September 2019

A Programming Model for Internetworked Things

Hao Jiang
The University of Western Ontario

Supervisor
Kontogiannis, Konstantinos
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Master of Science

Abstract

The Internet of Things (IoT) emerges as a system paradigm that encompasses a wide spectrum of technologies and protocols related to Internetworking, services computing, and device connectivity. The main objective is to achieve an environment whereby physical devices and everyday objects can communicate and interact with each other over the Internet. The Internet of Things is heralded as the next generation Internet, and introduces significant opportunities for novel applications in many different domains. What is missing right now is a programming model whereby developers as well as end-users can specify any addressable resource at a higher level of abstraction, and consequently utilize these abstractions to define compositions, or scripts, among resources that allow for the customizable exchange of data among the resources, the evaluation of conditions based on exchanged data, and the enactment of actions provided that specific events occur and specific conditions are met.

In this thesis, we investigate the problem of designing a programming model for composing resources or "things", with applications in the IoT domain, and implement a proof of concept prototype in order to evaluate the feasibility of such a programming model. More specifically, this thesis attacks the problem of devising an IoT programming model from three directions. The first direction is the design of a Meta-Object Facility meta-model, that allows for URI addressable entities to be specified at a higher level of abstraction. Such a meta-model can be considered as domain specific language that allows for the denotation of *types* of entities (resources) in different application domains. The second direction is the design of an actionable composition model for IoT devices and other URI addressable resources. In this respect, this thesis investigates the use of the Event-Condition-Action paradigm as a basis of a runtime environment whereby action models can be enacted once events occur and condition models are fulfilled. A resource composition model also allows for resources to exchange data through input and output plugs implemented on top of the OPC UA publish subscribe middleware. The third direction deals with the design of a layered architecture that allows for scalability, robustness, security, and fault tolerance to be considered. Such an architecture takes advantage of a publish subscribe framework and utilizes proxies and facades to efficiently connect with third party components.

Keywords: Internet of Things, Internet of Everything, Internetworked things, programming model, middleware, Event-Condition-Action, goal modeling

Acknowledgements

First of all, I would like to thank Dr. Konstantinos Kontogiannis, my supervisor, who has provided tremendous advice and guidance during the study of my M.Sc. His insight helped my work go in the right direction, and he taught me the principles of research work. I am extremely grateful for this. And I am so honored to continue my PhD study with him.

I have had the pleasure of being a member of the Software Engineering Group at Western University. In particular, I would like to thank my fellow lab mates, Marios Stavros Grigoriou, Konstantinos Tsiounis and Sanjay Ghanathey for the useful discussions during our group meetings and tea-breaks. I am also grateful to my friends, Zhongwen Zhang, Jingyi Ren, Yifang Liu for emotional support and valuable opinions.

My research was partly done at IBM Toronto Lab during the summer, 2018. It was indeed an opportunity for me to participate in the internship at IBM.

Finally, I would like to thank my parents for their support and encouragement throughout the master study and life, which made it all possible.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

In this chapter, we first introduce the concept of Internet of Things (IoT) and three key questions it brings about. We then discuss the problem statement, thesis contributions and thesis outline.

1.1 Internet of Things and Resources

During the past few decades we have witnessed the extraordinary success and usefulness of the Internet. The World Wide Web enables people to access global information and services, which include searching for information, shopping online, engaging in social networking and so on. However, the Internet is not only about the Web. It is also a suite of protocols that allow for a wide range of devices to use the Internet's global connectivity in order to engage in a variety of interactions. These interactions can range from simple exchange of data, to services computing. More recently, the emergence of resource oriented computing and the connectivity offered to a wide spectrum of inter-networked devices has given rise to what is referred to as the Internet of Things, or IoT. The Internet of Things is considered as the next generation of Internet use, which extends the Internet connectivity from software agents to physical devices and everyday connected objects.

The term Internet of Things was first coined by Kevin Ashton in 1999 in the context of supply chain management [26]. With the rapid development of technology, the definition of IoT has been more inclusive covering a variety of applications. According to [54], the Internet of Things is a system of physical objects that can be discovered or interacted with, by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider Internet. The

Internet of Things concept has been since extended to include what is referred to as the Internet of Everything (IoE). According to [63], the IoE encompasses people, data, process and things. Its major objective is to provide a platform that allows for a) the massive extraction of information from various sources (e.g., sensors) and b) utilization of intelligent layers to automate machine to machine and machine to people processes [9].

According to Cisco, 500 billion devices are expected to be connected to the Internet by 2030 [10]. Cisco also predicts that the global Internet of Things market will be $14.4 trillion by 2022. IoT has great potential in a wide range of application domains. IoT devices can be applied in home automation, which includes lighting and temperature control, intrusion detection, energy optimization, etc. IoT can also be applied to other areas such as healthcare by enabling remote health monitoring and emergency notification, transportation, manufacturing and agriculture.

However, the Internet of Things, and consequently the Internet of Everything, have introduced a wealth of new problems and challenges to address, ranging from modeling and programming issues, all the way to infrastructure, scalability, and security issues.

In this thesis, we investigate techniques which focus on modeling and composing resources as well as associating actions that these resources can perform as they exchange data and interact in an inter-networked IoT/IoE environment. More specifically, this thesis aims to address three key questions related to programming and composing resources. The first question deals with how we can model inter-networked resources (i.e., IoT resources) at a higher level of abstraction, and how we can utilize semantic web technologies for instantiating such abstract models of resources to concrete URI addressable resources that are accessible over the Internet communication protocols. The second question deals with the problem of providing a model of composing resources so that these can not only exchange data, but also enact actions as the result of such interactions and provided that certain conditions are met. In this respect, we aim for devising an initial programming model which can be used by application developers as well as IoT end-users to define plug and play applications that are based on the interaction of inter-networked resources.

The third question deals with the problem of what is an appropriate and scalable architecture that can be used to deploy such a system in a massive scale. In this respect, we investigate a layered architecture that utilizes publish-subscribe infrastructure middleware components. Such an architecture can be easily ported

to a distributed publish-subscribe system that utilizes distributed brokers.

In this respect, this thesis proposes a model whereby intelligence can be introduced on the interaction among "things" in a resource-oriented environment. This can provide a step towards achieving the Internet of Everything goal.

1.2 Problem Statement

The available Web services are constantly increasing in number and variety as the Internet expands. The last few years we have also witnessed the rapid increase of inter-networked IoT devices and resources consuming and producing data which come from various sources such as repositories, Web services or IoT devices. In such an IoT environment, data may come from either Web services or IoT devices. In such an environment, this thesis aims to address three major questions related to IoT system compositionality and programmability.

The first question to be addressed in this thesis is how to achieve a level of ease of programmability in an environment where resources or "things" exchange data, enact actions, and generate events. This question encompasses the problem of devising a meta-modeling framework to denote and compose resources at a higher level of abstraction and to provide a unified interface to access resources and their associated data representations, given that these may come from various information sources. Such a modeling and resource composition abstraction is the essential first step for facilitating a programming model for IoT application development. This thesis considers models of resources at two levels of abstraction: a) abstract resources and b) concrete resources. The abstract resources, which we refer to as Abstract Domain Resources (ADRs), denote conceptual high level representations or categories of actural resources that are addressable entities by Internet protocols such as HTTP and URI. The concrete resources which we refer to as Domain Resources (DRs), are instantiations of models of abstract resources. This instantiation is achieved through the use of semantic web technologies such as RDF/S and model transformers which generate concrete resource instances from abstract ones. The instantiation process takes into account a number of factors for choosing a candidate resource to instantiate an abstract one. Such factors may include cost, latency and, reliability of a resource so that the instantiation process can select the optimal combination of candidate resources to be used when instantiating an abstract resource.

The second question deals with devising a programming abstraction for a run-

time model for IoT devices and inter-networked resources composition and interoperation. More specifically, the problem here is that given a collection of data shared among resources, how to apply a feasible actionable composition model. In this respect, we consider an Event-Condition-Action type of system. Here, we have first to identify how to evaluate local conditions when a new event happens. One can employ a rule-based system which utilizes of a fact base and a rule base. Whenever an event occurs, the system checks all relevant rules to determine any consequent actions. The problem here is that developing a suitable rule framework is of substantial work and suffers from what is known plan-fullness (i.e. we are never sure we have a fully complete set of rules). The second problem is developing an action model whereby upon the availability of events and the satisfaction of conditions, resources can invoke or enact actions. For this thesis we utilize an existing action model that has been developed as an earlier thesis [27] which allows for actions to be specified as collections of tasks that aim to achieve an agent's goal.

The third question deals with the architectural choices that need be considered for designing such a system, given that scalability, robustness, security, and fault tolerance are important non-functional requirements to consider. For this thesis, we consider the use of layered, and event-driven architectures that utilize a publish-subscribe paradigm for data exchange and implicit invocation of services (i.e. evaluation of conditions, invocation of actions). In this respect, the middleware serves as a data exchange and invocation abstraction bridge between "things" and applications.

1.3 Thesis Contributions

In order to tackle the problems mentioned above, we first propose a meta-model to abstract IoT resources. Here the use of the metamodel serves as means to create a specification of Abstract Domain Resources in various domains (e.g. banking, insurance, healthcare) and for which serve as templates. These templates are used for describing a user's view and usage of a resource in the IoT domain. By means of semantic technologies and resource selection algorithms, templates are instantiated later and point to specific information sources. Ontologies are employed in an instantiation step to provide semantics to the model elements and create mappings between terms in different sources. We formulate the resource selection problem as a knapsack problem and implement a Dynamic Programming type of algorithms

to solve this problem in an efficient manner.

The second contribution of the thesis is modeling conditions and actions utilizing goal model frameworks [80]. More specifically, we employ goal modeling for specifying and evaluating condition models as well as action models for compiling action plans when the aforementioned condition models are satisfied. For the conditions, goals in a goal model represent conditions or states which are attached to the individual evaluators. The action model represents tasks or atomic actions the system has to perform in order to achieve the top level task (i.e. the root of the action model). Reasoners developed as part of previous theses are used first to evaluate conditions (i.e. goal models) using a fuzzy reasoner, and second to generate sequences of tasks that satisfy the agent's goals.

The third contribution of the thesis is to propose a scalable architecture that is based on the implicit invocation architectural style, and the use of publish-subscribe middleware technologies. To evaluate the feasibility of our architecture, this thesis proposes a prototype system that is based on semantic web technologies and the OPC UA middleware environment [60]. The OPC UA framework supports event-based communication between different software components. We employ OPC UA to integrate the server and client components of the running system. Server components register in event channels, while client components subscribe to channels and get notified when a new event is posted. When the condition is satisfied, the runtime of the prototype system executes corresponding actions as specified by the analysis of the action models. As part of a related project, the proposed system was also linked with the PADRES middleware [56] and the Node-RED framework [15].

1.4 Thesis Outline

The remainder of this thesis is organized as follows.

In Chapter 2, we provide background and related work information about the programming model, middleware architecture, modeling framework, ontology development, edge computing, resource oriented computing and gap analysis.

Chapter 3 discusses the proposed system architecture, which includes a general overview of the system and descriptions for each component.

In Chapter 4, we present programming abstractions and the instantiation process, which consists of semantic technologies and resource selection algorithms.

In Chapter 5, we describe the modeling and reasoning for the condition and

action by utilizing goal models. We also introduce a mapper model which associate output ports and input ports across different resource models, condition models and action models, facilitating thus the creation of "scripts" or "IoT programs".

In Chapter 6, we present a prototype that is based on the programming model and utilize OPC UA middleware framework.

Finally, in Chapter 7 we conclude the thesis and provide pointers for the future research.

Chapter 2

Background and Related Work

This chapter describes the background of the proposed work. The foundation of our proposed approach relies on programming model, middleware architecture, modeling framework, ontology development, edge computing and resource oriented computing. The last section presents the gap analysis.

2.1 Programming Model

IoT applications can control and interact a wide variety of devices. For example, using IoT applications, people can not only control their home appliances remotely from their smartphones, but also automate some everyday tasks. One way this automation can be achieved is by predefined rules, where users specify events of interest as well as corresponding actions to be taken whenever these events occur. Under certain circumstances, the execution of an action also requires the fulfillment of specific conditions. Such an event-driven architecture consists of event producers that generate the events, and event consumers that listen for the events and act upon receiving these events.

Such an event-driven architecture can be implemented using a publish subscribe style as depicted in Figure 2.1 [19]. In a publish/subscribe model, any event published to a channel is immediately received by all of the subscribers of this channel. In such an architecture, event producers and consumer are decoupled. Hence, the event-driven architecture is highly scalable and distributed. Based on the complexity of event processing required, event-driven architectures can be further divided into two programming paradigms: the Trigger-Action paradigm and the Event-Condition-Action paradigm. One concrete example of event-driven pro-

gramming model is Node-RED [15].

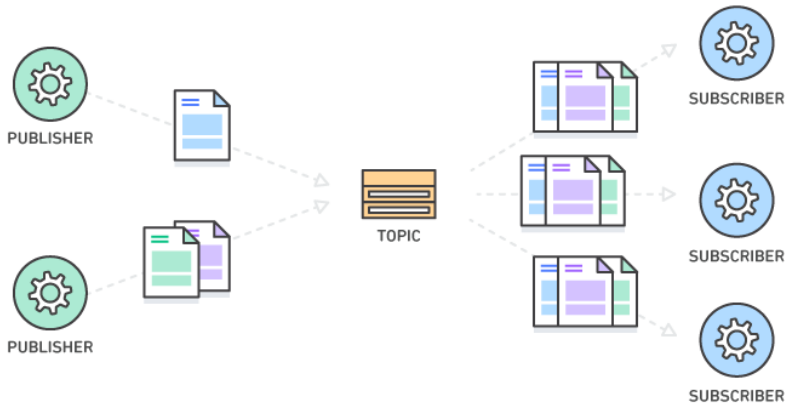


Figure 2.1: Publish Subscribe Model

2.1.1 Trigger-Action Programming

A concrete example of the trigger-action paradigm is the IFTTT service [8], which stands for “if this, then that”. For example, if a user’s Facebook profile picture changes, then such a service can update the user’s Twitter profile picture to match. Such if-then rules are easy to denote and cover many real-life scenarios. In [50], Ghiani et al. present a trigger-action rule editor that provides the possibility to create more flexible rules than IFTTT. Joëlle Coutaz and James L. Crowley propose AppsGate [45], an end-user development environment designed to empower people with tools to monitor and control their home. Corcella et al. [44] present a visual trigger-action tool for personalizing user’s IoT context-dependent applications, the users’ feedback is encouraging and promising. Besides simple “one trigger, one action” rules, Ur et al. [78] conduct three studies which prove that trigger-action programming with multiple triggers and multiple actions can be a practical approach to smart home programming.

Trigger-action programming also introduce many drawbacks. Many trigger-action programming interfaces lack feedback during rule creation [50]. Chandrakana Nandi and Michael D. Ernst [65] caution that even if the action block of a rule is implemented correctly, inadequate triggers can lead to too few firings of the rule. Brackenbury et al. [33] identify ten different categories of bugs that might arise in trigger-action programming, such as Priority Conflict, Missing Reversal, Infinite Loop, etc. Ur et al. [78] note that they are unable to evaluate the rela-

tive strengths and weaknesses of trigger-action programming due to insufficient control experiments.

2.1.2 Event-Condition-Action Pattern

Another programming paradigm of event-driven architecture is the Event-Condition-Action (ECA) pattern, which originates in database systems for efficiently responding to sources of incoming events or data [46]. ECA rules basically conform to the following form:

> "When some events occur, and
> If some conditions are true,
> Then perform some actions."

It is obvious that such rule-based systems are easier to model and program, than systems built on general purpose programming languages, and are powerful enough for many IoT applications. Actually, there are already a few commercial platforms which try to strike a balance between expressiveness and simplicity. In [21], users can write event-condition-actions rules for IoT devices via a smartphone app. Home Assistant [7] is an open source platform written in Python, in which automation programs are made up of event-condition-action rules. It can also attach delays to event handlers and to actions. A critical part of ECA rules is semantics, which influence both programmability and expressivity. Newcomb et al. [66] present the Internet of Things Automation (IOTA) calculus, which models an ECA language formalism with abstractions constructs related to time, state, and device aggregation, as well as ECA syntax and precise semantics.

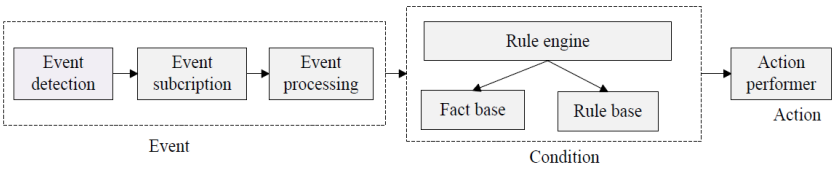


Figure 2.2: Event-Condition-Action based Platform

Cano et al. [35] focus on the safety and security issues of ECA rules in IoT environments. They propose an extension of ECA semantics by control theory and validate it with a case study. From a broader view, Bhandari et al. [31] propose an ECA framework, which includes four layers, namely device layer, service layer, ECA platform and event based application. The ECA platform is illustrated in Figure 2.2. As is shown, the event part is divided into event detection, subscription

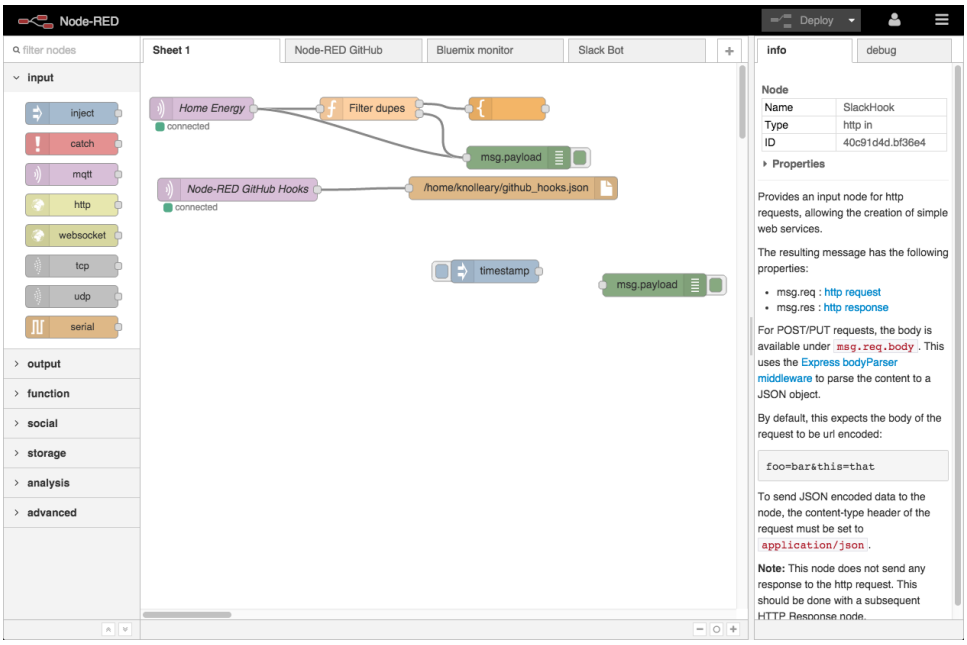


Figure 2.3: Node-RED Editor

and processing. The condition part handles the condition evaluation and it adopts a rule based system, which consists of a fact base and a rule base. The action part includes an action engine that is used to execute actions after condition is evaluated to true.

2.1.3 Node-RED

Node-RED is a programming tool for wiring together hardware devices, APIs and online services in new and interesting ways [15]. It was originally developed by IBM Emerging Technology Services and now is a JS Foundation project. Node-RED is implemented in JavaScript using the Node.js framework, which provides a visual browser-based flow editor. The developers can either drag, drop and wire up the nodes in the editor, or import JavaScript code in order to create applications.

The editor window of Node-RED has three main components as depicted in Figure 2.3:

1) Palette. The palette contains all the nodes that are available to use. The nodes are classified as several categories, such as input, output, function, etc. Typically, a JavaScript file describes the node's functionality, and an HTML file defines its properties, edit dialog and help text.

2) Workspace. The main workspace is where flows are developed by dragging

nodes from the palette and wiring them together. The wiring is achieved by connecting the output port of one node with the input port of the other node.

3) Sidebar. The sidebar provides a number of useful tools within the editor. For example, Information panel shows the properties about the current selected node. Debug panel displays log messages from the runtime. Other tools include Configuration Nodes and Context Data.

The Node-RED runtime is built on the Node.js event API, taking full advantage of its event-driven, non-blocking model. This enables the lightweight runtime to be easily deployed in the edge network as well as the cloud. Additionally, Node-RED offers powerful build-in nodes (e.g. HTTP and MQTT), which hide the complexity of interacting with the real word. In this way, the developers can focus on the application development, instead of on the programming details. These characteristics make Node-RED an ideal tool to create applications, especially applications that have an event-driven feature such as IoT applications.

The differences between Node-RED and the proposed framework are summarized in two main points. The first point is that the approach proposed in this thesis allows for the definition of arbitrary types of resources that can be instantiated at run-time by actual resources depending the system's operational context and user profile, while the Node-RED nodes have to be determined at specification time. The second point is that Node-RED is based on JavaScript technologies while the proposed approach allows for integration with any third party language and system. However, as part of a related project we have created a transformer that takes Node-Red specifications and generates a Mapper model (see Section 5.1).

2.2 Middleware Architecture

Due to the device and network heterogeneity, IoT application development is a very challenging task. Middleware addresses this problem by decoupling the applications from the underlying physical devices. It serves as the middle layer between the hardware and application layer. In this way, application developers can focus more on the development, instead of on how to interact with diverse physical devices.

Middleware development has been an active area of research in the IoT domain over the past few decades. Middleware is designed based on different architectural

styles. According to their unique features (design techniques, level of programming abstractions, infrastructure scale, etc.), each architecture can be classified differently. Based on design approaches taken in these middleware frameworks, in this section we discuss three categories of middleware architectures, namely the event based, service oriented and sematic oriented.

2.2.1 Event Based Middleware

Generally, an event-based architecture can also be viewed as an architecture following the publish subscribe, asynchronous, many-to-many communication model for distributed systems [48]. In this kind of middleware, all components interact with each other via events, where publishers also produce events and subscribers consume these events. Subscribers describe certain kinds of events that they are interested in and get notified when publishers post such events.

Publish subscribe systems can be divided into two forms: topic-based and content-based. The major distinction lies in how event subscribers express their interest in events. In topic-based systems, subscribers specify their interests in a topic (channel or subject) and receive all events published on this topic. These systems are easier to implement since they can adopt a group communication mechanism like IP multicast. However, they are inflexible as subscribers may need to filter events which come from general topics. In content-based systems, subscribers express their interests using event attributes. A subscription is often expressed in a subscription language that specifies a filter expression over events [72]. This form of publish subscribe achieves stronger expressiveness at the cost of increased overhead.

Figure 2.4 illustrates a distributed implementation of a publish/subscribe system [73]. Such implementation consists of two components, namely event clients and event brokers. Event clients can be publishers or subscribers and use the services provided by the middleware. The event brokers comprise the actual middleware which accept subscriptions and then distribute events from publishers to all registered subscribers.

There are several advantages of the event-based architectures. Firstly, it decouples space and time. Publishers and subscribers do not need to know about each other or run at the same time. Secondly, the event-based pattern can achieve greater scalability than a traditional client-server approach, because the loose coupling removes dependencies between clients. Lastly, it is capable of filtering events

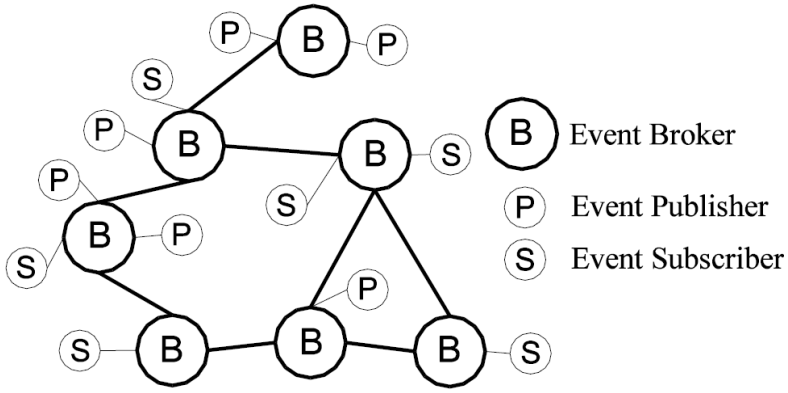


Figure 2.4: A Distribute Implementation of Publish Subscribe System

based on their attributes.

HERMES [72] is an event-based middleware architecture created for large-scale distributed systems. HERMES has type-based and attribute-based events, and encompasses two routing algorithms: type-based and type- and attribute-based routing. The former only supports subscriptions depending on the event type of event publications, while the latter extends the type-based routing with content-based filtering on event attributes in publications [72]. Another event-based architectures is Data Distribution Service (DDS) developed by Object Management Group (OMG) [69].

2.2.2 Service Oriented Middleware

Just like other software, IoT middleware solutions often follow the Service Oriented Architecture (SOA) approach. SOA aims to decompose complex and monolithic system into simpler independent components, which provide services through accessible interfaces. The SOA approach also allows for software and hardware reuse, because it does not impose a specific technology for the implementation of services [70].

In this context, OpenIoT is a popular open source cloud solution for the IoT domain, which also comprises a service-oriented middleware for collecting data from any sensor [76]. OpenIoT supports flexible configuration and deployment of algorithms for collection, and filtering data streams stemming from the internet-connected physical objects, while at the same time generating and processing events. The implementation of OpenIoT extends the Global Sensor Networks (GSN) sensor middleware and is called X-GSN [23].

2.2.3 Semantic Oriented Middleware

Semantic oriented middleware focuses on solving the interoperability problem in IoT, that is, different types of devices interact using different communication protocols and data models. Ontologies and semantic web technologies address this problem by providing a unified interface for data access.

Semantic web was originally designed to make information understandable by machines. In this way, interoperation among machines and integration of information is enhanced. Additionally, semantic web can provide context-awareness to applications, in which the search space for automatic service discovery and composition is reduced [52]. Last but not least, semantic web technologies facilitate reasoning of actionable knowledge from various heterogeneous data sources.

SemIoT is a middleware platform which employs semantic web technologies, existing ontologies and architectural style REST [58]. SemIoT applies the OSGi architecture [24] to allocate an independent service to a different type of devices. It also applies semantic web technologies such as RDF, OWL and SPARQL to achieve semantic interoperability. As for ontologies, SemIoT extends SSN ontology [42] to model and annotate devices' data.

2.3 Modeling Framework

The Meta Object Facility (MOF) [14] has emerged as the defacto standard for model-driven engineering of the Object Management Group (OMG). Its purpose is to provide the formal definition of modeling languages (including UML). Now MOF is one of the foundations of model-driven architecture (MDA). The MOF specification defines a hierarchy of four-layer models, and is designed to support extensions for more sophisticated metamodeling. The four layers are as follows:

- **Layer 1:** The M3 layer, or meta-meta model layer, is the highest abstraction layer. It is the top layer of the hierarchy and used by MOF to build M2 layer models, i.e. metamodels (such as UML). The meta-meta models at this layer are essentially the definitions of the languages used in the metamodel specification. This layer is self-referential, which means that the meta-meta models constructs in this layer can also be used to describe themselves.

- **Layer 2:** M2 layer or metamodel layer. A metamodel is an instance of a meta-meta model. The primary responsibility of this layer is to define a language for specifying metamodels. A typical example of this layer is UML.

- **Layer 3:** M1 layer or domain model layer. This layer describes a domain model that is used as a "schema" for defining entities in a specific application domain, such as banking, insurance and healthcare, to name a few. Such domain models are instantiated to form concrete information models at M0 layer.

- **Layer 4:** The M0 layer or information layer. This layer contains the runtime instances of data elements conforming to corresponding to a domain model, they are instances of. That is an M0 model stems from its corresponding domain model at M1 layer.

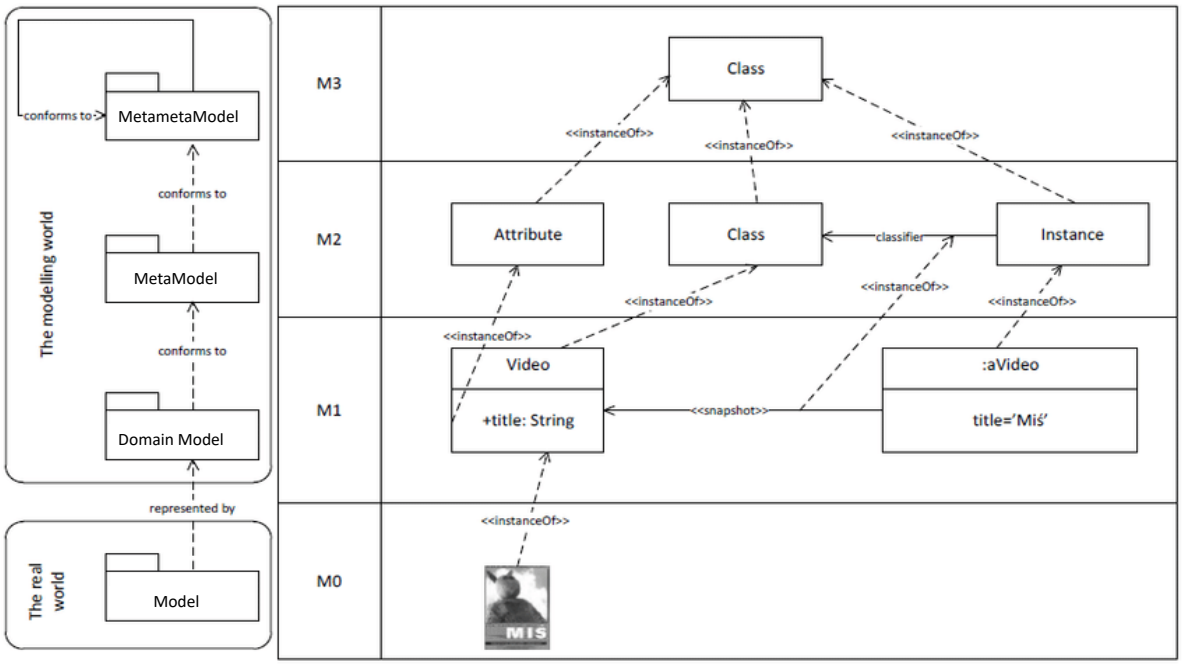


Figure 2.5: MOF Hierarchy

Figure 2.5 illustrates the MOF hierarchy [68]. The value of MOF lies in that it provides a straightforward framework for mapping MOF models to implementations like Java Metadata Interface (JMI). In addition, the MOF allows models to be stored with standards such as XML Metadata Interchange (XMI), which can be transferred to another application and extended easily later on.

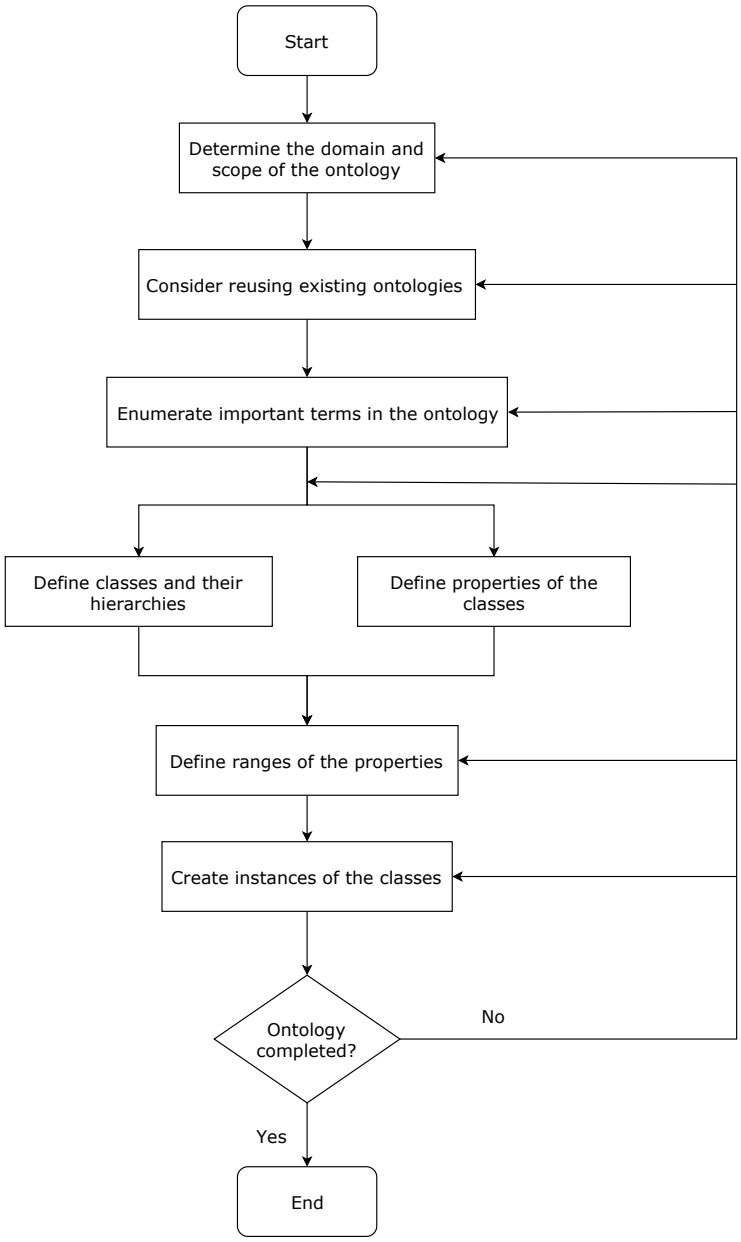


Figure 2.6: The ontology development process

2.4 Ontology Development

A widely quoted definition of an ontology is “a formal, explicit specification of a shared conceptualization” [53]. Here conceptualization refers to an abstract model of the way people think about things in the world. An explicit specification means that the concepts and relationships in the abstract model are given explicit names and definitions [79]. Formal means that there should not be any ambiguity about the specification, which is usually done by employing a logic-based language. In this respect, a common vocabulary for a certain domain can be established and ontologies can be shared and reused for other purposes.

Ontology generally characterizes different kinds of concepts and their relationships in a domain of interest. These concepts are called classes in the ontology, and they are usually the focus of an ontology. Just as in an object-oriented language, a class can have multiple subclasses which represent concepts that are more specific than the superclass. Various features and attributes of each concept are modeled as properties. An ontology together with a set of concrete instances (also called individuals) of the class constitutes a knowledge base. RDFS (Resource Description Framework Schema) [34] is a lightweight ontology language which allow us to define classes, properties as well as their hierarchies. OWL (Web Ontology Language) [62] is an extension to RDFS which provides much more powerful expressiveness and reasoning capability.

Ontology design is not an easy task. Noy and McGuiness [67] propose three fundamental rules in ontology design. Despite what approach is used for the design of ontology, their advice is helpful for making design decisions:

1) There is no single correct way to model a domain – there are always viable alternatives. The best solution almost always depends on the application that a modeler has in mind and the extensions that a modeler anticipates.

2) Ontology development is an iterative process.

3) Concepts in the ontology should be close to objects (physical or logical) and should stem from the domain of interest. These are most likely to be nouns (objects) or verbs (relationships) in sentences that describe the domain. [67]

As there is no a single unified and formal definition of the ontology, there are plenty of methodologies for building an ontology. Figure 2.6 provides a flowchart of the whole ontology development process as this is proposed by Noy and McGuiness [67].

2.4.1 Edge Computing

While IoT has great potential in various industry-specific and cross-industry use cases, it also brings about several issues, such as data storage, data processing, data analytics, etc. In the last few years, the integration of the IoT with cloud computing has overcome the computation and storage limitations [47]. Nevertheless, this also leads to an increase of latency in communications, especially for IoT applications in which devices usually span a large geographical area. To fulfill this gap, edge computing is introduced to provide computing and storage services at the edge of the network, instead of sending all the data to the cloud. Edge can perform computing offloading, data storage, caching and processing, as well as distribute request and delivery service from cloud to user [75]. Figure 2.7 [75] illustrates the edge computing paradigm. As is shown, "things" are not only data producers, but also data consumers. At the edge, "things" can not only collect data but also perform computing tasks. Therefore, the two-way computing stream between the edge and the cloud is achieved.

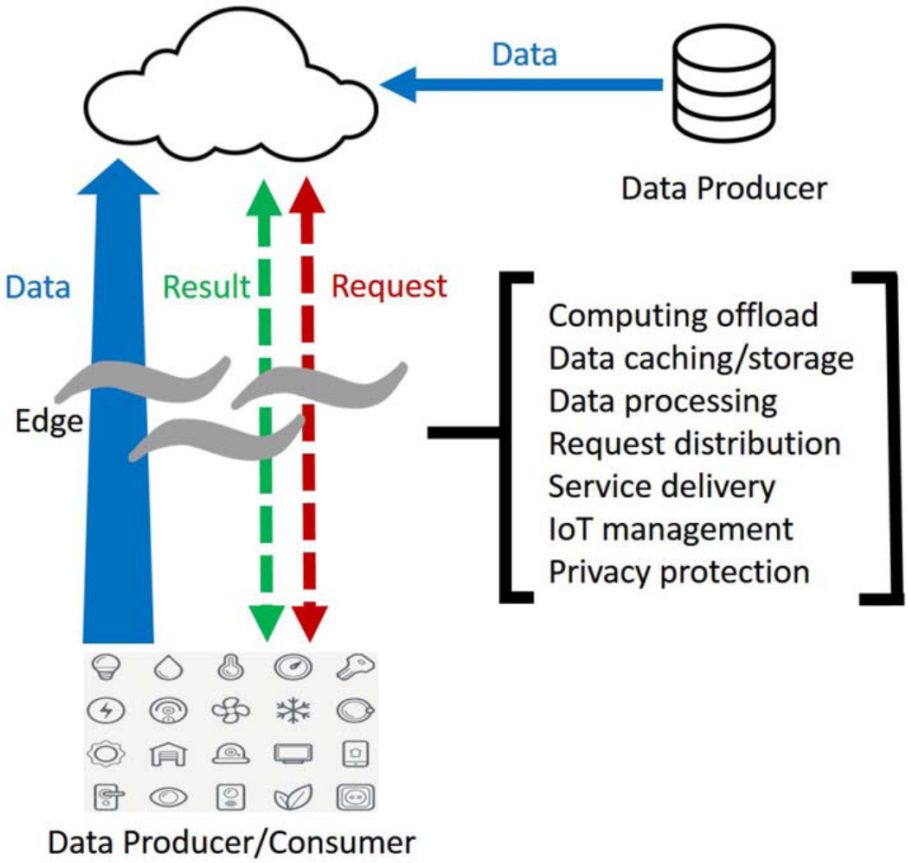


Figure 2.7: Edge Computing Paradigm

Generally, it is beneficial to support IoT applications by combining the high computation capacity and large storage of cloud computing with the advantages of edge computing. Specifically, edge computing-based IoT has following advantages [84]:

- `transmission`: By offloading the data processing and storage to end users, the latency, bandwidth and energy consumption are significantly reduced.
- `storage`: Edge computing-based storage is distributed to different edge nodes, which leverages load balancing and failure recovery technique to realize availability.
- `computation`: The computation task is also assigned to several edge nodes by utilizing the task scheduling scheme.

2.5 Resource Oriented Computing

REST (Representational State Transfer) was first introduced and defined in 2000 by Roy Fielding in his doctoral dissertation [49]. REST is a software architectural style for creating Web services by taking advantage of existing protocols. The REST architectural style is founded on a set of constraints. These include being stateless, having a client/server architecture, complying to a uniform interface, achieving cacheability, being a layered system and providing capabilities for offering code on demand. REST is not exclusively bound to a specific application layer protocol such as HTTP, but it is most commonly associated with it when we are talking about RESTful Web services. The central idea of REST revolves around the notion of a resource which is any component of an application that is worth being uniquely identified by a URI and linked to, utilizing an application layer protocol (e.g. HTTP) [55]. In this respect, resources can include physical devices (e.g., a temperature sensor), abstract concepts such as Web resources, but also dynamic concepts such as server-side states. When designing a RESTful API, there are five issues we need to address:

- **Resource Identification.** It is a common practice to utilize Uniform Resource Identifiers (URIs) to identify resources on the Web. Representations of resources also contain links to other resources. Clients of RESTful APIs can follow the links to find resources to interact with, just like browsing Web pages.

- **Resource Representation.** In order to represent data objects and attributes in a resource, we also need to agree on resource representation formats. For machine-oriented services, JavaScript Object Notation (JSON) and Extensible Markup Language (XML) have gained widespread support across server and client platforms. Besides, HTML representation increases the readability of resources for humans.

- **Uniform Interface.** In REST, interacting with resources and retrieving their representations are achieved through a uniform interface, which decouples clients from servers. On the Web, uniform interface is defined by HTTP, which provides four main methods to interact with resources: GET, PUT, POST and DELETE. GET is used to retrieve the representation of a resource. PUT updates the state of an existing resource or creates a resource if it does not exist. POST creates a new resource while DELETE removes a resource. Finally, the status of the response is represented by standard status codes in the header of the HTTP message.

- **Stateless Interactions.** Stateless means that interactions store no client context on the server between requests. This requires that when the client makes a request, it includes all the information for the server to fulfill that request. HTTP is a stateless protocol in that it has no knowledge beyond the request/response interaction. This helps increase the RESTful API's reliability by having all the data necessary to make the request. In addition, a stateless application is easier to distribute across load-balanced servers and cache.

The flexibility of REST allows for building the applications that meet both developers' and users' needs. Moreover, because of the decentralization and massive scalability inherent in the RESTful architecture, it is extremely useful in the IoT domain. There are millions of available resources and clients, with millions of concurrent interactions with one service provider. In such scenarios, RESTful architecture scales better than RPC-based client server type of architectures.

2.6 Gap Analysis

Over the last decade, a number of researchers and practitioners have investigated programming frameworks for IoT which support application development. As dis-

cussed in Section 2.1, those frameworks mostly employ an event-driven architecture which can be divided into two programming paradigms: the Trigger-Action paradigm and the Event-Condition-Action paradigm. The Trigger-Action paradigm is proved to be easy to use for the end-users in real-life scenarios like smart home. In such simple scenarios, one or two triggers are enough. However, in other complex scenarios which require multiple triggers, the semantics of composing them are complicated and confusing for the end-users. In addition, the Trigger-Action paradigm does not contain temporal information in triggers. Generally, the trigger could be an event, a condition, or some combination. If there are both event and condition involved in a rule, they do not compose well. For example, the exact moment someone shuts down the computer is unlikely to be the exact moment the computer is completely off.

Although the programming models based on the Event-Condition-Action paradigm do not have aforementioned drawbacks, they pose other limitations. First of all, most proposed frameworks use direct sensors (e.g., "temperature is 25 degrees" or "motion is detected") for event detection. They are simple to implement but constrained in terms of data acquisition. Ideally, data can come from either IoT devices and appliances (e.g., sensors) or inter-networked resources (e.g., Web resources). Secondly, to the best of our knowledge, all proposed ECA based programming frameworks utilize a rule-based approach, which consists of a rule engine, a fact base and a rule base for condition evaluation. The development and verification of a rule-based system are time consuming. Also, any changes to the rule base or fact base may introduce potential errors. Lastly, the proposed programming framework cannot express inherently vague concepts in conditions. For example, a condition may be "the weather is too hot", where "hot" is ambiguous and person-dependent. Therefore, there is a need for studying how to interpret concrete readings from sensors for condition evaluation.

To tackle the problems mentioned above, we develop a system which is built on the Event-Condition-Action paradigm. As depicted in Figure 3.1, the lowest layer is modeling. In our approach, the modeled resources not only include physical IoT devices (e.g., sensors), but also inter-networked resources (e.g., Web resources or Web services). In this case, any resource which is uniquely identified by a URI can server as a data source. Furthermore, instead of modeling one atomic resource at a time, we consider resources at a higher level of abstraction. Such abstract resources serve as "templates", which extract reusable parts of a resource for a specific domain. The instantiation process is performed at runtime and generates

concrete resources for the composition. Secondly, we employ goal modeling rather than rule-based system for the condition evaluation. Specifically, goals in a goal model represent conditions or states which are attached to the individual evaluators. After reasoning, we can obtain a truth value for the root node of the model and determine any consequent actions that need to be initiated. Goal modeling is also employed for specifying action models and compiling action plans. Another advantage of goal modeling is that it support more complicated relationships between two goals. Lastly, we utilize fuzzy logic for evaluating condition models. Fuzzy logic allows for reasoning on vague concepts (e.g., "hotness") and is robust to tolerate imprecise readings. The runtime system is developed using a publish/subscribe middleware in consideration of scalability and other non-functional requirements.

Chapter 3

System Architecture

In this chapter, we present the architecture of the proposed system, which is graphically depicted in Figure 3.1. Generally, the whole system has three subsystems: the *Modeling subsystem*, the *Instantiation subsystem* and the *Runtime system*. We then present the component view of the system followed by the detail descriptions of each component. To illustrate the workflow of the activities and actions, we also provide the activity diagram of the runtime system and a corresponding workflow example.

3.1 System Overview

The architecture of the proposed system is structured across seven layers, as depicted in Figure 3.1. Each layer builds upon the functionality provided by the layer below, and exposes interfaces to the layer above. In addition, layers are independent of each other. Each layer has its own implementation and it can be replaced by a different implementation if necessary. For example, if a more scalable Pub/-Sub middleware becomes available, the system can employ it without major modification. Next, we will discuss the responsibilities of each layer of the proposed architecture, starting with the lowest one.

- **Modeling Subsystem.**
  - **Modeling Layer.** The lowest layer is the modeling layer which hosts all the components for a user to draft and edit models related to abstract domain resources (ADRs), models related to conditions, models related to actions and models related to the composition of the above entities. This

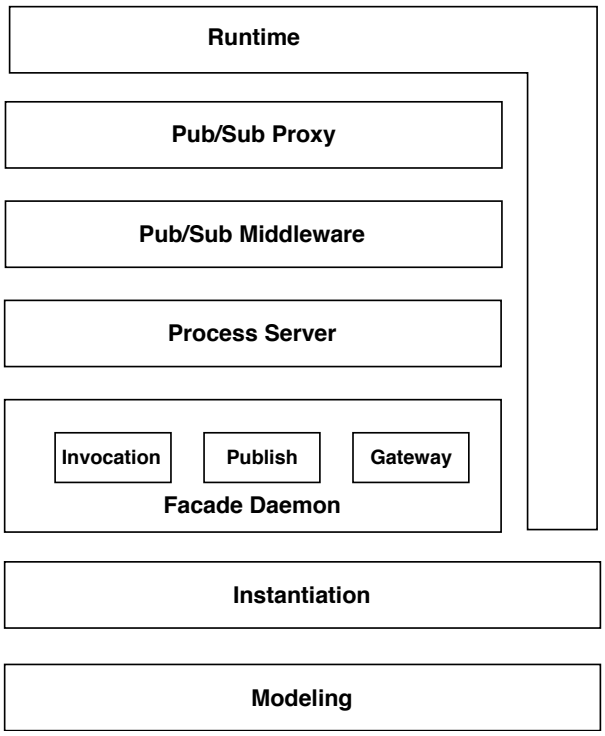


Figure 3.1: High Level Conceptual View of the System's Architecture

layer contains two components, namely the AbstractResourceCondition-Action Modeling component and the Composition Modeling component. In this layer, we model abstract domain resource, abstract condition, abstract action and composition.

- **Instantiation Subsystem.**

  - **Instantiation Layer.** This layer hosts components that instantiate abstract domain resource models to concrete ones (see Section 4.2). The instantiation process begins with the resource localization and then, with the help of domain ontology and a resource selection algorithm, a concrete resource is selected for instantiating an abstract domain resource.

- **Runtime Subsystem.**

  - **Facade Daemon Layer.** This layer is responsible for collecting data and handling responses. It provides an interface for the Process Server to transmit and receive data from an external medium (e.g. the Internet through RESTful Web services).

- **Process Server Layer.** The Process Server implements the functionality required by the runtime system. This functionality includes the sequencing and facilitation of data provision, condition evaluation and action evaluation.
- **Publish/Subscribe Middleware.** This layer represents a specific middleware employed in the runtime system. Specifically, we utilize the OPC UA system as a middleware framework for providing publish and subscribe services for the runtime system. The proposed system is also integrated with the PADRES middleware as part of a related project.
- **Publish/Subscribe Proxy Layer.** This layer acts as an intermediary between the Runtime and the Publish/Subscribe Middleware. It exposes Pub/SubProxyService as an interface to control the access to a specific Publish/Subscribe middleware (e.g. OPC UA, PADRES, etc.).
- **Runtime Layer.** This layer integrates services provided by the underlying layers and implements the prototype runtime system.

3.2 Component View

The component diagram of the proposed system is depicted in Figure 3.2 As is shown, there are three major subsystems in the whole system. The first subsystem is the *Modeling subsystem*, which provides services to model abstract resources, conditions, actions as well as compositions. The second subsystem is the *Instantiation subsystem*, which provides services to instantiate abstract models obtained from the Modeling subsystem and provides concrete models to the runtime environment. The third major subsystem is the *Runtime subsystem*. The Runtime subsystem employs an event driven architectural style. Specifically, it uses the Publish/Subscribe model to process events in the middleware.The detailed descriptions of the individual components in each module making the proposed architecture, are discussed in the next Section.

3.3 Component Descriptions

Here, we describe the functionality of each component in the proposed architecture along with the relationships between each component.

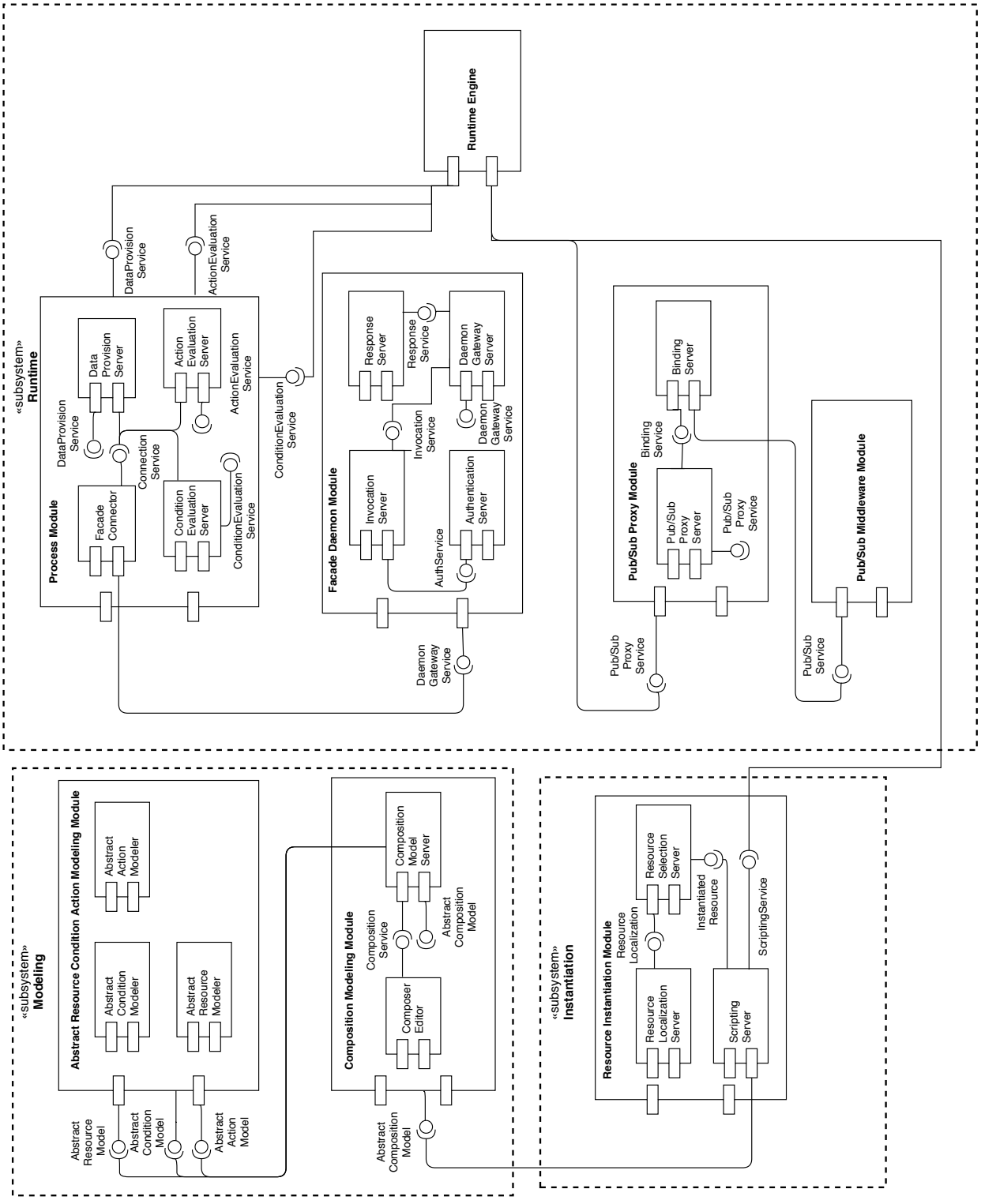


Figure 3.2: Component Diagram of System

3.3.1 Modeling Subsystem

Component	Descriptions
AbstractResourceModeler	This component provides services to read abstract domain resource specifications as well as initialize and load AbstractResourceModel(s) into the memory. It exposes the interface which is used by CompositionModelServer component.
AbstractConditionModeler	This component provides services to edit, read as well as initialize and load into the memory AbstractConditionModel(s). It exposes the interface which is used by CompositionModelServer component.
AbstractActionModeler	This component provides services to read abstract action specifications as well as to initialize and load into the memory AbstractActionModel(s). It exposes the interface which is used by CompositionModelServer component.

Table 3.1: AbstractResourceConditionAction Modeling (ARCAM) Module

Component	Descriptions
ComposerEditor	This component provides a textual editor service for modeling compositions. It exposes CompositionService as an interface.
CompositionModelServer	This component consumes CompositionService from ComposerEditor and abstract models generated from ARCA Modeling Module. CompositionModelServer initializes and loads AbstractCompositionModel into the memory. It exposes the interface which is used by ScriptingServer component.

Table 3.2: Composition Modeling (CM) Module

3.3.2 Instantiation Subsystem

Component	Descriptions
ResourceLocalizationServer	This component provides services to access the resource repository and retrieve all available resources for a particular domain based on user's preferences and context. It exposes ResourceLocalization service as its interface.
ResourceSelectionServer	This component consumes ResourceLocalization service and provides services to select the resource with the highest utility value with the help of Dynamic Programming algorithm. It generates InstantiatedResource as a result.
ScriptingServer	This component provides services to model elements in the AbstractCompositionModel so that a composition model can be created (see Section 5.1). It exposes ScriptingProvisionService as an interface.

Table 3.3: Resource Instantiation (RI) Module

3.3.3 Runtime Subsystem

Component	Descriptions
FacadeConnector	This component provides services to connect the required interface of DataProvisionServer, ConditionEvaluationServer and ActionEvaluationServer with the provided interfaces of Façade Daemon module (i.e. the DaemonGatewayServer).
DataProvisionServer	This component provides services to provision data for the runtime system. It consumes ConnectionService provided by FacadeConnector component and exposes DataProvisionService as an interface. Its aim is to decouple the process of provisioning data from an external sources from the evaluations of condition and action models.
ConditionEvaluationServer	This component provides services to evaluate condition models. It exposes ConditionEvaluationService as an interface.
ActionEvaluationServer	This component provides services to evaluate actions in an action model. It consumes ConnectionService provided by FacadeConnector component and exposes ActionEvaluationService as an interface.

Table 3.4: Process Module

Component	Descriptions
InvocationServer	This component performs HTTP requests to collect data from external sources. It consumes AuthenticationService (when it is provided) and exposes InvocationService as an interface.
ResponseHandler	This component handles the response of the HTTP request. It consumes DaemonGatewayService and exposes ResponseService as an interface.
AuthenticationServer	This component provides an access token which could be used in a HTTP request of InvocationService (optional), in order to authenticate a user or a session.
DaemonGatewayServer	This component provides an access point to the Pub/-Sub Proxy Module. It consumes Pub/SubProxyService and exposes DaemonGatewayService as an interface.

Table 3.5: Facade Daemon

Component	Descriptions
Pub/SubProxyServer	This component provides services to decouple the backend system (through its facade daemon) from the underlying middleware technology used.
BindingServer	This component provides services for binding of a specific underlying pub/sub framework (e.g. OPC UA, PADRES) used by the runtime system. It consumes AuthenticationService when it is provided and exposes the Pub/SubProxyService as an interface.

Table 3.6: Pub/Sub Proxy Module

3.4 System Workflow

After applying the modeling and the instantiation processes, we obtain concrete DomainResource models as well as concrete Condition and Action models. In order to build the sever and the client of the runtime system, we use the OPC UA publish/subscribe middleware framework. Specifically, the server creates folders in the middleware for DomainResoruce, Condition and Action. We also have four clients, namely Domain Resource Client, Daemon, Condition Client and Action Client, which subscribe to the corresponding folders and publish events to the middleware. The activity diagram of the runtime system is depicted in Figure 3.3.

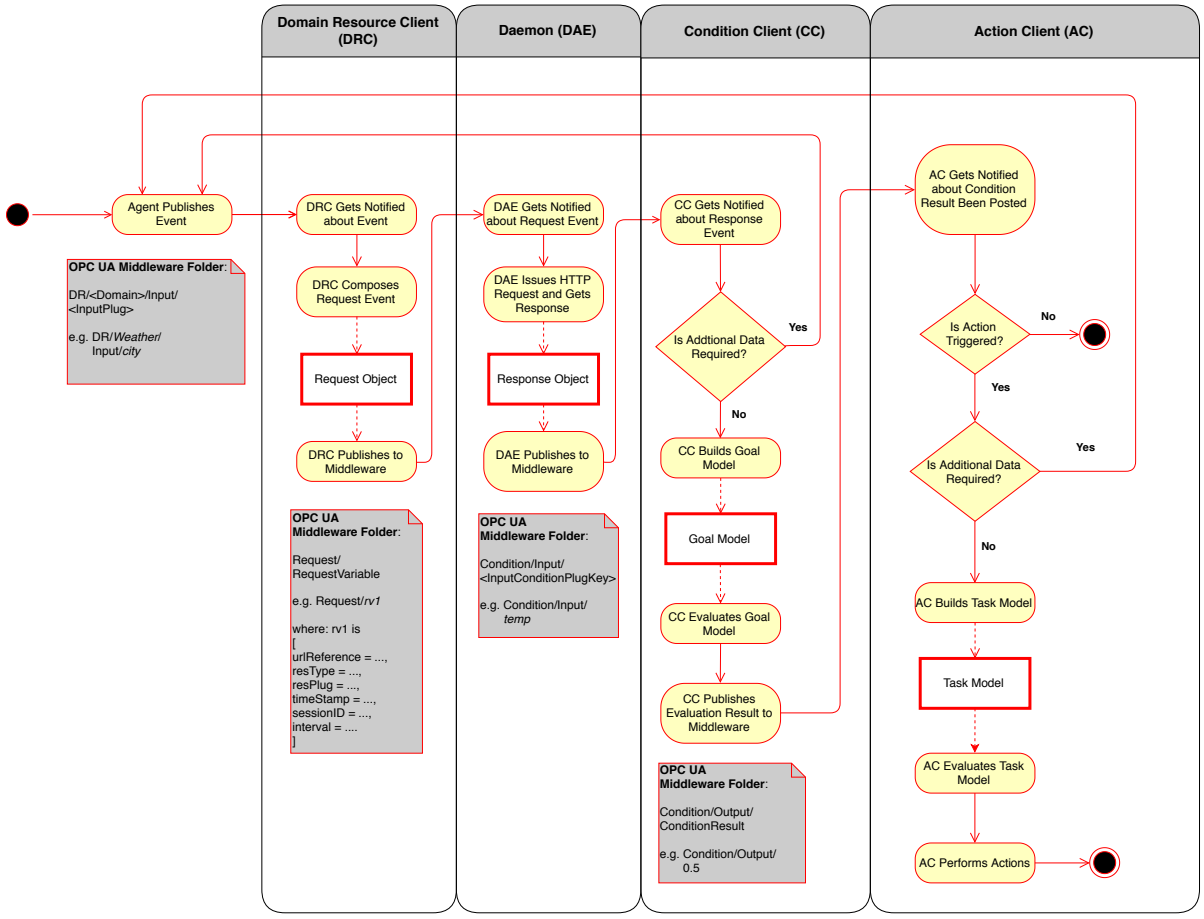


Figure 3.3: Activity Diagram of Runtime System

3.5 Workflow Example

In this section, a workflow example is given about the weather domain. Let us first assume that a concrete resource has the structure as depicted in Figure 3.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<iot:DomainResource xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:iot="http://org.eclipse.example/iot"
repositoryReference="repository/resourceRepository.ttl"
semanticReference="ontology/openweathermap.ttl"
label="current weather information" domain="Weather"
host="api.openweathermap.org"
urlReference="http://api.openweathermap.org/data/2.5/weather?q=${city}">
<resourceMetaModel>
<serialization xsi:type="iot:StringTemplateSerialization"/>
</resourceMetaModel>
<ActionInterface xsi:type="iot:Read" label="Find out weather information">
<output>
<outputMetaModel>
<outputPlug key="temp" label="Temperature in Fahrenheit"
   plugPath="$.main.temp"
   semanticsReference="http://www.openweathermap.com#Temp"/>
<outputPlug key="humidity" label="Humidity in percentage"
   plugPath="$.main.humidity"
   semanticsReference="http://www.openweathermap.com#Humidity"/>
<outputPlug key="pressure" label="AtmosphericPressure in hPa"
   plugPath="$.main.pressure"
   semanticsReference="http://www.openweathermap.com#Pressure"/>
</outputMetaModel>
</output>
<inputPlug key="city" label="Name of a city" type="string"/>
</ActionInterface>
</iot:DomainResource>
```

Reference to the resource repository containing metadata for each concrete resource considered in the system.

Reference to the ontology about the weather data.

Reference to the domain name.

Reference to the CRUD action interface.

Reference to the resource's output plugs.

Reference to the resource's input plug.

Figure 3.4: DomainResource Example for Weather Domain

Then, the steps that depict the system operation and correspond to the activity diagram in the Figure 3.3 are as follows:

1) An agent (e.g. an external actor, Condition Client or Action Client) publishes the event to the middleware folder (i.e. DR/***Weather***/Input/***city***).

2) The Domain Resource Client gets notified about the event, composes a request object and publishes to the middleware folder, (i.e. Request/Request-Variable).

3) Daemon gets notified about the request event, issues HTTP request and gets the response.

4) Daemon publishes the response data to the corresponding middleware folders, (i.e. Condition/Input/***temp***, Condition/Input/***humidity*** and Condition/Input/***pressure***).

5) Condition Client gets notified about the response event. If additional data is required, go back to step 1-3 (not shown for clarity in the corresponding Figure 3.3). If not, continue.

6) Condition Client obtains the reference to the goal model and builds it.

7) Condition Client evaluates the goal model and gets evaluation result.

8) Condition Client publishes the evaluation result to the middleware folder, i.e. Condition/Output/ConditionResult.

9) Action Client gets notified about the evaluation result been posted. If the condition is evaluated to false, terminate. If not, continue.

10) If additional data is required, go back to step 1-3 (not shown for clarity in the corresponding Figure 3.3). If not, continue.

11) Action Client obtains the reference to the task model and builds it.

12) Action Client evaluates the task model and compiles action plans.

13) Action Client performs actions.

14) System terminates.

Chapter 4

Resource Abstraction Metamodel and Instantiation

Service computing has emerged as a major paradigm for clients to access remote services. Such services can be invoked either through well defined message-oriented interfaces (as it is the case for classic message-oriented Web services), or through uniform resource identifiers as addressable resources (as it is the case for RESTful Web services). In either case, services are accessed through well defined end points. In this context, a major problem is for application developers and end users to choose the right end point (i.e. services) from many external ones. For instance, there are already over 1000 APIs under the *Mapping* category on ProgrammableWeb [18]. Of course there are thousands of APIs in other domains including the IoT domain. And this does not count even more resources which are available in IoT domain, such as sensors, actuators, processors, etc. On the other hand, developers may not be familiar with the details of service interfaces that the resources provide, which may include how to create an HTTP client to access service, parse the response data, etc. These issues guide us to consider an abstract model of resources (template), which provide abstractions for resource categories and service interfaces. Such abstractions are later instantiated with the help of the domain ontology and resource selection algorithm.

4.1 Resource Abstraction Metamodel

The Resource Abstraction Metamodel (RAMM) denotes the nature, capabilities and interfaces of RAMM resources. A RAMM resource serves as a "template", which

abstracts the reusable part of a resource. With the help of domain ontologies and a resource selection algorithm, the template can be instantiated in an automatic and flexible way. The RAMM is depicted in Figure 4.1.

Generally, the RAMM resources are parameterized with respect to some attributes, so that the templated resources can be configured and customized given the specific application scenario based on users' preferences and context. Such a template is more suitable in a dynamic environment where the users' preferences and context are constantly changing. In addition, this facilitates the sharing and reuse of templates, as well as the creation of template instances. The main element in RAMM is the *AbstractDomainResource*, which represents RAMM's resources for a particular domain. An AbstractDomainResource has references to the resource repository and the domain ontology which are useful in the instantiation process by discovering the proper resources. After instantiation, the "host" and "urlReference" attributes of DomainResource are given specific values. The former specifies the provider of the resource, and the latter provides the URL or the URL template of the resource, when the RAMM resource is associated with a single underlying instance resource. The descriptions for each class and its attributes are given in Section 4.1.1.

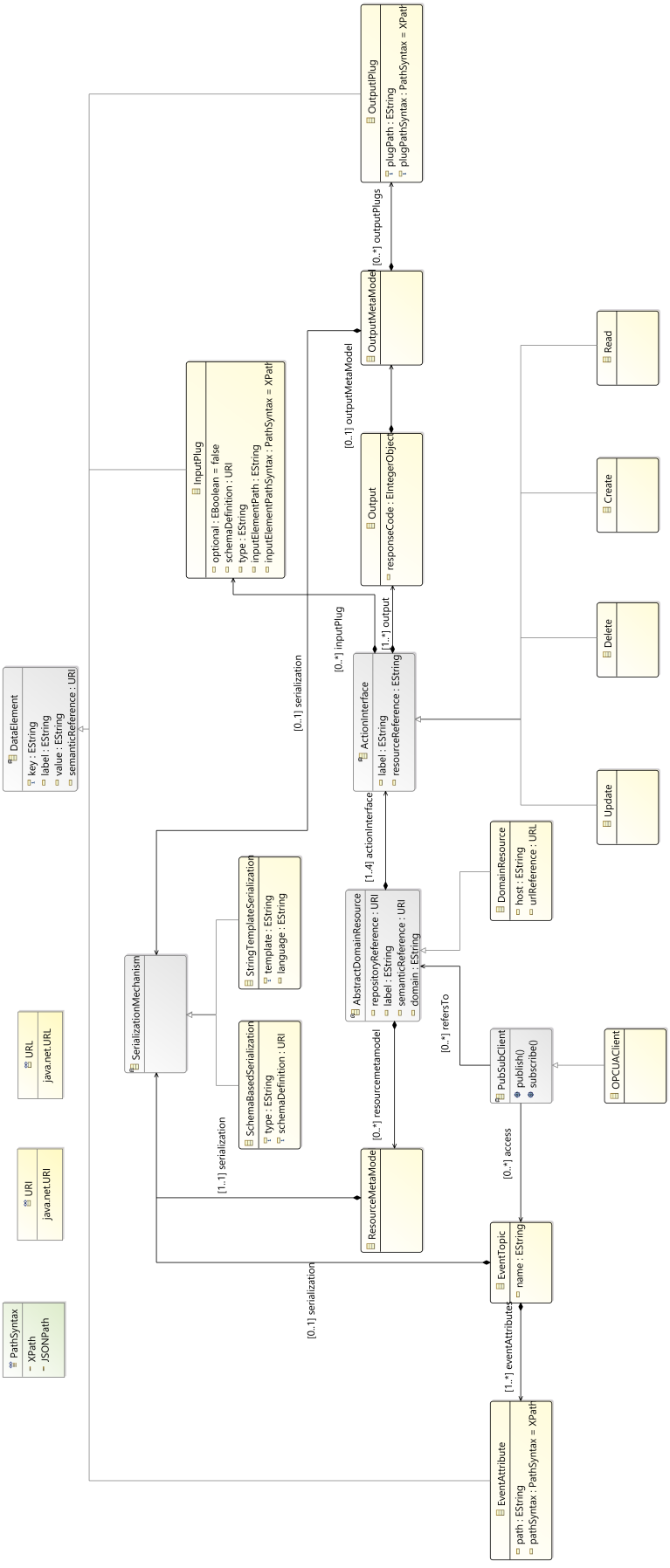


Figure 4.1: Resource Abstraction Metamodel

4.1.1 RAMM Classes and Attributes

AbstractDomainResource	This abstract class models general RAMM resources in a particular domain. It abstracts the reusable parts of IoT resources at design time. Generally, it is configured based on the user's preferences and context. At runtime, it is instantiated to a DomainResource element.
repositoryReference	This attribute specifies the URI of the resource repository associated with the AbstractDomainResource.
semanticReference	This attribute specifies the URI of the ontology associated with the AbstractDomainResource element.
label	This attribute describes the RAMM resources modeled by the AbstractDomainResource abstract class.
domain	This attribute specifies the domain which AbstractDomainResource abstract class models.
resourceMetamodel	This attribute associates an AbstractDomainResource with a structure description denoted by the ResouceMetaModel class.
actionInterface	This attribute associates an AbstractDomainResource with an interface denoted by the subclass of ActionInterface class.

Table 4.1: AbstractDomainResource Class

DomainResource	This class extends AbstractDomainResource class. It models a particular RAMM resource. More specifically, a DomainResource element specifies a semantically distinct, concrete resource which is subject to state manipulation or activity triggering. Typically, a DomainResource element is associated with a single underlying IoT resource.
host	This attribute specifies the identifier of the provider of the IoT resource associated with the DomainResource.
urlReference	This attribute specifies the URL or URL template of the IoT resource associated with the DomainResource.

Table 4.2: DomainResource Class

ResourceMetaModel	This class defines the structure of an associated DomainResource element by referring to an appropriate serialization mechanism.
serialization	This attribute associates the ResourceMetaModel class with the SerializationMechanism abstract class.

Table 4.3: ResourceMetaModel Class

ActionInterface	This abstract class provides a common abstraction for interaction points of an RAMM resource. The ActionInterface is extended by four concrete interface definition classes that denote the CRUD semantics, i.e. *Create, Read, Update* and *Delete*.
label	This attribute provides a description for the interface.
resourceReference	This attribute specifies the URL or URL template of an IoT resource associated with the action implemented by the interface (optional). When provided, the value of this attribute overrides the value of the urlReference DomainResoruce attribute.
inputPlug	The inputPlug attribute associates the ActionInterface class with the InputPlug class. Specifically, an interface can be associated with a set of data elements which are included in the request message.
output	The output attribute associates the ActionInterface class with the Output class.

Table 4.4: ActionInterface Class

Create	The Create class is a subclass of ActionInterface abstract class. It defines a specific action with resource creation semantics. Create is mapped to HTTP POST.

Table 4.5: Create Class

Read	The Read class is a subclass of ActionInterface abstract class. It defines a specific action with resource retrieval semantics. Read is mapped to HTTP GET.

Table 4.6: Read Class

Update	The Update class is a subclass of ActionInterface abstract class. It defines a specific action with resource modification semantics. Update is mapped to HTTP PUT.

Table 4.7: Update Class

Delete	The Delete class is a subclass of ActionInterface abstract class. It defines a specific action with resource deletion semantics. Create is mapped to HTTP DELETE.

Table 4.8: Delete Class

Output	The output class specifies the outcome of the interaction with the interface.
responseCode	This attribute denotes the status of the interaction.
outputMetaModel	This attribute associates the Output class with the OutputMetaModel class which describes the expected payload of the output of the interaction.

Table 4.9: Output Class

OutputMetaModel	This class models the response payload of an interface interaction. It also allows for the specification of the corresponding schema type.
serialization	This attribute associates the OutputMetaModel class with the SerializationMechanism class.

Table 4.10: OutputMetaModel Class

OutputPlug	This class specifies elements of the response payload that need to be distinctly identified so that they can be used in compositions and conditions.
plugPath	This attribute is used to locate the element that comprise the OutputPlug data element in the response payload structure.
plugPathSyntax	This attribute specifies the syntax that the value of the plugPath attribute conforms to. The possible values of plugPathSyntax are specified by the PathSyntax enumeration.

Table 4.11: OutputPlug Class

InputPlug	The InputPlug class models data elements included in request message payloads.
optional	This attribute specifies whether the input element is optional (true) or required (false) (optional).
schemaDefinition	This attribute specifies a schema document that includes the definition of the InputPlug element's type (optional).
type	This attribute specifies the element of the schemaDefinition document that constitutes the type of the InputPlug element (optional).
inputElementPath	This attribute is used to locate the InputPlug element in the exchanged message, when SchemaBasedSerialization is utilized to specify the structure of the message payload.
inputElementPathSyntax	This attribute is used when a value is specifies for the inputElementPath attribute. It denotes which syntax is used for the path expression. The possible values that this attribute can take are specifies by the PathSyntax enumeration.

Table 4.12: InputPlug Class

PathSyntax	This enumeration is used to specify the list of languages that can be used for the expressions contained in values of the inputElementPath and plugPath attributes. In this thesis, two language are considered, namely XPath and JSONPath.

Table 4.13: PathSyntax Class

DataElement	This abstract class denotes entities that are used as common abstractions for the classes InputPlug, OutputPlug and EventAttribute.
key	This attribute is used to uniquely identify the data element within the scope of a single use case scenario.
label	This attribute is used to describe the data element (optional).
value	This attribute is used to provide a fixed or default value for the data element (optional).
semanticReference	This attribute is used to assign semantics to the data element by pointing to a URI that identifies a corresponding ontology element (optional).

Table 4.14: DataElement Class

SerializationMechanism	This abstract class specifies the structure for the exchanged message. The SerializationMechanism class is extended by the StringTemplateSerialization and SchemaBasedSerialization classes.

Table 4.15: SerializationMechanism Class

StringTemplateSerialization	This class extends the SerializationMechanism class and specifies a string-based template to serialize and deserialize messages.
template	This attribute specifies the string-based template of the message.
language	This attribute specifies the identifier for the template language used (optional).

Table 4.16: StringTemplateSerialization Class

SchemaBasedSerialization	This class extends the SerializationMechanism class and specifies a schema-based template to serialize and deserialize messages.
type	This attribute provides the name of the schema type that specifies the structure of the message.
schemaDefinition	This attribute specifies a URI of a schema location which defines the structure of the message.

Table 4.17: SchemaBasedSerialization Class

EventTopic	This class specifies a type of event that the RAMM resource may publish or subscribe, when it utilizes a Publish/Subscribe client.
eventAttributes	This attribute associates the EventTopic class with the EventAttribute class. It allows for the specification of the particular events in the messages that are desired to be received by subscribers. There should be at least one EventAttribute specified for an EventTopic.
serialization	This attribute associates the EventTopic class with the SerializationMechanism class.

Table 4.18: EventTopic Class

Pub/SubClient	This abstract class provides a common abstraction for the middleware client which refers to DomainResource element. The Pub/-SubClient is extended by the concrete middleware specifications which provide publish and subscribe services.
publish	This method is used by Pub/Sub Client to publish response data which is acquired through the interaction with the interface.
subscribe	This method is used by Pub/Sub Client to subscribe events in the middleware and get notified when specific events occur.

Table 4.19: Pub/SubClient Class

OPCUAClient	The OPCUAClient class is a subclass of Pub/SubClient abstract class. It is a middleware implementation which provides publish and subscribe services for the runtime system.

Table 4.20: OPCUAClient Class

4.1.2 Abstract and Concrete Resource Example

In a smart home environment, light sensors can detect light levels for the purposes of saving energy and improving the security in the house. There are usually several light sensors in a house. Depending on the user's needs, different sensor may be used to detect light levels. Also, users may want the light sensor to function during a specific time period of the day. The obtained light data is useful for controlling lights in a smart home.

Figure 4.2 demonstrates an AbstractDomainResource used in the smart home for light detection. As it shows, the root element is AbstractDomainResource. It specifies the domain as "SmartHome" and shows that this resource models light level information. AbstractDomainResource has references to the resource repository and the domain ontology. The resourceMetaModel attribute specifies the serialization mechanism for this resource is SchemaBasedSerialization (XML).

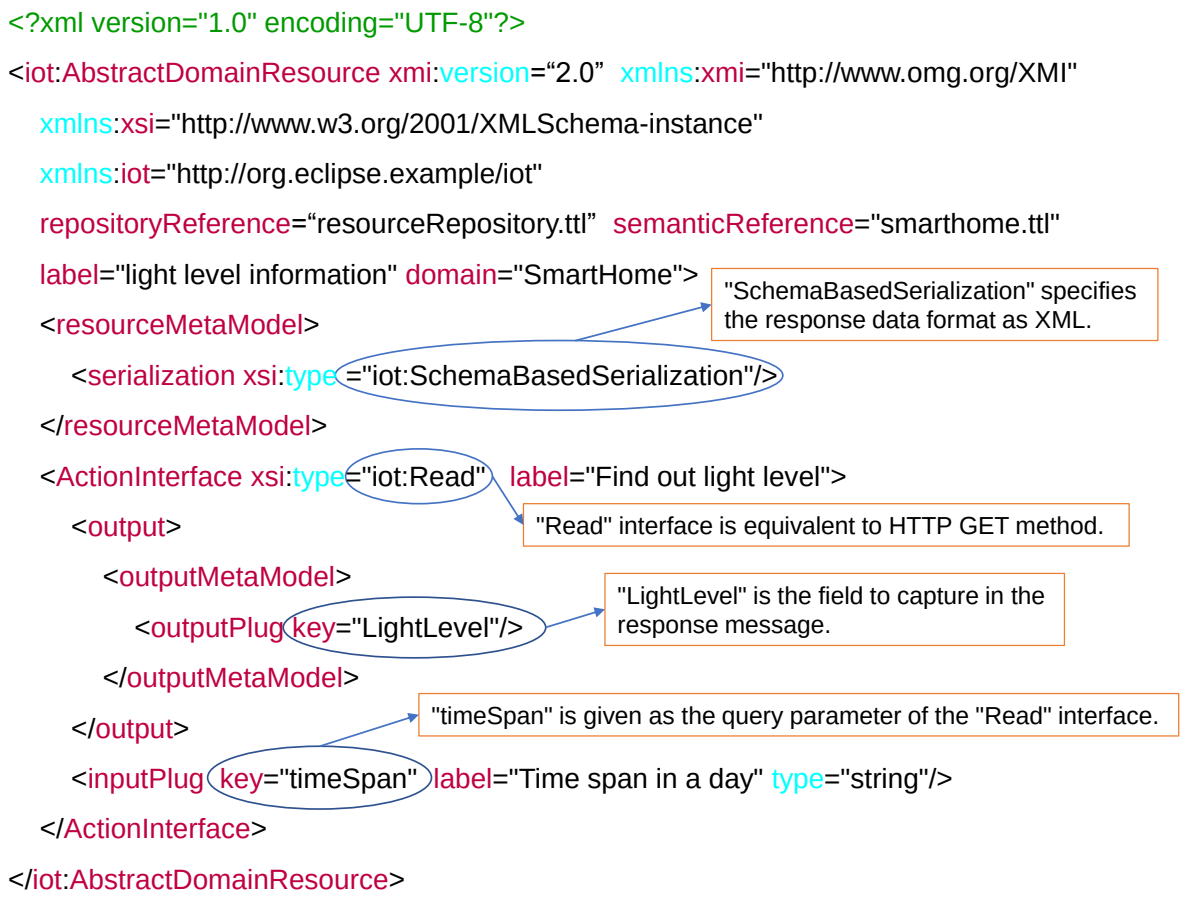


Figure 4.2: AbstractDomainResource Example for Smart Home

Subsequently, a Read interface is specified to capture the HTTP GET interaction point provided by the API. The response of the GET request returns a representation of the light level information in a particular time span of the day. In this smart home scenario, we are interested in one particular field included in the response message: LightLevel. This field is used later for condition evaluation and composition. Lastly, an inputPlug element specifies the timeSpan when we want to query light level information.

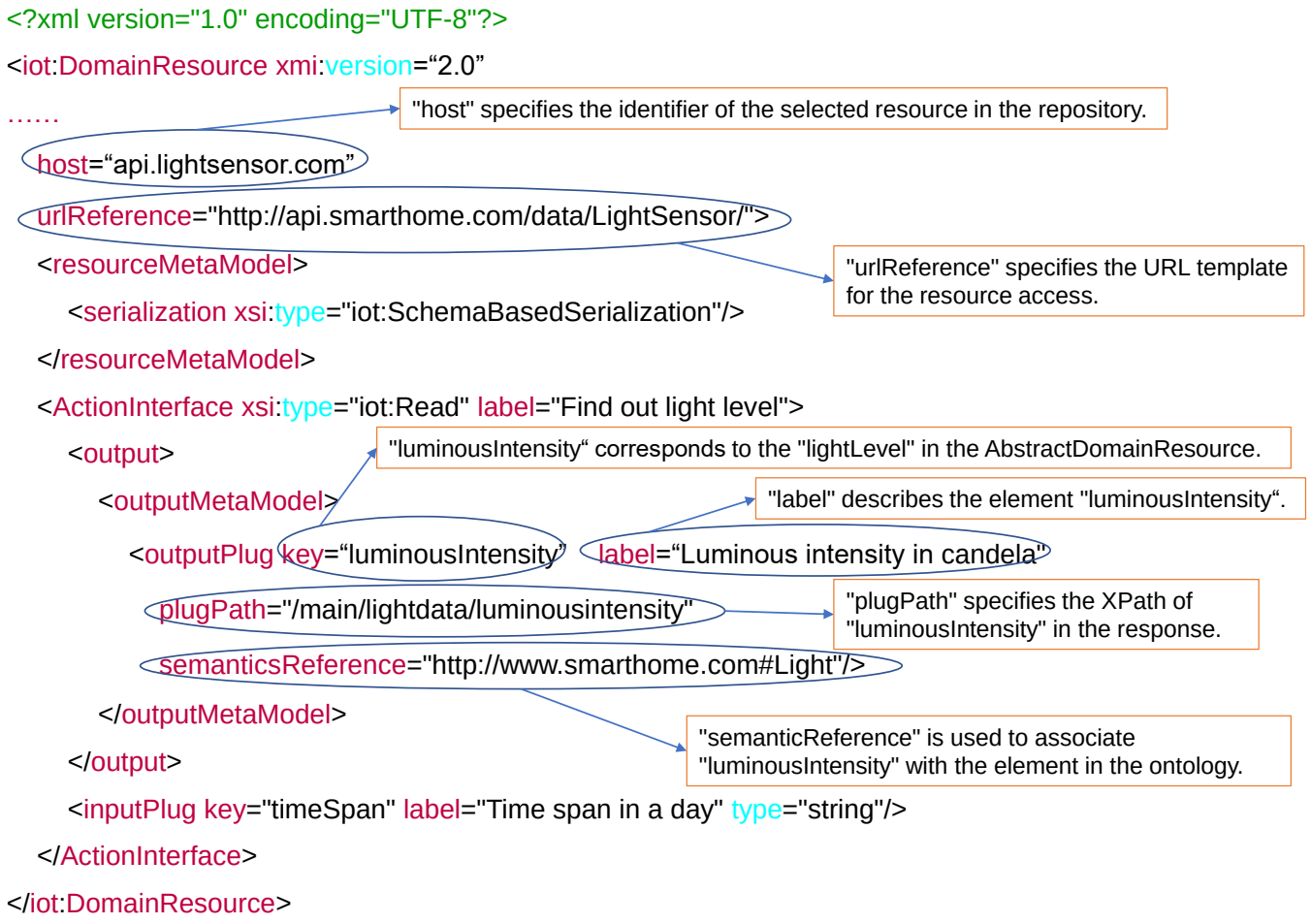


Figure 4.3: DomainResource Example for Smart Home

Figure 4.3 demonstrates a DomainResource used in the smart home example. After the instantiation process is applied (see Section 4.2), A DomainResource (i.e. a concrete resource) is generated as an instance of the AbstractDomainResource class. There are two new attributes added in DomainResource. The host attribute denotes that LightSensor resource is selected in the repository. The urlReference gives the URL template for the resource access. Subsequently, the fields in the outputPlug are instantiated in terms of LightSensor resource. Specifically, the

label describes the element (luminousIntensity) and plugPath specifies the XPath of luminousIntensity in the response message. The sematicReference is used to assign semantics and map to elements in other models.

4.2 Resource Instantiation Framework

In our approach, a RAMM resource template is first defined based on users' requirements. In other words, end users specify what kind of information they want the resource to denote as well as related inputs, outputs and preferred response format. The template instantiation process comprises two key components: semantic web modeling and a resource selection algorithm. Figure 4.4 demonstrates the whole resource instantiation process.

Before we delve into the details of the instantiation process, let us consider a simple example.

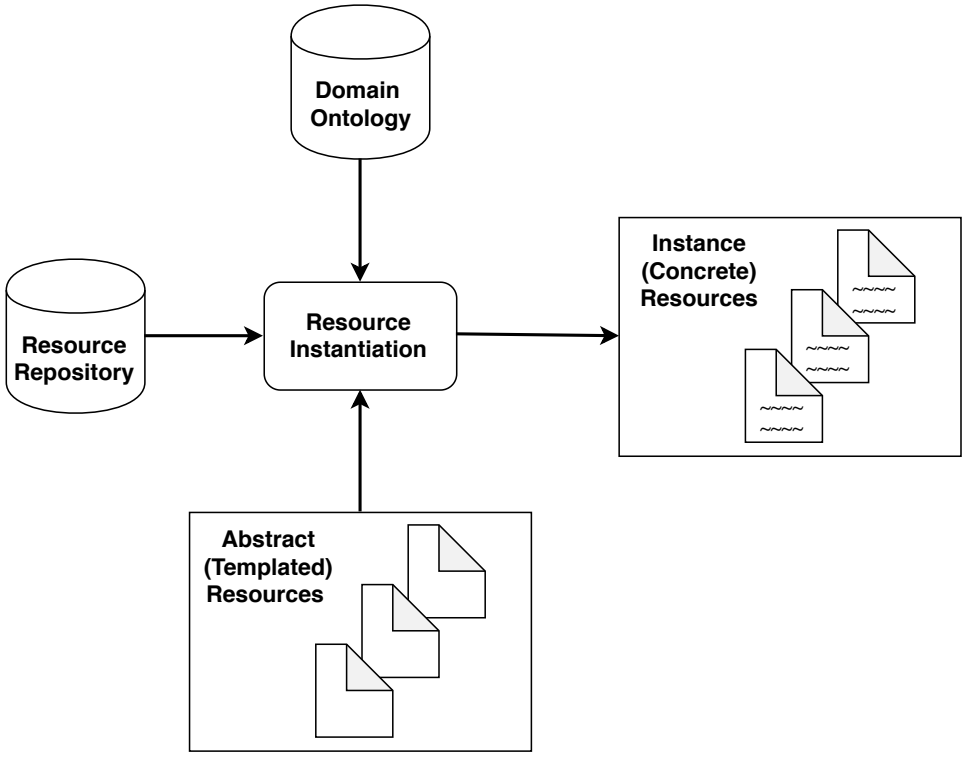


Figure 4.4: Resource Instantiation Process

Suppose we consider a system that involves three types of domain resources. The problem is for instantiation process to find one concrete resource in each resource type. Let us further assume that we have three possible Abstract Domain

Resource models (ADR_i, i = 1, 2, 3). Each of the ADRs describes its input, output and CRUD interface of the abstract resource in each domain. The instantiation process starts with finding all available resources in the repository. Here, let us assume for each AbstractDomainResource, there are three candidate resources available.

Figure 4.5. illustrates this scenario. ADR and DR are the abbreviations for Abstract Domain Resource and DomainResource, respectively. We consider that each candidate resource DR_{ij} (i = 1, 2, 3, j = 1, 2, 3) that can be used to instantiate the corresponding abstract domain resource ADR_i is associated with a five-dimensional vector which describes its QoS metrics, e.g. response time, cost, accuracy, availability and reliability. Based on the QoS metrics, we can also calculate a utility value for each resource. In addition, there is a total response time limit for the system. Hence, the goal of our resource selection step is to select exactly one resource from each domain such that the sum of the utility values is maximized without exceeding the total response time limit. In section 4.4, we propose two algorithms for the resource selection problem, namely Exhaustive Search and Dynamic Programming.

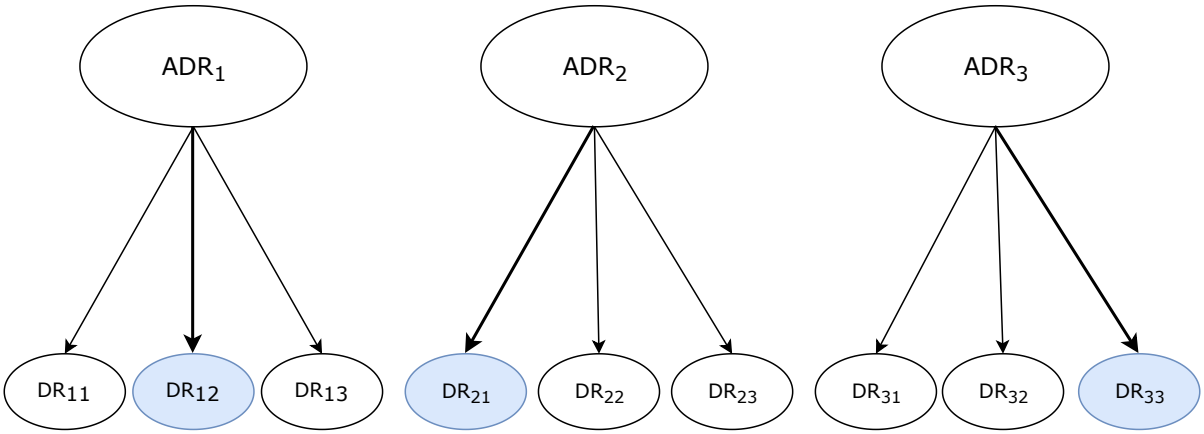


Figure 4.5: A Resource Instantiation Example

Suppose we select DR_{12}, DR_{21} and DR_{33} for the resource composition, the next step is to provide values for elements in each selected resource using domain knowledge. For this purpose, we have a global ontology which provides a shared vocabulary for each domain. For each resource in a domain, we also have a local ontology which corresponds to the global ontology. After resource selection, we can instantiate the AbstractDomainResource using mappings between the local and global ontology. The domain ontology development and semantic mapping steps are elaborated in the next section. In summary, after the resource instantia-

tion process commences, the Abstract Domain Resources (ADR_1, ADR_2, ADR_3) now have become concrete DomainResource (DR_{12}, DR_{21}, DR_{33}) after resource instantiation process.

4.3 Semantic Interoperation

With the rapid development of Linked Open Data and Knowledge Graph [32, 11], different datasets can be linked together to achieve better knowledge representation and sharing. As one of the most popular Linked Open Data sources, the DBpedia dataset [59] describes 6.0M entities which as of April 2016, includes 1.5M persons and 810k places [22]. While data can be collected from a variety of sources, it is a big challenge for us to integrate data across distributed heterogenous data sources. This is also known as interoperability problem. For example, the concept "Human" may be referenced as "Person" in one source and as "Individual" in another. The use of ontology and its description language is a promising approach to resolve the problem of semantic heterogeneity.

4.3.1 Domain Ontology Development

It is worth mentioning that Figure 2.6 only provides an outline for an iterative process for the ontology development. There is no need to strictly follow those steps one after another. Hereafter we use this methodology to develop an ontology for the weather domain. The WeatherDemo ontology is implemented in OWL using an open source ontology editor and the Java based knowledge management system Protégé 5.2.0 [6], which is developed by researchers at Stanford University.

First, we need to create an IRI (Internationalized Resource Identifier) for the ontology. Here the IRI for the WeatherDemo ontology is: **http://www.weatherdemo.com**. Since weather is not an open domain, we use a top-down process for the class hierarchy development. In OWL, every class is a subclass of *owl:Thing*. There are three main classes in our ontology, namely *City, WeatherData* and *WeatherSource*. *WeatherData* is further categorized into six subclasses, namely *Temperature, Humidity, AtomosphericPressure, Wind, Rain* and *CloudCover*. Then according to the source of weather data, *WeatherSource* class is specialized into *DeviceSource* and *ServiceSource*, meaning that weather data can be acquired through either a device (i.e. sensors) or a Web Service.

There are two main kinds of properties in OWL: *Object* properties and *Datatype*

properties. The former, link individuals to individuals, and the latter, link individuals to data values [29]. In this case, the domain and range are both classes for object properties. On the other hand, the domain and range for datatype properties are classes and literals respectively. OWL also supports constructs to express additional characteristics of properties. For instance, Listing 4.1 uses owl:inverseOf to suggest the relation of hasSource and hasProvided properties.

```
<owl:ObjectProperty rdf:about="http://www.weatherdemo.com#hasProvided">
<owl:inverseOf rdf:resource="http://www.weatherdemo.com#hasSource"/>
<rdfs:domain rdf:resource="http://www.weatherdemo.com#WeatherSource"/>
<rdfs:range rdf:resource="http://www.weatherdemo.com#WeatherData"/>
</owl:ObjectProperty>
```

Listing 4.1: Definition of hasProvided property

Finally, we can define instances in the ontology. For the sake of simplicity of our example, we only define three instances for the City class, namely Toronto, Beijing and Athens. The built ontology (not include imported ontologies) is visualized by OntoGraf plugin of Protégé in Figure 4.6, which depicts the concepts, their relationships and the instance of WeatherDemo ontology.

As is shown in Figure 2.6, we often need to consider reusing existing ontologies when we develop our own. In our WeatherDemo ontology, in order to represent weather data accurately, ontologies related to units of measurement can be adopted. Specifically, QUDT (Quantity, Unit, Dimension and Type) ontologies are imported to specify units of weather data [20]. Figure 4.7 shows how an instance of Temperature would be implemented without units of measurements. After the introduction of QUDT ontologies, “hasTemperatureValue” is transformed from a datatype property to an object property. It now links to a blank node which is an instance of QuantityValue class. The blank node has two properties: numericValue and unit. The datatype property numericValue refers to a literal. Another object property unit points to DegreeCelsius which is an instance of QUDT’s concept Unit. The resulting model is depicted in Figure 4.8, in which the upper part corresponds to the ontology layer and lower part corresponds to the data layer.

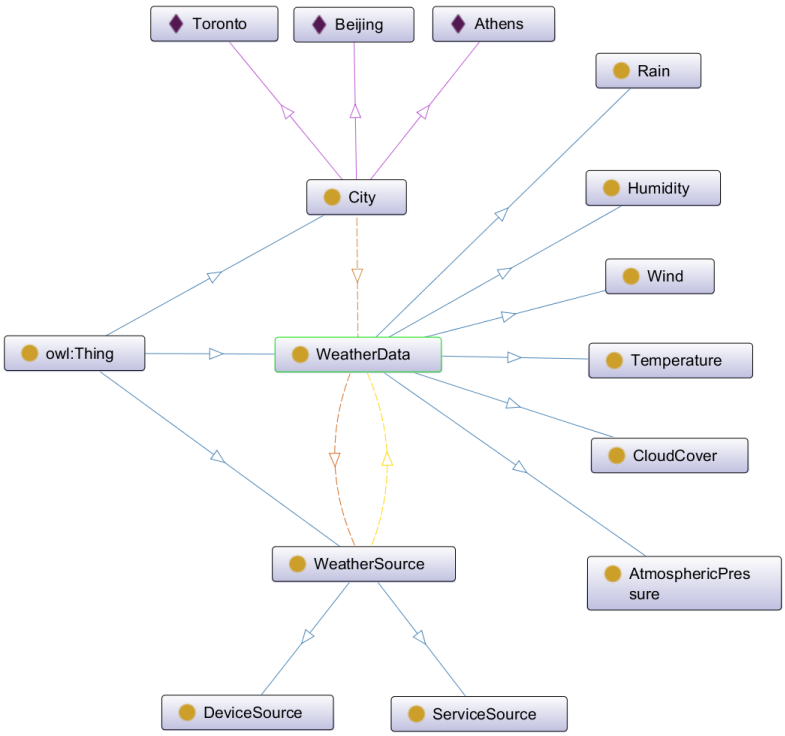


Figure 4.6: The WeatherDemo ontology

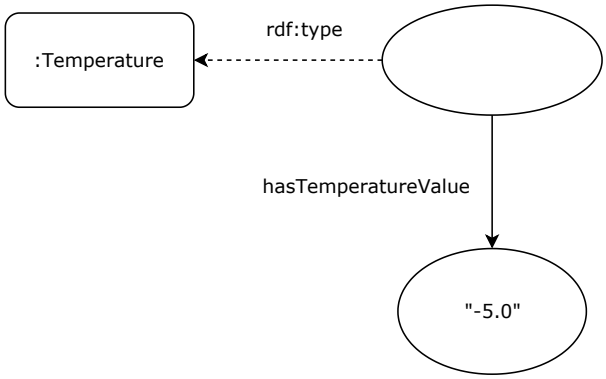


Figure 4.7: An instance of Temperature of -5.0 (without using a unit ontology)

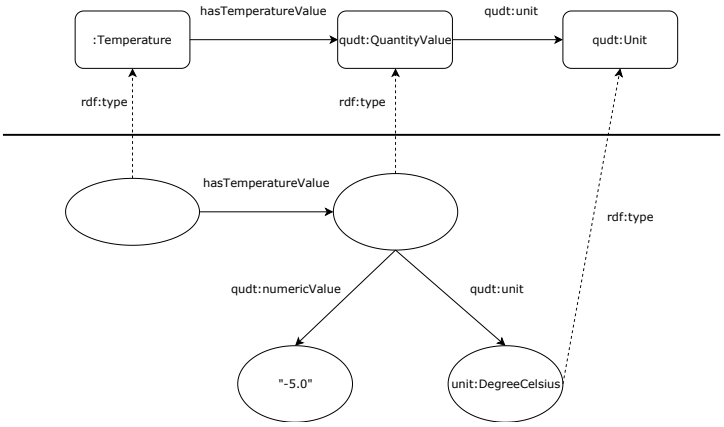


Figure 4.8: An instance of Temperature of -5.0 (using QUDT ontologies)

4.3.2 Semantic Mapping

The existence of a large number of heterogenous data sources on the Internet requires the use of a unified interface to access data. Data may come from numerous Web Services or ubiquitous IoT devices, e.g. sensors. Therefore, it is crucial for data integration and interoperability, to define a way of how to apply ontology to address issues like semantic heterogeneity from various data sources

In [82], the authors put forward three directions for semantic interoperability:

1) **Single ontology approach.** This approach uses one global ontology which provides a shared vocabulary for the specification of the semantics.

2) **Multiple ontologies approach.** In this approach each data source is described by its own ontology. This approach is more flexible than the single ontology approach.

3) **Hybrid approach.** In this approach a global ontology provides a shared vocabulary of a domain among local ontologies. The semantics of each source is represented by its own ontology.

In the third approach, new data sources can easily be integrated, and users can interact such sources through a unified interface. Due to these advantages, in this thesis we have opted to utilize the hybrid approach.

The hybrid approach provides a solution to a problem known as ontology mapping, that is how to create and denote associations between entities in the global ontology and entities in local ontologies. To resolve this issue, first local ontologies of each source are developed independently while capturing local specific

Global Ontology	Local Ontology 1	Local Ontology 2	Local Ontology 3
WeatherData	WeatherReport	WeatherInfo	WeatherRecord
Rain	Rainfall	Precipitation	PRCP
hasWeatherData	hasWeatherReport	hasWeatherInfo	hasWeatherRecord
Toronto	CityofToronto	TorontoCA	TRT

Table 4.21: Mapping between global and local ontology

information. Next, a global ontology is constructed by extracting common terms used in the local ontologies. The last step is to map semantically equivalent entities between them. OWL offers three built-in properties to link two entities: *owl:equivalentClass* for mapping same classes, *owl:equivalentProperty* for mapping same properties and *owl:sameAs* for mapping same individuals. In our example, weather domain, Table 4.21 demonstrates a sample mapping between local and global ontologies.

4.3.3 RDF and SPARQL

The Resource Description Framework (RDF) is a modeling language that has been developed in order to provide a flexible mechanism for describing web resources and relationships between them [43]. The underlying data structure of RDF is a collection of triplets, each consisting of three components: a subject, a predicate (or property) and an object. A set of triplets is called an RDF graph, as is indicated in the data layer of Figure 4.8. In order to facilitate sharing and exchange of RDF data on the Web, several serialization formats have been developed. Until now, those mainly include Turtle, N-Triples, JSON-LD, RDFa, RDF/XML, etc. In light of readability and compactness, in this thesis we adopt the Turtle format [30]. For instance, triples in Figure 4.8 could be written as in Listing 4.2.

```
@prefix : <http://www.weatherdemo.com#> .
```

```
@prefix source: <http://www.weatherdemo.com/source/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

source:10086 :hasTemperatureValue source:10000.

source:10000 qudt:numericValue "-5.0"^^xsd:double;
             qudt:unit unit:DegreeCelsius.
```

Listing 4.2: RDF example encoded in Turtle syntax

Two major disadvantages of RDF are that first it falls short of its capability to denote abstractions and second its limited capability of denoting semantic annotations. That is to say, RDF is not able to describe things which belong to a common set. Additionally, RDF can barely understand the meaning, or semantics, of the terms used in triples. This is where the ontology comes into play. By means of the ontology language, such as RDFS and OWL, the expressivity of RDF is significantly enhanced. One thing to notice is that RDFS and OWL can be serialized in RDF, hence they also have serialization formats like RDF/XML and Turtle.

Now that we have our own ontologies and RDF data, the next step is how to retrieve useful information from them. Like SQL is used to query relational database, RDF data is queried using a language called SPARQL [74]. SPARQL stands for SPARQL Protocol and RDF Query Language, which consists of two parts: query language and protocol. Besides its common query ability like SQL and XQuery, SPARQL differs in that it is capable of transmitting queries and results between a client and a SPARQL endpoint via HTTP protocol. SPARQL queries are based on the concept of graph pattern matching. A basic SPARQL query is simply a graph pattern with some variables [13]. Therefore, if RDF data matches a graph pattern, the specific value in RDF is returned as the result.

In this thesis, we utilize Apache Jena's SPARQL client library ARQ, which is a query engine that supports the SPARQL RDF Query Language [2]. In ARQ, the RDF dataset is first read into a data structure called Model using Jena's RDF API. The query is then executed along with the Model. Finally, the query result is handled as a stream of solutions and system memory is released. Listing 4.3 shows an example of querying all weather sources which are located in Canada using SPARQL.

```
PREFIX : <http://www.weatherdemo.com#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?s WHERE {
  ?s rdf:type :WeatherSource.
  ?s :locatedIn ?o.
  FILTER(?o = "Canada")
}
```

Listing 4.3: SPARQL code to query all weather sources which are located in Canada

4.4 Resource Selection Algorithm

Recently, there has been a growing number of Web Services and IoT devices. While it may seem tempting to have a diversity of ecosystem for prototyping, it is usually difficult and time consuming to find suitable IoT resources. Based on Quality of Service (QoS) metrics, this chapter discusses that the resource selection problem can be transformed to the 0-1 Multiple Choice Knapsack Problem (0-1 MCKP). We also propose two possible approaches to find a global optimal solution, namely Exhaustive Search and Dynamic Programming. Finally, the performance of these algorithms is compared by considering a simple scenario.

4.4.1 Problem Formulation

In the previous chapter, we discussed how to discover IoT resources of a certain domain using SPARQL queries in the resource repository 4.3. The next step is to select and combine those resources together to accomplish a complex task. The difficulty of this step lies in both the scale and complexity of IoT. In addition to an increasing number of Web Services active on the Internet, an even larger number of IoT devices are deployed in all kinds of application scenarios. Various aspects of IoT resources need to be considered before composing IoT applications.

The runtime performance of services is important for applications. For example, IoT applications such as disaster warning, smart transportation and emergency treatment may require a real-time response. QoS for Web Services refers to

various nonfunctional characteristics such as response time, throughput, availability, and reliability. Besides these characteristics, IoT resources may also need to take other measurement metrics, such as cost, accuracy and fidelity, into account. In this thesis, we consider five QoS metrics of IoT resources: response time, cost, accuracy, availability and reliability.

In the context of this thesis, we use the following terminology:

- **Atomic resource:** An atomic resource (or candidate resource) is associated with a QoS vector, which specifies parameters [61].
- **Resource class:** A resource class is a set of atomic resources that provide a common functionality like weather forecast.
- **Utility value:** Each atomic resource has an associated utility value, which is calculated by the utility function.

The 0-1 Multiple Choice Knapsack Problem (MCKP) is a generalization of the basic 0-1 Knapsack Problem. In 0-1 MCKP, we are given g groups $N_1, .., N_g$ of items to pack in a knapsack of capacity c. Each item $j \in N_i$ has a profit p_{ij} and a weight w_{ij}. The goal is to select exactly one item from each group such that the total profit P is maximized without the total weight W exceeding c. 0-1 MCKP is NP-hard as it contains the 0-1 KP as a special case [28]. Figure 4.9 illustrates the MCKP. We have to choose exactly one item from each group. At the same time, we must satisfy $\sum_{i=1}^{3} w_i \leq 50$ and maximize the total profit of the chosen items. It is important to note that there may be no solution, which means that no set of items satisfying the total weight constraint.

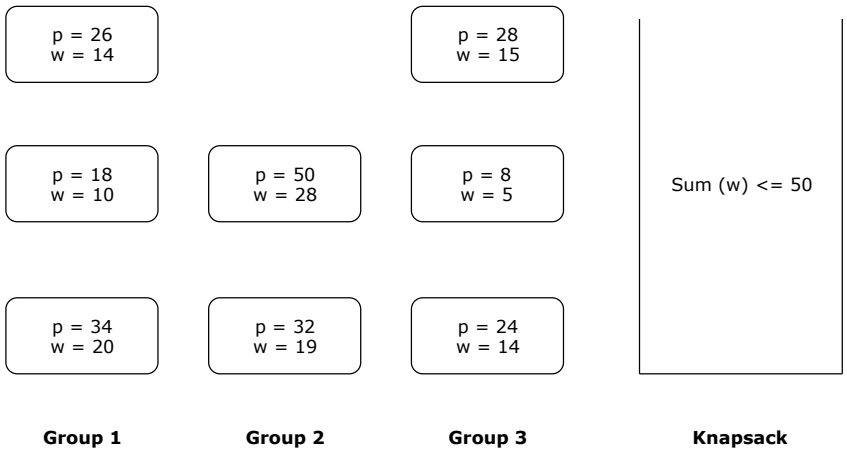


Figure 4.9: 0-1 Multiple Choice Knapsack Problem

Based on the definition of MCKP, we can formulate the resource selection problem as a MCKP as follows.

1) Each resource class is mapped to a group in MCKP.

2) Each atomic resource in a resource class is mapped to an item in a group in MCKP.

3) The response time of the atomic resource is mapped to the weight of the item in MCKP.

4) The utility value of the atomic resource is mapped to the profit of the item in MCKP.

5) The goal is to maximize the sum of the utility values without exceeding the total response time limit.

Suppose there are k resources class $(S_1, .., S_k)$ and total response time constraint is R. The mathematical form of the resource selection problem is as follows:

$$\begin{aligned} \max \quad & \sum_{i=1}^{k} \sum_{j \in S_i} u_{ij} x_{ij} \\ \text{s.t.} \quad & \sum_{i=1}^{k} \sum_{j \in S_i} r_{ij} x_{ij} \leq R, \\ & \sum_{j \in S_i} x_{ij} = 1, \qquad i = 1, \ldots, k, \\ & x_{ij} \in \{0, 1\}, \qquad i = 1, \ldots, k, j \in N_i \end{aligned} \tag{4.1}$$

where x_{ij} denotes whether the atomic resource j is selected for class S_i or not. u_{ij} and r_{ij} are the utility value and response time of the atomic resource j, respectively. The sum of response time of all selected atomic resources must be less than or equal to the total response time constraint R.

4.4.2 Resource Selection

Before diving into algorithms for solving the resource selection problem, we first need to calculate the utility value for each atomic resource. Since different QoS metrics have different scales and natures, combining the values of them directly may distort the ranges of values or lose information. Hence, the normalization

technique (specifically min-max normalization) is applied to the values of QoS metrics. There are two different kinds of metrics. For instance, a higher value for reliability indicates better quality, while a higher value for response time means the opposite effect. Equation 3.2 and 3.3 are used for the former and latter ones, respectively.

$$m.value = \begin{cases} \frac{m.value - m.min}{m.max - m.min}, & m.max - m.min \neq 0 \\ 1, & m.max - m.min = 0 \end{cases} \tag{4.2}$$

$$m.value = \begin{cases} \frac{m.max - m.value}{m.max - m.min}, & m.max - m.min \neq 0 \\ 1, & m.max - m.min = 0 \end{cases} \tag{4.3}$$

After normalization, the utility value of each atomic resource could be calculated by summing up the product of each normalized value and its corresponding weight as shown below.

$$u = \sum (m_i'.value * w_i) \tag{4.4}$$

Obviously, the candidate resource with the largest utility value has a higher quality of service than others in that resource class. If there is no constraint on any QoS metrics like response time, we can select the atomic resource with the largest utility value from each resource class efficiently. However, in reality, this is not the case for resource selection on the basis of the QoS parameters. From now on, we propose two approaches for finding the global optimal solution to resource selection problem.

Exhaustive Search Algorithm

This algorithm is straightforward, that is considering all possible resource combinations and select the best one from them. Without doubt it can find the global optimal solution, yet it is time consuming. As a result, exhaustive search algorithm only suits to occasions when both the number of resource classes and the number of atomics resources in each class are small. Assuming that there are k resource classes and each class has $n_i (i = 1, 2, .., k)$ candidates, the time complexity of this algorithm is $O(\prod_{i=1}^{k} n_i)$. Algorithm 1 gives the pseudo-code for this approach.

Dynamic Programming Algorithm

The resource selection problem can also be solved with dynamic programming technique by following the following steps:

1. **Characterize the optimal solution**
   Suppose there are G resource classes and total response time constraint is R. We first construct an array $M[i, r]$ to represent the maximum utility value with r response time limit for the first i resource classes. In this case, if we can compute all the entries of this array, then the array entry $M[G, R]$ will contain the maximum utility value that satisfies the response time constraint.

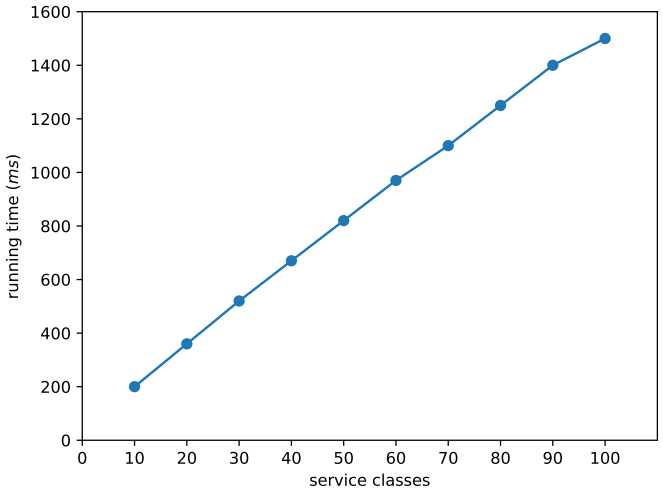
2. **Recursively define the value of the optimal solution**
   It is clear that $M[0, r] = 0$. For class $i = 1$, we should choose the atomic resource with the maximum utility value without violating the response time constraint. The same for class $i >= 2$, except that we must make sure one atomic resource from each previous class $k(k = 1, 2, .., i - 1)$ have been chosen.

3. **Compute value of the optimal solution**
   Based on step 1 and 2, now we can solve the problem using bottom-up method. Algorithm 2 gives the pseudo-code for this method. Assuming there are n candidate resources in total, it is not hard to derive that the complexity of this algorithm is $O(nR)$.

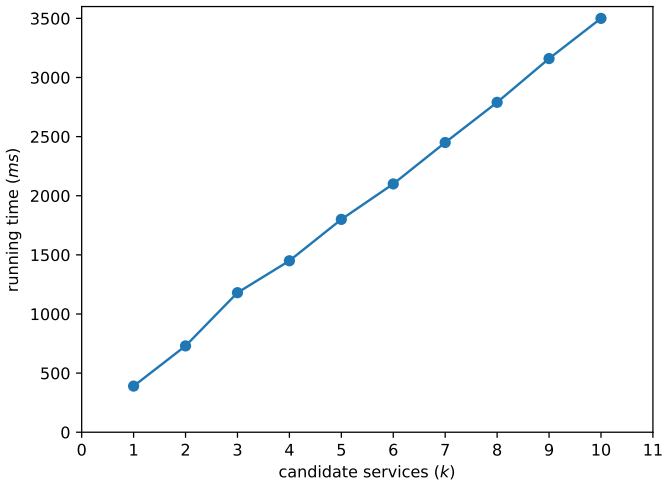
4. **Construct the optimal solution by backtracking**
   In step 3, we only get the maximum sum of the utility values. In order to construct the actual optimal solution, we add an auxiliary array $A[i, j]$ which is computed in line 14 and 24 in Algorithm 2. Suppose $A[i, j] = k$, it means that we decide to choose the $k-$th atomic resource in $M[i, j]$. Given this array, we can construct the optimal solution as is shown in Algorithm 3.

Performance Evaluation Results

To evaluate the performance of Exhaustive Search Algorithm and Dynamic Programming Algorithm, we conducted experiments on a 3.40 GHz Intel Core i7-4770 CPU with 32.0 GB RAM and JDK 1.8.0. In Figure 4.10 , we can conclude that the running time of Dynamic Programming Algorithm has linear correlation with both the number of resources classes and the number of candidate resources in each class.



(a) Each service class has 1000 candidate services



(b) Each test case has 50 service classes

Figure 4.10: Performance of Dynamic Programming algorithm

From Table 4.22 we know that Exhaustive Search Algorithm is time consuming, especially when problem size expands. Hence it is only suitable when the number of resource classes k and candidate resources of each class n are both small. On the other hand, Dynamic Programming Algorithm still performs well when k and n become larger. In conclusion, Dynamic Programming Algorithm is a feasible approach to get a global optimal solution to resource selection problem.

Algorithm 1: Exhaustive Search Algorithm

Input: R: total response time constraint
G: total number of resource classes
n_i: number of atomic resource in class i
r_{ij}: response time of j-th item in class i
u_{ij}: utility value of j-th item in class i
Output: s_i: select s_i-th item from class i

```
U ← 0
s_i ← 0, i = 1, ..., G
c_i ← 0, i = 1, ..., G
Recursive-SOLVE(G)
if s_1 = 0 then
    return null
else
    return s
end
Recursive-SOLVE(g)
    // Base case: all groups have been considered.
    if g = 0 then
        r ← 0
        u ← 0
        for i = 1 to G do
            r ← r + r_{ic_i}
            u ← u + u_{ic_i}
        end
        // Update the solution if needed.
        if r ≤ R and u > U then
            U ← u
            for i = 1 to G do
                s_i ← c_i
            end
        end
        return
    end
    // Recursive cases: there are n_g items in class g.
    for i = 1 to n_g do
        c_g ← i
        Recursive-SOLVE(g − 1)
    end
```

Algorithm 2: Dynamic Programming Algorithm

Input: R: total response time constraint
G: total number of resource classes
n_i: number of atomic resource in class i
r_{ij}: response time of j-th item in class i
u_{ij}: utility value of j-th item in class i
Output: M_{GR}: maximum sum of the utility values

$M_{ij} \leftarrow 0,\ i = 1, \ldots, G,\ j = 1, \ldots, R$
$A_{ij} \leftarrow 0,\ i = 1, \ldots, G,\ j = 1, \ldots, R$
for $i = 0$ *to* R **do**
  $M_{0i} \leftarrow 0$
end
for $i = 0$ *to* R **do**
  for $j = 1$ *to* n_1 **do**
    if $i \geq r_{1j}$ ***and*** $u_{1j} > M_{1i}$ **then**
      $M_{1i} \leftarrow u_{1j}$
      $A_{1i} \leftarrow j$
    end
  end
end
for $k = 2$ *to* G **do**
  for $i = 0$ *to* R **do**
    for $j = 1$ *to* n_k **do**
      $t \leftarrow i - r_{kj}$
      if $r_{kj} \leq i$ ***and*** $M_{k-1,t}$ ***and*** $u_{kj} + M_{k-1,t} > M_{ki}$ **then**
        $M_{ki} \leftarrow u_{kj} + Mk - 1, t$
        $A_{ki} \leftarrow j$
      end
    end
  end
end
return M_{GR}

Algorithm 3: Find Solution Algorithm

Input: R: total response time constraint
G: total number of resource classes
A_{ij}: for response time j, select A_{ij}-th item from class i
r_{ij}: response time of j-th item in class i
Output: s_i: select s_i-th item from class i
for $i = G$ *to* $i \geq 1$ **do**
  $t \leftarrow A_{iR}$
  $s_i \leftarrow t$
  $R = R - r_{it}$
end
if $s_G = 0$ **then**
  return *null*
else
  return s
end

Resource Classes	Running Time (ms)	
	Exhaustive Search	Dynamic Programming
5	4	8
6	18	9
7	150	9
8	1545	9
9	21436	9
10	183189	13
11	2138399	15
12	–	16

Table 4.22: Running Time Comparison (n=10)

Chapter 5

Condition and Action Modeling

This chapter discusses condition and action modeling. Apart from RAMM metamodel, we also consider a metamodel for Conditions, Actions as well as Mappers which are used to create compositions or "scripts". In this thesis, both Conditions and Actions reference goal models. Goal modeling is a requirements engineering technique which is used for capturing system or software requirements. In a goal model, goals usually represent stakeholders' objectives or expectations which a system should achieve or satisfy. Goal models are normally built in the early phase of a project to help understand whether and why a software should be developed. After building goal models for the conditions and actions, we utilize reasoning technique to evaluate condition goal model and compile action plan for action goal model.

5.1 Condition, Action and Mapper Metamodel

In the proposed approach, the Condition references a goal model, the evaluation of which will determine whether an Action will be triggered or not. Similarly, in our approach, Actions reference also goal models (we refer to these models as actions or task models), the evaluation of which yields a sequence of actions that achieve the top task (i.e., goal). Goal models used for denoting Conditions are evaluated using a reasoner originally developed in [38]. Goal models for actions are evaluated using an analyzer originally developed in [27]. Figure 5.1 shows the Condition and Action metamodel, and Figure 5.2 illustrates the Mapper metamodel. In the Condition metamodel, the root element is the Condition class, which prescribes a specific evaluation case. The Condition class has two attributes named "resultType" and

“resultValue”. The “resultType” attribute describes three possible data types of the evaluation result: Boolean, Probabilistic or Fuzzy. The “resultValue” represents the value of the evaluation result, which is used later for action triggering judgement. For instance, if the “resultType” is set to Boolean, then “resultValue” can be either 0 or 1 based on the evaluation process. The Condition class also contains elements for the evaluation that need to distinctly identified so that they can be mapped to OutputPlug in the RAMM meta-model. The Action metamodel is similar to the Condition metamodel, its “triggerInput” attribute is corresponding to the “result-Value”. Action also contains elements for the action plan compilation. Both the Condition and Action metamodel originate form goal modeling theory [80]. Goal models for condition and action models are utilized to support decision making and compile possible action plans, respectively. Goal models are further elaborated in the next section.

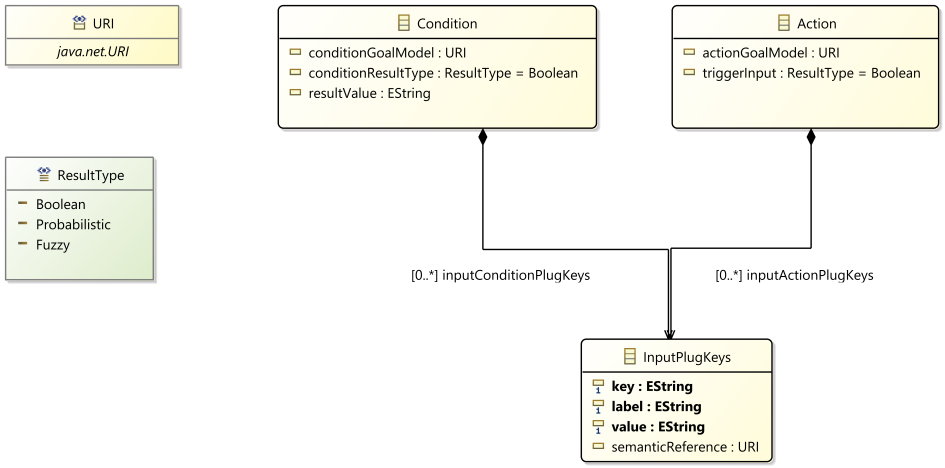


Figure 5.1: Condition and Action Metamodel

The Mapper meta-model is used to map between elements which may come from RAMM model, condition and action model. A Mapper class contains multiple Connection, which specifies a mapping between two model instances. Each Connection has two attributes named “source” and “destination” which acts as the URI of the target instance. Each Connection element, contains several Mapping elements. Each Mapping element has “from” and “to” attributes which specifies a particular element in the “source” and “destination”, respectively. Mapping also includes semantic information by pointing to a URI that refers to a corresponding ontology element.

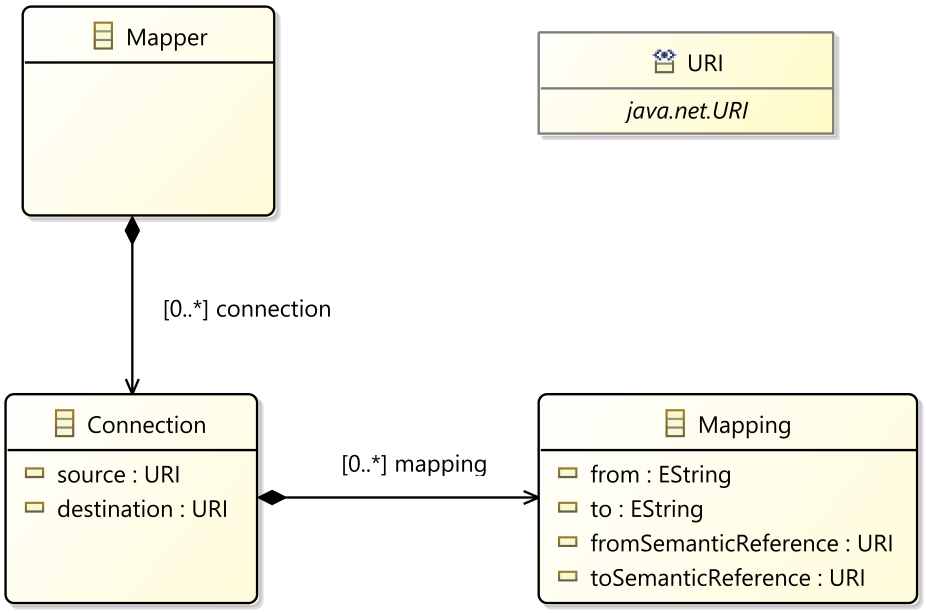


Figure 5.2: Mapper Metamodel

5.2 Goal Modeling and Reasoning

There is plenty of research work in goal modeling, including many approaches that analyze the satisfaction or denial of goals. Example goal modeling methodologies include KAOS [81], i* [83] ,Goal-oriented Requirements Language(GRL) [25] and Non-Functional Requirements(NFR) [41], to name a few. In NFR, goals are further divided to into hard goals and soft goals. The former describes a goal whose satisfaction is binary, while the latter refers to goals for which satisfaction do not follow clear-cut criteria (they are satisficed as opposed to satisfied). Although different goal modeling frameworks may present different concepts or notations, the essential part remains the same.

In this thesis, we focus on applying goal modeling to represent both Condition and Action models, in our quest to implement an IoT programming model which follows the Event-Condition-Action (ECA) paradigm. In the Condition model, the corresponding goal models represent conditions or states that a stakeholder would like to achieve. Similarly, tasks and actions are denoted also as goal models. Such a goal model associated with an action model is referred as task model. Besides modeling capability, goal models also allow for reasoning. In this thesis, we use a reasoner [38] to evaluate conditions for supporting decision making and we use a task model analyzer [27] to compile possible action plans.

5.2.1 Goal Modeling

The basic goal model defines the AND/OR decomposition of goals into sub-goals. An AND goal model node is satisfied if all of its sub-goals are satisfied, while an OR goal model node is satisfied if at least one of its sub-goals is satisfied. In the condition goal model, both AND goal model nodes and OR goal model nodes are called composite goal nodes, which means that they are necessarily completed by the composition of sub-goals. In addition, there are goals which cannot be further decomposed, in which case goals are named atomic goal nodes. Figure 5.3 depicts a simple goal tree. Pretty similar to goal models associated with Conditions, the task model follows the same tree structure except for adding some new resource nodes and links. In the task model, nodes are either tasks, actions or resources. Tasks can be further divided into AND task nodes and OR task nodes. Action nodes represent atomic activities, and resource nodes represent input parameters of the action nodes. Figure 5.4 depicts a simple task tree.

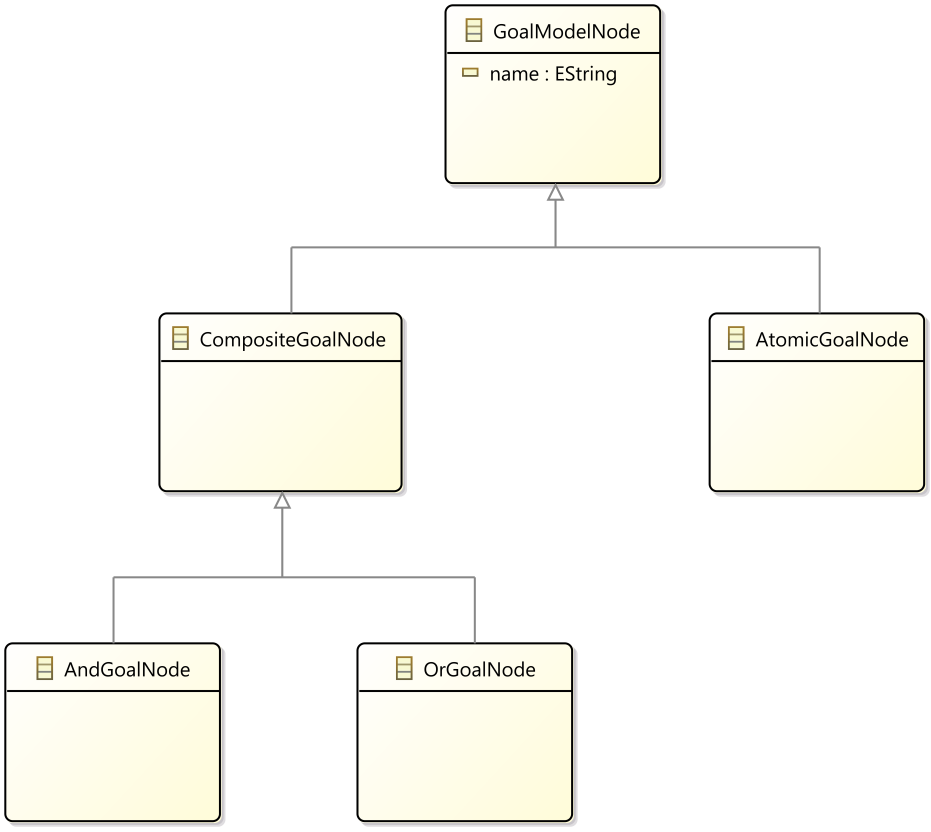


Figure 5.3: Conceptual Goal Tree

Apart from node decomposition, goal model may also contain binary links be-

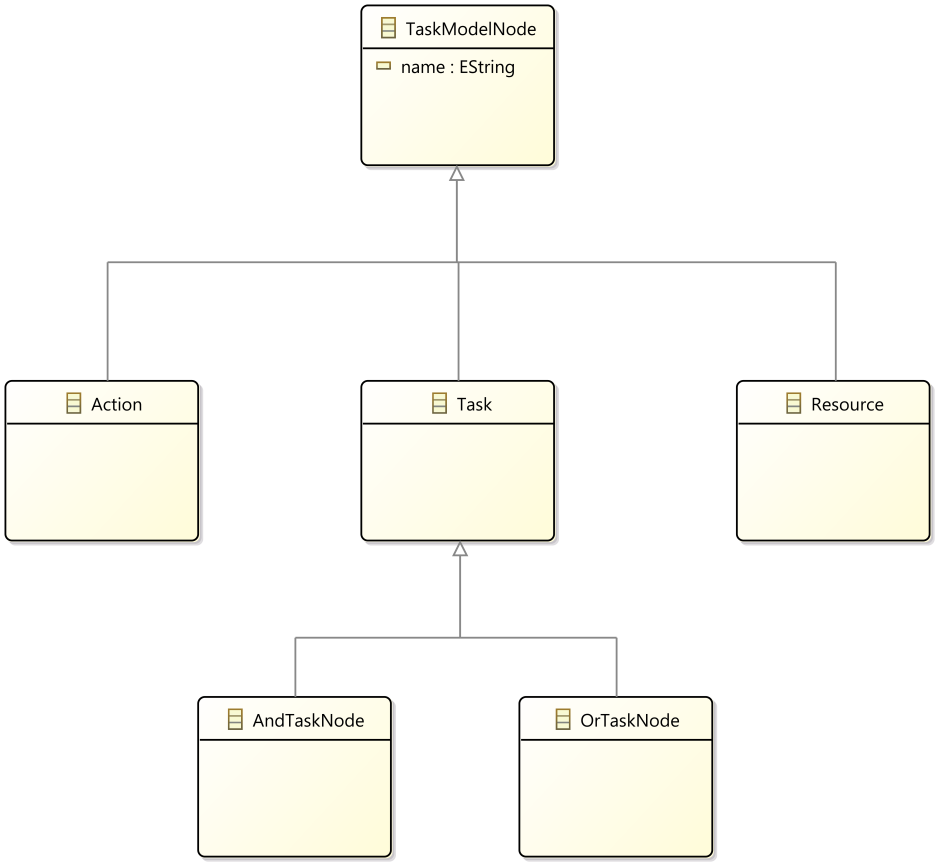


Figure 5.4: Conceptual Task Tree

tween nodes. There are five types of binary links in our complete metamodel, which is depicted in Figure 5.5. The most crucial links are called contribution links, which consist of four types. Adopting the notations from [40], ++S and –S means that the satisfaction of the source node leads to the satisfaction (denial) of the target node. ++D and –D means that the denial of the source node leads to the denial (satisfaction) of the target node. Besides contribution links, there are other binary links used in the action model. These include *Logical Precedence, Temporal Precedence, Resource Dependency, Timeout Link, Timedifference Link*. Logical precedence links are used when the target node can only be performed if the source node has already been performed. On the other hand, temporal precedence links implies that if both the source and target node are involved in an action plan, then source node must perform before the target node. In this way, temporal precedence is essentially a weaker precedence than logical precedence.

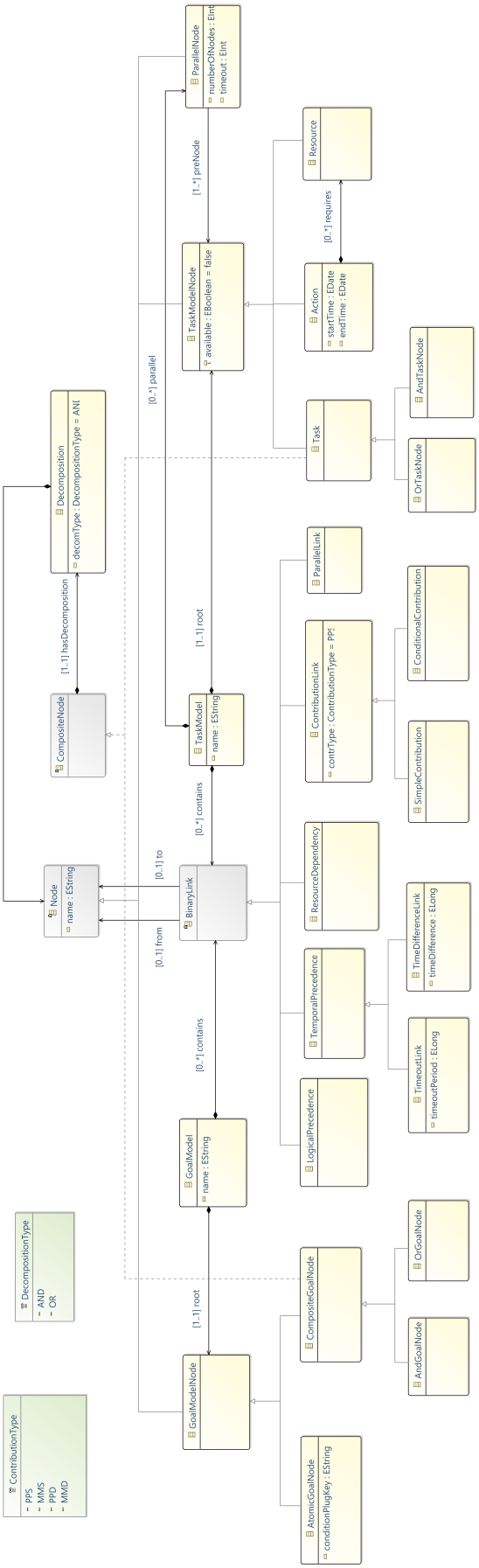


Figure 5.5: Goal Metamodel

What's more, resource dependency link indicates the fact that the source node (action) provides a value to the target node (resource), which can be easily transformed to a ++S contribution link. Parallel link points out that which nodes can be performed in parallel. A complete description of the semantics of the above links can be found in [27].

5.2.2 Condition Goal Model Reasoning

After building the goal model, the next step is evaluating the individual goal model nodes, so that we can obtain a truth value for the root node of the model and thus assist in the decision making process in the ECA environment. Reasoning on goal models has been extensively studied in the academia for run-time analysis. In [51], the authors propose a qualitative and a quantitative reasoning with goal models. Another quantitative reasoning approach is presented in [39], in which a Markov Logic Network (MLN) probabilistic reasoner is used to evaluate the root goal nodes.

Besides the reasoning techniques mentioned above, one can also utilize fuzzy logic for reasoning on goal models. Fuzzy logic uses specific membership functions and ad-hoc operators to model and manage information through a reasoning process that is similar to human reasoning [57]. Also, fuzzy logic is robust and tolerant when IoT devices like sensors have imprecise or unreliable readings. Last but not least, fuzzy logic is more intuitive to understand and implement comparing to other techniques which are based on probability theory. In this thesis we opt to use a fuzzy reasoner for goal models.

According to [38], fuzzy reasoning on the goal model consists of four steps:

1. **Conditional weighted fuzzy rules generation**
   This step can be further divided into two phases: Preprocessing goal model rules and weighted fuzzy rules generation. Because some goal nodes may serve as child nodes to both AND and OR decomposition rules, we need to transform those nodes by introducing pseudo-nodes. Secondly, [38] also describes the algorithm for generating weighted fuzzy rules.

2. **Fuzzification**
   This is a standard process in fuzzy reasoning which transforms observable characteristics to fuzzy values with the help of the membership functions.

3. **Inference**

Based on the step 1 and 2, we are able to deploy a reasoner to deduct the membership degrees for all goals.

4. **Defuzzification**
   In this step, the centroid defuzzification method is utilized to calculate the quantifiable result for fuzzy goals by combing the membership degrees.

Let us use an example to show the condition goal model reasoning. Listing 5.1 demonstrates an example of condition goal model, and it is visualized in Figure 5.6. As it shows, the root goal (AndGoalNode) is decomposed into two sub-goals, namely G1 and G2. G1 (AndGoalNode) is further decomposed into two atomic goals, namely G3 and G4. And G2 (OrGoalNode) is further decomposed into another two atomic goals, namely G5 and G6. There are also three contribution links between nodes, i.e. ++D link from G1 to G2, ++S link from G4 to G5 and –S link from G3 to G6.

```
<?xml version="1.0" encoding="UTF-8"?>
<goal:GoalModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:goal="http://org.eclipse.example/goal">
<root xsi:type="goal:CompositeGoalNode" name="WinterTime">
<hasDecomposition>
<decomposedTo xsi:type="goal:CompositeGoalNode" name="G1">
<hasDecomposition>
<decomposedTo xsi:type="goal:AtomicGoalNode" name="G3"/>
<decomposedTo xsi:type="goal:AtomicGoalNode" name="G4"/>
</hasDecomposition>
</decomposedTo>
<decomposedTo xsi:type="goal:CompositeGoalNode" name="G2">
<hasDecomposition decomType="OR">
<decomposedTo xsi:type="goal:AtomicGoalNode" name="G5"/>
<decomposedTo xsi:type="goal:AtomicGoalNode" name="G6"/>
</hasDecomposition>
</decomposedTo>
</hasDecomposition>
</root>
<contains xsi:type="goal:ContributionLink"
    from="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/@decomposedTo.1"
```

```
    to="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/@decomposedTo.0"
    contrType="PPS"/>
<contains xsi:type="goal:ContributionLink"
    from="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/@decomposedTo.0"
    to="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/@decomposedTo.1"
    contrType="MMS"/>
<contains xsi:type="goal:ContributionLink"
    from="//@root/@hasDecomposition/@decomposedTo.0"
    to="//@root/@hasDecomposition/@decomposedTo.1" contrType="PPD"/>
</goal:GoalModel>
```

Listing 5.1: Condition Goal Model Instance

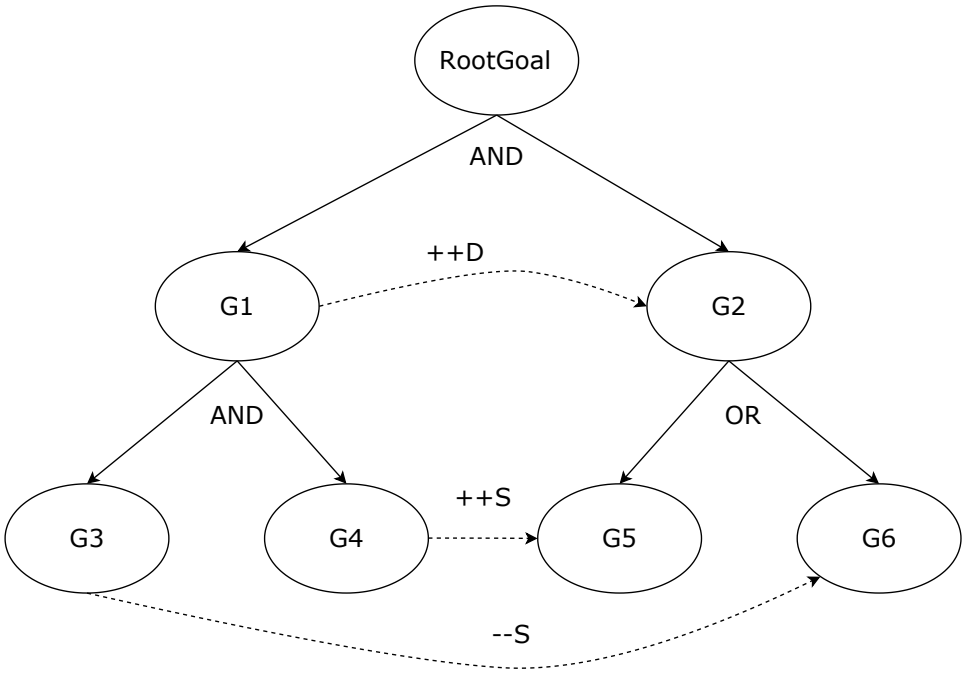


Figure 5.6: Condition Goal Model Visualization

Let us assume that the initial truth values of G3, G4, G5 and G6 are all 1. By utilizing the fuzzy reasoning on condition goal model depicted in Figure 5.6, we can compute the defuzzified value of the RootGoal which in this case is 79.52. In other words, the satisfaction degree of RootGoal is 79.52% (here the fuzzy reasoning process assumes values of parameters are 0, 0, 10, 60, and 40, 90, 100, 100 for the "low" and "high" membership functions, respectively). In another case, let us assume the initial truth values of G3, G4, G5 and G6 are 1, 0, 1, 0. After fuzzy reasoning, the defuzzified value of the RootGoal is 20.48, which means the

satisfaction degree of RootGoal in this case is 20.48%.

5.2.3 Task Model Reasoning

While the objective of goal model reasoning for conditions is determining the satisfaction degree of the root goal, task model reasoning aims to find a execution sequence of actions which can achieve the root task. The reasoning process normally involves two steps. The first step is identifying the possible combinations of actions for model resolution, and the second is computing a feasible scheduling of the selected actions. The first step is also called backward reasoning because it adopts the top-down procedure on the model. Several reasoning techniques applied for this step include propagation of labels, domain specific heuristics and SAT solvers. In a SAT solver based approach, the model is first encoded in a Boolean formula in conjunctive normal form (CNF), then the solver tries to find a truth assignment of the clauses to make the formula true. A lot of tools have been developed to solve the SAT problem in a reasonable amount of time, such as the zChaff tool [64]. Apart from reasoning in the static environment, the environment maybe dynamic due to context changes. In [37], the authors investigate the use of local search algorithms and Boolean expression evaluators for reasoning in the dynamic environment.

Once the actions of the final solution are identified, the next step is compiling a feasible execution sequence of the actions. In [36], the dependency graph analysis technique is applied for the task realization process. The main point here is capturing ordering information from the binary links. Suppose that for an action node a, $a \hat{\in} T$ indicates that a belongs to a set the task node T decomposes to, and $a \rightarrow b$ means that action b depends on action a. For the precedence links like logical precedence or temporal precedence, there are four rules for translating the link to dependencies based on the types of the node [36]:

1) Task $T_1 \xrightarrow{Precedence}$ Task $T_2 \iff \forall x \hat{\in} T_1, y \hat{\in} T_2 : x \rightarrow y$

2) Task $T \xrightarrow{Precedence}$ Action $a \iff \forall x \hat{\in} T : x \rightarrow a$

3) Action $a \xrightarrow{Precedence}$ Task $T \iff \forall x \hat{\in} T : a \rightarrow x$

4) Action $a \xrightarrow{Precedence}$ Action $b \iff a \rightarrow b$

Resource dependency links (RD links) can also contain ordering information between actions. For example, suppose a resource node has a parent action node

y and exactly one incoming resource dependency link from action node x. In this case, the temporal relationship is $x \rightarrow y$ because the execution of action y requires a value which could be produced by the action x.

Given the dependency information of selected actions, Action Dependency Graph is then constructed to compute a valid scheduling of actions [36]. An ADG can be viewed as a Directed Acyclic Graph (DAG), which is solvable by means of the topological sorting algorithm. One advantage of the algorithm is that its time complexity is linear to the sum of the number of actions and the number of dependencies.

Let us use an example to show the task model reasoning. Listing 5.2 demonstrates an example task model, and it is visualized in Figure 5.7. After task model reasoning, we can get an execution sequence of actions which is shown in Listing 5.3. As is shown, the execution sequence starts at node A1 and ends at node A4. There is a parallel node P0 which allows for the parallel executions of node A5 and A2. Also, the time allocated for the parallel execution is 459ms.

```
<?xml version="1.0" encoding="UTF-8"?>
<goal:TaskModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:goal="http://org.eclipse.example/goal" name="TaskModel1">
<root xsi:type="goal:Task" name="root">
<hasDecomposition>
<decomposedTo xsi:type="goal:Task" name="T1">
<hasDecomposition>
<decomposedTo xsi:type="goal:Action" name="A1">
<requires name="R1"/>
<requires name="R2"/>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A2">
<requires name="R3"/>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
<decomposedTo xsi:type="goal:Task" name="T2">
<hasDecomposition decomType="OR">
<decomposedTo xsi:type="goal:Task" name="T3">
<hasDecomposition>
<decomposedTo xsi:type="goal:Action" name="A3">
```

```
<requires name="R4"/>
<requires name="R5"/>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A4">
<requires name="R6"/>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A5">
<requires name="R7"/>
<requires name="R8"/>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
</hasDecomposition>
</root>
<contains xsi:type="goal:ContributionLink"
    from="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/@decomposedTo.0"
    to="//@root/@hasDecomposition/@decomposedTo.0" contrType="PPD"/>
<contains xsi:type="goal:LogicalPrecedence"
    from="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/@decomposedTo.0"
    to="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/@decomposedTo.0"/>
<contains xsi:type="goal:ResourceDependency"
    from="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/@decomposedTo.1"
    to="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/@decomposedTo.0/
@hasDecomposition/@decomposedTo.0/@requires.1"/>
<contains xsi:type="goal:TemporalPrecedence"
    from="//@root/@hasDecomposition/@decomposedTo.0"
    to="//@root/@hasDecomposition/@decomposedTo.1"/>
<parallel name="P0" timeout="459"
    preNode="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/@decomposedTo.1
    //@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/@decomposedTo.1"/>
</goal:TaskModel>
```

Listing 5.2: Task Model Instance

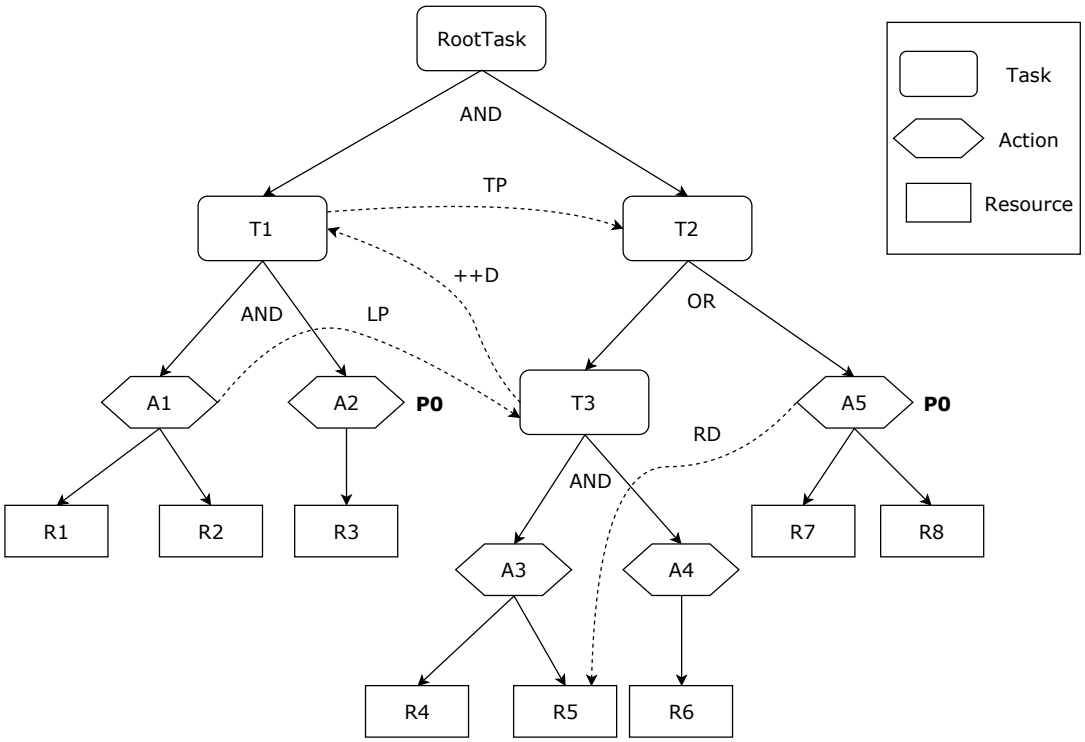


Figure 5.7: Task Model Visualization

```
One sequence is:
  SP A1 P0 A3 A4 JP

  One sequence for parallel P0 is:
    SP0 A5 JP0
    SP0 A2 JP0

    The nodes [A5, A2] will have 459ms to synchronize
```

Listing 5.3: Execution Sequence of Actions

Chapter 6

Implementation and Case Study

The implementation of the prototype system consists of three major parts: modeling and code generation; data collection; and prototype development. For the modeling and code generation, the Eclipse Modeling Framework (EMF) is employed to build the RAMM and goal metamodel, as well as to generate code from the metamodels. As it was presented in Chapter 4, data may come from various information sources, such as Web resources, IoT devices, etc. In this thesis, we have developed a Java client program to access various Web Services and retrieve relevant information, so that it can be used for evaluating and reasoning purposes. Finally, a prototype is developed and tested utilizing Open Platform Communications Unified Architecture (OPC UA) middleware as means to disseminate data between the various composed Domain Resources. The verified result proves that our proposed programming model is feasible for the IoT application development.

6.1 Modeling and Code Generation Framework

EMF is an open source modeling and code generation framework for building tools and other applications based on a structured data model. The core EMF framework includes an Ecore metamodel for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically [5]. Ecore can be seen as a simplified subset of the UML class diagram. Figure 6.1 shows four main components of Ecore [77].

- **EClass:** Represents a modeled class. It has a name, multiple EAttributes and multiple EReferences.

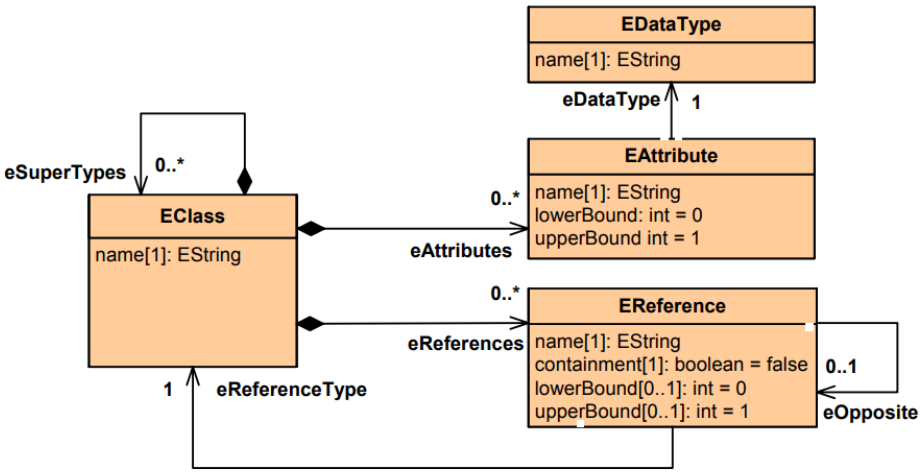


Figure 6.1: Ecore Metamodel

- **EAttribute:** Represents a modeled attribute. It has a name and EDataType.
- **EReference:** Represents one end of an association between classes. It has a name, a Boolean flag to indicate if it represents containment, and an EReference type which is another class.
- **EDataType:** Represents the type of an EAttribute. A type can be a primitive type like int or an object type like java.util.Date.

Figure 5.5 in Chapter 4 can be viewed as a metamodel for goal models. In the metamodel, GoalModel is an EClass and it has an EAttribute “name”. The “name” EAttribute’s EDataType is EString which is analogous to java.lang.String. Also, the GoalModel EClass has two containment EReference entities. One references the root element of the model which is an instance of a GoalModelNode EClass. The other references to abstract EClass BinaryLink which represents the relationship between nodes in the model.

Given the goal metamodel, we can also create a concrete model instance by the means of Eclipse runtime. The generated editor for a simple goal model instance is presented in Figure 6.2.

Besides the Ecore metamodel, EMF has another metamodel which is called Genmodel. While Ecore contains information about the defined classes and their relationships, Genmodel allows us to configure how the code should be generated. The generated Java code consists of three packages:

- **model:** Contains interfaces and the Factory to create Java classes.

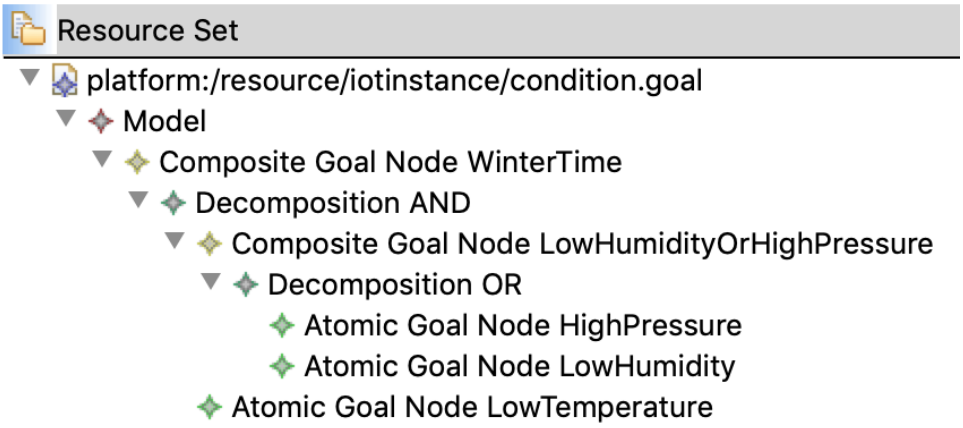


Figure 6.2: Goal Model Instance

- **model.impl:** Contains concrete implementations of the interfaces defined in the model.
- **model.util:** Contains an AdapterFactory that provides interfaces for editing and display.

Each generated interface extends the EObject interface. EObject is the root of every EMF class and it is equivalent to java.lang.Object. After code and instance creation, we can load the model instance using code demonstrated in Listing 6.1 , suppose the path to the instance file is “instances/my.goal”.

```
public GoalModel loadModel() {
   // Initialize the model
   GoalPackage.eINSTANCE.eClass();
   // Register the XMI resource factory
   Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
   Map<String, Object> m = reg.getExtensionToFactoryMap();
   m.put("goal", new XMIResourceFactoryImpl());
   // Obtain a new resource set
   ResourceSet resSet = new ResourceSetImpl();
   // Get the resource
   Resource resource = resSet.getResource(URI.
   createURI("instances/my.goal"), true);
   // Get the first model element and cast it to the right type
   GoalModel goalModel = (T) resource.getContents().get(0);
   return goalModel;
```

```
}
```

Listing 6.1: Load an EMF Model

6.2 Data Collection

Data collection is the process of collecting data from various information sources, such as the Web, sensors, etc. These sources should provide APIs for data access to the application developer or end user in either case. In this section, we demonstrate weather data collection procedure using OpenWeatherMap service [17]. OpenWeatherMap provides an API for accessing current weather data by city name, city ID, geographic coordinates or ZIP code. Depending on the request parameter, weather data is returned in JSON, XML or HTML format. JSON format is used by default. For instance, the current weather condition for Toronto, Canada is returned by sending an HTTP GET request to the following URL.

http://api.openweathermap.org/data/2.5/weather?q=Toronto,ca

An example of API response is shown in Listing 6.2 [3].

```
{"coord":{"lon":139,"lat":35},
"sys":{"country":"JP","sunrise":1369769524,"sunset":1369821049},
"weather":[{"id":804,"main":"clouds","description":"overcast
   clouds","icon":"04n"}],
"main":{"temp":289.5,"humidity":89,"pressure":1013,
"temp_min":287.04,"temp_max":292.04},
"wind":{"speed":7.31,"deg":187.002},
"rain":{"3h":0},
"clouds":{"all":92},
"dt":1369824698,
"id":1851632,
"name":"Shuzenji",
"cod":200}
```

Listing 6.2: API Response Example

For the client side, we employ the Java's API for RESTful Web Services (JAX-RS), specifically javax.ws.rs-api 2.0.1, to access web services [71]. JAX-RS provides a client API to retrieve resources on the Web. Listing 6.3 demonstrates a JAX-RS client API usage scenario. The code specifies the response format as JSON.

```
public void createClient(String url){

   Client client = ClientBuilder.newClient();
   WebTarget target = client.target(url);
   Response response = target.
      request(new MediaType("application", "JSON")).get();
   String responseStr = response.readEntity(String.class);

}
```

Listing 6.3: Create JAX-RS Client

Now that we have weather data from the selected resource (see Listing 6.2), the next step is to extract the elements that we are interested in from the response data as specified in the system's Domain Resource model. For example, if we refer to the resource in Listing 6.7, the inputPlug is denoted as "temp". Just as XPath to XML, there is also a similar tool for JSON which is called JSONPath. In this section, we adopt Jayway [12], a Java implementation of the JSONPath, to analyze, transform and selectively extract data out of JSON documents. JSONPath expressions refer to a JSON structure in the same way as XPath expressions are used in combination with an XML document. Since a JSON structure is always anonymous and doesn't necessarily has a "root member object", the root element in JSONPath is always referred to as $ regardless if it is an object or array. JSONPath expression can use the dot-notation or the bracket-notation. For example, the JSONPath to the element "temp" in Listing 6.2 is written as the following.

$.main.temp or **$['main']['temp']**

6.3 Prototype Development

OPC UA is a publish subscribe middleware framework that ensures the open connectivity, interoperability, security, and reliability of industrial automation devices

and systems [16]. OPC UA defines objects in terms of variables and methods. Figure 6.3 illustrates the OPC UA object model [1]. As it is shown, OPC UA provides services to access the objects and their components such as reading or writing a variable value, receiving events from the object or calling a method. The elements of the object model are represented in the address space as nodes. Each node is an instance of a node class including object, variable, method, etc. For example, Listing 6.4 depicts DataType node (RequestHolder) which is used to represent an HTTP request.

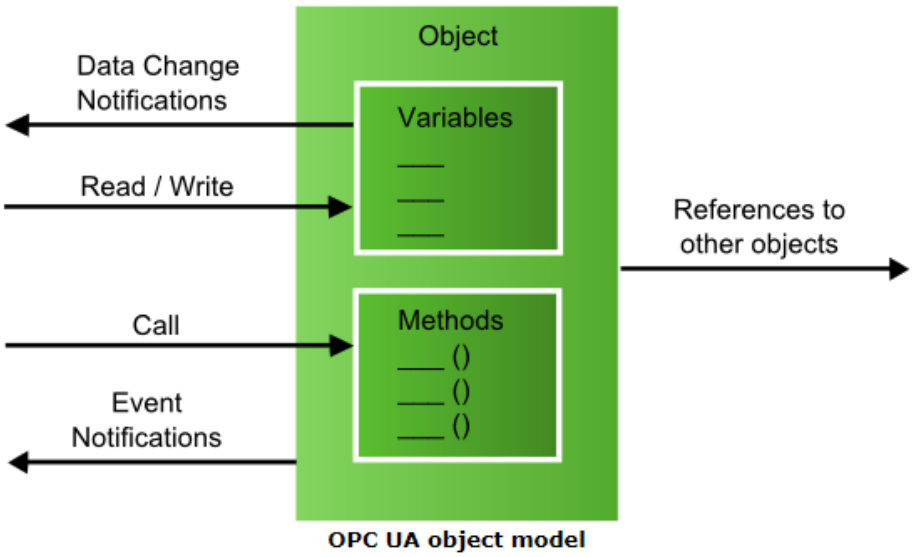


Figure 6.3: OPC UA Object Model

```
public class RequestHolder {

   private final String urlReference;
   private final String targetFolder;
   private final HashMap<String, String> inputPlugs;
   private final String responseFormat;
   private final HashMap<String, String> outputPlugs;
   private final String timeStamp;
   private final String sessionID;
   private final UInteger interval;
   ....
```

```
}
```

Listing 6.4: OPC UA DataType Node Example, see also activity diagram in Figure 3.3

Eclipse Milo is an open source Java implementation of OPC UA [4]. We employ Milo to build the OPC UA server and client of the runtime system. During runtime, OPC UA server is first initialized, and each address space is registered in the server. There are five clients involved in and their responsibilities are introduced below.

1) DomainResourceClient – Creates nodes in the server for all inputs in each DomainResource and subscribes to them. Upon receiving a new value from subscriptions, constructs a new RequestHolder node and writes it back.

2) ConditionClient – Loads all the condition goal models, creates nodes in the server for all inputs in each model and subscribes to them. Upon receiving a new value from subscriptions, re-evaluates the condition goal model and writes the result to the server.

3) ActionClient – Subscribes to the corresponding condition results. Upon receiving a new value from subscriptions, determines whether triggering the action or not, if yes, loading corresponding task model and performs the sequence of actions which is specified in the task model.

4) DaemonManager – Creates a thread pool based on the size of the DomainResource, subscribes to the RequestHolder node. Upon receiving a new value from subscriptions, dispatches the thread execution to the Daemon.

5) Daemon – Performs HTTP request to collect data and writes the response data to the corresponding nodes in the server.

Figure 6.4 depicts a simplified sequence diagram for the runtime system.

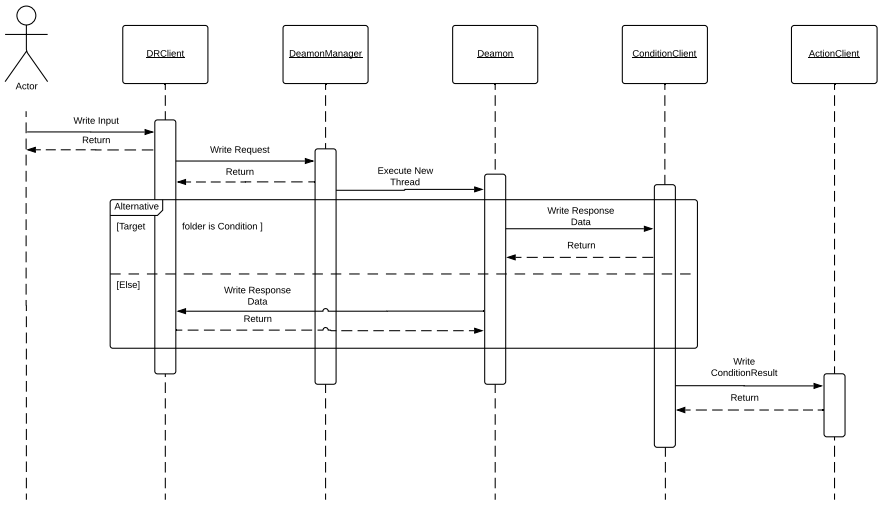


Figure 6.4: Runtime Sequence Diagram

In a nutshell, the runtime system proceeds as follows: An actor first initiates an event. DomainResourceClient receives the event, composes a HTTP request and send it to the server. Then DaemonManager receive the request and dispatches the data collection job to the Daemon. After Daemon retrieves the data and writes to the server, the ConditionClient evaluates the condition goal model and produces a fuzzy value result. At this point ActionClient gets notified and performs reasoning process based on action goal model and compiles possible action plans if needed. Every subscribe and write operation is returned with a StatusCode "Good" if it is successful.

6.4 Case Study: Winter Notification Example

In this section, we discuss a "Winter Notification" example (i.e. script) that follows the proposed programming model. This Winter Notification example is used to explain and demonstrate all components and models used by the runtime system.

6.4.1 Overview of The Winter Notification Example

The Winter Notification example specifies a simple scenario in which weather data in a city is retrieved. If the evaluation result of the condition model is satisfied, then an atomic action or action plan is performed or compiled. The Winter Notification example basically consists of four parts. The first part is modeling an AbstractDomainResource using RAMM and instantiating it via a DomainResource (concrete resource). The second part is specifying Condition, Action as well as Mapper models using the corresponding metamodels. The third part is modeling goal models for conditions and actions. The output of each model is a textual specification which is used in the runtime system. The last part is the result obtained by analyzing the task model.

6.4.2 AbstractDomainResource and DomainResource

Listing 6.5 demonstrates an AbstractDomainResource used in the Winter Notification example. As it is depicted, the root element is an AbstractDomainResource. It specifies the domain as “Weather” and shows that this resource models current weather information. AbstractDomainResource has references to the resource repository (i.e. "repository/resourceRepository.ttl") and the global ontology (i.e. "ontology/weatherdemo.ttl") as depicted in Listing 6.6 and Figure 4.6. The resourceMetaModel attribute specifies the serialization mechanism for this resource is StringTemplateSerialization.

```
<?xml version="1.0" encoding="UTF-8"?>
<iot:AbstractDomainResource xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:iot="http://org.eclipse.example/iot"
  repositoryReference="repository/resourceRepository.ttl"
  semanticReference="ontology/weatherdemo.ttl"
  label="current weather information" domain="Weather">
   <resourceMetaModel>
     <serialization xsi:type="iot:StringTemplateSerialization"/>
   </resourceMetaModel>
   <ActionInterface xsi:type="iot:Read" label="Find out weather
       information">
     <output>
       <outputMetaModel>
         <outputPlug key="Temperature"/>
         <outputPlug key="Humidity"/>
         <outputPlug key="AtmosphericPressure"/>
       </outputMetaModel>
     </output>
     <inputPlug key="city" label="Name of a city" type="string"/>
   </ActionInterface>
</iot:AbstractDomainResource>
```

Listing 6.5: AbstractDomainResource Example

```
@prefix : <http://www.weatherdemo.com#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://www.weatherdemo.com> .

:openweathermap rdf:type :WeatherSource ;
                  :endPoint "api.openweathermap.org";
                  :foundedDate 2012;
```

```
                :architecturalStyle "REST";
                :plugPath "$.main.";
                :hasURLReference
                "http://api.openweathermap.org/data/2.5/weather?q={cityName}".

:aerisapi rdf:type :WeatherSource;
            :endPoint "api.aerisapi.com";
            :foundedDate 2008;
            :architecturalStyle "REST";
            :plugPath "$.response.ob.";
            :hasURLReference
               "http://api.aerisapi.com/observations/{locationKey}".

:accuweather rdf:type :WeatherSource;
              :endPoint "developer.accuweather.com";
              :foundedDate 1962;
              :architecturalStyle "REST";
              :hasURLReference
                 "http://dataservice.accuweather.com/currentconditions/
                v1/{locationKey}".

....
```

Listing 6.6: Resource Repository Example

Subsequently, a Read interface is specified through the *ActionInterface* element to capture the HTTP GET interaction point provided by the API. The response of the GET request returns a representation of the current weather conditions in the specified city as depicted in the *outputPlug* element with key attributes "Temperature", "Humidity" and "Atmosphericpressure". In the Winter Notification example, we are interested in three particular fields included in the response message: Temperature, Humidity and AtmosphericPressure. These fields are used later for condition evaluation and composition. Lastly, an inputPlug element specifies the cityName as URL query parameter.

```
<?xml version="1.0" encoding="UTF-8"?>
<iot:DomainResource xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:iot="http://org.eclipse.example/iot"
  repositoryReference="repository/resourceRepository.ttl"
  semanticReference="ontology/openweathermap.ttl"
  label="current weather information" domain="Weather"
  host="api.openweathermap.org"
  urlReference="http://api.openweathermap.org/data/2.5/weather?q=${city}">
   <resourceMetaModel>
      <serialization xsi:type="iot:StringTemplateSerialization"/>
   </resourceMetaModel>
   <ActionInterface xsi:type="iot:Read" label="Find out weather
       information">
      <output>
         <outputMetaModel>
            <outputPlug key="temp" label="Temperature in Fahrenheit"
               plugPath="$.main.temp"
               semanticsReference="http://www.openweathermap.com#Temp"/>
            <outputPlug key="humidity" label="Humidity in percentage"
               plugPath="$.main.humidity"
               semanticsReference="http://www.openweathermap.com#Humidity"/>
            <outputPlug key="pressure" label="AtmosphericPressure in hPa"
               plugPath="$.main.pressure"
               semanticsReference="http://www.openweathermap.com#Pressure"/>
         </outputMetaModel>
      </output>
      <inputPlug key="city" label="Name of a city" type="string"/>
   </ActionInterface>
</iot:DomainResource>
```

Listing 6.7: An Instantiated DomainResource (from www.openweathermap.org)

Listing 6.7 demonstrates a DomainResource used in the Winter Notification example. After instantiation process, DomainResource is generated on the basis of AbstractDomainResource. There are three new attributes added in DomainRe-

source. The *host* attribute denotes that OpenWeatherMap resource is selected in the repository based on the utility value. The *semanticReference* specifies the URI of the ontology associated with DomainResource see Listing 6.8. The urlReference gives the URL template for the resource access. Subsequently, the fields in the outputPlug are instantiated in terms of OpenWeatherMap resource. Specifically, the label, plugPath and sematicReference are given values to assist in data collection and assign semantics.

```
@prefix : <http://www.openweathermap.com#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://www.openweathermap.com> .

<http://www.openweathermap.com> rdf:type owl:Ontology ;
                        owl:imports <http://www.weatherdemo.com> .

:WeatherObservation rdf:type owl:Class .

:Humidity rdf:type owl:Class ;
         owl:equivalentClass <http://www.weatherdemo.com#Humidity> ;
         rdfs:subClassOf :WeatherObservation ;
         rdfs:label "Humidity in percentage".

:Pressure rdf:type owl:Class ;
         owl:equivalentClass
            <http://www.weatherdemo.com#AtmosphericPressure> ;
         rdfs:subClassOf :WeatherObservation ;
         rdfs:label "AtmosphericPressure in hPa".

:Temp rdf:type owl:Class ;
         owl:equivalentClass <http://www.weatherdemo.com#Temperature> ;
         rdfs:subClassOf :WeatherObservation ;
         rdfs:label "Temperature in Fahrenheit".
```

....

Listing 6.8: Local Ontology for Openweathermap

6.4.3 Mapper

Listing 6.9 presents the Mapper model for the example, and it is visualized in Figure 6.5. As it is shown, it has two connections. One connects AbstractDomainResource with Condition, another connects AbstractDomainResource with Action. In the first connection, three mappings are established between fields in AbstractDomainResource and elements in Condition model. In the second connection, the mapping between “city” in AbstractDomainResource and “cityName” in Action is established.

```
<?xml version="1.0" encoding="UTF-8"?>
<iot:Mapper xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:iot="http://org.eclipse.example/iot">
<connection source="template.iot" destination="condition.iot">
<mapping from="Temperature" to="temp"
    fromSemanticReference="http://www.weatherdemo.com#Temperature"
    toSemanticReference="http://www.openweathermap.com#Temp"/>
<mapping from="Humidity" to="humidity"
    fromSemanticReference="http://www.weatherdemo.com#Humidity"
    toSemanticReference="http://www.openweathermap.com#Humidity"/>
<mapping from="AtmosphericPressure" to="pressure"
    fromSemanticReference="http://www.weatherdemo.com#AtmosphericPressure"
    toSemanticReference="http://www.openweathermap.com#Pressure"/>
</connection>
<connection source="template.iot" destination="action.iot">
<mapping from="city" to="cityName"
    fromSemanticReference="http://www.weatherdemo.com#City"
    toSemanticReference="http://www.openweathermap.com#CityName"/>
</connection>
</iot:Mapper>
```

Listing 6.9: Mapper Example

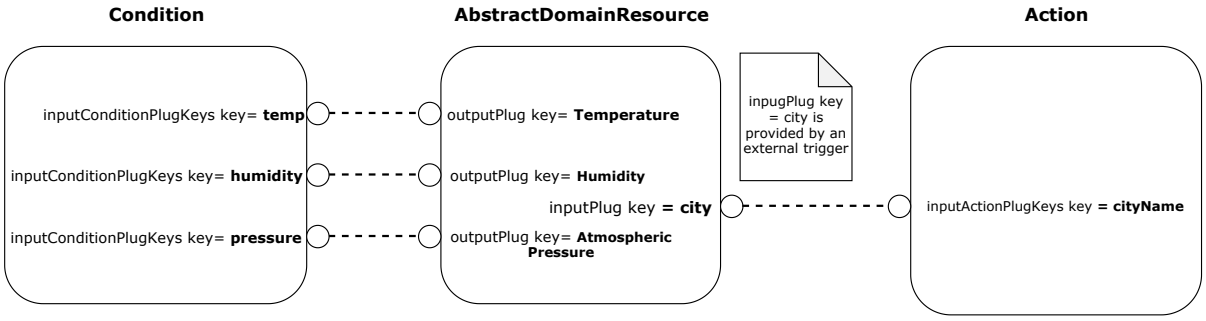


Figure 6.5: Mapper Diagram

6.4.4 Condition and Action

Listing 6.10 depicts the Condition model used in our Winter Notification example. Here, the Condition element has a reference to the goal model which is used to evaluate conditions. The resultType attribute specifies the fuzzy reasoning approach is employed for condition model reasoning e.g. boolean, fuzzy, probabilistic. Condition also contains three elements, namely temp, humidty and pressure which correspond to the fields in the response of AbstractDomainResource. The Condition specification see Listing 6.10 also references through the element ConditionGoalModel, a goal modle to be evaluated for this Condition. Note that the association between the inputConditionPlugKeys with keys "temp", "humidity" and "pressure" follows the keys in outputPlug elements in Listing 6.7.

```
<?xml version="1.0" encoding="UTF-8"?>
<iot:Condition xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:iot="http://org.eclipse.example/iot"
  conditionGoalModel="instances/condition.goal" resultType="Fuzzy">
   <inputConditionPlugKeys key="temp"
       semanticsReference="http://www.openweathermap.com#Temp"/>
   <inputConditionPlugKeys key="humidity"
       semanticsReference="http://www.openweathermap.com#Humidity"/>
   <inputConditionPlugKeys key="pressure"
       semanticsReference="http://www.openweathermap.com#Pressure"/>
</iot:Condition>
```

Listing 6.10: Condition Example

Listing 6.11 depicts the Action model for the example. Similar to the Condition

model, Action also has a reference to a goal model, i.e. task model. However, this goal model is utilized to compile possible action plans [27]. The triggerInput attribute in the Action corresponds to the resultType in the Condition. Here, Action only contains one element *cityName* which corresponds to the city attribute in inputPlug element of AbstractDomainResource see Listing 6.5.

```
<?xml version="1.0" encoding="UTF-8"?>
<iot:Action xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:iot="http://org.eclipse.example/iot"
  actionGoalModel="instances/action.goal" triggerInput="Fuzzy">
   <inputActionPlugKeys key="cityName"
      semanticsReference="http://www.openweathermap.com#CityName"/>
</iot:Action>
```

Listing 6.11: Action Example

6.4.5 Condition Goal Model and Task Goal Model

Listing 6.12 demonstrates a Condition Goal Model used in the Winter Notification Example, and it is visualized in Figure 6.6. As it is shown, the root goal is “WinterTime”. It is a composite goal which is decomposed into two sub-goals, namely “LowHumidityOrHighPressure” and “LowTemperature”. The former is also a composite goal which is further decomposed into two atomic goals, namely “HighPressure” and “LowHumidity”. The “LowTemperature” is already an atomic goal and cannot be decomposed. Each atomic goal in the ConditionGoalModel is attached to an Evaluator class which is used to evaluate the goal. For instance, Listing 6.13 shows the evaluator for “LowTemperature” goal.

```
<?xml version="1.0" encoding="UTF-8"?>
<goal:GoalModel xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:goal="http://org.eclipse.example/goal">
   <root xsi:type="goal:CompositeGoalNode" name="WinterTime">
     <hasDecomposition>
```

```
        <decomposedTo xsi:type="goal:CompositeGoalNode"
            name="LowHumidityOrHighPressure">
          <hasDecomposition decomType="OR">
            <decomposedTo xsi:type="goal:AtomicGoalNode"
                name="HighPressure" conditionPlugKey="pressure"/>
            <decomposedTo xsi:type="goal:AtomicGoalNode" name="LowHumidity"
                conditionPlugKey="humidity"/>
          </hasDecomposition>
        </decomposedTo>
        <decomposedTo xsi:type="goal:AtomicGoalNode" name="LowTemperature"
            conditionPlugKey="temp"/>
      </hasDecomposition>
    </root>
</goal:GoalModel>
```

Listing 6.12: Condition Goal Model Example

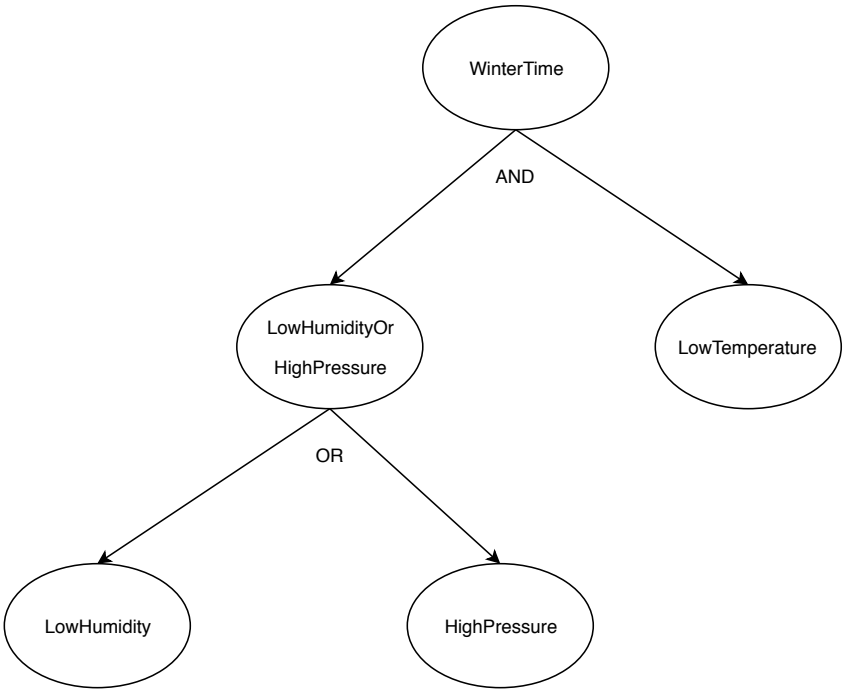


Figure 6.6: Condition Goal Model Diagram

```
public class TempEvaluator{
   String value;
   public boolean evaluate() {
      if(Double.parseDouble(value) < 32.0) return true;
      else return false;
   }
   ....
}
```

Listing 6.13: Temperature Evaluator Example

As it is mentioned above, the action can be as simple as sending a SMS message which notifies people that winter is coming. Action can also be complex enough to form an action plan. Listing A.1 demonstrates a complex Task Model used in the Winter Notification example, and it is visualized in Figure 6.7. As it shows, besides the tree structure, we also add some binary links between two nodes as discussed in Section 5.2.1. For example, there is a temporal precedence link from node T6 to node T5. And there is a resource dependency link from node A11 to R1.

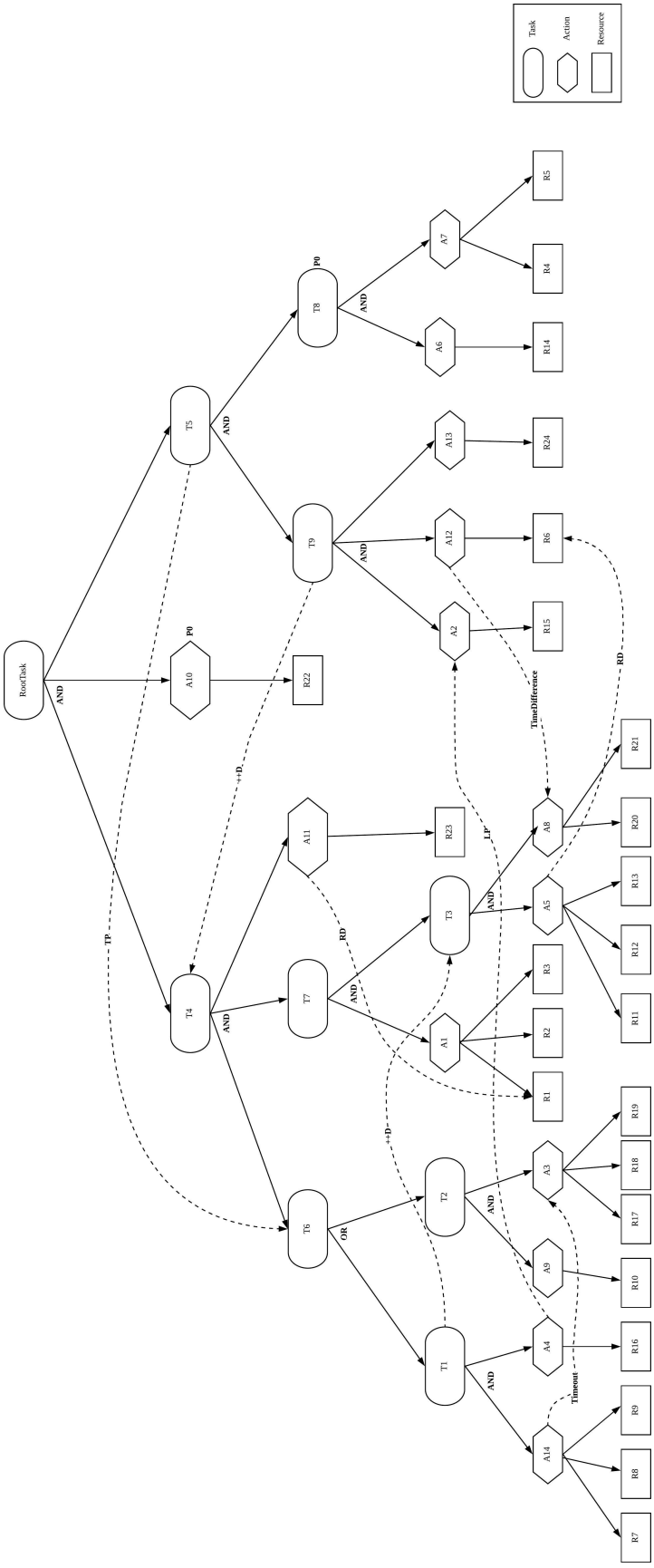


Figure 6.7: Action Model Diagram

6.4.6 Result Applying an Action Model

Suppose the root goal “WinterTime” in the Condition Goal Model is satisfied after applying fuzzy reasoning. For the sake of our example, let us consider a sample action model defined in Listing A.1 and in Figure 6.7. After applying task model reasoning using the system proposed in [27], there are several execution sequences of actions with time constraints which conform to the task model. Listing 6.14 depicts a possible action sequence that can be generated by the system proposed in [27]. The complete solution is shown in Listing B.

```
One sequence is:

   SP A5 A12 A13 P0 A4 A11 A1 A8 A2 A14 JP

   Constraints are: The max TimeDifference period between A12 and A8 is
       373ms

   One sequence for parallel P0 is:

      SP0 A10 JP0

      SP0 A7 JP0

      SP0 A6 JP0

      The nodes [A10, A7, A6] will have 323ms to synchronize
```

Listing 6.14: Action Plan Example

As is shown in Listing 6.14, the execution sequence starts at node A5 and ends at node A14. There is a parallel node P0 which allows for the parallel executions of node A10, A7 and A6. Also, the time allocated for the parallel execution is 323ms. In addition, there is a time difference constraint between node A12 and node A8, which means that A8 has to wait for at least 373ms to execute after the execution of A12.

Once such actions are executed, the output is written in a corresponding OPC UA middleware, so that if there are any other domain resources for which the input plugs can be fed by data provided by these actions, the whole process initi-

ates again (i.e. the conditions of the domain resources are evaluated and if these conditions succeed, the actions of these resources are triggered).

6.4.7 Evaluation

In this thesis the proposed programming model is applied using the Winter Notification Example case study. From this case study we obtain initial evidence that the proposed programming model can be used for IoT application development and goal-driven resource composition.

Firstly, we consider a generic definition of resources, which include not only physical IoT devices (e.g., sensors), but also other inter-networked resources such as Web resources and Web services. In this way, we overcome the limitation occurred by single data source. Moreover, we model resources at a high level of abstraction, which are referred as Abstract Domain Resources. Normally, abstract models (i.e., templates) of resources represent entities in a particular domain (e.g., Weather in the case study). The Abstract Domain Resources are instantiated to concrete resources (e.g., OpenWeatherMap in the case study) using Semantic Web technologies and Resource Selection algorithm. The performance results in Section 13 demonstrate the linear time complexity of our proposed algorithm.

Secondly, instead of utilizing a rule-based system for condition evaluation, we employ goal modeling technique to model both condition and actions. Goal models allows for expressing complex semantics (e.g., contribution link and temporal precedence), as shown in our built condition and action models. Another advantage of goal modeling is that it supports reasoning. There are quite a few off-the shelf reasoning approaches for condition and action goal models. Those approaches are proved to be efficient and easy to use. Hence, we can avoid checking an abundant rule base in the rule-based approach which is error prone. The result in the case study verifies the feasibility of both goal modeling and reasoning approaches.

Last but not least, we utilize fuzzy reasoning technique to evaluate conditions. Instead of relying on standard logic whose outcome is either true or false, the proposed system employs fuzzy logic which allows for reasoning about ambiguous concepts (e.g., LowTemperature and HighPressure in the case study). After applying fuzzy reasoning, the obtained result can be any value between 0 and 1, which assist in more accurate control and adjustment. For example, we can adjust the luminosity of light according to the conditions of the outside light, rather than turn on/off the light directly.

In summary, the case study indicates that the proposed programming model is promising for IoT application development, however more extensive evaluations should be performed. For instance, we could test the performance of the implementation when the number of resources scale up. Another thing that worth the evaluation is how our proposed work is compared to other competing approaches to IoT programming.

Chapter 7

Conclusion and Future Work

This chapter summarizes the delivered thesis work and provides ideas for the future work.

7.1 Conclusion

Internet is undergoing a major transformation leading to a new era which is known as the Internet of Things (IoT). To fully explore the potential of IoT, we need a programming model that will enable developers and end-users to easily compose and assemble IoT applications, utilizing concepts and constructs specified at a higher level of abstraction than the constructs found in general purpose programming languages. This thesis discusses the design and implementation of a framework that can serve as the basis of an IoT programming model that is founded on the Event-Condition-Action (ECA) paradigm.

The thesis addresses three major issues. The first issue deals with the design of a framework that allows for the denotation of different types of physical and cyber resources in the form of abstract entities, which we refer to as Abstract Domain Resources. An instantiation process that utilizes a utility-based function and a selection process as well as semantic web technologies, such as ontologies (OWL) linked data (RDF/S), allows for identifying and instantiating models or Abstract Domain Resources to concrete resources, we refer to as Domain Resources (or Concrete Resources). The instantiation process is based on Dynamic Programming for selecting optimal concrete resources for instantiating an abstract resource to a concrete one, given a collection of utility parameters such as cost, latency, accuracy, reliability and availability. In this respect, a model of an Abstract Domain Re-

source (e.g. the concept of a "banking account" as a resource) can be instantiated to a concrete resource (e.g. "Henry's HSBC savings banking account") according to the operational context in each system session and the user's profile.

The second issue this thesis addresses, is the definition of an event-driven architecture that is based on the collective composition of Abstract Domain Resources, Condition models and Action models, and the use of publish-subscribe middleware frameworks. The conceptual architecture is founded on seven layers (see Figure 3.1), and aims to decouple the run-time environment from the modeling and evaluation processes that are related to the Conditions and Actions in any given session. The architecture also decouples the run-time environment from the acquisition of data from external sources (e.g. the Web) through the use of Facades and Proxies. In the proposed concrete architecture (see Figure 3.2), a Daemon Module is responsible for issuing requests (i.e. invocations) and processing responses for the integration of the proposed system with external data sources and service providers.

The third issue is the design and development of a run-time environment as a proof of concept of the proposed programming model. For this thesis we have used the Eclipse Modeling Framework as the underlying meta-modeling foundation and the Open Platform Communications Unified Architecture (OPC UA) as the middleware of choice to build and test the prototype. OPC UA is a lightweight event-based middleware which employs a client server approach. For experimentation purposes we have also designed adapters with the PADRES distributed publish-subscribe middleware. In the run-time environment, the different software components communicate with each other by sending events and receiving events in the event channels. The prototype run-time environment indicates that the approach is feasible and can be extended so that it can be deployed to real-life applications.

7.2 Future Work

This thesis tackles the IoT programming model question by investigating the design and implementation of a prototype system. In this respect, there is a number of different directions which can be taken to extend the work presented in this thesis.

An interesting future direction is to employ other middleware solutions for the proposed programming model. While OPC UA is a topic-based publish/subscribe middleware, there is another category of publish/subscribe middleware which is

known as content-based. The major distinction lies in that how event subscribers express their interests in events. Content-based middleware usually allows more flexibility for the event filtering in that events are classified based on their properties. In OPC UA, events are classified according to topic names, such as temperature, humidity, pressure etc. It is worthwhile to explore the filtering mechanism in a content-based publish/subscribe middleware for the programming model.

The second possible issue to investigate is the use of virtualization and microservices to tackle issues related to scalability which poses an important requirement for real-life applications and deployments. As a result, we need to consider much larger computing system infrastructures that are based on container technology. In a real world scenario, there may be thousands of events producing and consuming at the same time. Also, IoT devices typically collect a tremendous amount of data. It is therefore necessary to consider distributed publish-subscribe environments as well as new deployment strategies so that distributed and complex event processing and scalable evaluation of condition and action models can be achieved. One way to solve this issue is adopting cloud computing and microservices technology.

A third possible avenue of work is to consider how security is incorporated in the current model. The architecture allows for authentication and proxy modules but these need to be defined and linked to particular security and cryptography solutions. Furthermore, it is important for the system to be able to protect an individual's privacy and business secrets in such a pervasive and inter-connected environment. When designing a programming model and middleware for IoT, we need to take into account additional security issues, such as access control, user authorization and data provenance, to name a few.

Lastly, a fourth possible direction is to investigate the application of data integration techniques. As we mentioned earlier, data may come from various information sources. In this thesis, we select only one source with the highest utility value based on QoS metrics. However, there may be situations where data from different sources have to be combined in order to be useful. This poses the question of investigating techniques for data and schema integration as a necessary step for IoT application development.

Concluding, we say that we embark a new and exciting era of the use of Internet Technologies, which has the potential to create very useful everyday applications for the benefit of the users and the public. Smart cities, health care, and Industry 4.0 applications may be only the beginning of this new era.

Bibliography

[1] Address space concepts. https://documentation.unified-automation.com/uasdkhp/1.0.0/html/_l2_ua_address_space_concepts.html. Accessed: 2019-03-25.

[2] Arq - a sparql processor for jena. https://jena.apache.org/documentation/query/. Accessed: 2019-03-20.

[3] Current weather data. https://openweathermap.org/current. Accessed: 2019-03-25.

[4] Eclipse milo. https://projects.eclipse.org/projects/iot.milo. Accessed: 2019-03-25.

[5] Eclipse modeling framework (emf). https://www.eclipse.org/modeling/emf/. Accessed: 2019-03-25.

[6] A free, open-source ontology editor and framework for building intelligent systems. https://protege.stanford.edu/. Accessed: 2019-03-20.

[7] Homeassistant. https://www.home-assistant.io/. Accessed: 2019-03-25.

[8] ifttt. https://ifttt.com/. Accessed: 2019-03-25.

[9] The internet of everything global public sector economic analysis. https://www.cisco.com/c/dam/en_us/about/business-insights/docs/ioe-value-at-stake-public-sector-analysis-faq.pdf. Accessed: 2019-03-25.

[10] Internet of things at a glance. https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45-731471.pdf. Accessed: 2019-03-25.

[11] Introducing the knowledge graph: things, not strings. https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html. Accessed: 2019-03-20.

[12] Jsonpath. https://github.com/json-path/JsonPath. Accessed: 2019-03-25.

[13] Learn sparql. https://www.cambridgesemantics.com/blog/semantic-university/learn-sparql/. Accessed: 2019-03-20.

[14] Meta object facility. https://www.omg.org/spec/MOF/. Accessed: 2019-03-25.

[15] Node-red. https://nodered.org/. Accessed: 2019-03-25.

[16] Opc foundation and microsoft: Accelerating the future of manufacturing. https://cloudblogs.microsoft.com/industry-blog/manufacturing/2017/03/21/microsoft-and-opc-foundation-accelerating-the-future-of-manufacturing/. Accessed: 2019-03-25.

[17] openweathermap. https://openweathermap.org/. Accessed: 2019-03-25.

[18] programmableweb. https://www.programmableweb.com/category/mapping/api. Accessed: 2019-03-25.

[19] pub-sub-messaging. https://aws.amazon.com/cn/pub-sub-messaging/. Accessed: 2019-03-25.

[20] Qudt ontologies overview. http://www.qudt.org/pages/QUDToverviewPage.html. Accessed: 2019-03-20.

[21] Smartrules. http://smartrulesapp.com/. Accessed: 2019-03-25.

[22] Yeah! we did it again ;) – new 2016-04 dbpedia release. https://blog.dbpedia.org/2016/10/19/yeah-we-did-it-again-new-2016-04-dbpedia-release/. Accessed: 2019-03-20.

[23] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Global sensor networks. *EPFL, Lausanne, Tech. Rep*, 2006.

[24] OSGi Alliance. *Osgi service platform, release 3*. IOS press, 2003.

[25] Daniel Amyot. Introduction to the user requirements notation: learning by example. *Computer Networks*, 42(3):285–301, 2003.

[26] Kevin Ashton et al. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

[27] Michalis Bachras. Model-based management of the internet of things infrastructure. Master's thesis, Athens Greece, 2018.

[28] M Bansal and V Venkaiah. Improved fully polynomial time approximation scheme for the 0-1 multiple-choice knapsack problem.

[29] Sean Bechhofer, Frank Van Harmelen, Jim Hendler, Ian Horrocks, Deborah L McGuinness, Peter F Patel-Schneider, Lynn Andrea Stein, et al. Owl web ontology language reference. *W3C recommendation*, 10(02), 2004.

[30] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. Rdf 1.1 turtle. *World Wide Web Consortium*, 2014.

[31] Shiddartha Raj Bhandari and Neil W Bergmann. An internet-of-things system architecture based on services and events. In *2013 IEEE Eighth International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pages 339–344. IEEE, 2013.

[32] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: The story so far. In *Semantic services, interoperability and web applications: emerging concepts*, pages 205–227. IGI Global, 2011.

[33] Will Brackenbury, Abhimanyu Deora, Jillian Ritchey, Jason Vallee, Weijia He, Guan Wang, Michael L Littman, and Blase Ur. How users interpret bugs in trigger-action programming. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, page 552. ACM, 2019.

[34] Dan Brickley, Ramanathan V Guha, and Brian McBride. Rdf schema 1.1. *W3C recommendation*, 25:2004–2014, 2014.

[35] Julio Cano, Eric Rutten, Gwenaël Delaval, Yazid Benazzouz, and Levent Gurgen. Eca rules for iot environment: a case study in safe design. In *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 116–121. IEEE, 2014.

[36] George Chatzikonstantinou, Michael Athanasopoulos, and Kostas Kontogiannis. Towards a goal driven task personalization specification framework. In *2013 IEEE Ninth World Congress on Services*, pages 180–184. IEEE, 2013.

[37] George Chatzikonstantinou, Michael Athanasopoulos, and Kostas Kontogiannis. Task specification and reasoning in dynamically altered contexts. In *International Conference on Advanced Information Systems Engineering*, pages 625–639. Springer, 2014.

[38] George Chatzikonstantinou and Kostas Kontogiannis. Run-time requirements verification for reconfigurable systems. *Information and Software Technology*, 75:105–121, 2016.

[39] George Chatzikonstantinou, Kostas Kontogiannis, and Ioanna-Maria Attarian. A goal driven framework for software project data analytics. In *International Conference on Advanced Information Systems Engineering*, pages 546–561. Springer, 2013.

[40] Amit K Chopra, Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Reasoning about agents and protocols via goals and commitments. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 457–464. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[41] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

[42] Michael Compton, Payam Barnaghi, Luis Bermudez, RaúL GarcíA-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, et al. The ssn ontology of the w3c semantic sensor network incubator group. *Web semantics: science, services and agents on the World Wide Web*, 17:25–32, 2012.

[43] World Wide Web Consortium et al. Rdf 1.1 concepts and abstract syntax. 2014.

[44] Luca Corcella, Marco Manca, Fabio Paternò, and Carmen Santoro. A visual tool for analysing iot trigger/action programming. In *International Con-*

ference on Human-Centred Software Engineering, pages 189–206. Springer, 2018.

[45] Joëlle Coutaz and James L Crowley. A first-person experience with end-user development for smart homes. *IEEE Pervasive Computing*, 15(2):26–39, 2016.

[46] Umeshwar Dayal, Barbara Blaustein, Alex Buchmann, Upen Chakravarthy, Meichun Hsu, R Ledin, Dennis McCarthy, Arnon Rosenthal, Sunil Sarin, Michael J. Carey, et al. The hipac project: Combining active databases and timing constraints. *ACM Sigmod Record*, 17(1):51–70, 1988.

[47] Manuel Díaz, Cristian Martín, and Bartolomé Rubio. State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. *Journal of Network and Computer applications*, 67:99–117, 2016.

[48] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[49] Roy Thomas Fielding. Rest: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California*, 2000.

[50] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. Personalization of context-dependent applications through trigger-action rules. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 24(2):14, 2017.

[51] Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani. Reasoning with goal models. In *International Conference on Conceptual Modeling*, pages 167–181. Springer, 2002.

[52] Aitor Gómez-Goiri and Diego López-de Ipiña. A triple space-based semantic distributed middleware for internet of things. In *International Conference on Web Engineering*, pages 447–458. Springer, 2010.

[53] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[54] Dominique Guinard and Vlad Trifa. *Building the web of things: with examples in node. js and raspberry pi*. Manning Publications Co., 2016.

[55] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the internet of things to the web of things: Resource-oriented architecture and best practices. In *Architecting the Internet of things*, pages 97–129. Springer, 2011.

[56] Hans-Arno Jacobsen, Alex Cheung, Guoli Li, Balasubramaneyam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. The padres publish/subscribe system. In *Principles and Applications of Distributed Event-Based Systems*, pages 164–205. IGI Global, 2010.

[57] Gupta M.M. Kaufmann, A. *Introduction to fuzzy arithmetic: theory and applications*. Van Nostrand Reinhold Company, 1985.

[58] Maxim Kolchin, Nikolay Klimov, Ivan Shilin, Daniil Garayzuev, Alexey Andreev, and Dmitry Mouromtsev. Semiot: an architecture of semantic internet of things middleware. In *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 416–419. IEEE, 2016.

[59] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[60] Stefan-Helmut Leitner and Wolfgang Mahnke. Opc ua–service-oriented architecture for industrial applications. *ABB Corporate Research Center*, 48:61–66, 2006.

[61] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P King, and Richard Franck. Web service level agreement (wsla) language specification. *Ibm corporation*, pages 815–824, 2003.

[62] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[63] Mahdi H Miraz, Maaruf Ali, Peter S Excell, and Rich Picking. A review on internet of things (iot), internet of everything (ioe) and internet of nano things (iont). In *2015 Internet Technologies and Applications (ITA)*, pages 219–224. IEEE, 2015.

[64] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[65] Chandrakana Nandi and Michael D Ernst. Automatic trigger generation for rule-based smart homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 97–102. ACM, 2016.

[66] Julie L Newcomb, Satish Chandra, Jean-Baptiste Jeannin, Cole Schlesinger, and Manu Sridharan. Iota: a calculus for internet of things automation. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 119–133. ACM, 2017.

[67] Natalya F Noy, Deborah L McGuinness, et al. Ontology development 101: A guide to creating your first ontology, 2001.

[68] Cezary Orłowski, Artur Ziółkowski, Aleksander Orłowski, Paweł Kapłański, Tomasz Sitek, and Witold Pokrzywnicki. Ontology of the design pattern language for smart cities systems. In *Transactions on Computational Collective Intelligence XXV*, pages 76–100. Springer, 2016.

[69] Gerardo Pardo-Castellote. Omg data-distribution service: Architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206. IEEE, 2003.

[70] James Pasley. How bpel and soa are changing web services development. *IEEE Internet Computing*, 9(3):60–67, 2005.

[71] Santiago Pericas-Geertsen and Marek Potociar. Jax-rs: Java™ api for restful web services. *Oracle Corporation*, pages 1–84, 2013.

[72] Peter R Pietzuch. Hermes: A scalable event-based middleware. Technical report, University of Cambridge, Computer Laboratory, 2004.

[73] Peter R Pietzuch and Jean M Bacon. Hermes: A distributed event-based middleware architecture. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pages 611–618. IEEE, 2002.

[74] Eric Prud, Andy Seaborne, et al. Sparql query language for rdf. 2006.

[75] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[76] John Soldatos, Nikos Kefalakis, Manfred Hauswirth, Martin Serrano, Jean-Paul Calbimonte, Mehdi Riahi, Karl Aberer, Prem Prakash Jayaraman, Arkady Zaslavsky, Ivana Podnar Žarko, et al. Openiot: Open source internet-of-things in the cloud. In *Interoperability and open-source solutions for the internet of things*, pages 13–25. Springer, 2015.

[77] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[78] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 803–812. ACM, 2014.

[79] Michael Uschold and Michael Gruninger. Ontologies and semantics for seamless connectivity. *ACM SIGMod Record*, 33(4):58–64, 2004.

[80] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings fifth ieee international symposium on requirements engineering*, pages 249–262. IEEE, 2001.

[81] Axel Van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on software engineering*, 26(10):978–1005, 2000.

[82] Holger Wache, Thomas Voegele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. Ontology-based integration of information-a survey of existing approaches. In *IJCAI-01 workshop: ontologies and information sharing*, volume 2001, pages 108–117. Citeseer, 2001.

[83] Eric Yu. Modelling strategic relationships for process reengineering. *Social Modeling for Requirements Engineering*, 11:2011, 2011.

[84] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the internet of things. *IEEE access*, 6:6900–6919, 2017.

Appendix A

Task Model Example

```
<?xml version="1.0" encoding="UTF-8"?>
<goal:TaskModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:goal="http://org.eclipse.example/goal" name="TaskModel1">
<root xsi:type="goal:Task" name="root">
<hasDecomposition>
<decomposedTo xsi:type="goal:Task" name="T4">
<hasDecomposition>
<decomposedTo xsi:type="goal:Task" name="T6">
<hasDecomposition decomType="OR">
<decomposedTo xsi:type="goal:Task" name="T1">
<hasDecomposition>
<decomposedTo xsi:type="goal:Action" name="A14">
<requires name="R7"/>
<requires name="R8"/>
<requires name="R9"/>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A4">
<requires name="R16"/>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
<decomposedTo xsi:type="goal:Task" name="T2">
<hasDecomposition>
<decomposedTo xsi:type="goal:Action" name="A9">
<requires name="R10"/>
```

```
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A3">
<requires name="R17"/>
<requires name="R18"/>
<requires name="R19"/>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
<decomposedTo xsi:type="goal:Task" name="T7">
<hasDecomposition>
<decomposedTo xsi:type="goal:Task" name="T3">
<hasDecomposition>
<decomposedTo xsi:type="goal:Action" name="A5">
<requires name="R11"/>
<requires name="R12"/>
<requires name="R13"/>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A8">
<requires name="R20"/>
<requires name="R21"/>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A1">
<requires name="R1"/>
<requires name="R2"/>
<requires name="R3"/>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A11">
<requires name="R23"/>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
<decomposedTo xsi:type="goal:Task" name="T5">
```

```
<hasDecomposition>
<decomposedTo xsi:type="goal:Task" name="T8">
<hasDecomposition>
<decomposedTo xsi:type="goal:Action" name="A7">
<requires name="R4"/>
<requires name="R5"/>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A6">
<requires name="R14"/>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
<decomposedTo xsi:type="goal:Task" name="T9">
<hasDecomposition>
<decomposedTo xsi:type="goal:Action" name="A12">
<requires name="R6"/>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A2">
<requires name="R15"/>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A13">
<requires name="R24"/>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
</hasDecomposition>
</decomposedTo>
<decomposedTo xsi:type="goal:Action" name="A10">
<requires name="R22"/>
</decomposedTo>
</hasDecomposition>
</root>
<contains xsi:type="goal:ContributionLink"
    from="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/@decomposedTo.1"
    to="//@root/@hasDecomposition/@decomposedTo.0" contrType="PPD"/>
<contains xsi:type="goal:LogicalPrecedence"
    from="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.0/
```

```
@hasDecomposition/@decomposedTo.0/@hasDecomposition/@decomposedTo.1"
    to="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/@decomposedTo.1
/@hasDecomposition/@decomposedTo.1"/>
<contains xsi:type="goal:ContributionLink"
    from="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.0/@hasDecomposition/@decomposedTo.0"
    to="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.1/@hasDecomposition/@decomposedTo.0" contrType="PPD"/>
<contains xsi:type="goal:ResourceDependency"
    from="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.1/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.0"
    to="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/
@decomposedTo.1/@hasDecomposition/@decomposedTo.0/@requires.0"/>
<contains xsi:type="goal:ResourceDependency"
    from="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.2"
    to="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.1/@hasDecomposition/@decomposedTo.1/@requires.0"/>
<contains xsi:type="goal:TimeoutLink"
    from="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.0/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.0"
    to="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.0/@hasDecomposition/@decomposedTo.1/@hasDecomposition/
@decomposedTo.1"/>
<contains xsi:type="goal:TimeDifferenceLink"
    from="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/
@decomposedTo.1/@hasDecomposition/@decomposedTo.0"
    to="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.1/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.1" timeDifference="373"/>
<contains xsi:type="goal:TemporalPrecedence"
    from="//@root/@hasDecomposition/@decomposedTo.1"
    to="//@root/@hasDecomposition/@decomposedTo.0/@hasDecomposition/
@decomposedTo.0"/>
<parallel name="P0" timeout="323"
    preNode="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/
```

```
@decomposedTo.0 //@root/@hasDecomposition/@decomposedTo.2"/>
<parallel name="P0" timeout="323"
    preNode="//@root/@hasDecomposition/@decomposedTo.1/@hasDecomposition/
@decomposedTo.0 //@root/@hasDecomposition/@decomposedTo.2"/>
</goal:TaskModel>
```

Listing A.1: Task Model Example [27]

Appendix B

Experiment Result Example

```
One sequence is:

SP A5 A12 A13 P0 A4 A11 A8 A1 A2 A14 A3 A9 JP

Constraints are: Timeout period between A14 and A3 is 0,
TimeDifference period between A12 and A8 is 373

One sequence for parallel P0 is:

SP0 A10 JP0

SP0 A7 JP0

SP0 A6 JP0

The nodes [A10, A7, A6] will have 323 to synchronize


One sequence is:

SP A5 A12 A8 A13 A2 P0 A9 A11 A1 A3 JP

Constraints are: TimeDifference period between A12 and A8 is 373

One sequence for parallel P0 is:
```

```
SP0 A10 JP0

SP0 A7 JP0

SP0 A6 JP0


The nodes [A10, A7, A6] will have 323 to synchronize


One sequence is:

SP A5 A12 A13 P0 A4 A11 A1 A8 A2 A14 JP

Constraints are: TimeDifference period between A12 and A8 is 373

One sequence for parallel P0 is:

SP0 A10 JP0

SP0 A7 JP0

SP0 A6 JP0


The nodes [A10, A7, A6] will have 323 to synchronize
```

Listing B.1: Experiment Result Example [27]

Curriculum Vitae

Name:	Hao Jiang
Post-Secondary Education and Degrees:	Jinan University Guangzhou, Guangdong, China 2010 - 2014 B.E. Western University (University of Western Ontario) London, ON, Canada 2017 - 2019 M.Sc.
Honours and Awards:	National Scholarship, China 2012-2013
Related Work Experience:	Teaching Assistant Western University 2017 - 2019 Research Assistant Western University 2017 - 2018