. Já *PasseioDeBicicleta* e *PasseioDeUber* são habilidades compostas. *PasseioDeBicicleta* depende da Habilidade *PegarBicicleta*, que representa a habilidade de um *Ator* de pegar uma bicicleta, e também da *Habilidade Pedalar*, cuja *precondition* e *postcondition* sinalizam condições para que a ação *Pedalar* comece, bem como, o que acontece após seu término.

PasseioDeUber é composto pelas habilidades *ChamarUber*, *AndarDeUber* e *PagarUber*. A primeira delas, representa a atividade de chamar o Uber através de seu aplicativo. Para isso, o *Ator*, agora visto como *Passageiro*, deve estar no $?pontoOrigem$ e possuir um *Smartphone* ($?s$) com o aplicativo *uberApp*. A segunda *Habilidade* (*AndarDeUber*) representa o deslocamento do Uber do $?pontoOrigem$ ao $?pontoDestino$. Por último, temos a *Habilidade PagarUber*, cuja *precondition* avalia se o passageiro possui dinheiro ($?m$) para pagar a viagem ($?valorUber$) e se o mesmo já se encontra no destino ($?em(?p, ?pontoDestino)$). Como *postcondition*, o motorista do Uber ($?x$) e o passageiro ($?p$) devem estar em locais diferentes. Isso ocorre, pois concluído o deslocamento, o motorista segue viagem para buscar o próximo passageiro.

Aplicando-se o planejamento hierárquico para esse problema, poderíamos ter as árvores de decomposição apresentada na Figura 32. Perceba que como as habilidades *PasseioAPé*, *PasseioDeBicicleta* e *PasseioDeUber* apresentam a mesma *precondition*, podemos aplicar qualquer uma delas para realizar a *Capacidade Deslocamento* em qualquer cenário.

Figura 32: Possíveis decomposições de *Deslocar* e paralelização

Fonte: Próprio Autor

Dessa característica que surgem as extensões feitas nos algoritmos de planejamento adotados neste trabalho. O algoritmo *HIERARCHICAL_SEARCH* (RUSSELL, 2010) e o algoritmo ND-SHOP2 (KUTER, 2006) realizam a busca do plano de forma sequencial. Nossas modificações permitem paralelizar a busca do plano, no momento em que é identificado que mais de uma habilidade pode realizar uma dada capacidade. A Figura 32 ilustra esse comportamento.

A listagem de código 20 apresenta a versão paralela do algoritmo de planejamento hierárquico *HIERARCHICAL_SEARCH*. As linhas destacadas em azul (14 a 16) representam os pontos de modificação feitos nesse algoritmo quando comparado ao original.

Listagem de Código 20: O algoritmo de planejamento MTHierarquicoPlan. Adaptado de (RUSSELL, 2010)

```

1: def MTHierarquicoPlan(problema, hierarquia, plano = [Capacidade]): #retorna uma solução ou falso
2:   while True:
3:     if plano == []:
4:       return False
5:     capacidade = peguePrimeiraCapacidade(plano)
6:     habilidades_antes = habilidadesAntesDe(capacidade, plano)
7:     habCap_depois = habilidadesECapacidadesDepoisDe(capacidade, plano)
8:     estadoAtual = aplicarHabilidades(problema.estadoInicial, habilidades_antes)
9:     if capacidade == Null:      # Não existe mais capacidades a ser realizadas
10:      if satisfaz(estadoAtual, problem.meta):
11:        return plano
12:     else:
13:       sequencias = decomposicao(capacidade, estadoAtual, hierarquia)
14:       for sequencia in sequencias[1:]:
15:         plano = habilidades_antes + sequencia + habCap_depois
16:         threadPool.add(new thread(MTHierarquicoPlan, problema, hierarquia, plano))
17:       plano = habilidades_antes + sequencias[0] + habCap_depois

```

Fonte: Próprio Autor.

O algoritmo *MTHierarquicoPlan* recebe como parâmetro o problema de planejamento (problema), o conjunto de capacidades e habilidades (hierarquia) e a variável plano. O valor da variável plano inicialmente é igual a capacidade a ser realizada. Se a variável plano se tornar vazia, *MTHierarquicoPlan* retornará False (linhas 3-4). Caso contrário, inicia o processo de realização da *Capacidade* (linhas 5-8).

O processo de realização da *Capacidade* consiste na seleção da primeira capacidade presente na variável plano, para, em seguida, identificar as habilidades que antecedem e sucedem a *capacidade*. O objetivo dessa separação é descobrir o estadoAtual do sistema, obtido por meio da aplicação das habilidades que antecedem a *capacidade*. Caso a *capacidade* seja nula, todas as habilidades presentes em plano são aplicadas. Por conta disso, a linha 9 avalia se *capacidade = Null*, pois caso positivo, o algoritmo avalia se o estadoAtual é um estado meta, ou seja, um estado que satisfaz o problema. Em caso de

satisfação, o conteúdo da variável plano é retornado (linha 11).

As demais linhas apresentam a substituição da primeira capacidade da variável plano pelas habilidades que a realizam (linha 13). Em caso de haver mais de uma habilidade, *threads* são criadas para cada uma delas (linha 14-16) e assim, o algoritmo prossegue até falhar ou retornar um plano.

Listagem de Código 21: Versão multithreading do algoritmo ND-Shop2. Adaptado de (KUTER, 2006)

```

1: def MTND-SHOP2(OPEN, G,  $\pi$ , solved)
2:   if OPEN =  $\emptyset$ : return( $\pi$ )
3:   selecione uma dupla ( $s, w$ )  $\in$  OPEN e a remova de OPEN
4:   if  $s \in G$ : solved = solved  $\cup$  { $s$ }
5:   else if  $s \notin S_\pi$ :
6:     habilidades = { $h$ : Habilidade |  $h \in w$  e  $h$  não possui predecessores}
7:     não-deterministicamente escolha uma  $h$  de habilidades
8:     if  $h$  é uma habilidade primitiva:
9:        $w' = w - \{h\}$ 
10:       $\pi' = \text{append}(s, h, \pi)$ 
11:       $\pi = \pi'$ 
12:      OPEN' = OPEN  $\cup$  {( $s', w'$ ) |  $s' \in \text{aplicarHabilidade}(s, w, h)$ }
13:   else
14:     capacidades = { $c$ : Capacidade |  $c \in w$  e  $c$  não possui predecessores}
15:     não-deterministicamente escolha uma  $c$  de capacidades
16:     sequencias = decomposicao( $c, s$ , hierarquia)
17:     if sequencias =  $\emptyset$ : return Falha
18:     habilidades_antes = habilidadesAntesDe(capacidade, plano)
19:     habCap_depois = habilidadesECapacidadesDepoisDe(capacidade, plano)
20:     for seq in sequencias[1:]:
21:        $w' = \text{habilidades\_antes} + \text{seq} + \text{habCap\_depois}$ 
22:       OPEN' = OPEN  $\cup$  ( $s, w'$ )
23:       threadPool.add(new thread(ND-SHOP2, OPEN', G,  $\pi$ , solved))
24:   else if  $s$  não possui um descendente em (StatesOf(OPEN)  $\cup$  solved)  $\setminus S_\pi$ :
25:     return Falha
26:   return ND-SHOP2(OPEN', G,  $\pi$ , solved)

```

Fonte: Próprio Autor.

O segundo algoritmo implementado, consiste em uma versão paralela para o algoritmo de planejamento SHOP2 não determinístico apresentado em (KUTER, 2006). Estendemos esse algoritmo para uma versão *multithreading* denominada *MTND-SHOP2*, que permite a busca de planos em paralelo. A Listagem de Código 21 apresenta o algoritmo. A entrada do algoritmo é o conjunto de estados de metas G , o plano (π), inicialmente vazio, um conjunto vazio denominado *solved* e o conjunto *OPEN*, que é inicialmente (s, w) | $s \in S_0$ onde S_0 é o conjunto de estados iniciais e w é a rede de tarefas inicial.

Em cada chamada, o algoritmo primeiro seleciona uma tupla (s, w) do conjunto *OPEN* e a remove. Em seguida, decompõe recursivamente as capacidades em w em capacidades e habilidades menores, até que uma habilidade primitiva (isto é, uma habilidade que não pode ser decomposta) seja gerada para o estado atual s . Então, o algoritmo aplica a

habilidade h para s e gera os estados sucessores (linha 12). Isto produz a rede de tarefa sucessora w' a ser realizada em cada estado $s' \in aplicarHabilidade(s, w, h)$. O processo de planejamento procede com cada sucessor (s', w') até que não haja tarefas a serem realizadas.

Listagem de Código 22: ChamarUber não determinístico

```

1:  ChamarUber
2:    precondition
3:      ?p[instrumento hasValue ?s] memberOf Passageiro and em(?x, ?pontoOrigem) and
4:      ?s[app hasValue uberApp] memberOf Smartphone
5:    postcondition
6:      ?x memberOf MotoristaUber and (em(?x, ?pontoOrigem) or
7:      (em(?x, ?lugar) and ?lugar  $\neq$  ?pontoOrigem)).

```

Fonte: Próprio Autor.

Para ilustrar um exemplo de aplicação do algoritmo, faremos uma pequena modificação do problema de deslocamento, apresentado na Listagem de Código 19, com o objetivo de tornar o problema não-determinístico. Considere que a habilidade primitiva *ChamarUber* pode resultar em duas situações. A primeira, já ilustrada, onde o motorista do Uber se dirige até o *?pontoOrigem* e uma segunda, onde o motorista não comparece. A mudança em *ChamarUber* é ilustrada na Listagem de Código 22.

Figura 33: Decomposição com não-determinismo

Fonte: Próprio Autor

Com essa mudança, temos a decomposição ilustrada na Figura 33. Os nós apresentam o valor da dupla $(s, w) \in OPEN$, bem como o valor do plano que está sendo calculado.

Quando existem mais de um nó em um mesmo nível, o valor de *OPEN* naquele nível é determinado por $OPEN_{no_1} \cup OPEN_{no_2} \cup \dots \cup OPEN_{no_n}$.

No momento em que alguma habilidade não determinística é aplicada, *OPEN* recebe a adição de uma dupla (s', w') , onde s' representa um dos possíveis estados resultantes da aplicação da habilidade e w' representa a nova rede de tarefas. No exemplo, isso ocorre quando a habilidade *ChamarUber* é aplicada. No caso do não comparecimento do motorista, o sistema permanece no mesmo estado, porém, a nova rede de tarefas deixa de apresentar a habilidade *ChamarUber*. O segundo cenário representa a situação desejável, no qual o Uber comparece e o sistema modifica do estado *s0* para o estado *s1*. É válido notar que o algoritmo converge para esse exemplo, pois o ramo que apresentou falha, permanece no estado *s0*. E o estado *s1*, que converge, é alcançável pelo estado *s0*.

4.3.3 Invocador de Serviços

Nesse trabalho não implementamos, nem agregamos um motor sofisticado para invocação dos serviços. Deixamos esse módulo como trabalho futuro. O Invocador de Serviços atual, representado pela classe *Invocador*, invoca sequencialmente, um ou mais serviços, de acordo com o que foi especificado pelo módulo de composição automática de serviços.

4.4 A Ferramenta SWoTPAD Designer

Todas ideias apresentadas nesse capítulo são disponibilizadas para utilização através de uma ferramenta denominada *SWoTPAD Designer*. Basicamente, o *SWoTPAD Designer* é um conjunto de *plugins* para o eclipse que visa auxiliar desenvolvedores na concepção de novos sistemas de *IoT*. A seguir, apresentamos algumas telas deste sistema para elucidar seu funcionamento.

A Figura 34 apresenta o menu adicionar da ferramenta. Nesse menu, podemos adicionar um novo ambiente, *device*, *Web service*, ontologia ou requisição. A criação de um novo ambiente consiste no desenvolvimento de uma descrição JSON que corresponde ao ambiente que receberá o sistema de *IoT*. A criação de *device* e *webService* consiste no desenvolvimento de uma descrição de serviço em SWoTPADL, que apresenta os cenários em que cada um desses elementos podem atuar. Adicionalmente, a especificação sinaliza quais mudanças esses elementos proporcionam ao ambiente após sua execução. Em adicionar ontologia, podemos criar uma especificação de ontologia em WSML. Por último, adicionar requisição permite criar requisições empregando a linguagem SWoTPADL.

Figura 34: SWoTPAD Designer Menu Adicionar

Fonte: Próprio Autor

A Figura 35 apresenta o menu gerar presente na ferramenta. Basicamente, esse menu

Figura 35: SWoTPAD Designer Menu Gerar

Fonte: Próprio Autor

conta com duas opções. A primeira delas permite gerar código python, ou seja, gerar o código de serviços Restful a partir das descrições semânticas de serviço apresentada pelo projetista. A segunda opção, gerar semântica, é responsável pela geração de ontologias a

partir das descrições de serviços desenvolvidas pelo projetista.

A Figura 36 apresenta o menu executar. Basicamente, esse menu conta com duas

Figura 36: SWoTPAD Designer Menu Executar

Fonte: Próprio Autor

opções. Registrar serviços, utilizado para registrar serviços no repositório de serviços. Após registro, o serviço já está apto a ser descoberto, bem como ser utilizado como parte de uma composição de serviços. O menu executar requisição permite solicitar um serviço atômico ou composto, por meio de uma requisição. As demais funcionalidades são derivadas dessas duas primeiras.

4.5 Conclusões

Esse capítulo teve como objetivo apresentar os vários elementos que compõem o *framework* SWoTPAD. O objetivo de SWoTPAD é prover um conjunto de soluções que visam facilitar o desenvolvimento de novas aplicações para IoT.

As soluções fornecidas pelo *framework* são distribuídas através de um modelo de três camadas, composto pela camada de modelagem, camada semântica e camada funcional. A camada de modelagem fornece a especificação formal e o modelo conceitual das entidades básicas de um sistema de IoT. A camada semântica provê, através da linguagem SWoTPADL, formas de possibilitar a associação e construção de um vocabulário semântico ao sistema de IoT. Por último, a camada funcional apresenta aplicações de descoberta e composição automática de serviços, úteis a um sistema de IoT.

Assim, temos um *framework* flexível, cujos artefatos de cada camada podem ser apli-

cados em um sistema de IoT, em diferentes estágios do seu ciclo de desenvolvimento.

5 ESTUDOS DE CASO

Esse capítulo tem como objetivo apresentar dois estudos de caso adotados nesse trabalho para validação de nossa proposição. Esse capítulo apresenta a seguinte organização. Na Seção 5.1 é apresentado o Estudo de Caso 1, que simula a necessidade por gerência e composição de serviços em uma aplicação para segurança residencial. A Seção 5.2 apresenta o Estudo de Caso 2, que simula a demanda por composição de serviços em um ambiente de computação em nuvem.

5.1 Estudo de Caso 1: Aplicação de Segurança Residencial

Esta seção apresenta a modelagem de um sistema de segurança residencial para ilustrar a abordagem proposta. Os vídeos relativos a esse estudo de caso, podem ser encontrados em (SILVA, 2016). O sistema de segurança residencial é um aplicativo cliente-servidor, no qual o módulo cliente simula a navegação de uma pessoa em uma casa. A pessoa pode ser o proprietário ou um invasor. Quando o proprietário se move pela residência, o sistema reage com o serviço de acendimento de luzes. Quando o invasor entra na residência, o sistema pode operar em modo pânico ou silencioso. Esses modos são mais complexos, pois são acionados por *requests* que demandam a composição de serviço. O sistema também simula princípios de incêndio, vazamento de água ou vazamento de gás.

Além do módulo cliente, o sistema apresenta um servidor que hospeda os serviços que compõem a solução SWoTPAD, os serviços RESTfull do ambiente e as ontologias. O servidor é uma estação intel core i5-5200U de 2,2 GHz com 8 GB de RAM e 256 GB de SSD. O cliente é um tablet Galaxy Note N-8010 com o processador Quad-core 1.4-GHz Cortex-A9 com 2GB de RAM e memória interna de 32 GB.

A Figura 37 mostra o cenário modelado no cliente Android. A casa contém sete ambientes: cozinha, sala de estar, corredor, banheiro, quintal e dois quartos. Esses ambientes possuem um conjunto de dispositivos de IoT para segurança, tais como sensores de ja-

Figura 37: Aplicação de segurança residencial

Fonte: Próprio Autor

nela e porta, sensores de movimento, câmera, detecção de vazamentos, sirenes, lâmpadas inteligentes, sensor de temperatura e detecção de gás. Adicionalmente, existem alguns dispositivos auxiliares, tais como *smartphone* e roteador sem fio. Quando esses elementos cooperam, eles podem fornecer um alto grau de segurança. A residência apresenta serviços de proteção contra incêndio, vazamento de gás, roubo e inundação. Destacamos apenas os serviços de proteção contra roubo. Na próxima seção, descreveremos cada um deles.

5.1.1 Artefatos de Entrada

O *framework* exige que o projetista descreva os ambientes. Para isso, adotamos o formato JSON. A Listagem de Código 23 mostra um trecho desta descrição em relação a planta apresentada na Figura 37. As linhas 2-5 definem o lar, que consiste na *Cozinha*, *SaladeEstar*, *Banheiro*, *Corredor*, *Quarto1*, *Quarto2* e *Quintal*. O campo *field* define a localização do ambiente. A propriedade *memberOf* determina o tipo de conceito descrito. As definições da *Cozinha* (linhas 6-9) e *SalaDeEstar* (linhas 14-17) são feitas

Listagem de Código 23: Descrição do ambiente em JSON

```

1: {
2:   "Home": {"environments": ["Kitchen", "LivingRoom", "BathRoom", "Corridor", "BedRoom1", "BedRoom2", "FrontYard"],
3:     "coordinates": [639, 524, -78, -63],
4:     "memberOf": "swot#Environment"
5:   },
6:   "Kitchen": {"devices": ["msKitchen", "lampKitchen", "w1Kitchen", "w2Kitchen"],
7:     "memberOf": "swot#Enironment",
8:     "coordinates": [302, 111, -78, -63]
9:   },
10:  "lampKitchen": {"memberOf": "Lamp"},
11:  "msKitchen": {"memberOf": "MotionSensor"},
12:  "w1Kitchen": {"memberOf": "Window"},
13:  "w2Kitchen": {"memberOf": "Window"},
14:  "LivingRoom": {"devices": ["lampLivingRoom", "msLivingRoom", "lockLivingRoom", "modemLivingRoom"],
15:    "memberOf": "swot#Environmet",
16:    "coordinates": [279, 357, -76, 145]
17:  },
18:  "lampLivinRoom": {"memberOf": "Lamp"},
19:  "msLivinRoom": {"memberOf": "MotionSensor"},
20:  "lockLivinRoom": {"memberOf": "Lock"},
21:  "modemLivinRoom": {"memberOf": "Modem"},
22:  ...
23: }
```

Fonte: Próprio Autor.

de forma semelhante. As linhas 10-13 e 18-21 descrevem os *dispositivos*.

Além do arquivo de descrição JSON, o *projetista* deve definir os serviços SWoTPADL. A seguir, apresentamos alguns dos serviços modelados para realização desse estudo de caso.

Um serviço básico para prevenção de roubo em residências é a detecção de intrusos. Este serviço pode ser operado por sensores de movimento que relatam a presença de um suspeito em um lugar da casa. A Listagem de Código 24 apresenta o código SWoTPADL que descreve esse serviço. Basicamente, a pré-condição de *DetectIntruderService* de-

Listagem de Código 24: O Serviço de detecção de intrusos

```

1: _"http://poli.usp.br/swotpad/services/detectintruderservice#"
2: service DetectIntruderService
3:   import {heo _"http://poli.usp.br/sowtpad/services/detectintruderservice#" }
4:   precondition
5:     ?sensor [state hasValue _string("anomaly_detected")] memberOf heo#Sensor.
6:   postcondition
7:     ?situation[notifyEv.value hasValue _string("notify_on")] memberOf heo#HomeFeatures.
8:     ?intruder[hasLocalization hasValue ?place] memberOf heo#Intruder .
```

Fonte: Próprio Autor

clara que um sensor detecta a presença de uma pessoa suspeita (linhas 4 a 6). Como consequência, temos a localização do intruso, bem como restrições que orientam a execu-

ção de outros serviços (linhas 7 a 13).

Serviços simples para ajudar na prevenção do roubo são a ativação automática de luzes e alarmes. A descrição desses serviços é apresentada nas listagens de código 25 e 26, respectivamente.

Listagem de Código 25: O serviço *TurnOnLightService*

```

1: _"http://www.usp.lsi.br/wsmplpe/services/turnonlampservice#"
2: service TurnOnLightService extends Lights
3:   import { heo _"http://poli.usp.br/swotpad/services/detectintruderservice#" }
4:   precondition
5:     ?situation [notifyEv.value hasValue _string(notify_on)].
6:     ?lamp[hasState hasValue _string("off")] memberOf heo#Lamp.
7:   postcondition
8:     ?lamp[hasState hasValue _string("on")] memberOf heo#Lamp.

```

Fonte: Próprio Autor

Listagem de Código 26: O serviço *PlaySirenService*

```

1: _"http://www.usp.lsi.br/wsmplpe/services/playsirenservice#"
2: service PlaysirenService
3:   import{ heo _"http://poli.usp.br/swotpad/services/detectintruder#" }
4:   precondition
5:     ?situation[notifyEv.value hasValue _string("notify_on")].
6:     ?siren[hasState hasValue _string("mute")] memberOf heo#Siren.
7:   postcondition
8:     ?siren[hasState hasValue _string("noisy")] memberOf heo#Siren .

```

Fonte: Próprio Autor

O serviço *TurnOnLightService* altera o estado da lâmpada de *off* (linha 6) para *on* (linhas 8). Da mesma forma, *PlaySiren* altera o estado da sirene de *mute* (linhas 6) para *noisy* (linhas 8).

Além de fornecer vídeo em tempo real, o dispositivo da câmera pode tirar fotos de um suspeito. Esse comportamento é alcançado pelo serviço *TakeAPhotoService*, ilustrado na Listagem de Código 27. Como precondições, *TakeAPhotoService* usa informações de localização obtidas pelo sensor de movimento e rastreia uma câmera do dispositivo próximo do intruso (linhas 5 a 8). Depois, ele fotografa o intruso (linha 10).

Listagem de Código 27: O serviço *TakeAPhotoService*

```

1: _"http://www.lsi.usp.br/services/takeaphotoservice#"
2: service TakeAPhotoService
3:   import{ heo _"http://poli.usp.br/swotpad/services/detectintruderservice#" }
4:   precondition
5:     ?situation [notifyEv.value hasValue _string("notify_on")].
6:     ?intruder [hasLocalization hasValue ?place] memberOf heo#Intruder.
7:     ?cam[hasLocalization hasValue ?placeCam] memberOf heo#Camera and nextTo(?place, ?placeCam).
8:   postcondition
9:     ?photo[WhomIs hasValue ?intruder]memberOf heo#Photo.

```

Fonte: Próprio Autor

O conjunto de informações obtidas pelos dispositivos pode ser útil para evitar o roubo. Adicionalmente, eles podem ser úteis às autoridades policiais para localizar e identificar o suspeito. O *ContactPoliceStationService*, apresentado na Listagem de Código 28, executa essa tarefa. Com base na localização e na foto tirada pelo sensor de movimento e pela câmera (linhas 5 a 9), ele envia essas informações para uma delegacia (linhas 11 a 12).

Listagem de Código 28: O serviço ContactPoliceStationService

```

1: _"http://www.usp.lsi.br/wsmlpe/services/contactpolicestationservice#"
2: service ContactPoliceStationService extends Notify
3:   import {heo _"http://poli.usp.br/swotpad/services/detectintruderservice#"}
4:   precondition
5:     ?inputEvent [sensor hasValue _string("anomaly_detected")].
6:     ?intruder[hasLocalization hasValue ?place] memberOf heo#Intruder.
7:     ?photo[WhomIs hasValue ?intruder, date hasValue ?time] memberOf heo#Photo.
8:   postcondition
9:     ?rsp[header hasValue _string("Notification received by the Police Station")].

```

Fonte: Próprio Autor

Para um usuário que tenha uma residência e serviços como anteriormente ilustrado, é fácil modelar uma solicitação para garantir sua proteção doméstica. A Listagem de Código 29 mostra um exemplo de uma solicitação que pode ser aplicada a este caso. O *SecurityGoal* apenas descreve o efeito direcionado pelo usuário.

Listagem de Código 29: O serviço SecurityGoal

```

1: _"http://www.usp.lsi.br/wsmlpe/goals/securitygoal#"
2: request SecurityGoal
3:   import {heo _"http://poli.usp.br/swotpad/services/detectintruderservice#"}
4:   postcondition
5:     ?rsp[header hasValue _string("Notification received by the Police Station")].
6:     ?siren[hasState hasValue _string("noisy")] memberOf Siren .
7:     ?lamp[hasState hasValue _string("on")] memberOf Lamp.
8: compCapacities: P1: Security

```

Fonte: Próprio Autor

Note que cada pós-condição (linhas 5 a 8) pode ser alcançada pelos serviços apresentados. No entanto, existem restrições relacionados à dependência de dados que devem ser obedecidas. Essas restrições afetam a ordem de execução dos serviços. Portanto, uma requisição não é suficiente para especificar os serviços e a ordem a ser executada.

Para estes casos, o uso da composição automática de serviços é desejável. O algoritmo deve resolver esses problemas, procurando outros serviços que atendam a essas dependências, bem como define a ordem correta de execução. A importância da hierarquia de serviços SWoTPADL é que ajuda a reduzir o número de serviços procurados. Por exemplo, para atender ao *SecurityGoal*, não seria sensato buscar serviços não relacionados à proteção contra roubo.

5.1.2 Planejamento e Composição Automática

Os serviços modelados na aplicação de segurança residencial, seguiram a taxonomia apresentado na Listagem de Código 30. Os elementos que aparecem na taxonomia entre colchetes representam *capacidades*. Os demais elementos representam serviços, mais especificamente, sua *habilidade*. Serviços introduzidos por meio de '::<=' representam serviços compostos. Esses serviços fazem uso do construtor *compCapacities* em sua definição. Serviços não introduzidos por '::<=' correspondem a serviços simples. Usando o vocabulário comumente usado em planejamento, serviços compostos correspondem aos métodos, serviços atômicos correspondem às tarefas primitivas e capacidades representam *tarefas não primitivas*. Uma *ação* corresponde a uma serviço simples acompanhado de valores para cada um de seus parâmetros.

Listagem de Código 30: Taxonomia dos Serviços de acordo com as habilidades

1:	[Security]
2:	TurnOnLampService
3:	AlertMode ::= [Lights] PlaySirenService TakeAPhotoService [Notify]
4:	SilentMode ::= TakeAPhotoService [Notify]
5:	FireGasFlood ::= [MeasureProblem] [DecreaseProblem] [Notify]
6:	[Lights]
7:	TurnOnLightService
8:	TurnOffLightService
9:	BlinkLights ::= TurnOnLightService TurnOffLightService [Lights]
10:	[Notify]
11:	ContactPoliceStationService
12:	ContactUserService
13:	ContactFireStationService
14:	NotifyUserAndPolice ::= ContactUserService ContactPoliceStationService
15:	NotifyUserAndFirefighters ::= ContactUserService ContactFireStationService
16:	[MeasureProblem]
17:	DegreeOfSmokeService
18:	DegreeOfFloodService
19:	DegreeOfGasService
20:	[DecreaseProblem]
21:	ActivateFireService
22:	LeakageService
23:	OpenWindowService

Fonte: Próprio Autor

Na Listagem de Código 30, podemos ver que a aplicação de segurança residencial apresenta cinco capacidades, são elas *Security*, *Lights*, *Notify*, *MeasureProblem* e *DecreaseProblem*. A *Capacidade Security* representa um conjunto de serviços que atuam em um determinado aspecto da segurança da residência. Nessa *capacidade* estão presentes os serviços *TurnOnLampService*, *AlertMode*, *SilentMode* e *FireGasFlood*.

TurnOnLampService é um serviço atômico para acendimento de lâmpadas. Já *AlertMode* é um serviço composto para acionamento do sistema de segurança em caso de invasão por

algum intruso. Ele é composto pela capacidade para acionamento de luzes [*Lights*], o serviço atômico para disparo de sirene *PlaySirenService*, o serviço atômico para captura de imagens *TakeAPhotoService* e pela capacidade para notificação *Notify*.

O serviço composto *SilentMode* também aciona o sistema de segurança em caso de invasão, porém, o intruso não recebe notificação desse acionamento. *SilentMode* é composto pelo serviço atômico *TakeAPhotoService* e pela capacidade para notificação *Notify*. Por último, temos o serviço *FireGasFlood* que atua na resolução de problemas ligados a vazamento de gás, água ou incêndio. Ele é composto pela capacidade *MeasureProblem*, que avalia a gravidade do problema, pela capacidade *DecreaseProblem*, que lança algumas ações na intenção de amenizar o problema e pela capacidade de notificação *Notify*.

A *Capacidade Lights* reúne serviços voltados para iluminação. Ela é composta por três serviços, o serviço atômico *TurnOnLightService* que realiza o acendimento da iluminação e o serviço *TurnOffLightService*, que desliga a iluminação e o serviço composto *BlinkLights* formado pelos serviços atômicos *TurnOnLightService* e *TurnOffLightService*, alternando-os no sentido de fazer a iluminação piscar.

A *Capacidade Notify* é composta por cinco serviços. Os serviços simples *ContactPoliceStationService*, *ContactUserService* e *ContactFireStationService* que notificam, respectivamente, a estação de polícia, os moradores da residência e o corpo de bombeiros sobre a ocorrência de alguma anomalia. Os serviços compostos *NotifyUserAndPolice* e *NotifyUserAndFirefighters* são serviços compostos para notificação, sendo o primeiro para notificação dos moradores e a estação de polícia e o segundo para notificação dos moradores e o corpo de bombeiros.

A *Capacidade MeasureProblem* é composta por três serviços simples, todos eles fazem uso de sensores para avaliar o grau da anomalia relatada. O serviço simples *DegreeOfSmokeService* é usado no cenário de incêndio, para avaliar o quanto de fumaça há no ambiente sensoriado. De maneira análoga, os serviços simples *DegreeOfFloodService* e *DegreeOfGasService* avaliam quão grave se encontram o vazamento de água ou gás, respectivamente.

Por último, temos a capacidade *DecreaseProblem*. Ela é composta pelos serviços simples *ActivateFireService*, *LeakageService*, *OpenWindowService*. *ActivateFireService* é usado para acionar os *sprinklers*, chuveiros automáticos usados na contenção de incêndio. Já o serviço *LeakageService* fecha o registro da residência, no sentido de evitar maiores prejuízos com vazamento. O serviço *OpenWindowService* é responsável por abrir

as janelas da residência, com intuito de arejar o ambiente no caso de vazamento por gás.

Figura 38: Plano obtido para AlertMode

Fonte: Próprio Autor

Por meio dessa taxonomia de *Capacidades* e *Habilidades*, o *framework* SWoTPADL realiza a composição automática de serviços para o sistema de segurança residencial. A Figura 38 apresenta um exemplo de plano obtido a partir de *Security*. O plano escolhido faz uso de *AlertMode*. A escolha por uma habilidade decorre de sua *precondition*, de forma que, para ocorrência de *AlertMode* o sistema deve apresentar as seguintes ocorrências: (1) existência de um intruso nas margens ou no interior da residência e (2) usuário ter configurado o sistema para o modo *AlterMode*, em caso de invasão.

AlertMode é decomposto na Capacidade *Lights*, no serviço atômico *PlaySirenService*, no serviço atômico *TakeAPhotoService* e na Capacidade *Notify*. A escolha de *BlinkLights* para realizar a *Capacidade Lights* foi feita por *AlertMode* se tratar de um serviço que visa expulsar o intruso. Perceba que *BlinkLights* chama a capacidade [*Lights*] de forma a relizar um ciclo recursivo de chamadas. Ele mantém esse comportamento até que o intruso saia do domínio do sensor que fez a detecção. Na Figura 38, está ilustrado um cenário onde duas piscadas de luzes foram suficientes. Em seguida são chamados os serviços *PlaySirenService* que aciona um alarme sonoro e o serviço *TakeAPhotoService* que usa a câmera da residência com intuito de realizar registros da

invasão. Por último, temos a *Capacidade Notify*, que é realizada pelo serviço composto *NotifyUserAndPolice*, escolhido por se tratar de uma invasão. O *NotifyUserAndPolice* gera assim, uma notificação para os moradores e a estação policial, através dos serviços *ContactUserService* e *ContactPoliceStateService*, respectivamente.

5.2 Estudo de Caso 2 - Serviços na Nuvem

O segundo estudo de caso trata da composição automática de serviços para aplicativos de computação em nuvem. Um problema clássico na computação em nuvem é a implantação de serviços. Dada a configuração inicial de uma infraestrutura de nuvem, a implantação de serviços consiste em encontrar uma sequência de ações de implantação relativas aos serviços que compõem o aplicativo desejado. Embora não seja um problema de IoT, queremos avaliar, com esse estudo de caso, se é possível adotar o framework SWoTPAD em outros domínios.

Para a modelagem do problema de implantação, adotamos o modelo Aeolus (COSMO; ZACCHIROLI; ZAVATTARO, 2012; GEORGIEVSKI, 2015). Esse modelo apresenta vários componentes, que solicitam e fornecem funcionalidades. Solicitar e fornecer funcionalidades implica estabelecer interdependências entre as componentes. A solicitação de um aplicativo é feita por uma sequência de ações de baixo nível, como instalar um serviço, executar um serviço, etc.

O estudo de caso usa um paradigma comum nas distribuições linux, como o Debian, onde o software está disponível em pacotes. Pacotes apresentam software, suas configurações e metadados. As informações de metadados mostram os requisitos de software, como dependências de instalação e dependências de execução.

Aeolus formaliza e abstrai esse modelo de pacote. Um pacote apresenta três estados: não instalado, instalado e em execução. Quando um pacote está no primeiro estado, ele não está presente no sistema. O pacote é instalado quando a ação de instalação é executada. Esta ação requer que todas as dependências de instalação do pacote sejam atendidas. O pacote muda para execução quando o usuário ou sistema executa a ação de executar o pacote. O pacote é alterado quando todas as dependências são satisfeitas. A figura 39 mostra a visão geral dos diagramas de estado do modelo Aeolus.

As dependências de instalação e execução do pacote ocorrem no nível de aplicativo ou serviço. Cada pacote tem um ou mais aplicativos ou serviços disponíveis quando o pacote está instalado ou em execução. A Figura 40 apresenta uma versão simplificada

Figura 39: Máquina de Estados de aplicações em um ambiente de nuvem

Fonte: Adaptado de (GEORGIEVSKI, 2015)

das dependências de pacotes para alguns serviços. Os pacotes wordpress, apache2 e mysql estão instalado, em execução e não instalado, respectivamente. Observe que a figura ilustra uma configuração válida porque todas as dependências de instalação e execução foram atendidas. Por exemplo, o pacote wordpress foi instalado porque sua dependência de instalação, que é o serviço httpd, foi resolvida pelo pacote apache2. Se wordpress passar para o estado em execução, é necessário que suas dependências de execução sejam resolvidas. Na configuração atual, o pacote mysql deve mudar para execução, pois wordpress requer o serviço mysql_{up}. Não há necessidade de alterar o apache2, já que ele se encontra em execução e fornece o serviço httpd. No momento em que o wordpress é executado, o serviço wordpress é disponibilizado.

Figura 40: Descrição básica do modelo de serviços e pacotes.

Fonte: Adaptado de (COSMO; ZACCHIROLI; ZAVATTARO, 2012)

O problema de implantação foi mapeado da seguinte maneira. Geramos problemas de implantação em um número crescente de componentes, variando de 3 a 100 componentes. Para os casos de teste, usamos um conjunto de componentes $c_1, \dots, c_n$, onde cada c_i solicita e provê serviços. Dado que queremos ter o componente c_n mais à direita no seu estado de execução, as dependências entre componentes exigem primeiro instalar as

componentes de c_1 a c_n e, então, executar a transição do estado *não instalado* para *instalado* na ordem inversa das instâncias componentes (c_1 a c_n), e, finalmente, a transição do estado *instalado* para o estado *em execução* na ordem de c_1 a c_n . A seguir descrevemos os artefatos de entrada desse estudo de caso.

5.2.1 Artefatos de Entrada

A seguir apresentamos um subconjunto das descrições adotadas nesse estudo de caso. Este subconjunto é suficiente para compreender como ocorre as tarefas de instalação e execução de componentes.

A Listagem de código 31 apresenta *service Instalar*, que implementa a capacidade *InstalarServico*. Para que uma instalação de componente seja feita a partir desse

Listagem de Código 31: Serviço Instalar

```

1: _"http://www.lsi.usp.br/services/cloud#"
2: service Instalar extends InstalarServico
3:   precondition
4:     ?s memberOf component and desinstalado(?s) and
5:       forall?y(?s[di hasValue ?y] and instalado(?y))
6:   postcondition
7:     instalado(?s)

```

Fonte: Próprio Autor

serviço, se faz necessário que o componente *?s* esteja *desinstalado* e suas dependências de instalação (*s.di*) estejam *instaladas*. Esse comportamento é expresso nas linhas 4 e 5 da definição. Como *postcondition*, *service Instalar* muda *?s* para o estado *instalado(?s)*.

De maneira análoga, temos o *service Executar*, ilustrado na Listagem de Código 32. *Executar* é responsável por alterar o estado de um componente *?s* para o estado

Listagem de Código 32: Serviço Executar

```

1: _"http://www.lsi.usp.br/services/cloud#"
2: service Executar extends ExecutarServico
3:   precondition
4:     ?s memberOf component and naf executando(?s) and
5:       forall?y(?s[de hasValue ?y] and executando(?y))
6:   postcondition
7:     executando(?s)

```

Fonte: Próprio Autor

executando. Para que o componente execute, é necessário que ele não esteja em execução, representado por *naf executando(?s)*. Para que ele passe para o estado executando, é necessário que todas as dependências de execução *?s.de* estejam em execução. Esse

comportamento é descrito nas linhas 4 e 5. Como *postcondition*, temos o componente agora no estado *executando(?s)*.

O *service InstDepend*, ilustrado na Listagem de Código 33, também implementa a

Listagem de Código 33: Serviço InstDepend

```

1: _"http://www.lsi.usp.br/services/cloud#"
2: service InstDepend extends InstalarServico
3:   precondition
4:     ?s memberOf component and desinstalado(s) and
5:       exists?y(?s[di hasValue ?y] and desinstalado(?y))
6:   postcondition
7:     instalado(?s)
8:   compCapacities:
9:     SEQ(P1: InstalarServico(?s.di), P2: Instalar(?s))

```

Fonte: Próprio Autor

capacidade *InstalarServico*. Diferente de *Instalar*, *InstDepend* instala as dependências de *?s*, caso alguma dependência de instalação (*?s.di*) não esteja instalada.

Como *precondition*, o componente *?s* precisa estar *desinstalado* e é necessário que exista ao menos uma dependência de instalação (*?s.di*) não instalada (linhas 4 e 5). A pós-condição de *InstDepend* é a mudança de *?s* para o estado *instalado(?s)*. Perceba que o *service InstDepend* é composto por outros serviços. Ele é realizado por uma composição sequencial de serviços, sendo o primeiro, qualquer serviço *P1* que realize a capacidade *InstalarServico*, recebendo como parâmetros as dependências de instalação de *?s*. Já o segundo serviço é *Instalar*, que recebe como parâmetro o componente *?s* (linha 9).

Por último, temos o serviço *ExecEInst*, ilustrado na Listagem de Código 34. Ele rea-

Listagem de Código 34: Serviço ExecEInst

```

1: _"http://www.lsi.usp.br/services/cloud#"
2: service ExecEInst extends ExecutarServico
3:   precondition
4:     ?s memberOf component and parado(s) and
5:       exists?y(?s[de hasValue ?y] and parado(?y))
6:   postcondition
7:     executando(?s)
8:   compCapacities:
9:     SEQ(P1: InstalarServico(?s), P2: ExecutarServico(?s.de), P3: Executar(?s))

```

Fonte: Próprio Autor

liza a capacidade *ExecutarServico*, porém só é chamado quando o componente *?s* a ser executado apresenta alguma dependência de execução (*?s.de*) no estado *parado* (linhas 4-5). Como *postcondition*, a componente *?s* muda para o estado *executando(?s)*. Ele é composto pelos serviços *P1*, *P2* e *P3* (linha 9). *P1* é qualquer serviço que realize

a capacidade *InstalarServiço*, que recebe como parâmetro o componente *?s*. Já *P2* é qualquer serviço que realize a capacidade *ExecutarServiço*, recebendo como parâmetro as dependências de execução de *?s*. Por último, temos *P3*, que chama o serviço primitivo *Executar* para o componente *?s*.

5.2.2 Planejamento e Composição Automática

Os serviços modelados na aplicação Serviços na Nuvem seguiram a taxonomia apresentado na Listagem de Código 35. Os elementos que aparecem na taxonomia entre colchetes representam *capacidades*. Os demais elementos representam serviços, mais especificamente, sua *habilidade*. Serviços introduzidos por meio de '::=' representam serviços compostos. Esses serviços fazem uso do construtor *compCapacities* em sua definição. Serviços não introduzidos por '::=' correspondem a serviços simples. Usando o vocabulário comumente adotado em planejamento, serviços compostos correspondem aos métodos, serviços atômicos correspondem às tarefas primitivas e capacidades representam tarefas não primitivas. Uma ação corresponde a uma serviço simples acompanhado de valores para cada um de seus parâmetros.

Listagem de Código 35: Taxonomia dos Serviços para Cloud de acordo com as habilidades

1:	[ExecutarServiço(s)]
2:	ExecEInst(s) ::= [InstalarServiço(s)] [ExecutarServiço(s.de)] Executar(s)
3:	Executar(s)
4:	FazNada
5:	[InstalarServiço(s)]
6:	InstDepend(s) ::= [InstalarServiço(s.di)] Instalar(s)
7:	Instalar(s)
8:	FazNada
9:	[DesinstalarServiço(s)]
10:	DesEPar ::= [PararServiço(s)] [DesinstalarServiço(s.ddi)] Desinstalar(s)
11:	Desinstalar(s)
12:	FazNada
13:	[PararServiço(s)]
14:	PararDepend(s) ::= [PararServiço(s.dde)] Parar(s)
15:	Parar(s)
16:	FazNada

Fonte: Próprio Autor

Na Listagem de Código 35, podemos ver que a aplicação de implantação de serviços em nuvem apresentam quatro capacidades, são elas: *ExecutarServiço*, *InstalarServiço*, *DesinstalarServiço* e *PararServiço*. Cada capacidade apresenta como entrada uma lista de serviços, denominada por *s*.

A *Capacidade ExecutarServiço(s)* apresenta um conjunto de serviços necessários para a execução da lista de serviço *s*. Ela é composta pelo serviço composto *ExecEInst(s)*

que avalia a necessidade de instalação dos serviços de s , através da Capacidade *InstalarServico(s)*. Em seguida, usa *Capacidade ExecutarServico* para $s.de$, que representa o conjunto de dependências de execução de cada serviço de s . Por fim, usa o serviço atômico *Executar(s)* que realiza a execução dos serviços em s . Os outros serviços que compõem *Capacidade ExecutarServico(s)* são o serviço atômico *Executar(s)* que só executa os serviços de s caso todos estejam instalados e suas dependências de execução já tenham sido atendidas e *FazNada* que, como o próprio nome indica, é um serviço que não apresenta comportamento.

A *Capacidade InstalarServico(s)* apresenta serviços para instalação de serviços. Ela é formada pelo serviço composto *InstDepend(s)*, que instala as dependências de instalação ($s.di$) através da capacidade *InstalarServico(s.di)*, e pelo serviço atômico *Instalar(s)* que instala todos os serviços da lista (s). Adicionalmente, a *Capacidade InstalarServico(s)* pode ser realizada puramente pelo serviço atômico *Instalar(s)*, no caso de todas as dependências de instalação de s já terem sido atendidas ou pelo serviço *FazNada*.

A Capacidade *Desinstalar(s)* apresenta um conjunto de serviços necessários para a desinstalação da lista de serviço s . Ela é composta pelo serviço composto *DesEPar(s)* que avalia a necessidade de parar serviços de s , através da Capacidade *PararServico(s)*. Em seguida, usa a *Capacidade DesinstalarServico* para $s.ddi$, que representa o conjunto de serviços que dependem de s para instalação. Por fim, usa o serviço atômico *Desinstalar(s)* que realiza a desinstalação dos serviços de s . Os outros serviços que compõem *Capacidade DesinstalarServico(s)* são o serviço atômico *Desinstalar(s)*, que só desinstala os serviços de s caso todos estejam parados e os serviços que dependem dele para instalação já tenham sido desinstalados, e o serviço *FazNada*.

A *Capacidade ParaServico(s)* apresenta serviços para parar os serviços de s . Ela é formada pelo serviço composto *PararDepend(s)*, que para os serviços que apresentam dependências de execução ($s.dde$) com os serviços de s e faz isso através da capacidade *PararServico(s.dde)*, e pelo serviço atômico *Parar(s)* que para todos os serviços de (s). Adicionalmente, a *Capacidade PararServico(s)* pode ser realizada pelo serviço atômico *Parar(s)*, no caso de todos os serviços que apresentam dependência de execução com serviços de s já terem sido parados ou pelo serviço *FazNada*.

Essa taxonomia de *Capacidades* e *Habilidades* permite ao *framework* SWoTPADL realizar a composição automática de serviços para o sistema de implantação de serviços na nuvem. A Figura 41 apresenta um exemplo de plano obtido a partir de uma requisição para

Figura 41: Plano obtido para execução do serviço Wordpress.

Fonte: Próprio Autor

passar o serviço *wordpress* para o estado de execução. Para esse exemplo, considere que tanto *wordpress*, como todas as suas dependências se encontram no estado desinstalado. No exemplo, adotamos as dependências de *wordpress* apresentadas na Figura 40, ou seja, o pacote *apache2* como dependência de instalação e o pacote *mysql* como dependência de execução.

A capacidade *ExecutarServico(wordpress)* é realizado pelo serviço composto *ExecutaEInstala(wordpress)* que é decomposto nas *Capacidades* *InstalarServico(wordpress)*, *ExecutarServico(mysql)* e no serviço simples *Executar(wordpress)*. A *Capacidade* *InstalarServico(wordpress)* resulta na instalação da dependência *apache2*, na instalação do serviço *wordpress*, ambas realizadas através do serviço composto *InstDepend(wordpress)*. Em seguida, a *Capacidade* *ExecutarServico(mysql)* é realizada pelo serviço simples *Executar(mysql)*. A sequência de serviços obtidas por meio dessa decomposição é *Instalar(apache2)*, *Instalar(wordpress)*, *Executar(mysql)*, *Executar(wordpress)*, ou seja, a sequência que resolve os problemas de dependência para a execução do serviço *wordpress*.

5.3 Resultados e Discussões

A seguir, apresentamos os resultados e fazemos uma breve discussão dos estudos de caso das seções 5.1 e 5.2. O sistema de segurança residencial pode lidar com os seguintes tipos de solicitações: detecção de intrusão (*AlertMode* e *SilentMode*), ativação de luz

(*Lights*), detecção de vazamento *flood*), detecção de incêndio (os últimos três, resolvidos por *FireGasFlood*). A Tabela 1 apresenta as requisições, o tempo para a descoberta do serviço, o número de serviços no plano, o tempo para criar e executar o plano.

Tabela 1: Desempenho do Sistema de Segurança Residencial

Requisição	Descoberta (seg.)	Número de Serviços	Composição (seg.)	Execução do Plano (em seg.)
AlertMode	0.3	8	2.5	10.5
Lights	0.2	1	-	1.1
SilentMode	0.3	6	1.8	4.2
Flood	0.2	4	2.0	3.8
Fire	0.2	4	2.1	3.5
Gas	0.2	4.1	2.1	3.6

Fonte: Próprio Autor

Em seguida, realizamos testes mais profundos, alternando o número de solicitações, para avaliar o comportamento da solução em um cenário de estresse. O resultado deste expe-

Tabela 2: Sistema de Segurança Residencial - Cenário de Estresse .

ID	Request	Quantity	Discovery (sec.)	Number of services	Planning (sec.)	Execution (sec.)	Overhead (%)
1	AlertMode	10	5.2	80	9.6	20.4	72.55
2	AlertMode	30	12.2	240	16.2	35.3	80.45
3	AlertMode	50	40.4	400	22.7	55.2	114.31
4	Lights	10	0.9	10	-	2.3	39.13
5	Lights	30	2.3	30	-	4.7	48.94
6	Lights	50	3.2	50	-	6.1	52.46
7	SilentMode	10	3.3	60	4.7	11.6	68.97
8	SilentMode	30	8.3	180	8.4	15.7	106.37
9	SilentMode	50	12.6	300	10.5	22.3	103.59
10	Flood	10	2.3	40	3.1	6.7	80.60
11	Flood	30	5.1	120	5.2	12.3	83.74
12	Flood	50	6.6	200	8.9	15.5	100.00
13	Fire	10	2.2	40	3.2	5.9	91.53
14	Fire	30	5.1	120	5.0	10.3	98.06
15	Fire	50	6.5	200	9.3	14.9	106.04
16	Gas	10	1.9	20	2.8	3.1	151.61
17	Gas	30	4.0	60	3.9	4.5	175.56
18	Gas	50	6.1	100	8.5	6.6	221.21

Fonte Próprio Autor.

rimento é ilustrado na Tabela 2. Nós introduzimos uma nova métrica, denominada *sobrecarga*. Em nossos experimentos, consideramos como sobrecarga a relação entre a soma do

tempo de descoberta e planejamento dividido pelo tempo de execução. Dessa forma, sobrecarga corresponde ao atraso introduzido pela composição automática caso já existisse o serviço composto. Na Tabela 2, vemos que o *AlertMode* apresentou tempo de resposta mais longo porque exige um maior número de serviços para sua realização. Além disso, apresenta maior troca de informações e exige mais largura de banda quando comparado a outras solicitações.

Para o estudo de caso 2, testamos cenários em que variamos a quantidade de componentes implantados no sistema. A implantação de um componente consiste em instalar/desinstalar ou executar/parar um componente. A escolha do componente e os estados finais de cada um deles é feita de forma aleatória. Além disso, para este estudo de caso, não foi usado o componente de descoberta, pois o problema de composição consiste apenas em resolver as dependências de instalação e execução de cada componente. O tamanho do plano é a quantidade de dependências necessárias para instalar, executar, parar ou desinstalar um componente. A sobrecarga é a divisão entre o tempo de planejamento e a execução do plano. Os resultados deste estudo de caso são apresentados na Tabela 3.

Tabela 3: Tempo de implantação dos serviços no estudo de caso de computação em nuvem

ID	Número de Componentes	Tamanho do Plano	Planejamento (seg.)	Execução (seg.)	Overhead (%)
1	3	14	0.9	3.2	28.12
2	6	30	1.1	5.3	20.75
3	10	50	2.8	10.9	25.68
4	20	100	3.1	16.8	18.45
5	30	120	2.9	17.2	16.86
6	50	165	3.4	19.0	17.89
7	70	225	4.1	20.8	19.71
8	100	317	4.7	23.8	19.74

A avaliação de desempenho comparou o tempo de execução dos aplicativos com o tempo total de descoberta, planejamento e execução. Apesar de ser uma métrica de comparação injusta, essa comparação apresenta a sobrecarga introduzida com a descoberta e a composição do serviço automático. A Figura 42 mostra os gráficos relacionados aos estudos de caso das seções 5.1 e 5.2, respectivamente. O eixo Y representa o tempo de execução e o eixo X o ID do experimento. Linhas tracejadas referem-se ao experimento sem descoberta e composição. As linhas contínuas, o experimento com descoberta e composição.

(a) Descoberta e Planejamento vs Execução - Segurança Residencial (b) Descoberta e Planejamento vs Execução - Serviços na Nuvem

Figura 42: Avaliação de desempenho dos estudos de caso

A partir dos resultados, para o estudo de caso 1, tivemos uma sobrecarga média de desempenho em torno de 99,73%. No pior cenário, chegou a 221,21% e, no melhor cenário, 39,13%. A sobrecarga elevada é esperada, pois os serviços apresentam tempo de execução baixo. No entanto, as solicitações que os usuários realizam a esses serviços são complexas, o que faz com que o mecanismo de inferência leve bastante tempo para descobrir e compor os serviços. No estudo de caso 2, onde temos um conjunto de estados e requisitos mais simples e bem definidos, tivemos uma sobrecarga média de desempenho em torno de 20,90%, sendo o pior cenário de 28,12% e no melhor cenário, de 16,86%.

Embora a sobrecarga introduzida seja aceitável, pretendemos, em trabalhos futuros, avaliar se os valores obtidos permanecem semelhantes em um estudo de caso mais complexo. Adicionalmente, faltam na literatura ferramentas de *benchmark* para que possamos avaliar o grau de aceitação dos resultados. Não comparamos com os resultados de outros trabalhos, devido à ausência de material para *download*. Sem isso, não podemos reproduzir um estudo de caso fiel o suficiente para comparar o desempenho de nossa abordagem comparada a de terceiros.

5.4 Conclusões

Esse capítulo apresentou dois estudos de caso usados na validação do *framework* SWoTPAD. O primeiro estudo de caso, o sistema de segurança residencial, se trata de um problema de IoT. Por conta disso, fez uso de grande parte dos artefatos do *framework*. O segundo estudo de caso, implantação de serviços em nuvem, embora não seja do domínio de IoT, é relevante pois valida a ideia de que o *framework* pode ser usado em outros domínios além dos de IoT.

Nos estudos de Caso 1 e 2 tivemos tempos aceitáveis. No estudo de caso 1, levando-se em consideração a habilitação do serviço mais lento que foi o *AlertMode*, tivemos um tempo para descoberta e composição de 2.8 segundos. Dado que *AlertMode* é um serviço composto, formado por outros oito serviços e que o mesmo atua na notificação de um intruso em uma residência, podemos afirmar que o tempo total de 13.3 segundos é aceitável. Para isso, basta imaginarmos a realização manual ou assistida de cada uma das ações contidas no plano da Figura 38. Mesmo que tivéssemos um aplicativo *mobile*, é possível que um ser-humano operando o aplicativo manualmente, leve mais tempo para habilitar os oito serviços.

Para o Estudo de Caso 2, a mesma ideia é válida. Dos resultados obtidos, vimos que a sobrecarga introduzido pela descoberta e planejamento, no pior cenário, teve um acréscimo de 28.12% quando comparada a manual. Dado que para manual é necessário que o usuário saiba, antecipadamente, todos os pacotes e dependências, tanto para instalação como execução da aplicação, um acréscimo de 28.12% pode ser considerado pequeno. Considere, por exemplo, o cenário 3, que apresenta 10 componentes. Se considerarmos o tempo que o usuário leva para realizar as 50 ações necessárias, facilmente o usuário consumirá mais que 2.8 segundos para executar os serviços.

6 TRABALHOS RELACIONADOS

Esse capítulo apresenta o levantamento bibliográfico sobre trabalhos que apresentam relação direta com o tema aqui proposto: a criação de um *framework* de suporte ao desenvolvimento de aplicações para SWoT. Nosso levantamento bibliográfico consistiu em três partes: na Seção 6.1, são apresentados trabalhos cuja ênfase é a composição automática de serviços, em seguida, na Seção 6.2, apresentamos soluções de suporte ao projeto de sistemas para Web semântica das coisas, por último, a Seção 6.3 descreve algumas soluções de suporte a representação de conhecimento semântico.

6.1 Composição Automática de Serviços

Nessa seção, descrevemos alguns trabalhos que apresentam abordagens para composição automática de serviços. Não necessariamente para o domínio de ambientes pervasivos ou de IoT. Nosso objetivo é apresentar o conjunto de técnicas que podem ser adotadas para resolução desse problema.

Na literatura, algumas soluções optam pelo uso de teoria dos grafos na tentativa de solucionar o problema de composição automática de serviços. Spidernet(GU; NAHRSTEDT; YU, 2004) é um *framework* para composição de serviços para redes ponto a ponto, que adota uma infraestrutura de *middleware* distribuída e mapeia automaticamente requisições feitas por seus usuários para instâncias de serviços distribuídos. Cada ponto da rede é composto por serviços que se registram e informam sua localização, bem como seus requisitos de qualidade de serviço. O módulo de requisição faz uso de um grafo de funções, que mapeia as funções solicitadas pelo usuário para serviços reais. Adicionalmente, a requisição é composta pelos requisitos de qualidade de serviço (QoS, acrônimo de Quality of Service), tais como atraso e taxa de perda de dados.

Ainda no domínio de teoria dos grafos, o trabalho desenvolvido em (SHIN; LEE; SUDA, 2009) propõe um método de composição que usa um modelo de grafo, que representa a semântica funcional do serviço Web. A tarefa de composição, adota um grafo de

relação de serviços, composta por três partes: um grafo que representa a dependência de dados entre serviços; um grafo que representa a relação entre as ações realizadas pelos serviços e o mapeamento entre esses dois grafos.

O grafo de dependência de dados é construído através do encadeamento dos serviços disponíveis e suas entradas e saídas. O grafo de ação representa conceitos de ação em nós. Todo nó serviço do grafo de dependência de dados é mapeado para um conceito de ação que o serviço provê.

Em (AGGARWAL et al., 2004), são empregadas técnicas de resolução de problemas por satisfação de restrições (RUSSELL, 2010) e *workflow*. METEOR-S é uma abordagem de composição de serviços orientada a restrições. A ideia consiste em reduzir a complexidade do problema de composição de serviços a um problema de satisfação de restrições. Para construção dessa proposta foi adotada Web semântica para representar os requisitos de cada serviço no processo. Após descoberta, serviços candidatos são selecionados com base em técnicas de *workflow* e algoritmos de satisfação de restrições. A abordagem apresenta múltiplas fases, que consiste na representação de restrições, estimativa de custos e otimização, tanto para análise de restrição, bem como para seleção ideal dos serviços.

(AGARWAL; LAMPARTER, 2005) introduzem lógica difusa para representar restrições do usuário sobre atributos de *QoS*. Nesta abordagem, o usuário pode definir suas preferências usando regras *IF-THEN*. A regra difusa expressa em que grau de satisfação está situado os atributos do usuário, baseado em seus valores combinados. O usuário pode combinar preferências atômicas, expressa em conjuntos *fuzzy*, usando conjunção, disjunção e negação. O objetivo desta proposta é a classificação das soluções que satisfazem a meta do usuário com base em seu grau de satisfação. Para cada composição, com base na qualidade calculada, avalia-se os graus do cumprimento das regras. Este grau determina a posição da composição particular no *ranking*.

Técnicas de *workflow* também costumam ser empregadas para resolução do problema de composição automática de serviços. (ZENG et al., 2003) apresentam uma plataforma chamada AgFlow que permite a composição de serviços web baseados em qualidade. Nesta plataforma, os parâmetros de *QoS* de serviços Web são avaliados. Em seguida, são selecionados serviços Web que podem otimizar *QoS* totais.

Na abordagem de otimização local, a seleção de serviço é realizada individualmente para cada tarefa e o serviço com melhor pontuação é escolhido. No planejamento global, levam-se em conta as restrições globais. Eles usam programação linear (KARLOFF, 2008) para solução do problema.

Alguns trabalhos mapeiam o problema da composição automática para um problema de busca de planos. Em (HATZI et al., 2012; HATZI et al., 2013) apresentam o *framework* PORSCE II. PORSCE-II realiza a composição automática de serviços Web semântico através do uso de técnicas de planejamento. Os passos necessários para obter a composição de serviços incluem a tradução do problema decomposição de serviços para um planejador, seguido por soluções de aquisição e, por último, tradução das soluções obtidas para uma linguagem de descrição de serviços.

Ao longo de todo processo, o sistema explora a informação semântica extraída de descrições de redes semântica dos serviços Web disponíveis, bem como ontologias associadas. Dessa forma, a execução da composição é feita sob consciência semântica. Ele usa OWL-S (MARTIN et al., 2004) para descrição dos serviços e PDDL (Planning Domain Description Language) para descrição do problema de planejamento. Requisitos não funcionais e características relacionadas ao contexto não são levadas em consideração no processo de composição.

Yale e McDermott (MCDERMOTT, 2002) usam PDDL para a especificação dos planos, porém a estende para introduzir um novo tipo de conhecimento chamado valor de uma ação que serve para representar a passagem de informação entre passos dos planos. Esta característica facilita distinguir a transformação da informação e a mudança de estados produzida pela execução do serviço. O planejador usado é um planejador baseado em regressão (encadeamento para trás) que permite a geração de planos condicionais com ramificações, necessários para o caso de serviços Web onde o ambiente é não determinista.

(WU et al., 2003) usam o planejador SHOP2 (“Simple Hierarchical Ordered Planner 2”) (NAU et al., 2003) para a composição automática de serviços Web. As entradas para o planejador são especificadas em OWL-S. SHOP2 é um planejador baseado em uma rede hierárquica de tarefas (HTN) (GHALLAB; NAU; TRAVERSO, 2004). Um dos problemas desta proposta é o fato de não poder gerar planos com estruturas de controle concorrentes (i.e. *Split* e *Split+Join*). Esse tipo de estrutura é útil na composição de serviços, sendo parte da estrutura dos processos compostos de OWL-S.

É comum mesclar planejamento com outras técnicas. Em (MCILRAITH; SON, 2002), adotam planejamento e cálculo de situações em sua abordagem. Eles adaptam e estendem a linguagem Golog para realizar a composição automática de serviços Web. O trabalho de Traverso e Pistore (TRAVERSO; PISTORE, 2004) emprega técnicas de *workflow*, planejamento e verificação simbólica de modelos. Eles apresentam uma técnica para a composição automática de SW descritos em OWL-S que permite a geração automática

de processos executáveis em BPEL4WS.

A abordagem adotada em (KLUSCH; GERBER; SCHMIDT, 2005) emprega conceitos de planejamento, grafos e algoritmos para resolução de problemas por busca. OWLS-Xplan é uma ferramenta que converte OWL-S para PDDL. O planejador PDDL é chamado por OWLS-Xplan, responsável por gerar composições. As descrições de serviço incluem pre/pós-condições, que são simples conjunções de predicados. OWLS-Xplan adota planejamento baseado em grafos para extensão de ações e o usa em conjunção com planejamento hierárquico.

Em (LIN; KUTER; SIRIN, 2008) é apresentada uma abordagem para composição personalizada de serviços Web, que além de empregar planejamento, também emprega conceitos de dominância de Pareto (VOORNEVELD, 2003). Ela considera restrições rígidas e suaves durante a composição. Restrições rígidas são declarações de restrição de valor. Restrições suaves expressam preferências do usuário.

Algumas soluções para composição automática fazem uso de soluções bioinspiradas. (CANFORA et al., 2005) apresenta uma proposta baseada em algoritmo genético para resolver o problema da composição. A função de aptidão é usada para comparação de soluções. Restrições são levadas em consideração pela função de aptidão e todas os tipos de *workflow* comumente usados em processos de negócios são levados em conta.

(YANG et al., 2010) propõe um algoritmo denominado ACAGA_WSC, que combina algoritmo de otimização de colônia de formigas e algoritmo genético para resolver o problema de composição de serviços. Algoritmo genético é empregado, com o intuito de definir os parâmetros do algoritmo de colônia de formigas.

As abordagens apresentadas até aqui lidam com o problema de composição automática de serviços no âmbito de serviços Web. A seguir, apresentamos abordagens específicas para o contexto de ambientes pervasivos ou de IoT. Em (LOPES et al., 2012), os autores propõem mecanismos para composição e seleção de serviços ubíquos. Tais mecanismos são fornecidos através do pacote OpenCopi, uma plataforma projetada para integrar um *middleware* de provisão de contexto. Na abordagem proposta, provedores de serviços publicam seus serviços usando a linguagem OWL-S. Aplicações ubíquas são consumidores de serviços e OpenCopi é o mediador que provê acesso uniforme a serviços usados pelas aplicações ubíquas. Aplicações são definidas através de *workflows* semânticos.

Em (KALDELI et al., 2013), é apresentada uma abordagem para composição automática de serviços para ambientes pervasivos baseada em problema de satisfação por restrições. Toda vez que uma meta é emitida, o planejador gera um plano, que é uma

sequência de operações de serviços, cuja execução muda a depender do estado do ambiente e de acordo com as propriedades prescritas pela meta. O plano é passado para o orquestrador, que traduz a composição para comandos de invocação de serviços de baixo nível e executa cada serviço, passo a passo, de maneira síncrona. No caso de uma operação de serviço retornar uma falha permanente, a execução de plano é finalizada e o serviço é removido do registro de dispositivos ativos. Adicionalmente, é solicitada um planejamento alternativo com a mesma meta.

(GEORGIEVSKI, 2015) apresenta uma abordagem para composição de serviços em ambientes pervasivos que adota planejamento hierárquico. Esta abordagem apresenta algumas extensões a planejamento hierárquico, pois dá suporte ao uso de variáveis numéricas nos estados. Adicionalmente, resolve o problema *phantomization*, que acontece quando em um cenário de composição automática de serviços, uma tarefa anteriormente adotada na composição, já realizou o comportamento de uma tarefa sucessora.

Diante do exposto, vimos que existem diferentes abordagens para tratar o problema de composição automática de serviços. Poucas abordagens lidam com aspectos semânticos, e, em alguns casos, resumem-se a validação dos tipos associados a entradas e saídas do serviço. Também é notável que poucas abordagens são disponibilizadas como *framework*, impossibilitando sua utilização em aplicações desenvolvidas por terceiros.

6.2 *Frameworks* para Internet das Coisas

Web semântica das coisas (*SWoT*) herda grande parte dos conceitos, tecnologias, metodologias, ontologias e linguagens aplicados com sucesso na Web Semântica. Para a representação do conhecimento e modelagem de serviços, é comum a adoção de linguagens, como o WSMML e OWL-S. As ideias expostas por essas especificações permearam o surgimento de *frameworks* para apoiar o desenvolvimento de aplicativos *SWoT*. A seguir, apresentamos alguns desses trabalhos.

SPITFIRE (PFISTERER et al., 2011) contribui para área de *SWoT* através do fornecimento de abstrações para coisas. Adicionalmente, provê serviços essenciais para pesquisa e anotação. SPITFIRE faz isso por meio da técnica de dados ligados que consiste na publicação e associação de dados estruturados na Web (BIZER; HEATH; BERNERS-LEE, 2009). Através dela, associa sensores e coisas aos dados do projeto *LOD* (acrônimo para *Linked Open Data*), que é uma base de dados aberta e ligada. A ligação entre os sensores e dados é feita de forma semi-automática, através de técnicas de agrupamento.

CityPulse (TÖNJES et al., 2014; BARNAGHI et al., 2014) é um *framework* para cidades inteligentes que processa fluxos de dados de IoT em grande escala. Ele enriquece o fluxo de dados com anotações semânticas e provê processamento adaptativo, agregação e federação de dados. Adicionalmente, o *framework* consegue lidar com adaptação inteligente, suporte à decisão centrada no usuário e processamento de informações confiáveis para aplicações de cidades inteligentes. O projeto facilita a criação e o fornecimento de aplicações de tempo real para cidades inteligentes. Dessa forma, consegue reunir a computação baseada em conhecimento e os testes de confiabilidade.

OpenIoT (SOLDATOS et al., 2015) é uma plataforma IoT de código aberto que permite a interoperabilidade semântica de serviços IoT na nuvem. No núcleo de OpenIoT encontra-se a ontologia SSN (COMPTON et al., 2012), que fornece um modelo comum baseado em padrões para a representação de sensores físicos e virtuais. Além disso, eles incluem um *middleware* de sensores que facilita a coleta de dados de praticamente qualquer sensor e, ao mesmo tempo, assegura sua própria anotação semântica. Funcionalidades fornecidas por esta abordagem: (1) integração de dados e aplicações IoT em infra-estruturas de computação em nuvem; (2) implantação e acesso seguro a aplicativos semanticamente interoperáveis; e (3) manipulação de sensores móveis e parâmetros de *QoS* associados.

Ploennigs et al (PLOENNIGS; SCHUMANN; LÉCUÉ, 2014) apresenta o *Semantic Smart Building Diagnoser*, uma abordagem que ilustra como as tecnologias e o raciocínio em Web semântica podem ajudar as aplicações do mundo real a derivar modelos complexos para predição e diagnóstico automaticamente. As técnicas semânticas foram utilizadas para integrar dados heterogêneos. Eles também estendem a ontologia SSN (COMPTON et al., 2012) para promover a criação e configuração de modelos físicos e automaticamente derivar algumas regras de diagnóstico dos modelos.

Wu et al (WU et al., 2017) apresenta o SWoT4CPS, um framework para *SWoT* para sistemas cibernéticos. SWoT4CPS fornece uma solução híbrida com métodos de engenharia de conhecimento através da extensão da ontologia SSN e métodos de aprendizado de máquina com base em um modelo de dados ligados. Ele faz dois estudos de caso para validação: um sistema de controle automático de temperatura e um detector de anomalias. Em ambos, o método de ligação relacionou o conhecimento do domínio com o DBpedia (AUER et al., 2007) com uma precisão média de 74.2%.

(URBIETA et al., 2017) propõem um *framework* para composição de serviços adaptáveis que suporta raciocínio dinâmico. O *framework* usa o wEASEL, um modelo de serviço abstrato que representa serviços e tarefas de usuários em relação à assinatura, especifica-

ção (isto é, consciência de contexto, condições, pós-condições e efeitos) e execução (ou seja, comportamento associados a fluxo de dados e restrições de fluxo de contexto).

Gyrard *et al* (GYRARD et al., 2015) propõem o framework M3 para auxiliar projetistas de sistemas de IoT na criação de aplicativos *SWoT*. A estrutura funciona com uma abordagem de três passos e oferece ontologias, conjuntos de dados, regras e consultas SPARQL (HARRIS; SEABORNE; PRUD'HOMMEAUX, 2013) relacionadas ao domínio do aplicativo. Ele também oferece algumas sugestões, notificações ou instruções sobre atuadores que serão analisados e exibidos em uma interface amigável.

Poucas abordagens têm uma preocupação real com o desenvolvedor, produzindo artefatos em tempo de projeto. Embora façam o uso de conceito de framework, poucas disponibilizam sua solução por meio de artefatos abertos e extensíveis. A única exceção é o trabalho de Gyrard *et al* (GYRARD et al., 2015), cuja abordagem pode ser aplicada em vários estágios do projeto de uma aplicação *SWoT*. Ele desempenha bem o papel de remover preocupações semânticas do desenvolvedor.

6.3 Representações e formalismos

Apesar das tentativas de criar linguagens ou padrões que ajudem no projeto de ambientes pervasivos, as soluções comerciais ainda possuem problemas de interoperabilidade e escalabilidade. A interoperabilidade na computação pervasiva é um tema de pesquisa amplamente investigado (SERRANO et al., 2015; INVERARDI; TIVOLI, 2013; NAKAZAWA et al., 2010) e é um sério problema em várias aplicações empresariais.

Uma das razões para esse problema é que, atualmente, os fornecedores de tecnologia de IoT se concentram em construir um relacionamento exclusivo com seus clientes e alavancar seus investimentos econômicos ao estabelecer barreiras contra novos fabricantes que entram no mercado. Dentro dessa estrutura ampla e heterogênea, a lógica do mercado é amarrar os consumidores a um protocolo de IoT particular, o que os obriga a comprar somente dispositivos compatíveis para manter um nível consistente de interoperabilidade (MIORI; RUSSO; ALIBERTI, 2010).

Outro problema é a escalabilidade. Tecnologias orientadas a serviço escalam se oferecem um grau significativo de descoberta automática, classificação, seleção, composição, invocação de serviços, bem como dados, protocolos e instalações de mediação de processos. A automação só pode ser alcançada se as descrições de serviços processáveis pela máquina estiverem disponíveis. Assim, a semântica é necessária para obter uma integra-

ção escalável e robusta (FENSEL et al., 2011).

Uma abordagem interessante que melhora a interoperabilidade e a escalabilidade é usar alguns conceitos adotados para serviços. Na arquitetura orientada a serviços (SOA), os provedores e os solicitantes devem se comunicar de forma significativa entre si, apesar da natureza heterogênea das estruturas de informação subjacentes, artefatos empresariais e outros documentos. Este requisito é conhecido como interoperabilidade semântica (BANDYOPADHYAY; SEN, 2011). Para garantir a interoperabilidade entre diferentes aplicações, é necessário fornecer dados com formatos, modelos e descrições semânticas adequados. Assim, as aplicações podem suportar o raciocínio automatizado, uma característica fundamental para permitir a adoção bem-sucedida de tal tecnologia em larga escala (MIORANDI et al., 2012).

As ontologias são freqüentemente usadas para resolver problemas de interoperabilidade semântica e escalabilidade em IoT (HACHEM; TEIXEIRA; ISSARNY, 2011; WANG et al., 2012b; MEREZEANU; VASILESCU; DOBRESU, 2016). (HACHEM; TEIXEIRA; ISSARNY, 2011) apresentam uma abordagem que descreve dispositivos que usam três ontologias: ontologia de dispositivos, ontologia de domínio físico e ontologia de estimativas. As ontologias são usadas para compor serviços para descoberta de dispositivo e para interação de dispositivo.

Wang *et al.* fornecem uma ontologia abrangente para a representação do conhecimento em IoT (WANG et al., 2012b). Kosek *et al.* (KOSEK; SYED; KERRIDGEY, 2010) destacam os problemas de interoperabilidade na computação pervasiva e apresenta uma solução para dispositivos com capacidades de processamento limitados através do uso de uma ontologia para formulação de conhecimento e interoperabilidade semântica.

Merezeanu *et al.* (MEREZEANU; VASILESCU; DOBRESU, 2016) apresentam uma abordagem semântica para integrar as tecnologias de redes de sensores sem fio, IoT e computação em nuvem em um *framework* que admite capacidades de detecção, computação e comunicação sensíveis ao contexto em aplicações industriais. A modelagem da camada IoT é descrita em OWL.

O projeto SOUPA faz parte de um esforço contínuo de Web semântica do *UBIComp Special Interest Group 1*, um grupo internacional de pesquisadores da academia e indústria que usa a linguagem OWL para aplicações de computação ubíqua e define casos de uso orientados por ontologia, demonstrando aspectos da visão computacional onipresente. O projeto visa definir ontologias para suportar aplicações de computação ubíqua (CHEN; FININ; JOSHI, 2005).

USDL (NAKAZAWA et al., 2010) é um idioma para descrever entidades de computação pervasiva, incluindo objetos que fornecem serviços digitais, espaços que contêm objetos e humanos que os gerenciam e/ou usam. Basicamente, adota os seguintes princípios: (1) divisão da descrição em duas partes (tipos e entidades), (2) habilitando descrições baseadas em XML legíveis para humanos e máquinas, e (3) descrevendo tanto informações específicas de tipo e entidades. O USDL fornece aplicativos com documentos embutidos para determinar mecanicamente a melhor estratégia para compartilhar os recursos entre múltiplos usuários em um ambiente de computação ubíqua.

Perv-ML cite munoz2005 é uma linguagem de domínio específico (DSL, acrônimo do termo em inglês Domain-Specific Language) projetada para fornecer um conjunto de elementos que permitem descrever o sistema pervasivo independentemente da tecnologia. Promove a separação de papéis e classifica os desenvolvedores como analistas e arquitetos. O Perv-ML segue o padrão de arquitetura orientada a modelos (MDA acrônimo do termo Model Driven Architecture) para definir a linguagem de domínio específico e a etapa de geração automática de código. A linguagem é simples e uma solução eficaz para modelagem de ambientes pervasivos.

Uma possível solução para interoperabilidade e problemas de escalabilidade é a adoção de descrição semântica e métodos formais. (CÁMARA et al., 2006) propõem uma abordagem que descreve a formalização dos processos de negócios em WS-BPEL (BLOCH et al., 2003) e que adiciona informações de protocolo à especificação de serviços Web interativos. Além disso, usa CSS (MILNER, 1980), uma linguagem de álgebra de processo, para modelar seu comportamento dinâmico. O uso de CSS permite a análise formal e a inferência de propriedades relevantes aos sistemas que estão sendo criados. Embora promissor, o uso do WS-BPEL restringe a abordagem, porque WS-BPEL não possui todos os elementos necessários para modelar descrições de serviço e pedidos de usuários.

Wang *et al.* apresentam um modelo formal e não ambíguo em Object-Z de WSMO que define diferentes aspectos da linguagem em um único *framework* unificado (WANG et al., 2012a). Este modelo pode ser usado para desenvolver ferramentas, bem como para validar automaticamente especificações WSMO. Devido a essas vantagens, nosso trabalho também usa Object-Z para modelar SWoTPADL.

Nossa abordagem consolida algumas idéias apresentadas nesta seção. SWoTPADL é uma linguagem que usa e adota conceitos de serviços Web semânticos. Adicionalmente, apresenta uma especificação formal em Object-Z para evitar a ambiguidade e habilitar a verificação automática.

6.4 Conclusões

Nesse capítulo apresentamos algumas abordagens que apresentam relação com o *framework* desenvolvido SWoTPAD. Por se tratar de um *framework*, que apresenta um módulo específico para composição automática de serviço e propõe um novo vocabulário para especificação de serviços em IoT, apresentamos trabalhos relacionados a esses três domínios.

Diante do que foi exposto, é perceptível a grande quantidade de soluções que podem ser aplicadas para o problema de suporte ao projeto de IoT. No entanto, a literatura não apresenta trabalho similar ao nosso, que propôs a seguinte solução: a criação de uma *framework* para IoT que apresenta (1) uma camada específica para modelagem, dando suporte a essa etapa do projeto através de um modelo conceitual formalizado em uma linguagem de especificação formal; (2) uma camada para o desenvolvimento de especificações semânticas das entidades que compõe o sistema de IoT, através da Linguagem SWoTPADl; e (3) uma camada funcional, adotada para codificação e realização de testes, que já apresenta os serviços de registro, descoberta e composição automática de serviços.

7 CONCLUSÕES

Esse trabalho teve como objetivo o desenvolvimento de um *framework* para dar suporte a utilização de semântica em projetos de *IoT*. Apesar da forte perspectiva de crescimento da área de *IoT*, a mesma prossegue com problemas clássicos, tais como a questão da interoperabilidade e escalabilidade de sistemas. A adoção de semântica em projetos de *IoT* pode solucionar esse problema, no entanto, o conhecimento especializado, necessário para o desenvolvimento de anotações, ontologias e regras, têm sido uma barreira natural para sua adoção por grande parte dos desenvolvedores. Esse trabalho defende a ideia de que, a melhor forma de aplicar semântica e estimular seu crescimento, é através da disponibilização de elementos facilitadores, que despertem o interesse e necessidade da comunidade de desenvolvedores.

Para isso, propomos o *framework* SWoTPAD, embutido na ferramenta IoTPAD *Designer*, disponibilizada através de um conjunto de *plugins* para a IDE Eclipse e oferece os seguintes recursos:

- I. A possibilidade de descrever serviços e requisições, adotando a linguagem (SWoTPADL), que apresenta sintaxe mais leve que WSMML e mais adequada ao domínio de *IoT*. Adicionalmente, a ferramenta permite obter o *grounding* automático, a partir de especificações semânticas.
- II. Geração dos módulos para descoberta e composição automática de serviços. Integração desses módulos ao sistema de *IoT* desenvolvido.
- III. Um modelo conceitual de sistemas de *IoT*. Esse modelo representa o conhecimento de domínio de nosso *framework*. Adicionalmente, oferece um espaço para troca, extensão e integração de conhecimento.
- IV. A possibilidade de descrever ambientes utilizando descrições em GeoJSON e a conversão automática desse conhecimento em ontologias WSMML.

A abordagem desenvolvida esta de acordo com os objetivos gerais e específicos apresentados na Seção 1.3, conforme descrito a seguir:

- I. **Desenvolvimento de um framework formal para composição automática de serviços para IoT.** Conforme apresentado no Capítulo 4.
- II. **Desenvolvimento de uma linguagem formal para especificação de serviços e requisições apropriadas para o domínio de IoT.** Conforme apresentado no Capítulo 4, nas seções 4.2.1 e 4.2.4.
- III. **Adaptação de algoritmos para descoberta e composição automática.** Conforme apresentado no Capítulo 4, seções 4.3.1 e 4.3.2.
- IV. **Desenvolvimento de um modelo conceitual que possa ser associado a aplicações de IoT de diferentes domínios.** Conforme apresentado no Capítulo 3.
- V. **Disponibilização desse conjunto de soluções por meio de um *framework* extensível, disponibilizado no âmbito de uma IDE de desenvolvimento, para dar apoio a projetos de IoT.** Conforme apresentado no Capítulo 4, na Seção 4.1.

Acreditamos que com essa abordagem, projetos de IoT passam a ter mais uma opção de apoio sólido, que afasta das mãos dos projetistas, algumas preocupações associadas a projetos dessa natureza.

7.1 Limitações

O fato de termos adotado WSML como base de nossa solução, apesar de suficiente para validação de nossas ideias, nos trouxe algumas limitações. O motor de inferência adotado por WSML, o IRIS (BISHOP; FISCHER, 2008), não lida bem com recursividade, nem com operadores existencial e universal. O motor de inferência não foi capaz de apresentar resultados. Apesar dessa limitação, esse problema pode ser facilmente resolvido, com o desenvolvimento ou adaptação de um novo motor de inferência.

Do ponto de vista da composição automática, os planejadores adotados em nossa abordagem não lidam com temporalidade. Nos estudos de caso desenvolvidos, não sentimos falta desse recurso, porém, julgamos ser necessária seu desenvolvimento como trabalhos

futuros. É válido lembrar que um dos motivos que levaram o presente trabalho a apresentar sua solução como *framework*, foi o de justamente permitir sua extensão e fácil acomodação de novas soluções.

7.2 Trabalhos Futuros

Como trabalhos futuros, destacamos três questões, são elas: aprofundamento de questões semânticas, desenvolvimento de outras técnicas para composição automática de serviços, adaptação e especialização dessa abordagem para outros domínios do conhecimento.

Avaliamos algumas possibilidades de trabalhos futuros para nosso *framework* que julgamos consoante com a área de SWoT. Para dar melhor suporte a sistemas de IoT já implantados, acreditamos que a extensão desse *framework*, no sentido a dar suporte a técnicas de aprendizado de máquina e detecção padrões, com a intenção de inferir automaticamente ontologias para elementos de IoT é uma demanda necessária.

Também é desejável a extração automática do conhecimento de domínio referenciado neste conjunto de dados e redesenhá-lo de forma interoperável. Isso permitirá atualizar automaticamente as bases de conhecimento. Para alcançar esse desafio, é necessário melhorar as ferramentas da Web semântica para incentivar melhores práticas e uma melhor interoperabilidade para interligar facilmente o conhecimento do domínio.

No âmbito do planejador, há espaço para melhora. Um primeiro ponto, refere-se às preferências do usuário, questões temporais e espaciais. Nesse sentido, é necessário estudos mais detalhados sobre a melhor forma de utilizar essa informação no âmbito do planejador. Em nossa abordagem, essas propriedades são traduzidas para axiomas (regras), que em algum momento, as aplica por meio de *requests* e são realizadas, pela execução de um serviço.

Um outro ponto não abordado nesse trabalho é a real necessidade de um planejador mais robusto, com suporte, por exemplo, a temporalidade e ordenação parcial. Nos estudos de casos apresentados nesse trabalho, o planejador se mostrou suficiente. No entanto, não é seguro afirmar se, em um ambiente de IoT com uma ampla gama de serviços, ou ainda, em um ambiente de IoT com pontos cegos quanto ao sensoriamento, se o planejador aqui proposto também atenderia bem essas demandas. Adicionalmente, é interessante ao *framework*, o uso de outras abordagens para composição automática, como algumas apresentadas na Seção 6.1, com o intuito de avaliar a abordagem que melhor se adequa ao domínio de IoT.

REFERÊNCIAS

- ABANDA, F. H.; TAH, J. H.; KEIVANI, R. Trends in built environment semantic web applications: Where are we today? *Expert Systems with Applications*, Elsevier, v. 40, n. 14, p. 5563–5577, 2013.
- ADOLPHS, P. et al. Reference architecture model industrie 4.0 (rami4. 0). *ZVEI and VDI, Status Report*, 2015.
- AGARWAL, S.; LAMPARTER, S. User preference based automated selection of web service compositions. In: *ICSOC Workshop on Dynamic Web Processes*. [S.l.: s.n.], 2005. v. 12, p. 1–12.
- AGGARWAL, R. et al. Constraint driven web service composition in METEOR-S. In: IEEE. *Services Computing, 2004.(SCC 2004). Proceedings. 2004 IEEE International Conference on*. [S.l.], 2004. p. 23–30.
- AHMED, N.; RAHMAN, H.; HUSSAIN, M. I. Scalability analysis of medium access control protocols for internet of things. In: SPRINGER. *Proceedings of International Conference on Communication and Networks*. [S.l.], 2017. p. 601–611.
- AL, G.; DOSS, R.; CHOWDHURY, M. Adjusting matryoshka protocol to address the scalability issue in iot environment. In: SPRINGER. *International Conference on Future Network Systems and Security*. [S.l.], 2017. p. 84–94.
- ALONSO, G. et al. *Web services: concepts, architectures and applications*. [S.l.]: Springer Berlin, 2004.
- ASHTON, K. That 'Internet of Things' Thing. *RFiD Journal*, v. 22, n. 7, 2011.
- AUER, S. et al. Dbpedia: A nucleus for a web of open data. *The semantic web*, Springer, p. 722–735, 2007.
- BANDYOPADHYAY, D.; SEN, J. Internet of Things: Applications and Challenges in Technology and Standardization. *Wirel. Pers. Commun.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 58, n. 1, p. 49–69, maio 2011. ISSN 0929-6212.
- BARNAGHI, P. et al. Citypulse: Real-time IoT stream processing and large-scale data analytics for smart city applications. In: *European Semantic Web Conference (ESWC)*. [S.l.: s.n.], 2014. v. 2014.
- BARNAGHI, P. et al. Semantics for the internet of things: early progress and back to the future. *International Journal on Semantic Web and Information Systems (IJSWIS)*, IGI Global, v. 8, n. 1, p. 1–21, 2012.
- BAUER, M. et al. Internet of things-architecture iot-a deliverable d1.3–updated reference model for iot v1.5. 2013.

- BERMUDEZ-EDO, M. et al. IoT-Lite: a lightweight semantic model for the Internet of Things. In: IEEE. *Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, 2016 Intl IEEE Conferences. [S.l.], 2016. p. 90–97.
- BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. The Semantic Web. *Scientific American*, v. 284, n. 5, p. 34–43, maio 2001. Disponível em: <<http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21>>.
- BISHOP, B.; FISCHER, F. Iris-integrated rule inference system. In: *International Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008)*. [S.l.: s.n.], 2008.
- BIZER, C.; HEATH, T.; BERNERS-LEE, T. Linked data-the story so far. *Semantic services, interoperability and web applications: emerging concepts*, p. 205–227, 2009.
- BLOCH, B. et al. Web services business process execution language. *OASIS Open Inc*, p. 21, 2003.
- BOUROS, P.; PANEPISIMIOPOLIS, T. *Semantic Web Services: A conceptual comparison of OWL-S, WSMO and METEOR-S approaches*. [S.l.], 2005.
- BRAY, T. The javascript object notation (json) data interchange format. 2017.
- BRICKLEY, D.; MILLER, L. *FOAF vocabulary specification 0.91*. [S.l.]: Technical report, ILRT, 2007.
- BRÖRING, A. et al. Enabling iot ecosystems through platform interoperability. *IEEE software*, IEEE, v. 34, n. 1, p. 54–61, 2017.
- BRUIJN, J. D. et al. *Web Service Modeling Ontology (WSMO)*. 2005. W3C member submission. Disponível em: <<http://www.w3.org/Submission/WSMO/>>.
- BRUIJN, J. D. et al. *D16.1v1.0 The Web Service Modeling Language WSML*. ESSI WSML working group, 2008. Disponível em: <<http://www.wsmo.org/TR/d16/d16.1/v1.0/>>.
- BUYYA, R.; DASTJERDI, A. V. *Internet of Things: Principles and paradigms*. [S.l.]: Elsevier, 2016.
- CÁMARA, J. et al. Formalizing WSBPEL Business Processes Using Process Algebra. *Electron. Notes Theor. Comput. Sci.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 154, n. 1, p. 159–173, maio 2006. ISSN 1571-0661.
- CANFORA, G. et al. An approach for QoS-aware service composition based on genetic algorithms. In: ACM. *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. [S.l.], 2005. p. 1069–1075.
- CHEN, H.; FININ, T.; JOSHI, A. The SOUPA ontology for pervasive computing. In: *Ontologies for agents: Theory and experiences*. [S.l.]: Springer, 2005. p. 233–258.
- COMPTON, M. et al. The SSN ontology of the W3C semantic sensor network incubator group. *Web semantics: science, services and agents on the World Wide Web*, Elsevier, v. 17, p. 25–32, 2012.

- COSMO, R. D.; ZACCHIROLI, S.; ZAVATTARO, G. Towards a formal component model for the cloud. In: SPRINGER. Thessaloniki, Greece, 2012. (International Conference on Software Engineering and Formal Methods), p. 156–171.
- DEAN, M. et al. Owl web ontology language reference. *W3C Recommendation February*, v. 10, 2004.
- DERRICK, J.; BOITEN, E. *Refinement in Z and Object-Z*. [S.l.]: Springer, 2001. v. 30:4.
- DUKE, R.; ROSE, G. *Formal object-oriented specification using Object-Z*. [S.l.]: Macmillan, 2000.
- DUSTDAR, S.; SCHREINER, W. A survey on web services composition. *International journal of web and grid services*, Inderscience Publishers, v. 1, n. 1, p. 1–30, 2005.
- FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. *Communications of the ACM*, ACM, v. 40, n. 10, p. 32–38, 1997.
- FENSEL, D. et al. *Semantic web services*. [S.l.]: Springer Science & Business Media, 2011.
- FERRIS, B.; JACOBSON, K. RECommentadions Ontology. 2010.
- FORTINO, G. et al. Towards multi-layer interoperability of heterogeneous iot platforms: The inter-iot approach. In: *Integration, Interconnection, and Interoperability of IoT Systems*. [S.l.]: Springer, 2018. p. 199–232.
- GEORGIEVSKI, I. *Coordinating services embedded everywhere via hierarchical planning*. Tese (Doutorado) — University of Groningen, October 2015.
- GHALLAB, M.; NAU, D.; TRAVERSO, P. *Automated Planning: theory and practice*. [S.l.]: Elsevier, 2004.
- GHALLAB, M.; NAU, D.; TRAVERSO, P. *Automated Planning and Acting*. [S.l.]: Cambridge University Press, 2016.
- GLIMM, B.; STUCKENSCHMIDT, H. 15 years of semantic web: An incomplete survey. *KI-Künstliche Intelligenz*, Springer, v. 30, n. 2, p. 117–130, 2016.
- GRAVINA, R. et al. *Integration, Interconnection, and Interoperability of IoT Systems*. [S.l.]: Springer, 2018.
- GRINBERG, M. *Flask web development: developing web applications with python*. [S.l.]: "O'Reilly Media, Inc.", 2014.
- GRUBER, T. R. A translation approach to portable ontology specifications. *Knowledge acquisition*, Elsevier, v. 5, n. 2, p. 199–220, 1993.
- GU, X.; NAHRSTEDT, K.; YU, B. SpiderNet: An integrated peer-to-peer service composition framework. In: IEEE. *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*. [S.l.], 2004. p. 110–119.
- GUARINO, N.; OBERLE, D.; STAAB, S. What is an ontology? In: *Handbook on ontologies*. [S.l.]: Springer, 2009. p. 1–17.

- GUBBI, J. et al. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, Elsevier, v. 29, n. 7, p. 1645–1660, 2013.
- GUINARD, D. et al. From the internet of things to the web of things: Resource-oriented architecture and best practices. In: *Architecting the Internet of things*. [S.l.]: Springer, 2011. p. 97–129.
- GUREVICH, Y.; ROSSMAN, B.; SCHULTE, W. Semantic essence of AsmL. *Theor. Comput. Sci.*, v. 343, n. 3, p. 370–412, 2005. Disponível em: <<http://research.microsoft.com/apps/pubs/default.aspx?id=77435>>.
- GYRARD, A. et al. Assisting iot projects and developers in designing interoperable semantic web of things applications. In: IEEE. *Data Science and Data Intensive Systems (DSDIS), 2015 IEEE International Conference on*. [S.l.], 2015. p. 659–666.
- GYRARD, A. et al. Lov4iot: A second life for ontology-based domain knowledge to build semantic web of things applications. In: IEEE. *Future Internet of Things and Cloud (FiCloud), 2016 IEEE 4th International Conference on*. [S.l.], 2016. p. 254–261.
- GYRARD, A. et al. Semantic Web meets Internet of Things (IoT) and Web of Things (WoT). In: *The 15th International Conference on Semantic Web (ISWC).(Oct 2016)*. [S.l.: s.n.], 2016.
- GYRARD, A. et al. Workshop on semantic interoperability in the iot and wot. In: *in Global IoT Summit*. [S.l.: s.n.], 2018.
- HACHEM, S.; TEIXEIRA, T.; ISSARNY, V. Ontologies for the Internet of Things. In: *Proceedings of the 8th Middleware Doctoral Symposium*. New York, NY, USA: ACM, 2011. (MDS '11), p. 3:1–3:6. ISBN 978-1-4503-1072-7.
- HALL, M. et al. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, ACM, v. 11, n. 1, p. 10–18, 2009.
- HARRIS, S.; SEABORNE, A.; PRUD'HOMMEAUX, E. SPARQL 1.1 query language. *W3C recommendation*, v. 21, n. 10, 2013.
- HATZI, O. et al. The PORSCE II framework: Using AI planning for automated semantic web service composition. *The Knowledge Engineering Review*, Cambridge University Press, v. 28, n. 2, p. 137–156, 2013.
- HATZI, O. et al. An integrated approach to automated semantic web service composition through planning. *IEEE Transactions on Services Computing*, IEEE, v. 5, n. 3, p. 319–332, 2012.
- HAUSWIRTH, M.; DECKER, S. Semantic reality-connecting the real and the virtual world. In: *Microsoft SemGrail Workshop*. [S.l.: s.n.], 2007. p. 21–22.
- HEPP, M. *Accommodation ontology language reference*. [S.l.], 2013.
- HOBBS, J. R.; PAN, F. Time ontology in OWL. *W3C working draft*, v. 27, p. 133, 2006.
- HOLOVATY, A.; KAPLAN-MOSS, J. *The definitive guide to Django: Web development done right*. [S.l.]: Apress, 2009.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computations*. 3rd. ed. [S.l.]: Prentice Hall, 2006. ISBN 0321455363.

HORROCKS, I.; PADGHAM, L.; THOMSON, L. Feasibility of optimised disjunctive reasoning for approximate matching. *Advanced Topics in Artificial Intelligence*, Springer, p. 328–339, 1999.

HOVORUSHCHENKO, T.; POMOROVA, O. Ontological approach to the assessment of information sufficiency for software quality determination. CEUR-WS, 2016.

INVERARDI, P.; TIVOLI, M. Automatic Synthesis of Modular Connectors via Composition of Protocol Mediation Patterns. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 3–12. ISBN 978-1-4673-3076-3. Disponível em: <<http://dl.acm.org/citation.cfm?id=2486788.2486790>>.

JARA, A. J. et al. Semantic Web of Things: an analysis of the application semantics for the iot moving towards the iot convergence. *International Journal of Web and Grid Services*, Inderscience Publishers Ltd, v. 10, n. 2-3, p. 244–272, 2014.

KALDELI, E. et al. Coordinating the web of services for a smart home. *ACM Transactions on the Web (TWEB)*, ACM, v. 7, n. 2, p. 10, 2013.

KAMARUDDIN, L. A.; SHEN, J.; BEYDOUN, G. Evaluating Usage of WSMO and OWL-S in Semantic Web Services. In: *Proceedings of the Eighth Asia-Pacific Conference on Conceptual Modelling - Volume 130*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2012. (APCCM '12), p. 53–58. ISBN 978-1-921770-11-1.

KARAU, H. et al. *Learning spark: lightning-fast big data analysis*. [S.l.]: "O'Reilly Media, Inc.", 2015.

KARLOFF, H. *Linear programming*. [S.l.]: Springer Science & Business Media, 2008.

KASSEL, G.; SMITH, G. Model checking Object-Z classes: Some experiments with FDR. In: IEEE. *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*. [S.l.], 2001. p. 445–452.

KLUSCH, M.; GERBER, A.; SCHMIDT, M. Semantic web service composition planning with owls-xplan. In: *Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*. [S.l.: s.n.], 2005. p. 55–62.

KORTUEM, G. et al. Smart objects as building blocks for the internet of things. *IEEE Internet Computing*, IEEE, v. 14, n. 1, p. 44–51, 2010.

KOSEK, A.; SYED, A.; KERRIDGEY, J. RDF recipes for context-aware interoperability in pervasive systems. In: *Computers and Communications (ISCC), 2010 IEEE Symposium on*. [S.l.: s.n.], 2010. p. 1017–1022. ISSN 1530-1346.

KUTER, U. *Planning under Uncertainty: Moving Forward*. Tese (Doutorado) — University of Maryland, College Park, 2006. Disponível em: <<http://www.cs.umd.edu/~ukuter/papers/kuter06thesis.pdf>>.

- LANO, K. Book Review: Formal Object-Oriented Specification Using Object-Z, by Roger Duke and Gordon Rose, Macmillan Press. *Softw. Test., Verif. Reliab.*, v. 11, n. 1, p. 55, 2001. Disponível em: <<http://dblp.uni-trier.de/db/journals/stvr/stvr11.html#Lano01>>.
- LARA, R. et al. Web Services: European Conference, ECOWS 2004, Erfurt, Germany, September 27-30, 2004. Proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. cap. A Conceptual Comparison of WSMO and OWL-S, p. 254–269. ISBN 978-3-540-30209-4.
- LAUSEN, H.; FARRELL, J. Semantic annotations for wsdl and xml schema. *W3C recommendation, W3C*, v. 69, 2007.
- LI, S.; XU, L. D.; ZHAO, S. The internet of things: a survey. *Information Systems Frontiers*, Springer, v. 17, n. 2, p. 243–259, 2015.
- LIN, N.; KUTER, U.; SIRIN, E. Web service composition with user preferences. *The Semantic Web: Research and Applications*, Springer, p. 629–643, 2008.
- LIN, S.-W. et al. Industrial internet reference architecture. *Industrial Internet Consortium (IIC), Tech. Rep*, 2015.
- LIU, W. et al. Adaptive resource discovery in mobile cloud computing. *Computer Communications*, Elsevier, v. 50, p. 119–129, 2014.
- LOPES, F. A. d. S. et al. Dynamic and semantic web services composition for ubiquitous computing. In: ACM. *Proceedings of the 18th Brazilian symposium on Multimedia and the web*. [S.l.], 2012. p. 151–160.
- MAKONIN, S.; BARTRAM, L.; POPOWICH, F. A smarter smart home: Case studies of ambient intelligence. *IEEE Pervasive Computing*, IEEE, v. 12, n. 1, p. 58–66, 2013.
- MALIK, P.; UTTING, M. CZT: A framework for Z tools. *ZB*, Springer, v. 3455, p. 65–84, 2005.
- MANYIKA, J. et al. The Internet of Things: Mapping the Value Beyond the Hype. *McKinsey Global Institute*, p. 3, 2015.
- MARTIN, D. et al. *OWL-S: Semantic Markup for Web Services*. 2004. Disponível em: <<http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/>>.
- MCDERMOTT, D. V. Estimated-Regression Planning for Interactions with Web Services. In: *AIPS*. [S.l.: s.n.], 2002. v. 2, p. 204–211.
- MCILRAITH, S.; SON, T. C. Adapting golog for composition of semantic web services. *KR*, v. 2, p. 482–493, 2002.
- MEDJAHED, B.; BOUGUETTAYA, A. *Service composition for the Semantic Web*. [S.l.]: Springer Science & Business Media, 2011.
- MEREZEANU, D.; VASILESCU, G.; DOBRESCU, R. Context-aware Control Platform for Sensor Network Integration in IoT and Cloud. *Studies in Informatics and Control*, NATL INST R&D INFORMATICS-ICI PUBL DEPT, 8-10 AVERESCU BLVD, SECTOR 1, BUCHAREST, 011455, ROMANIA, v. 25, n. 4, p. 489–498, 2016.

- MILNER, R. A calculus of communicating systems. Springer, 1980.
- MIORANDI, D. et al. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, Elsevier, v. 10, n. 7, p. 1497–1516, 2012.
- MIORI, V.; RUSSO, D. Domotic evolution towards the IoT. In: IEEE. *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*. [S.l.], 2014. p. 809–814.
- MIORI, V.; RUSSO, D.; ALIBERTI, M. Domotic Technologies Incompatibility Becomes User Transparent. *Commun. ACM*, ACM, New York, NY, USA, v. 53, n. 1, p. 153–157, jan. 2010. ISSN 0001-0782.
- NAKAZAWA, J. et al. A Description Language for Universal Understandings of Heterogeneous Services in Pervasive Computing. In: *Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC), 2010 IEEE International Conference on*. [S.l.: s.n.], 2010. p. 161–168.
- NAU, D. S. et al. SHOP2: An HTN planning system. *Journal of artificial intelligence research*, v. 20, p. 379–404, 2003.
- PANDA, S. Security issues and challenges in internet of things. In: *Handbook of Research on Advanced Wireless Sensor Network Applications, Protocols, and Architectures*. [S.l.]: IGI Global, 2017. p. 369–385.
- PANZIERA, L. et al. WSML or OWL? A Lesson Learned by Addressing NFP-based Selection of Semantic Web Services. In: *Proceedings of the Non Functional Properties and SLA Management in Service-Oriented Computing Workshop*. [S.l.: s.n.], 2010. (NFPSLAM-SOC 2010).
- PAOLUCCI, M. et al. Semantic matching of web services capabilities. In: SPRINGER. *International Semantic Web Conference*. [S.l.], 2002. p. 333–347.
- PAPAZOGLU, M. P. et al. Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, World Scientific, v. 17, n. 02, p. 223–255, 2008.
- PEDREGOSA, F. et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, v. 12, n. Oct, p. 2825–2830, 2011.
- PFISTERER, D. et al. SPITFIRE: toward a semantic web of things. *IEEE Communications Magazine*, IEEE, v. 49, n. 11, p. 40–48, 2011.
- PLOENNIGS, J.; SCHUMANN, A.; LÉCUÉ, F. Adapting semantic sensor networks for smart building diagnosis. In: SPRINGER. *International Semantic Web Conference*. [S.l.], 2014. p. 308–323.
- POSLAD, S. *Ubiquitous computing: smart devices, environments and interactions*. [S.l.]: John Wiley & Sons, 2011.
- PRAJOGO, D.; OHLAGER, J. Supply chain integration and performance: The effects of long-term relationships, information technology and sharing, and logistics integration. *International Journal of Production Economics*, Elsevier, v. 135, n. 1, p. 514–522, 2012.

- RAJ, P.; RAMAN, A. C. *The Internet of Things: Enabling Technologies, Platforms, and Use Cases*. [S.l.]: CRC Press, 2017.
- RAO, J.; SU, X. A survey of automated web service composition methods. In: SPRINGER. *SWSWPC*. [S.l.], 2004. v. 3387, p. 43–54.
- RIEHLE, D. *Framework design*. Tese (Doutorado) — Diss. Technische Wissenschaften ETH Zürich, Nr. 13509, 2000, 2000.
- RUSSELL, P. N. S. *Artificial Intelligence: A Modern Approach*. 3rd. ed. [S.l.]: Prentice Hall, 2010. (Prentice Hall Series in Artificial Intelligence). ISBN 0136042597, 9780136042594.
- SERRANO, M. et al. *Cross-Domain Interoperability Using Federated Interoperable Semantic IoT/Cloud Testbeds and Applications: The FIESTA-IoT Approach*. [S.l.]: River Publishers, 2017.
- SERRANO, M. et al. The stack for service delivery models and interoperability in the Internet of Things: A practical case with OpenIoT-VDK. *Selected Areas in Communications, IEEE Journal on*, PP, n. 99, p. 1–1, 2015. ISSN 0733-8716.
- SHARIFI, O.; BAYRAM, Z. A Critical Evaluation of Web Service Modeling Ontology and Web Service Modeling Language. In: SPRINGER. *International Symposium on Computer and Information Sciences*. [S.l.], 2016. p. 97–105.
- SHELBY, Z.; HARTKE, K.; BORMANN, C. The constrained application protocol (coap). 2014.
- SHENG, Q. Z. et al. Web services composition: A decade's overview. *Information Sciences*, Elsevier, v. 280, p. 218–238, 2014.
- SHETH, A. Internet of Things to Smart IoT Through Semantic, Cognitive, and Perceptual Computing. *IEEE Intelligent Systems*, v. 31, n. 2, p. 108–112, Mar 2016. ISSN 1541-1672.
- SHIN, D.-H.; LEE, K.-H.; SUDA, T. Automated generation of composite web services based on functional semantics. *Web Semantics: Science, Services and Agents on the World Wide Web*, Elsevier, v. 7, n. 4, p. 332–343, 2009.
- SILVA, A. L. M. *WSML-PE Wiki*. 2016. <<https://gitlab.com/andre Luis-ms/WSML-PE/wikis/home>>. Accessed: 23.02.2016.
- SMITH, G. *The Object-Z specification language*. [S.l.]: Springer Science & Business Media, 2012. v. 1.
- SOHRABI, S.; MCILRAITH, S. A. On planning with preferences in htn. In: *Proc. of the 12th Int'l Workshop on Non-Monotonic Reasoning (NMR)*. [S.l.: s.n.], 2008. p. 241–248.
- SOLDATOS, J. et al. Openiot: Open source internet-of-things in the cloud. In: *Interoperability and open-source solutions for the internet of things*. [S.l.]: Springer, 2015. p. 13–25.

- SPIESS, P. et al. Soa-based integration of the internet of things in enterprise services. In: IEEE. *Web Services, 2009. ICWS 2009. IEEE International Conference on*. [S.l.], 2009. p. 968–975.
- SPIVEY, J. The z notation: A reference manual, 1998. *Published by the author at <http://spivey.oriel.ox.ac.uk/~mike/zrm>*, 1998.
- TÖNJES, R. et al. Real time IoT stream processing and large-scale data analytics for smart city applications. In: *poster session, European Conference on Networks and Communications*. [S.l.: s.n.], 2014.
- TRAVERSO, P.; PISTORE, M. Automated composition of semantic web services into executable processes. In: SPRINGER. *International Semantic Web Conference*. [S.l.], 2004. v. 3298, p. 380–394.
- URBIETA, A. et al. Adaptive and context-aware service composition for IoT-based smart cities. *Future Generation Computer Systems*, p. –, 2017. ISSN 0167-739X. Disponible em: <<http://www.sciencedirect.com/science/article/pii/S0167739X16308688>>.
- VOORNEVELD, M. Characterization of pareto dominance. *Operations Research Letters*, Elsevier, v. 31, n. 1, p. 7–11, 2003.
- WANG, H. et al. A formal model of the Semantic Web Service Ontology (WSMO). *Information Systems*, Elsevier, v. 37, n. 1, p. 33–60, March 2012. Disponible em: <<http://eprints.soton.ac.uk/273195/>>.
- WANG, W. et al. A Comprehensive Ontology for Knowledge Representation in the Internet of Things. In: *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. [s.n.], 2012. p. 1793 – 1798. Disponible em: <<http://epubs.surrey.ac.uk/716727/>>.
- WEISER, M. The computer for the 21st century. *Mobile Computing and Communications Review*, v. 3, n. 3, p. 3–11, 1999.
- WHITE, T. *Hadoop: The definitive guide*. [S.l.]: "O'Reilly Media, Inc.", 2012.
- WING, J. M. A specifier's introduction to formal methods. *Computer*, IEEE, v. 23, n. 9, p. 8–22, 1990.
- WINTER, K.; DUKE, R. Model checking Object-Z using ASM. In: SPRINGER. *International Conference on Integrated Formal Methods*. [S.l.], 2002. p. 165–184.
- WITCHALLS, C.; CHAMBERS, J. The Internet of Things business index: A quiet revolution gathers pace. *The Economist Intelligence Unit*. Retrieved from <http://www.economistinsights.com/analysis/internet-things-business-index>, 2013.
- WU, D. et al. Automating DAML-S web services composition using SHOP2. *The Semantic Web-ISWC 2003*, Springer, p. 195–210, 2003.
- WU, Z. et al. Towards a Semantic Web of Things: A Hybrid Semantic Annotation, Extraction, and Reasoning Framework for Cyber-Physical System. *Sensors*, Multidisciplinary Digital Publishing Institute, v. 17, n. 2, p. 403, 2017.

YANG, Z. et al. A dynamic web services composition algorithm based on the combination of ant colony algorithm and genetic algorithm. *Journal of Computational Information Systems*, v. 6, n. 8, p. 2617–2622, 2010.

YU, P. et al. Application mobility in pervasive computing: A survey. *Pervasive and Mobile Computing*, Elsevier, v. 9, n. 1, p. 2–17, 2013.

ZENG, L. et al. Quality driven web services composition. In: ACM. *Proceedings of the 12th international conference on World Wide Web*. [S.l.], 2003. p. 411–421.