Enabling High-Level Application Development for the Internet of Things

Pankesh Patel

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Pankesh PATEL

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Environnement de développement d'applications pour l'Internet des objets

Enabling High-Level Application Development for the Internet of Things

soutenue prévue le 25 Novembre 2013

devant le jury composé de :

Dr. Michel BANÂTRE	Rapporteur
Pr. Apostolos ZARRAS	Rapporteur
Dr. Olivier BARAIS	Examinateur
Pr. Maria POTOP-BUTUCARU	Examinateur
Dr. Valérie ISSARNY	Directrice de thèse
Dr. Animesh PATHAK	Co-Directeur de thèse

Abstract

Application development in the Internet of Things (IoT) is challenging because it involves dealing with a wide range of related issues such as lack of separation of concerns, and lack of high-level of abstractions to address both the large scale and heterogeneity. Moreover, stakeholders involved in the application development have to address issues that can be attributed to different life-cycles phases when developing applications. First, the application logic has to be analyzed and then separated into a set of distributed tasks for an underlying network. Then, the tasks have to be implemented for the specific hardware. Apart from handling these issues, they have to deal with other aspects of life-cycle such as changes in application requirements and deployed devices.

Several approaches have been proposed in the closely related fields of wireless sensor network, ubiquitous and pervasive computing, and software engineering in general to address the above challenges. However, existing approaches only cover limited subsets of the above mentioned challenges when applied to the IoT. This thesis proposes an integrated approach for addressing the above mentioned challenges. The main contributions of this thesis are: (1) a development methodology that separates IoT application development into different concerns and provides a conceptual framework to develop an application, (2) a development framework that implements the development methodology to support actions of stakeholders. The development framework provides a set of modeling languages to specify each development concern and abstracts the scale and heterogeneity related complexity. It integrates code generation, task-mapping, and linking techniques to provide automation. Code generation supports the application development phase by producing a programming framework that allows stakeholders to focus on the application logic, while our mapping and linking techniques together support the deployment phase by producing device-specific code to result in a distributed system collaboratively hosted by individual devices. Our evaluation based on two realistic scenarios shows that the use of our approach improves the productivity of stakeholders involved in the application development.

Table of Contents

List of Figures

List of Tables

Chapter I

Introduction

> *"The* 19^{th} *century was a century of empire, the* 20^{th} *century was a century of nation states and the* 21^{st} *century will be a century of cities."*
>
> - Former Denver Mayor W. Webb

Cities are the future of humankind, with more than 50% of the world's population now living in cities [Harrison and Donnelly, 2011]. By 2050, the UN predicts this number will increase to 70% due to massive economical growth in the current urban areas [Nations, 2010]. Some of this growth will be in 27 mega cities with greater than 10 million people, but more than this half of the growth will be in cities that have fewer than 500,000 people [Naphade et al., 2011]. This urban growth and migration are putting significant stress on city infrastructure and raising challenges in domains including energy, transportation, healthcare, safety, and security of citizens [Haubensak, 2011].

A possible solution of the above challenges may be in recent technological advances in computer and communication technology. These advances have been fueling a tremendous growth in the number of smart objects (or *things*) [Vasseur and Dunkels, 2010, p. 3]. In 2010, the number of everyday physical objects was around 12.5 billion. Cisco forecasts that this number is expected to double to 25 billion in 2015 as the number of smart devices per person increase, and to a further 50 billion by 2020 [Evans, 2011]. Figure I.1 illustrates the electronic skin of the city consisting of millions of smart objects: temperature sensors, lights, smoke detectors, smart phones, fire alarms, air pollution controllers, car, parking space controllers, etc. These smart objects sense the physical world by obtaining information from sensors, affect the physical world by triggering actions using actuators, engage users by interacting with them whenever necessary, and process captured data and communicate it to outside world.

As a means to realize the above vision, the *Internet of Things* (IoT) enables a variety of physical objects or things – such as sensors, actuator, mobile phones, etc. – to communicate with each other and cooperate with their neighbors to reach a common goal [Atzori et al., 2010].

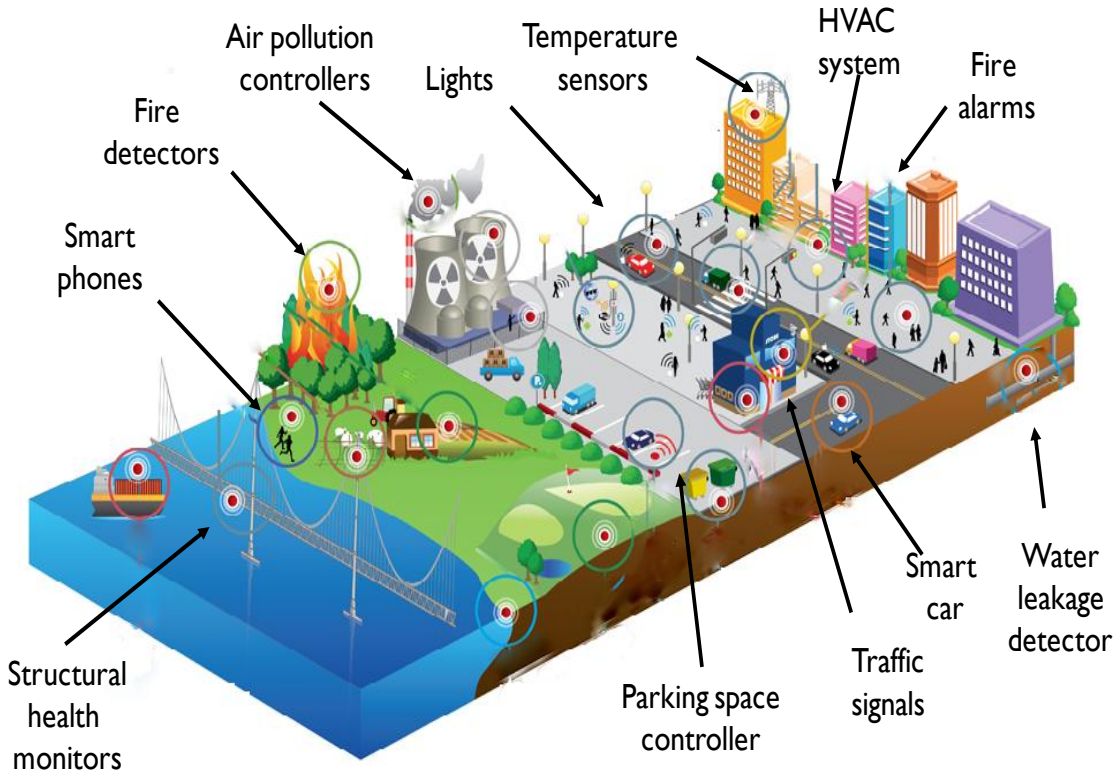

Figure I.1 – Smart objects in urbanized areas (figure credit : http://www.libelium.com/)

Though the precise definition of the IoT is still evolving, we base our work on the definition below, proposed by the CASAGRAS project [1] :

> "*[The Internet of Things is a] global network infrastructure, linking physical and virtual objects through the exploitation of data capture and communication capabilities. This infrastructure includes existing and evolving Internet and network developments. It will offer specific object-identification, sensor and connection capability as the basis for the development of independent cooperative services and applications. These will be characterized by a high degree of autonomous data capture, event transfer, network connectivity and interoperability.*"

In the IoT, "things" acquire intelligence thanks to the fact that they access information that has been aggregated by other things. For example, a building interacts with its residents and surrounding buildings in case of fire for safety and security of residents, offices adjust themselves automatically accordingly to user preferences while minimizing energy consumption, or traffic signals control in-flow of vehicles according to the current highway status [de Saint-Exupery,

1. http://www.grifs-project.eu/data/File/CASAGRAS%20FinalReport%20(2).pdf

2009].

As evident above, IoT applications will involve interactions among extremely large numbers of disparate devices, many of them directly interacting with their physical surroundings. An important challenge that needs to be addressed in the IoT, therefore, is to enable the rapid development of IoT applications with minimal effort by the various stakeholders involved in the process. Similar challenges have already been addressed in the closely related fields of Wireless Sensor Networks (WSNs) [Vasseur and Dunkels, 2010, p. 11] and ubiquitous and pervasive computing [Vasseur and Dunkels, 2010, p. 7], regarded as precursors to the modern day IoT. While the main challenge in the former is the *large scale* – hundreds to thousands of largely similar devices, the primary concern in the latter has been the *heterogeneity* of devices and the major role that the user's own interaction with these devices plays in these systems (cf. the classic "smart home" scenario where a user controls lights and receives notifications from his refrigerator and toaster.) It is the goal of our work to enable the development of such applications. In the following, we discuss one of such applications.

1 Application example

As discussed above, smart cities present an excellent platform for the execution of IoT applications. We discuss below the details of one such application, many of which will come together to empower smart cities. We first give a general idea about the domain, followed by the details of the smart building application.

Building automation domain. The authors in [Vasseur and Dunkels, 2010, p. 361] say that "*Building automation is the instrumentation, mechanization, and data aggregation of a variety of discrete building systems to make monitoring and controlling of building equipment more efficient.*" This building system might consist of several buildings, with each building in turn consisting of one or more floors, each with several rooms. It may consist of a large number of heterogeneous devices equipped with sensors to sense environment, actuators to influence the environment, storage that stores the persistent information, and user interfaces through which users interact with other devices. Figure I.2 describes the building automation domain with various devices. Many applications can be developed using these devices, one of which we discuss below. Details of our applications can be found in Chapter V.

Smart building application. In 2004, building consumption in European Union was 37% of total energy, greater than 28% in industry and 32% in transport [Nguyen and Aiello, 2012]. The improvement in buildings has the potential to save a considerable amount of energy, and

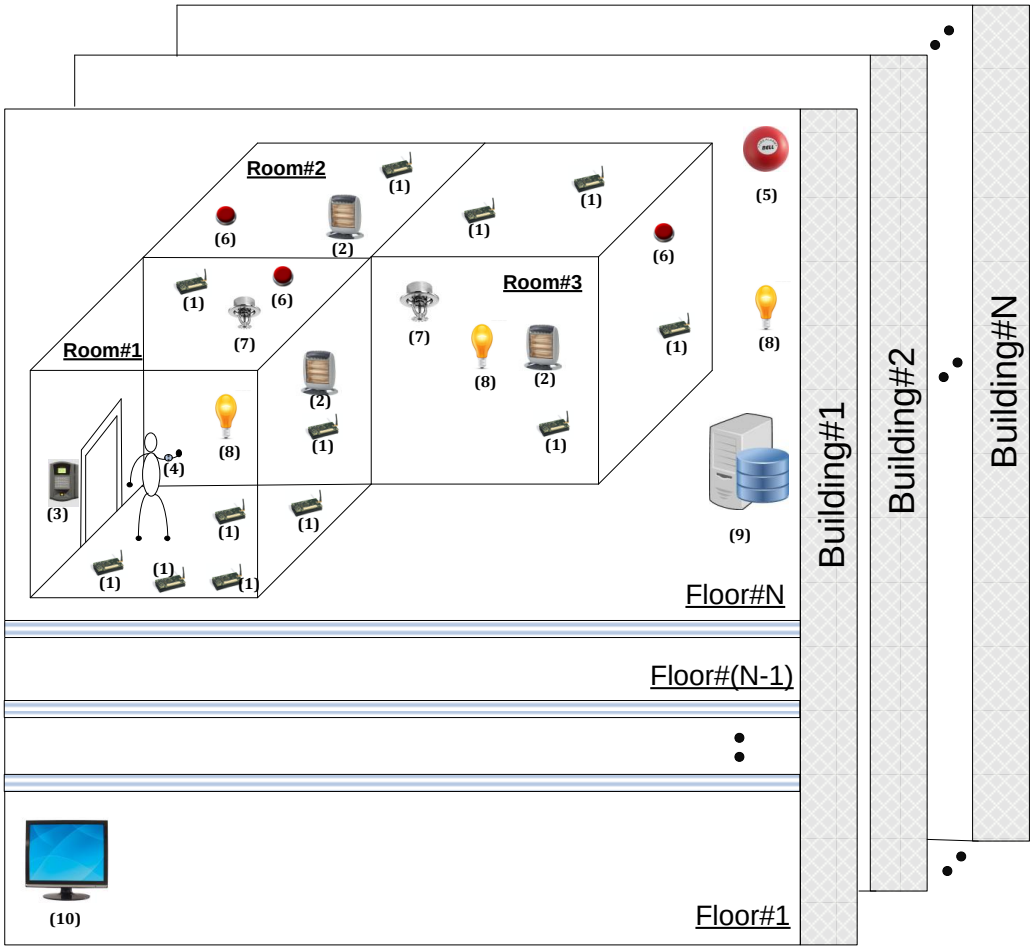

Figure I.2 – A cluster of multi-floored buildings with deployed devices with (1) temperature sensor, (2) heater, (3) badge reader, (4) badge, (5) alarm, (6) smoke detector, (7) sprinkler, (8) light, (9) data storage, and (10) monitor.

therefore money. Data shows that much of the energy consumption by commercial buildings is spent on lighting (26%), followed by heating and cooling is 13% and 14% respectively [Chen et al., 2009; Perez-Lombard et al., 2008]. Therefore, switching off lights and heating and cooling systems within a building when rooms become unoccupied would be one of the ways to save energy. Another way to save energy is to engage the residents of the building [Smith et al., 2011]. A system can generate situation awareness by displaying general information about the building such as current temperature or energy usage of the building on dashboard placed on a central location of a building. It can thus encourage residents to save energy to reduce the overall energy demand [Chen et al., 2009].

We consider a hypothetical building utilized by a company. To accommodate an increasingly mobile work force, the company has designated its sites for mobile workers. A mobile worker

does not have a pre-assigned office. On a typical day, he finds a convenient room based on personal preferences such as the amount of sunlight available in the room, or the proximity to other employees working on the same project. To accommodate the mobile worker's preference in the reserved room, a database is used to keep the profile of each worker, including his preferred lighting and temperature level. A badge reader in the room detects the worker's entry event and queries the database for the worker's preference. Based on this, the thresholds used by the room's devices are updated. To reduce electricity waste when a person leaves the room, detected by badge disappeared event, lighting and heating level are automatically set to the lowest level; all according to the building's policy. The system may also include user interfaces that allow a late worker to control heater of his room and request the profile database to get his lighting and temperature preferences.

Moreover, the system generates the current status (e.g., temperature, energy consumption) of each room, which is then aggregated and used to determine the current status of each floor and, in turn, the entire building. A monitor installed at the building entrance presents the information to the building operator for situational awareness.

2 IoT application characteristics

This section identifies the characteristics of IoT applications, as gleaned from our analysis of applications such as the one discussed in the previous section, and provides a necessary basis for discussing the challenges in IoT application development. While most of these characteristics are present in all software applications and have been identified and analyzed in software engineering literature, the IoT context does bring in some peculiarities that have not been addressed in recent work. In our view, IoT applications have the following characteristics:

Commonality at various levels. We note that there are a significant amount of common features between different IoT systems. These arise from commonality between the different *deployments* of the same application (e.g., the same building management system deployed across two companies' offices), and between different applications in the same *domain* (e.g., two applications in the building automation domain). A domain manifests itself in applications concerned with similar entities of interest. An entity of interest is an object (e.g., room, book, plant), including the attributes that describe it, and its state that is relevant from a user or an application perspective [Haller, 2010, p. 1]. For instance, the building automation domain contains entities of interest such as rooms, floors, and buildings. This forms the basis for breaking down development effort into concerns and responsibilities of stakeholders, which underpin our

approach.

Multiple concerns. An IoT application may involve a number of concerns: (1) domain-specific features (e.g., the smart building application reason in terms of rooms and floors, the smart city applications are expressed in terms of sectors.), (2) application-specific features (e.g., regulating temperature, fire detection), (3) operating system-specific features (e.g., Android-specific APIs to get data from sensors, vendor-specific database such as MySQL), and (4) deployment-specific features (e.g., understanding of the specific target area where the application is to be deployed, mapping of processing components to devices in the target deployment).

Heterogeneous devices. An IoT application may execute on a network consisting of different types of devices. For example, the smart building application consists of devices, including sensing devices (e.g., temperature sensor, badge reader), actuating devices (e.g., heater, light), user interface devices (e.g., smart phone, monitor), storage devices (e.g., profile storage on different database systems such as MySQL or MongoDB).

Heterogeneous platforms. An IoT application may execute on a network with heterogeneous platforms. These platforms are operating system-specific. For instance, a device could be running Android mobile OS, Java SE on laptops, or a server OS such as GNU/Linux etc.

Heterogeneous interaction modes. The devices could be different in terms of how data can be accessed from them. The interaction mode could be one of the following:

- Publish/Subscribe: It provides subscribers with the ability to express their interest in an event, generated by a publisher, that matches their registered interest [Eugster et al., 2003]. For instance, a badge detect event is notified to subscribers, when a user enters into a room.
- Request/Response: A request is a one-to-one interaction with a return value. In order to fetch data, a requester sends a request message containing an access parameter to a responder. The responder receives and processes the request message, ultimately returns an appropriate message as a response [Berson, 1996]. For instance, in the smart building application, profile data is retrieved by querying the profile storage.
- Command: This is a unidirectional mode of communication, an instance of message passing [Andrews, 1991], used for controlling actuators. In contrast to request/response, no values are returned from the receiving software component. For instance, in the smart building application, a "set temperature" command is given to regulate a temperature level in a room.

Scale. An IoT application may execute on a network consisting of hundreds to thousands of devices, and its goal is usually obtained by composing multiple activities. For instance,

temperature values are computed at per-room and then per-floor levels to calculate an average temperature value of a building.

Evolution. Over time, IoT applications may need to be modified in response to the changes in the application requirements or the underlying hardware. When this happens, all or parts of the applications need to be re-developed. For instance, it could be changes in functionality of an application (e.g., the smart building application is extended by including fire detection functionality), addition and replacement of devices (e.g., more temperature sensors are added to sense accurate temperature values in the building), technological advances with new software features (e.g., the smart building application is extended by providing support for Android devices), or changes in distribution of target devices (e.g., moving temperature sensing devices from one floor to other).

3 IoT application development challenges

This section reviews the application development challenges due to the IoT application characteristics discussed in the previous section. As earlier, we note that although the software engineering community has discussed and analyzed similar challenges in the general case, this has not been applied to the case of IoT in particular. The challenges we address in this work are as follows:

Lack of division of roles. IoT application development is a multi-disciplined process where knowledge from multiple concerns intersects. Traditional IoT application development assumes that the individuals involved in the application development have similar skills. This is in clear conflict with the varied set of skills required during the process, including domain expertise, deployment-specific knowledge, application design and implementation knowledge, and platform-specific knowledge, a challenge recognized by recent works such as [Chen et al., 2012; Picco, 2010].

Heterogeneity. IoT applications execute on a network consisting of heterogeneous devices in terms of types, interaction modes, as well as different platforms. The heterogeneity largely spreads into the application code and makes the portability of code to a different deployment difficult. Ideally, the same application should execute on a different deployment (e.g., the same smart building application on different offices with different devices).

Scale. As mentioned above, IoT applications execute on distributed systems consisting of hundreds to thousands of devices, involving the coordination of their activities. Requiring the ability of reasoning at such levels of scale is impractical in general, as has been largely the view in the

WSN community. Consequently, there is a need of adequate abstractions that allow stakeholders to express their requirements in a compact manner regardless of the scale.

Different life cycle phases. Stakeholders have to address issues that are attributed to different life cycles phases, including development, deployment, and maintenance [Bischoff and Kortuem, 2007] illustrated in Figure I.3. At the **development phase**, the application logic has to be analyzed and separated into a set of distributed tasks for the underlying network consisting of a large number of heterogeneous entities. Then, the tasks have to be implemented for the specific platform of a device. At the **deployment phase**, the application logic has to be deployed onto a large number of devices. Apart from handling these issues, stakeholders have to keep in mind evolution issues both in the development (e.g., change in functionality of an application) and deployment phase (e.g. adding, removing devices in deployment scenarios) at the **maintenance phase**. Manual effort in all above three phases for hundreds to thousands of heterogeneous devices is a time-consuming and error-prone process. Ideally, automation should be provided that can reduce this manual effort by stakeholders.

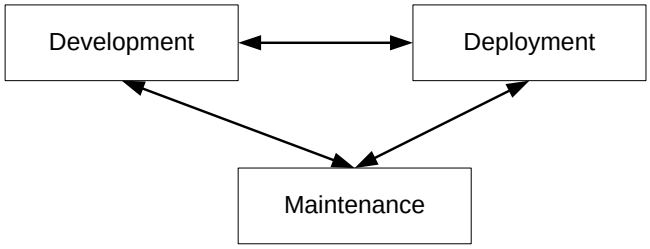

Figure I.3 – Different life cycle phases of IoT application development

Lack of special-purpose modeling languages. An IoT application corresponds to a specific application domain (e.g., building automation) consisting of domain-specific features (e.g., the applications in the building automation domain reason in terms of room and floors). Existing modeling languages for IoT application development remain general purpose (e.g., the modeling language proposed by PervML [Serral et al., 2010] relies on generic UML notations). Thus, they provide little guidance to stakeholders about the application domain. We believe that further customization of this approach is needed with respect to domain-specific features. In this manner, domain-specific knowledge should be available to stakeholders and can be reused across applications.

The challenges discussed above are not completely new. In fact they have been investigated at length in the domains of software engineering and model-driven design, and more specifically in wireless sensor network macroprogramming, the precursor to our work. In the next section, we summarize the application development techniques from those domains, which we base our

work on.

4 Application development approaches

This section studies existing application development approaches available to stakeholders for IoT application development. Throughout this thesis, we will use the term **stakeholders** as used in software engineering to mean – people, who are involved in the application development. Examples of stakeholders defined in [Taylor et al., 2009] are software designer, developer, domain expert, technologist, etc. The various application development approaches currently available to stakeholders are discussed next. For a detailed discussion of various systems available for application development, we refer the reader to Chapter II.

4.1 Node-centric and Macro-programming

Currently, development in the IoT is performed at the *node level*, by experts in embedded and distributed systems, who are directly concerned with operations of each device individually. They think in terms of activities of individual devices and explicitly encode their interaction with other devices. For example, they write a program that reads sensing data from appropriate sensor devices, aggregates data pertaining to the some external events, decides where to send it addressed by ID or location, and communicates with actuators if needed. Stakeholders in WSN, for example, use general-purpose programming languages (such as nesC [Gay et al., 2003], galsC [Cheong and Liu, 2005], or Java) and target a particular middleware API or node-level service [Costa et al., 2007; Frank and Römer, 2005; Whitehouse et al., 2004]. The Gaia [Román et al., 2002] distributed middleware infrastructure for pervasive environments supports application development using C++.

Although node-centric programming allows for the development of extremely efficient systems based on complete control over individual devices, it is not easy to use for IoT applications due to the large size and heterogeneity of systems. For instance, to develop an IoT application, stakeholders have to specify functions individually at each device – one each for sensing the environment, communicating with other devices, processing sensed data, as well as controlling actuators attached to each device.

An alternative approach to development of IoT applications comes from the WSN domain in the form of *sensor network macroprogramming*. This approach aims to aid stakeholders by providing the ability to specify their applications at a global level rather than individual nodes. In macroprogramming systems, abstractions are provided to specify high-level collaborative

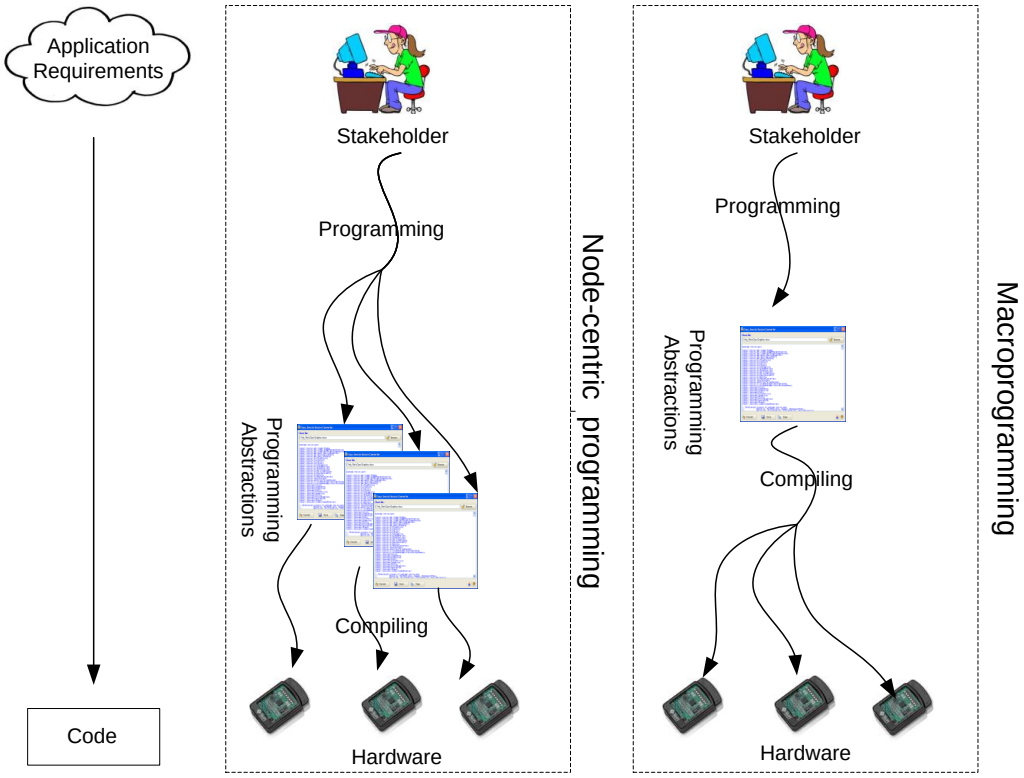

Figure I.4 – Comparing node-centric and macroprogramming

behaviors, while hiding low-level details such as message passing or state maintenance from stakeholders. Stakeholders describe their application using these abstractions, which is then compiled to node-level code. Pathak et al. [Pathak et al., 2007] define this compilation process as *the semantic-preserving transformation of a high level application specification into a distributed software system collaboratively hosted by the individual nodes.*

As illustrated in Figure I.4, stakeholders using WSN macroprogramming reason at a high-level of abstraction compared to node-centric programming, while the process of transforming the high-level specification to a node-level code is delegated to a compiler. Consequently, macro-programming is a viable approach compared to the node-centric programming, as demonstrated by the several efforts [Bakshi et al., 2005; Hnat et al., 2008; Newton et al., 2007]. However, most of macroprogramming systems largely focus on development phase (cf. Figure I.4) while ignoring the fact that it represents a tiny fraction of the application development life-cycle [Bischoff and Kortuem, 2007; Losilla et al., 2007]. The lack of a software engineering methodology to support the entire application development life-cycle commonly results in highly difficult to maintain, reuse, and platform-dependent design, which can be tackled by the model-driven approach discussed in the next section.

4.2 Model-driven development

The primary goal of Model-driven development (MDD) approach is to allow specifications of applications independently of specific implementation platforms such as programming languages and middleware [France and Rumpe, 2007; Mellor et al., 2003; Schmidt, 2006]. This approach applies the basic separation of concerns [Parnas, 1972] principal *vertically* by separating the specification of the system functionality from its specification on a specific platform. The former is defined as a platform independent model (PIM), the latter as platform specific model (PSM). (The object management group (OMG) defines a platform as "*a set of subsystem and technologies that provide a coherent set of functionality through interfaces and usage patterns*".) Examples of platforms are operating systems, programming languages, vendor specific databases, user interfaces, middleware solutions, etc. Figure I.5 illustrates the relationship between PIM and PSM. The mapping from PIM to PSM is performed using model transformation. In this view, code generation is regarded as a special kind of model transformation, as the code really is a model description in accordance with the meta-model of the actual programming language [Solberg et al., 2005].

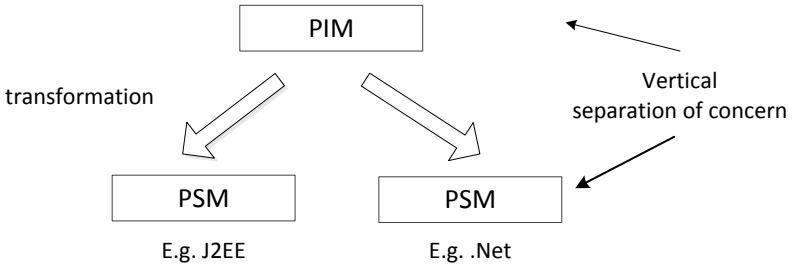

Figure I.5 – OMG's Model-driven development : main concepts

Several MDD approaches [Kulkarni and Reddy, 2003; Solberg et al., 2005] argue that vertical separation is not enough to reduce application development complexity, and mechanisms for both vertical and horizontal separation of concerns should be provided. Vertical separation reduces the application development complexity through abstraction and horizontal separation of concern reduces it by describing the system using different system views, each view describing a certain facet of the system. Kulkarni et al. describe this concept in graphical form shown in Figure I.6. Application development starts with an abstract specification A, which is to be transformed into a concrete implementation C. A can be separated into different views $A_1 \ldots A_n$, each view defining a set of properties corresponding to the concern it models. Each A_i can be transformed into C_i, with application level composition of $C_1 \ldots C_n$ giving C, the

intended implementation of A.

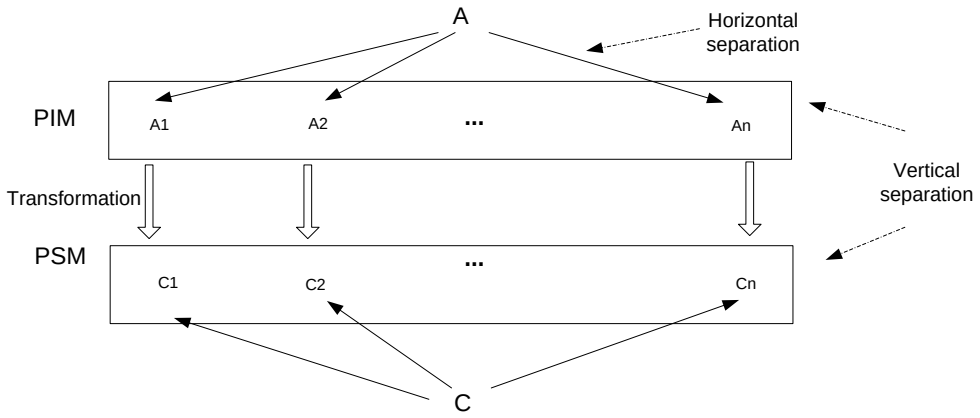

Figure I.6 – Model-driven development : horizontal and vertical separation of concerns

By following MDD guideline, numerous benefits can be achieved [Picek and Strahonja, 2007]. These benefits came from the basic idea that by separating different concerns of a system at a certain level of abstraction, and by providing transformation engines to convert these abstractions to a target code, productivity (e.g., reusability, maintainability) in the application development process can be improved.

5 Aim of thesis

The aim of this thesis is to make IoT application development easy for stakeholders as is the case in software engineering in general, by taking inspiration from the MDD approach and building upon work in sensor network macroprogramming. In particular, the aim of thesis is to achieve the following objectives :

- To separate IoT application development into different concerns, so stakeholders can deal with them individually at evolution and reuse them across applications.
- To integrate the identified development concerns into a well-defined and structured development process, so stakeholders have a precise sequence of steps to follow, thus smoothing the application development.
- To provide high-level modeling languages addressing IoT characteristics. This helps stakeholders to reduce both complexity and development effort associated with IoT applications.
- To automate IoT application development where possible. This helps to reduce development effort of stakeholders.

6 Thesis contributions

This thesis proposes a development methodology [2] for IoT application development, based on techniques presented in the domains of sensor network macroprogramming and model-driven development. It separates IoT application development into different concerns and integrates a set of high-level languages to specify them. This methodology is supported by automation techniques at different phases of IoT application development. We now present the contributions of this thesis described below:

Development methodology. We propose a development methodology that defines a precise sequence of steps to be followed to develop IoT applications, thus facilitating IoT application development. These steps are separated into four concerns, namely, domain, functional, deployment, and platform. This separation allows stakeholders to deal with them individually and reuse them across applications. Each concern is matched with a precise stakeholder according to skills. The clear identification of expectations and specialized skills of each type of stakeholders helps them to play their part effectively.

Development framework. To support the actions of each stakeholder, the development methodology is implemented as a concrete development framework [3]. It provides a set of modeling languages, each named after "Srijan", [4] and offers automation techniques at different phases of IoT application development, including the following:

- **A set of modeling languages.** To aid stakeholders, the development framework integrates three modeling languages that abstract the scale and heterogeneity-related complexity: (1) Srijan Vocabulary Language (SVL) to describe domain-specific features of an IoT application, (2) Srijan Architecture Language (SAL) to describe application-specific functionality of an IoT application, (3) Srijan Deployment Language (SDL) to describe deployment-specific features consisting information about a physical environment where devices are deployed. SAL and SDL are customized with respect to concepts defined using SVL. This enables knowledge sharing and reusability of domain knowledge across IoT applications.

2. Sommerville [Sommerville, 2010, 9^{th} edition] defines development methodology as "*a set of related activities that leads to the production of a software product.*" These involves phases, including software specification, design, implementation, validation, and evolution.

3. It includes support programs, code libraries, high-level languages or other software that help stakeholders to develop and glue together different components of a software product. Various literature use different terms in different context such a software framework, tool-driven development methodology, compilation framework, etc. We use term "development framework" in this thesis.

4. *Srijan* is the sanskrit word for "creation", and our work builds upon ideas introduced in the Srijan toolkit for sensor network macroprogramming [Pathak and Gowda, 2009].

– **Automation techniques.** The development framework is supported by code-generation, task-mapping, and linking techniques. These three techniques together provide automation at various phases of IoT application development. Code generation supports the application development phase by producing a programming framework that reduces the effort in specifying the details of the components of an IoT application. Mapping and linking together support the deployment phase by producing device-specific code to result in a distributed system collaboratively hosted by individual devices.

Our work on the above is supported at the lower layers by a middleware that enables delivery of messages across physical regions, thus enabling our abstractions for managing large scales in the Internet of Things.

7 Thesis structure

The remainder of this thesis is organized as follows:

– Chapter II reviews related work. We explore state of the art approaches for developing IoT applications and compare them with respect to the research challenges.
– Chapter III presents our development methodology and its development framework. This includes details of on modeling languages, automation techniques, and our approach for handling evolutions.
– Chapter IV presents an implementation of our development framework. We present tools, technologies, and programming languages used to implement this development framework.
– Chapter V evaluates the development framework in a quantitative manner. We explore three aspects to evaluate our approach: effort to develop applications using our approach, reusability of implementations and specification across applications, and code quality of the framework generated by our approach.
– Chapter VI summarizes this thesis and describes briefly some future directions of this work.
– For completeness, we include the grammar of three modeling languages (SVL, SAL, and SDL) in Appendix 1. Two application specifications of the building automation domain written using these modeling languages are presented in Appendix 2.

Chapter II

Related work

This chapter focuses on existing works in literature that would address the research challenges discussed in Chapter I. As stated earlier, while the application development life-cycle has been discussed in general in the software engineering domain, a similar structured approach is largely lacking in the IoT for the development of Sense-Computer-Control (SCC) [Taylor et al., 2009, p. 97] applications. Consequently, in this chapter we present existing approaches geared towards the IoT, but also its precursor fields of Pervasive Computing and Wireless Sensor Networking. These are mature fields, with several excellent surveys available on programming models [Mottola and Picco, 2011; Sugihara and Gupta, 2008] and middleware [Henricksen and Robinson, 2006]. We organize this chapter based on the perspective of the system provided to the stakeholders by the various approaches. Section 1 presentes the node-level programming approaches, where the developer has significant control over the actions of each device in the system, which comes at the cost of complexity. Section 2 summarizes approaches that aim to abstract the entire (sensing) system as a database on which one can run queries. Section 3 presents the evolution of these approaches to macroprogramming inspired by general-purpose programming languages, where abstractions are provided to specify high-level collaborative behaviors at the system-level while hiding low-level details from stakeholders. Section 4 then describes the macroprogramming approaches more grounded in model-driven development techniques, which aim to provide a cleaner separation of concerns during the application development process. Finally, Section 5 compares the discussed approaches with respect to the research challenges discussed in Chapter I.

1 Node-centric programming

In this approach, which is most commonly used, stakeholders think in terms of single device activities and explicitly encode its interaction with other devices. For example, they write a

program in nesC for TinyOS [Levis et al., 2005] that reads sensing data from appropriate sensing devices, aggregates data pertaining to the some external events, decides where to send it addressed by ID or location, and communicates with actuators if needed. Stakeholders use general-purpose programming language (e.g., Java, C++) and target a particular middleware API or node-level service to develop an application. The node-centric programming makes it possible to develop extremely efficient applications by allowing control over each device. However, due to a large size of the systems, continuous evolution in hardware and software components, and a limited expertise of stakeholders, this approach is not easy to use for IoT application development. In the following, we present systems that adopt the node-centric approach.

In the pervasive computing domain, Olympus [Ranganathan et al., 2005] is a programming model on top of Gaia [Román et al., 2002] – a distributed middleware infrastructure for pervasive environments. Stakeholders write a C++ program that consists of a high-level description about active space entities (including service, applications, devices, physical objects, and locations) and common active operations (e.g., switching devices on/, starting/stopping applications). The Olympus framework takes care of resolving high-level description based on properties specified by stakeholders. While this approach certainly simplifies the SCC application development involving heterogeneous devices, stakeholders have to write a lot of code to interface hardware and software components, as well as to interface software components and its interactions with a distributed system. This makes it tedious to develop applications involving a large number of devices.

The Context toolkit [Dey et al., 2001; Salber et al., 1999] simplifies the context-aware application development on top of heterogeneous data sources by providing three architectural components, namely, widgets, interpreters, and aggregators. These components separate application semantics from platform-specific code. For example, an application does not have to be modified if an Android-specific sensor is used rather than a Sun SPOT sensor. It means stakeholders can treat a widget in a similar fashion and do not have to deal with differences among platform-specific code. Although context toolkit provides support for acquiring the context data from the heterogeneous sensors, it does not support actuation that is an essential part of IoT applications.

Henricksen et al. [Bettini et al., 2010; Henricksen and Indulska, 2006] propose a middleware and a programming framework to gather, manage, and disseminate context to applications. This work introduces context modeling concepts, namely, context modeling languages, situation abstraction, and preference and branching models. This work presents a software engineering process that can be used in conjunction with the specified concepts. However, the clear separa-

tion of roles among the various stakeholders is missing. Moreover, this framework limits itself to context gathering applications, thus not providing the actuation support that is important for IoT application development.

Physical-Virtual mashup. As indicated by its name, it connects web services from both the physical and virtual world through visual constructs directly from web browsers. The embedded device runs tiny web servers [Duquennoy et al., 2009] to answer HTTP queries from users for checking or changing the state of a device. For instance, users may want to see temperature of different places on map. Under such requirements, stakeholders can use the mashup to connect physical services such as temperature sensors and virtual services such as Google map. Many mashup prototypes have been developed that include both the physical and virtual services [Blackstock and Lea, 2012; Castellani et al., 2012; Ghidini et al., 2012; Guinard et al., 2010; Gupta et al., 2010]. The mashup editor usually provides visual components representing web service and operations (such as add, filter) that stakeholders need to connect together to program an application. The framework takes care of resolving these visual components based on properties specified by stakeholders and produces code to interface software components and distributed system. The main advantage of this mashup approach is that any service, either physical or virtual, can be mashed-up if they follow the standards (e.g., REST). The Physical-Virtual mashup significantly lowers the barrier of the application development. However, stakeholders have to manage a potentially large graph for an application involving a large number of entities. This makes it difficult to develop applications containing a large number of entities.

2 Database approach

Since a large part of these systems consists of devices that sense the environment, a natural abstraction for thinking of their functionality at a high level is that of a *database*, providing access to sensed data. This approach views the whole sensor network as a virtual database system. It provides an easy-to-use interface that lets stakeholders issue queries to a sensor network to extract the data of interest.

In TinyDB [Madden et al., 2005] and Cougar [Yao and Gehrke, 2002] systems, an SQL-like query is submitted to a WSN. On receiving a query, the system collects data from the individual device, filters it, and sends it to the base station. They provide a suitable interface for data collection in a network with a large number of devices. However, they do not offer much flexibility for introducing the application logic. For example, stakeholders require extensive

modifications in the TinyDB parser and query engine to implement new query operators.

The work on SINA (Sensor Information Networking Architecture) [Shen et al., 2001] overcomes this limitation on specification of custom operators by introducing an imperative language with an SQL query. In SINA, stakeholders can embed a script written in Sensor Querying and Tasking Language (SQTL) [Jaikaeo et al., 2000] in the SQL query. By this hybrid approach, stakeholders can perform more collaborative tasks than what SQL in TinyDB and Cougar can describe.

The TinyDB, Cougar, and SINA systems are largely limited to homogeneous devices. The IrisNet (Internet-Scale Resource-Intensive Sensor Network) [Gibbons et al., 2003] allows stakeholders to query a large number of distributed heterogeneous devices. For example, Internet-connected PCs source sensor feeds and cooperate to answer queries. Similar to the other database approaches, stakeholders view the sensing network as a single unit that supports a high-level query in XML. This system provides a suitable interface for data collection from a large number of different types of devices. However, it does not offer flexibility for introducing the application logic, similar to TinyDB and Cougar.

Semantic Streams [Whitehouse et al., 2006] allows stakeholders to pose a declarative query over semantic interpretations of sensor data. For example, instead of querying raw magnetometer data, stakeholders query whether a vehicle is a car or truck. The system infers this query and decides sensor data to use to infer the type of vehicle. The main benefit of using this system is that it allows people, with less technical background to query the network with heterogeneous devices. However, it presents a centralized approach for sensor data collection that limits its applicability for handling a network with a large number of devices.

A number of systems have been proposed to expose functionality of devices accessible through standardized protocols without having worry about the heterogeneity of underlying infrastructure [Mohamed and Al-Jaroodi, 2001]. They logically view sensing devices (e.g., motion sensor, temperature sensor, door and window sensor) as service providers for applications and provide abstractions usually through a set of services. We discuss these examples below.

TinySOA [Avilés-López and García-Macías, 2009] is a service-oriented architecture that provides a high-level abstraction for WSN application development. It allows stakeholders to access WSNs using service-oriented APIs provided by a gateway. The APIs provide functions to obtain information about the network, listing devices, and sensing parameters. The gateway component acts as a bridge between a WSN and an application. The gateway consists of a WSN infrastructure registry and discovery components. Through these components, the gateway allows stakeholders to access data from a WSN without dealing with low-level details. This system provides suitable interfaces from a large number of different types of devices for data

collection. However, every device in TinySOA is pre-configured and sends data to the base station only. Thus, the flexibility for in-network processing is limited, similar to TinyDB, Cougar, and IrisNet.

Priyantha et al. [Priyantha et al., 2008] present an approach based on SOAP [Box et al., 2000] to enable an evolutionary WSN where additional devices may be added after the initial deployment. To support such a system, this approach has adopted two features. (1) structured data: the data generated by sensing devices are represented in a XML format for that may be understood by any application. (2) structured functionality: the functionality of a sensing device is exposed by Web Service Description Language (WSDL) [Chinnici et al., 2007]. While this system addresses the evolution issue in a target deployment, the authors do not demonstrate the evolution scenarios such as a change in functionality of an application, technological advances in deployment devices.

A number of approaches based on REST [Fielding, 2000] have been proposed to overcome the resource needs and complexity of SOAP-based web services for sensing and actuating devices. TinyREST [Luckenbach et al., 2005] is one of first attempts to overcome these limitations. It uses the HTTP-based REST architecture to access a state of sensing and actuating devices. The TinyREST gateway maps the HTTP request to TinyOS messages and allows stakeholders to access sensing and actuating devices from their applications. The aim of this system is to make services available through standardized REST without having to worry about the heterogeneity of the underlying infrastructure; that said, it suffers from a centralized structure similar to TinySOA.

3 General-purpose macroprogramming languages

To provide a flexibility for any-to-any device collaboration (one of limitations of database approach) while preserving global network abstractions, a number of macroprogramming languages have been proposed. They allow stakeholders to specify an application as a global program, which can then be decomposed into smaller programs that execute on devices. In the following, we present macroprogramming languages for IoT application development, which are grounded in traditional general purpose programming langauges (whether imperative or functional) in order to provide developers with familiar abstractions.

Kairos [Gummadi et al., 2005] allows stakeholders to program an application in a Python-based language. The Kairos developers write a centralized program of a whole application. Then, the pre-processor divides the program into subprograms, and later its compiler compiles

it into binary code containing code for accessing local and remote variables. Thus, this binary code allows stakeholders to program distributed sensor network applications. Although Kairos makes the development task easier for stakeholders, it targets homogeneous network where each device executes the same application.

Regiment [Newton et al., 2007] provides a high-level programming language based on Haskell to describe an application as a set of spatially distributed data streams. This system provides primitives that facilitate processing data, manipulating regions, and aggregating data across regions. The written program is compiled down to an intermediate token machine language that passes information over a spanning tree constructed across the WSN. In contrast to the database approaches, this approach provides greater flexibility to stakeholders when it comes to the application logic. However, the regiment program collects data to a single base station. It means that the flexibility for any-to-any device collaboration for reducing scale is difficult.

MacroLab [Hnat et al., 2008] offers a vector programming abstraction similar to Matlab for applications involving both sensing and actuation. Stakeholders write a single program for an entire application using Matlab like operations such as `addition`, `find`, and `max`. The written macroprogram is passed to the MacroLab decomposer that generates multiple decompositions of the program. Each decomposition is analyzed by the cost analyzer that calculates the cost of each decomposition with respect to a cost profile (provided by stakeholders) of a target deployment. After choosing a best decomposition by the cost analyzer, it is passed to the compiler that converts the decomposition into a binary executable. The main benefit is that it offers flexibility of decomposing code according to cost profiles of the target platform. While this system certainly separates the deployment aspect and functionality of an application, this approach remains general purpose and provides little guidance to stakeholders about the application domain.

4 Model-based macroprogramming

As an evolution of the approaches discussed above which are based on general-purpose programming languages, model-driven approaches to macroprogramming aim to provide greater coverage of the software development lifecycle. They aim to specify an application using high-level abstract models that can be transformed into concrete implementations by automated code generators. A number of model-driven approaches have been proposed to make IoT application development easy.

PervML [Serral et al., 2010] allows stakeholders to specify pervasive applications at a high-

level of abstraction using a set of models. This system raises the level of abstraction in program specification, and code generators produce code from these specifications. Nevertheless, it adopts generic UML notations to describe them, thus provides little guidance to stakeholders about the specific application domain. In addition to this, the main focus of this work is to address the heterogeneity associated with pervasive computing applications, and the consideration of a large number of devices in an application is missing. PervML integrates the mapping process at the deployment phase. However, stakeholders have to link the application code and configure device drivers manually. This manual work in the deployment phase is not suitable for IoT applications involving a large number of devices. Moreover, the separation between deployment and domain-specific features are missing. These limitations would restrict PervML to a certain level.

DiaSuite [Cassou et al., 2012] is a suite of tools to develop pervasive computing applications. It combines design languages and covers application development life-cycle. The design language defines both a taxonomy of an application domain and an application architecture. Stakeholders define entities in a high-level manner to abstract heterogeneity. However, the consideration of a large number of devices in an application is largely missing. Moreover, the application deployment for a large number of heterogeneous devices using this approach is difficult because stakeholders require manual effort (e.g., mapping of computational services to devices).

Our work takes inspiration from ATaG [Pathak and Prasanna, 2011], which is a WSN is a macroprogramming framework to develop SCC applications. The notion of abstract task, abstract data item, and abstract channel are the core of this framework. An abstract task encapsulates the processing of data items. The flow of information among abstract tasks is specified in terms of input and output data items. Abstract channels connect abstract tasks to data items. ATaG presents a compilation framework that translates a program, containing abstract notations, into executable node-level programs. Moreover, it tackles the issue of scale reasonably well. The ATaG linker and mapper modules support the application deployment phase by producing device-specific code to result in a distributed software system collaboratively hosted by individual devices, thus providing automation at deployment phase. Nevertheless, the clear separation of roles among the various stakeholders in the application development, as well as the focus on heterogeneity among the constituent devices are largely missing. Moreover, the ATaG program notations remains general purpose and provides little guidance to stakeholders about the application domain.

RuleCaster [Bischoff and Kortuem, 2006, 2007] introduces an engineering method to provide support for SCC applications, as well as evolutionary changes in the application development. The RuleCaster programming model is based on a logical partitioning of the network into spatial

regions. Each region is in one discrete state. A rule-based language is used to specify a transition of the state, where each rule consists of two parts: (1) body and (2) head. The body part specifies a fire condition, and the head part specifies one or more actions when the condition becomes true. The RuleCaster compiler takes as input the application program containing rules and a network model that describes device locations and its capabilities. Then, it maps processing tasks to devices. Similar to ATaG, this system handles the scale issue reasonably well by partitioning the network into several spatial regions. Moreover, it supports automation at the deployment phase by mapping computational components to devices. However, the clear separation of roles among the various stakeholders, support for application domain, as well as the focus on heterogeneity among the constituent devices are missing.

Pantagruel [Drey et al., 2009] is a visual approach dedicated to the development of home automation applications. The Pantagruel application development consists of three steps: (1) specification of taxonomy to define entities of the home automation domain (e.g., temperature sensor, alarm, door, smoke detector, etc.), (2) specification of rules to orchestrate these entities using the Pantagruel visual language, and (3) compilation of the taxonomy and orchestration rules to generate a programming framework. The novelty of this approach is that the orchestration rules are customized with respect to entities defined in the taxonomy. While this system reduces the requirement of having domain-specific knowledge for other stakeholders, the clear separation of different development concerns, support for large scale, automation both at the development and deployment phase are largely missing. These limitations make it difficult to use for IoT application development.

5 Chapter summary and conclusion

This chapter has investigated existing works for IoT application development. Table II.1 compares approaches with respect to the challenges discussed in Chapter I. The symbol "$\surd$" means the challenge is supported by the approach, "$\times$" means "not supported", and "$\sim$" means "no adequately supported". Due to similarity, we pick one representative system in some cases. For instance, TinyDB and Cougar have adopted SQL-based interface for collecting data. So, we take only TinyDB as a representative example.

As illustrated in Table II.1, our investigation has revealed that many ideas have been proposed for addressing the research challenges individually. However, none of the examined approach addresses all of the identified challenges to a sufficient extent. More specifically, we note the following observations based on our literature survey:

Division of roles. Most of the application development approaches do not divide roles of stakeholders to a sufficient extent. This is in clear conflict with the varied set of skills required during IoT application development, including domain expertise, deployment-specific knowledge, application design and implementation skill, and ability to deal with operating system and hardware-specific details. The clear identification of expectations and specialized skills of each type of stakeholder is necessary to play their part effectively to smoothen the application development.

Heterogeneity and Scale. Existing approaches address the scale and heterogeneity challenge, but individually. None of the examined approach addresses these two challenges to a sufficient extend. IoT applications are heterogeneous in terms of device types, interaction modes, and different implementations. Moreover than this, they may execute on systems consists of hundreds to thousands of devices. Therefore, it is important to address both the issues in one approach.

General purpose modeling languages. Some approaches customize modeling languages with respect to an application domain. However, most of approaches remain general purpose (e.g., a modeling language proposed by PervML relies on generic UML notations), thus provide little guidance to stakeholders about the application domain. The key advantage of the customization is that domain-specific knowledge is made available to stakeholders and can be reused across applications of the same application domain, and thus the "one-size-fits-all" approach needs to be revisited for IoT application development.

Evolution. An IoT application may involve a number of changes such as technological advances with new software features and changes in a distribution of devices in a target deployment, deployment of the same application on a different deployment, changes in functionality of an application. Existing approaches do not address these evolutionary changes to a sufficient extent. Therefore, it is necessary to consider them comprehensively for IoT application development.

Automation at development phase. IoT applications involve a large number of heterogeneous devices. Consequently, stakeholders face the prospect of spending a lot of time in manual effort that is attributed to the development and deployment phases of the application development. While some approaches address poor automation problem at development phase to allow stakeholders to focus on the application logic, they do not address automation at deployment phase to a sufficient extend. Therefore, automation at both the deployment and development phase remains missing in the existing approaches.

In conclusion, there is a need of designing a new approach that utilizes advantages and promising features of the existing works to develop a comprehensive integrated approach while focusing on ease of IoT application development.

Existing Approaches		Division of roles	Hetro.	Scale	General purpose modeling lang.	Evolution	Automation	
							Development phase	Deployment phase
Node-centric Programming	ContextToolkit (2001)	×	∼	×	×	∼	×	×
	TinyOS (2005)	×	×	×	×	∼	×	×
	Olympus (2005)	×	√	×	×	×	×	×
	Henricksen et al. (2010)	×	∼	×	×	×	∼	×
	Dominique et al. (2010)	×	√	×	×	∼	∼	×
Database Approach	TinyDB (2000)	×	×	∼	×	×	Not clear	×
	IrisNet (2000)	×	∼	∼	×	×	Not clear	×
	SINA (2000)	×	×	√	×	×	Not clear	×
	TinySOA (2000)	×	√	∼	×	∼	×	×
	TinyREST (2005)	×	√	×	×	∼	×	×
	Semantic Streams (2006)	×	∼	×	×	×	∼	×
	Priyantha et al. (2008)	×	√	×	×	∼	×	×
General-Purpose Macroprogramming Languages	Kairos (2005)	×	×	√	×	×	∼	∼
	Regiment (2007)	×	×	∼	×	×	∼	∼
	MacroLab (2008)	×	×	√	×	∼	∼	√
Model-driven Macroprogramming	RuleCaster (2007)	×	×	√	×	∼	∼	∼
	Pantagruel (2009)	∼	∼	×	√	∼	∼	×
	PervML (2010)	∼	√	×	×	∼	∼	∼
	DiaSuite (2011)	∼	√	×	√	∼	√	×
	ATaG (2011)	×	∼	√	×	∼	∼	√

Table II.1 – Comparison of existing approaches. √ – Supported, × – No supported, ∼ – No adequately supported.

Chapter III

Our approach to IoT application development

Chapter I presented the challenges for IoT application development. Chapter II investigated the existing works. The investigation revealed that many ideas have been proposed for addressing the research challenges *individually*. However, none of the examined approaches addresses all of the identified challenges to a sufficient extent. This chapter presents a novel IoT application development approach that addresses the identified research challenges in an integrated manner.

This chapter is organized into three sections. Section 1 provides an overview of our approach. Section 2 describes our approach in detail. Finally, Section 3 concludes this chapter by summarizing how the identified research challenges are addressed by our approach.

1 Overview

A conceptual model often serves as a base of knowledge about a problem area [Fowler, 1996]. It represents the concepts as well as the associations among them and also attempts to clarify the meaning of various terms. Leveraging the conceptual model for IoT applications we present in Section 1.1, Section 1.2 presents our overall development methodology.

1.1 Conceptual model

Applying the separation of concerns design principal from software engineering, we break the identified concepts and associations into different categories. We note that we are not the first ones who have used separation of concerns as a fundamental method for implementing applications. Taking inspiration from previous efforts [Bischoff and Kortuem, 2007; Cassou et al., 2012; Doddapaneni et al., 2012], we have identified four major concerns for IoT application development. Figure III.1 illustrates the concepts and their associations along with these

four separate concerns: (1) domain-specific concepts, (2) functionality-specific concepts, (3) deployment-specific concepts, and (4) platform-specific concepts.

1.1.1 Domain-specific concepts

The concepts that fall into this category are specific to a target application domain (e.g., building automation, transport, etc.). For example, the building automation domain is reasoned in terms of rooms and floors, while the transport domain is expressed in terms of highway sectors. Furthermore, each domain has a set of entities of interest (e.g., average temperature of a building, smoke presence in a room), which are observed and controlled by sensors and actuators respectively. Storages store information about entities of interest, and user interfaces enable users to interact with entities of interest (e.g., receiving notification in case of fire in a building, controlling the temperature of a room). We describe these concepts in detail below:

- An **Entity of Interest (EoI)** is an object (e.g., room, book, plant), including attributes that describe it, and its state that is relevant from a user or an application perspective [Haller, 2010, p. 1]. The entity of interest has an observable property called *phenomenon*. Typical examples are the temperature value of a room and a tag ID.
- A **resource** is a conceptual representation of a sensor, an actuator, a storage, or a user interface. We consider the following types of resources:
  - A **sensor** has the ability to detect changes in the environment. Thermometer and tag readers are examples of sensors. The sensor **observes** a phenomenon of an EoI. For instance, a temperature sensor observes the temperature phenomenon of a room.
  - An **actuator** makes changes in the environment through an action. Heating or cooling elements, speakers, lights are examples of actuators. The actuator **affects** a phenomenon of an EoI by performing actions. For instance, a heater is set to control a temperature level of a room.
  - A **storage** has the ability of storing data in a persistent manner. The storage **stores** information about a phenomenon of an EoI. For instance, a database server stores information about an employee's temperature preference.
  - A **user interface** represents tasks available to users to interact with entities of interest. For the building automation domain, a task could be receiving a fire notification in case of emergency or controlling a heater according to a temperature preference.
- A device is located in a **region** [Tubaishat and Madria, 2003]. The region is used to specify the location of a device. In the building automation domain, a region (or location) of a device can be expressed in terms of building, room, and floor IDs.

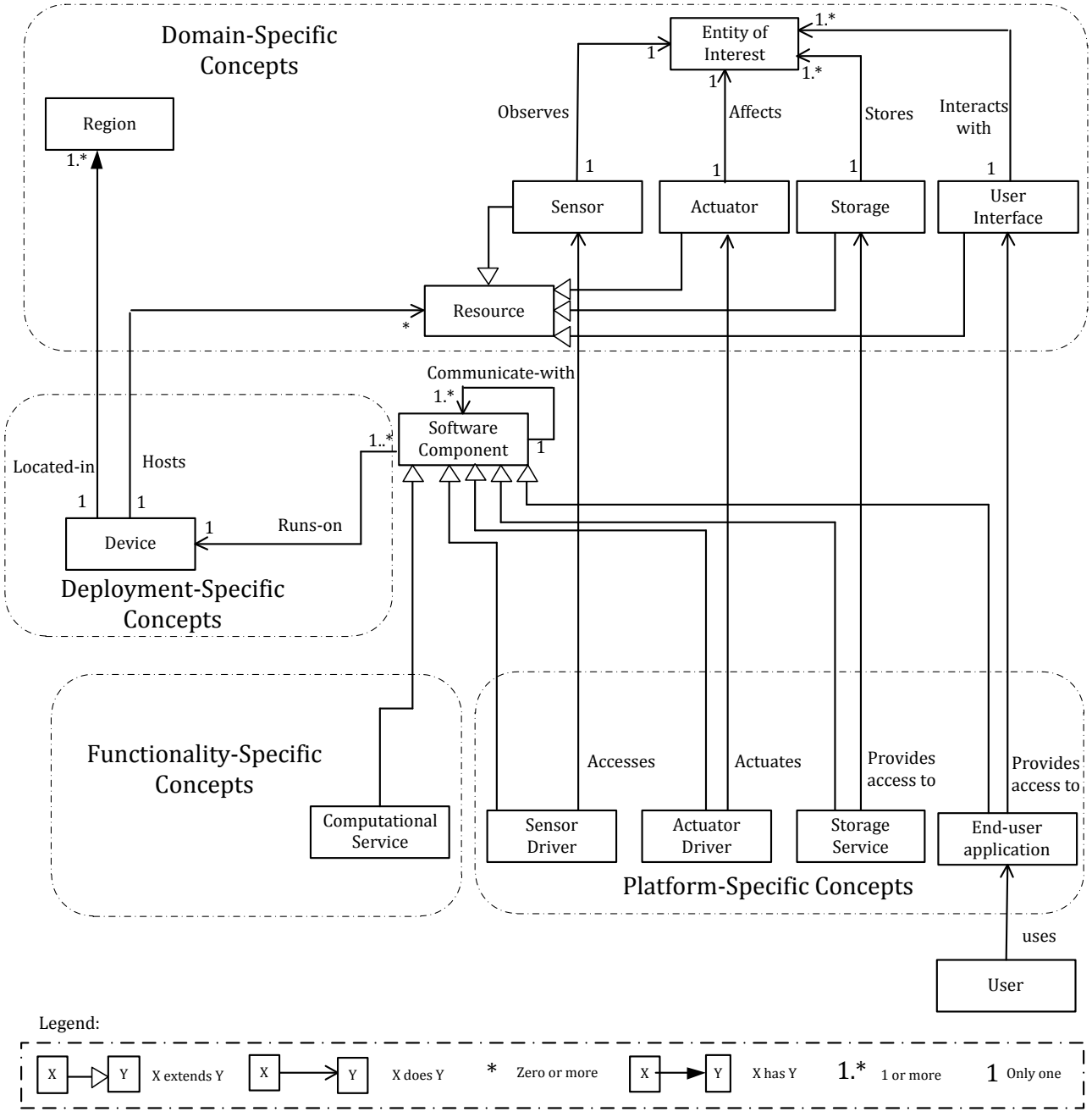

Figure III.1 – Conceptual model for IoT applications

1.1.2 Functionality-specific concepts

The concepts that fall into this category describe computational elements of an application and interactions among them. A computational element is a type of software component, which is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface [Taylor et al., 2009, p. 69]. We use the term **application logic** to refer a functionality of a software component. An example of the application logic is to open a window when the average temperature value of a room is greater than $30°C$.

The conceptual model contains the following functionality-specific software component, a **computational service**, which is a type of software component that consumes one or more units of information as inputs, processes it, and generates an output. An output could be data message that is consumed by others or a command message that triggers an action of an actuator. A computational service is a representation of the processing element in an application.

A software component **communicates-with** other software components to exchange data or control. These interactions might contain instances of various interaction modes such as request-response, publish-subscribe, and command. Note that this is in principle an instance of the component-port-connector architecture used in software engineering.

1.1.3 Deployment-specific concepts

The concepts that fall into this category describe information about devices. Each device **hosts** zero or more resources. For example, a device could host resources such as a temperature sensor to sense, a heater to control a temperature level, a monitor to display a temperature value, a storage to store temperature readings, etc. Each device is **located-in** regions. For instance, a device is located-in room#1 of floor#12 in building#14. We consider the following definition of a device:

- A **device** is an entity that provides resources the ability of interacting with other devices. Mobile phones, and personal computers are examples of devices.

1.1.4 Platform-specific concepts

The concepts that fall into this category are computer programs that act as a (operating system-specific) translator between a hardware device and an application. We identify the following platform-specific concepts:

- A **sensor driver** is a type of software component that operates on a sensor attached to a device. It **accesses** data observed by the sensor and generates the meaningful data

that can be used by other software components. For instance, a temperature sensor driver generates temperature values and its meta-data such as unit of measurement, time of sensing. Another software component takes this temperature data as input and calculates the average temperature of the room.

- An **actuator driver** is a type of software component that controls an actuator attached to a device. It translates a command from other software components and **actuates** the actuator appropriately. For instance, a heater driver translates a command "turn the heater on" to regulate the temperature level.
- A **storage service** is a type of software component that provides a read and write access to a storage. A storage service **provides access to** the storage. Other software components access data from the storage by requesting the storage service. For instance, MySQL storage service provides access to a database server.
- An **end-user application** is a type of software component that is designed to help a user to perform tasks (e.g., receiving notifications, submitting information). It **provides access to** available tasks. For instance, in the smart building application a user could provide his temperature preferences using an application installed on his smart phone.

The next section presents a development methodology that links the above four concerns and provides a conceptual framework to develop IoT applications.

1.2 A development methodology for IoT applications

To make IoT application development easy, stakeholders should be provided a structured and well-defined application development process (referred to as *development methodology*). This section presents a development methodology that integrates different development concerns discussed in Section 1.1 and provides a conceptual framework for IoT application development. In addition to this, it assigns a precise role to each stakeholder commensurate with his skills and responsibilities.

As stated in chapter I, IoT application development is a multi-disciplined process where knowledge from multiple concerns intersects. So far, IoT application development assumes that the individuals have similar skills. While this may be true for simple/small applications for single-use deployments, as the IoT gains wide acceptance, the need for sound software engineering approaches to adequately manage the development of complex applications arises.

Taking inspiration from ideas proposed in the 4+1 view model of software architecture [Kruchten, 1995], collaboration model for smart spaces [Chen et al., 2012], and tool-based methodology for pervasive computing [Cassou et al., 2012], we propose a development methodology that pro-

vides a conceptual framework to develop an IoT application (detailed in Figure III.2). The development methodology divides the responsibilities of stakeholders into five distinct roles —domain expert, software designer, application developer, device developer, and network manager. Note that although these roles have been discussed in the software engineering literature in general, e.g., domain expert and software designer in [Taylor et al., 2009, p. 657], application developer [Cassou et al., 2012, p. 3], their clear identification for IoT applications is largely missing. Due to the existance of various, slightly varying, definitions in literature, we summarize the skills and responsibilities of the various stakeholders in Table III.1

Role	Skills	Responsibilities
Domain expert	Understands domain concepts, including the data types produced by the sensors, consumed by actuators, accessed from storages, user's interactions, and how the system is divided into regions.	Specify the vocabulary of an application domain to be used by applications in the domain.
Software designer	Software architecture concepts, including the proper use of interaction modes such as publish-subscribe, command, and request-response for use in the application.	Define the structure of an IoT application by specifying the software components and their generate, consume, and command relationships.
Application developer	Skilled in algorithm design and use of programming languages.	Develop the application logic of the computational services in the application.
Device developer	Deep understanding of the inputs/outputs, and protocols of the individual devices.	Write drivers for the sensors, actuators, storages, and end-user applications used in the domain.
Network manager	Deep understanding of the specific target area where the application is to be deployed.	Install the application on the system at hand; this process may involve the generation of binaries or bytecode, and configuring middleware.

Table III.1 – Roles in IoT application development

An application corresponds to a specific application domain (e.g., building automation, health-care, transport) consisting of domain-specific concepts. Keeping this in mind, we separate the domain concern from other concerns (see Figure III.2, **stage 1**). The main advantage of this separation is that domain-specific knowledge can be made available to stakeholders and reused across applications of a same application domain.

IoT applications closely interact with the physical world. Consequently, changes in either of them have a direct influence on the other. The changes could be technological advances with new software features, a change in functionality of an application, a change in distribution of

devices, and adding or replacing devices. Considering this aspect, we separate IoT application development into the platform, functional, and deployment concern at the second stage (see Figure III.2, **stage 2**). Thus, stakeholders can deal with them individually and reuse them across applications. The final stage combines and packs the code generated by the second stage into packages that be deployed on devices (see Figure III.2, **stage 3**).

2 Multi-step IoT application development process

2.1 Overview

To support actions of stakeholders, the development methodology discussed in Section 1.2 is implemented as a concrete development framework. This section presents this development framework that provides a set of modeling languages, each named after *Srijan*[1], and offers automation techniques at different phases of IoT application development for the respective concerns.

2.1.1 Domain concern

This concern is related to domain-specific concepts of an IoT application. It consists of the following steps:

- **Specifying domain vocabulary.** The domain expert specifies a domain vocabulary using the Srijan Vocabulary Language (SVL). The vocabulary includes specification of resources, which are responsible for interacting with entities of interest. In the vocabulary, resources are specified in a high-level manner to abstract low-level details from the domain expert. Moreover, the vocabulary includes definitions of regions that define spatial partitions (e.g., room, floor, building) of a system.
- **Compiling vocabulary specification.** Leveraging the vocabulary, the development framework generates: (1) a vocabulary framework to aid the device developer, (2) a customized architecture grammar according to the vocabulary to aid the software designer, and (3) a customized deployment grammar according to the vocabulary to aid the network manager. The key advantage of this customization is that the domain-specific concepts defined in the vocabulary are made available to other stakeholders and can be reused across applications of the same application domain.

1. the Sanskrit word for “creation”

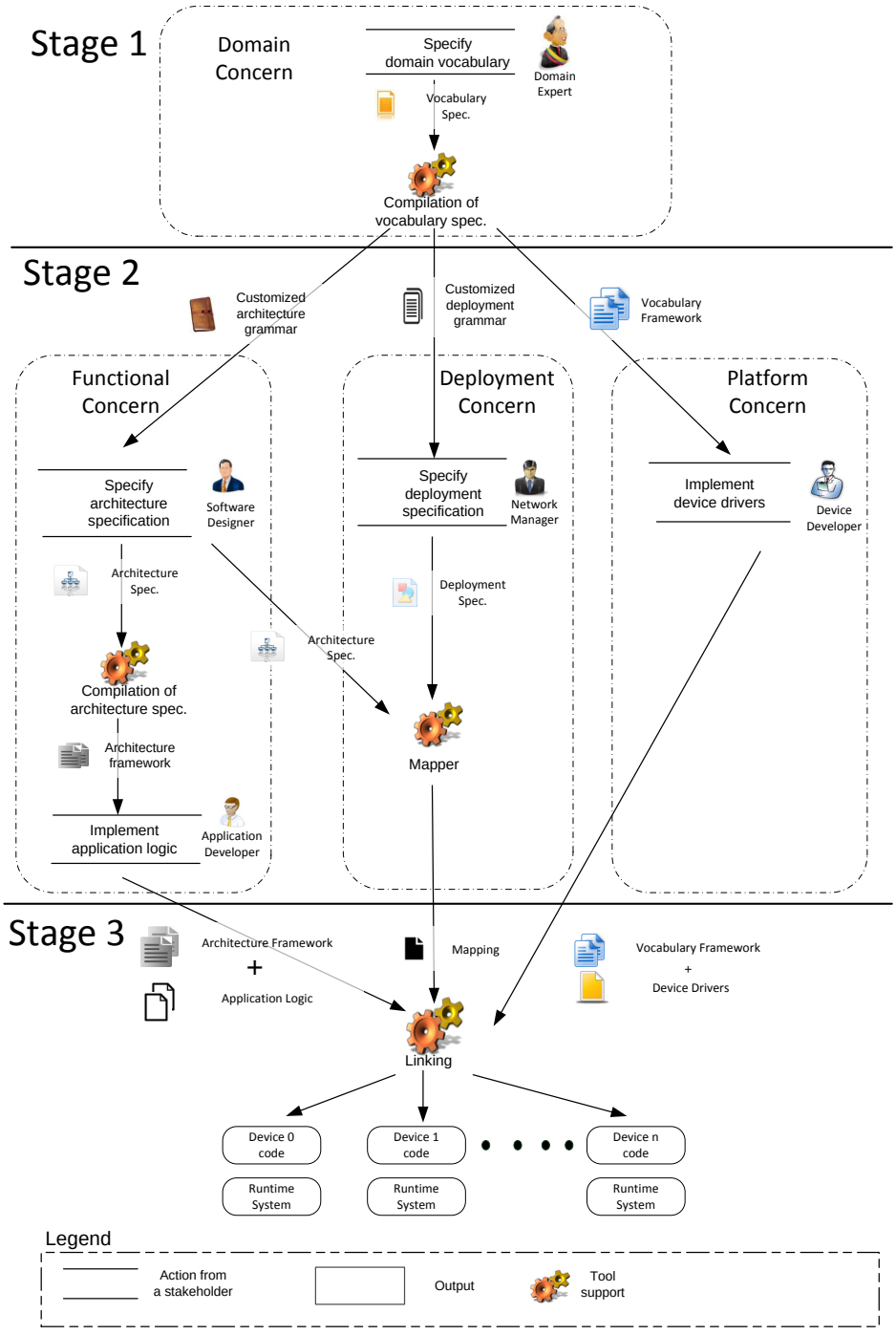

Figure III.2 – IoT application development : overall process

2.1.2 Functional concern

This concern is related to functionality-specific concepts of an IoT application. It consists of the following steps:

- **Specifying application architecture.** Using a customized architecture grammar, the software designer specifies an application architecture using the Srijan Architecture Language (SAL). SAL is an architecture description language (ADL) designed for specifying computational services and their interactions with other software components. To facilitate scalable operations within IoT applications, SAL offers scope constructs. These constructs allow the software designer to group devices based on their spatial relationship to form a cluster (e.g., "devices are in room#1") and to place a cluster head to receive and process data from that cluster. The grouping and cluster head mechanism can be recursively applied to form a hierarchical clustering that facilitates the scalable operations within IoT applications.
- **Compiling architecture specification.** The development framework leverages an architecture specification to support the application developer. To describe the application logic of each computational service, the application developer is provided an architecture framework, pre-configured according to the architecture specification of an application, an approach similar to the one discussed in [Cassou et al., 2009].
- **Implementing application logic.** To describe the application logic of each computational service, the application developer leverages a generated architecture framework. It contains abstract classes [2], corresponding to each computational service, that hide interaction details with other software components and allow the application developer to focus only on application logic. The application developer implements only the abstract methods of generated abstract classes.

2.1.3 Deployment concern

This concern is related to deployment-specific concepts of an IoT application. It consists of the following steps:

- **Specifying target deployment.** Using a customized deployment grammar, the network manager describes a deployment specification using the Srijan Deployment Language (SDL). The deployment specification includes the details of each device, including its regions (in terms of values of the regions defined in the vocabulary), resources hos-

2. We assume that the application developer uses an object-oriented language.

ted by devices (a subset of those defined in the vocabulary), and the type of the device. Ideally, the same IoT application could be deployed on different target deployments (e.g., the same inventory tracking application can be deployed in different warehouses). This requirement is dictated by separating a deployment specification from other specifications.

– **Mapping.** The mapper produces a mapping from a set of computational services to a set of devices. It takes as input a set of placement rules of computational services from an architecture specification and a set of devices defined in a deployment specification. The mapper decides devices where each computational service will be deployed.

2.1.4 Platform concern

This concern is related to platform-specific concepts of an IoT application. It consists of the following step :

– **Implementing device drivers.** Leveraging the vocabulary, our system generates a vocabulary framework to aid the device developer. The vocabulary framework contains *interfaces* and *concrete classes* corresponding to resources defined in the vocabulary. The concrete classes contain concrete methods for interacting with other software components and platform-specific device drivers. The interfaces are implemented by the device developer to write platform-specific device drivers.

2.1.5 Linking

The linker combines and packs code generated by various stages into packages that can be deployed on devices. It merges generated architecture framework, application logic, mapping files, device drivers, and vocabulary framework. This stage supports the application deployment phase by producing device-specific code to result in a distributed software system collaboratively hosted by individual devices, thus providing automation at the deployment phase [3].

2.1.6 Handling evolution

Evolution is an important aspect in IoT application development where new resources and computational services are added, removed, or extended. To deal with these changes, our development framework separates IoT application development into different concerns and allows an iterative development [Sommerville, 2010] for these concerns.

3. We assume that a middleware is already installed on the deployed devices. The installed middleware enables inter-device communication among devices.

This next section provides the details of our approach including three modeling languages (SVL, SAL, and SDL), programming frameworks to aid stakeholders, and an approach for handling evolution. This section refers to the building automation domain discussed in Chapter I for describing examples.

2.2 Specifying domain concern with the Srijan vocabulary language (SVL)

The domain concern describes an application domain of an IoT application. The domain expert specifies it using SVL. A vocabulary includes specification of resources that are responsible for interacting with entities of interest, including sensors, actuators, storages, and user interfaces. Moreover, it includes region definitions specific to the application domain. We now present SVL for describing the domain concern.

SVL is designed to enable the domain expert to describe a domain vocabulary domain. It offers constructs to specify concepts that interact with entities of interest. Figure III.3 illustrates domain-specific concepts (defined in the conceptual model Figure III.1) that can be specified using SVL. These concepts can be described as $\mathcal{V} = (\mathcal{P}, \mathcal{D}, \mathcal{R})$. $\mathcal{P}$ represents the set of regions, $\mathcal{D}$ represents the set of data structure, and $\mathcal{R}$ represents the set of resources. We describe these concepts in detail as follows:

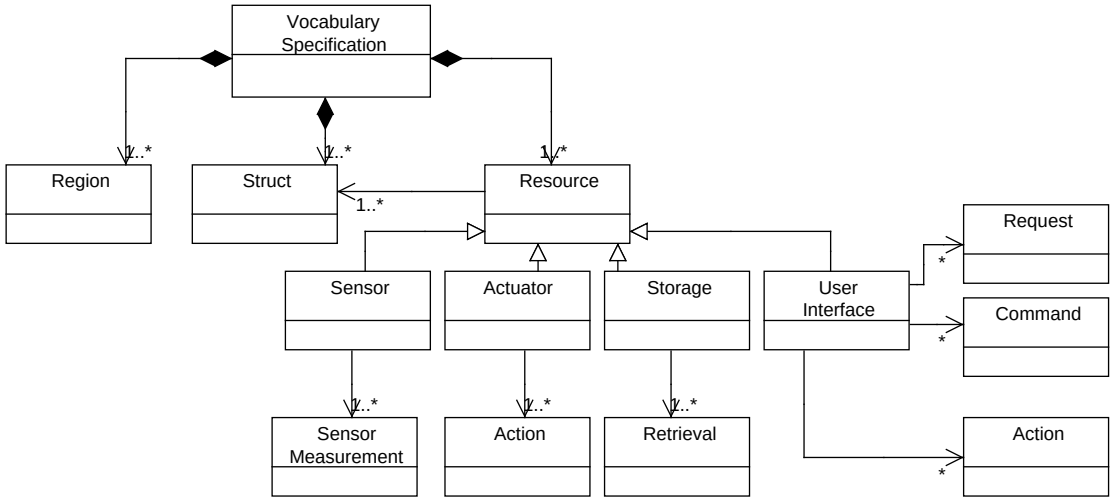

Figure III.3 – Class diagram of domain-specific concepts

regions **($\mathcal{P}$).** It represents the set of regions that are used to specify locations of devices. A region definition includes a region label and region type. For example, the building automation is

reasoned in terms of rooms and floors (considered as region labels), while the transport domain is expressed in terms of highway sectors. Each room or floor in a building may be annotated with an integer value (e.g. room: 1 interprets as room number 1) considered as region type. This construct is declared using the `regions` keyword. Listing III.1 (lines 1-4) shows region definitions for the building automation domain.

data structures **($\mathcal{D}$).** Each resource is characterized by types of information it generates or consumes. A set of information is defined using the `structs` keyword (Listing III.1, line 5). For instance, a temperature sensor may generate a temperature value and unit of measurement (e.g., Celsius or Fahrenheit). This information is defined as `TempStruct` and its two fields (Listing III.1, lines 9-11).

resources **($\mathcal{R}$).** It defines resources that might be attached with devices, including sensors, actuators, storages, or user interfaces. It is defined as $\mathcal{R} = (\mathcal{R}_{sensor}, \mathcal{R}_{actuator}, \mathcal{R}_{storage}, \mathcal{R}_{ui})$. $\mathcal{R}_{sensor}$ represents a set of sensors, $\mathcal{R}_{actuator}$ represents a set of actuators, $\mathcal{R}_{storage}$ represents a set of storages, and R_{ui} represents a set of user interfaces. We describe them in detail as follows:

- ***sensors*** **($\mathcal{R}_{sensor}$):** It defines a set of various types of sensors (e.g., temperature sensor, smoke detector). A set of sensors is declared using the `sensors` keyword (Listing III.1, line 13). $\mathcal{S}_{generate}$ is a set of sensor measurements produced by $\mathcal{R}_{sensor}$. Each sensor ($\mathcal{S} \in \mathcal{R}_{sensor}$) produces one or more sensor measurements ($op \in \mathcal{S}_{generate}$) along with the data-types specified in the data structure ($\mathcal{D}$). A sensor measurement of each sensor is declared using the `generate` keyword (Listing III.1, line 17). For instance, a temperature sensor generates a temperature measurement of `Tempstruct` type (lines 16-17) defined in data structures (lines 9-11).
- ***actuators*** **($\mathcal{R}_{actuator}$):** It defines a set of various types of actuator [4] (e.g., heater, alarm). A set of actuators is declared using the `actuators` keyword (Listing III.1, line 18). $\mathcal{A}_{action}$ is a set of actions performed by $\mathcal{R}_{actuator}$. Each actuator ($\mathcal{A} \in \mathcal{R}_{actuator}$) has one or more actions ($a \in \mathcal{A}_{action}$) that is declared using the `action` keyword. An action of an actuator may take inputs specified as parameters of an action (Listing III.1, line 21). For instance, a heater may has two actions. One is to switch off the heater and second is to set the heater according to a user's temperature preference illustrated in Listing III.1, lines 19-21. The `SetTemp` action takes a user's temperature preference shown in line 21.

4. Since a deployment infrastructure may be shared among a number of different IoT applications and users, it is likely that these applications may have actuation conflicts. This work assumes actuators are pre-configured which can resolve actuation conflicts.

- ***storages*** **($\mathcal{R}_{storage}$)**: It defines a set of storages[5] (e.g., user's profile storage) that might be attached to a device. A set of storages is declared using the `storages` keyword (Listing III.1, line 22). $\mathcal{ST}_{generate}$ represents a set of retrievals of $\mathcal{R}_{storage}$. A retrieval ($rq \in \mathcal{ST}_{generate}$) from the storage ($\mathcal{ST} \in \mathcal{R}_{storage}$) requires a parameter. Such a parameter is specified using the `accessed-by` keyword (Listing III.1, line 24). For instance, a user's profile is accessed from profile storage by his unique badge identification illustrated in Listing III.1, lines 23-24.
- ***user interfaces*** **($\mathcal{R}_{ui}$)**: It defines a set of tasks (e.g., controlling a heater, receiving notification from a fire alarm, or requesting preference information from a database server) available to users to interact with other entities. A set of user interfaces is declared using the `user interfaces` keyword (Listing III.1, line 25). The user interface provides the following tasks:
  - ***command*** **($\mathcal{U}_{command}$)**: It is a set of commands available to users to control actuators, represented as $\mathcal{U}_{command}$. A user can control an actuator by triggering a command (e.g., switch off the heater) declared using the `command` keyword (Listing III.1, line 27).
  - ***action*** **($\mathcal{U}_{action}$)**: It is a set of actions that can be invoked by other entities to notify users, represented as $\mathcal{U}_{action}$. The other resources may notify a user (e.g., notify the current temperature) by invoking an action provided by the user interface. The notification task is declared using the `action` keyword (Listing III.1, line 28).
  - ***request*** **($\mathcal{U}_{request}$)**: It is a set of request though which a user can request other resources for data, represented as $\mathcal{U}_{request}$. A user can retrieve data by requesting a resource (e.g., retrieve my temperature preference). This is declared using the `request` keyword (Listing III.1, line 29).

```
1 regions:
2     Building: integer;
3     Floor: integer;
4     Room: integer;
5 structs:
6     BadgeDetectedStruct
7         badgeID: string;
8         timeStamp: long;
9     TempStruct
```

5. Even though IoT applications may include rich diverse set of storages available today on the Internet (*e.g.*, RDBMs and noSQL databases, using content that is both user generated such as photos as well as machine generated such as sensor data), we restrict our work to key-value data storage services.

```
10          tempValue: double;
11          unitOfMeasurement: string;
12  resources:
13      sensors:
14          BadgeReader
15              generate badgeDetected: BadgeDetectedStruct;
16          TemperatureSensor
17              generate tempMeasurement: TempStruct;
18      actuators:
19          Heater
20              action Off();
21              action SetTemp(setTemp: TempStruct);
22      storages:
23          ProfileDB
24              generate  profile: TempStruct accessed-by badgeID:
                    string;
25      userinterfaces:
26          EndUserGUI
27              command Off();
28              action DisplayData(displayTemp: TempStruct);
29              request profile(badgeID);
```

Listing III.1 – Code snippet of the building automation domain using SVL. Keywords are printed in blue. For a full Listing of the vocabulary specification, see Appendix 2.

The regions ($\mathcal{P}$), data structures ($\mathcal{D}$), and resources ($\mathcal{R}$) defined using SVL in the vocabulary are used to customize the grammar of SAL, and can be exploited by tools to provide support such as code completion to the software designer, discussed next.

2.3 Specifying functional concern

This concern describes computational services and how they interact with each other to describe functionality of an application. We describe the computational services and interactions among them using SAL (discussed in Section 2.3.1). The development framework customizes the SAL grammar to make domain-specific knowledge defined in the vocabulary available to the software designer and use it to generate an architecture framework. The application developer leverages this generated framework and implements the application logic on top of it (discussed

in Section 2.3.2).

2.3.1 Srijan architecture language (SAL)

Based on a vocabulary, the SAL grammar is customized to enable the software designer to design an application. Specifically, sensors ($\mathcal{R}_{sensor}$), actuators ($\mathcal{R}_{actuator}$), storages ($\mathcal{R}_{storage}$), user interfaces ($\mathcal{R}_{ui}$), and regions ($\mathcal{P}$) defined in the vocabulary become possible set of values for certain attributes in SAL (see underlined words in Listing III.2). Appendix 1 presents a customized SAL grammar with respect to the building automation domain.

Figure III.4 illustrates concepts related-to a computational service that can be specified using SAL. It can be described as $\mathcal{A}_v = (\mathcal{C})$. $\mathcal{C}$ represents a set of computational services. It is described as $\mathcal{C} = (\mathcal{C}_{generate}, \mathcal{C}_{consume}, \mathcal{C}_{request}, \mathcal{C}_{command}, \mathcal{C}_{in-region}, \mathcal{C}_{hops})$. $\mathcal{C}_{generate}$ represents a set of outputs produced by computational services. $\mathcal{C}_{consume}$ is a set of inputs consumed by computational services. The inputs could be data produced by other computational services or sensors ($\mathcal{R}_{sensor}$). $\mathcal{C}_{request}$ represents a set of request by computational services to retrieve data from the storages ($\mathcal{R}_{storage}$). $\mathcal{C}_{command}$ represents a set of commands to invoke actuators ($\mathcal{R}_{actuator}$) or user interfaces ($\mathcal{R}_{ui}$). $\mathcal{C}_{in-region}$ is a set of regions ($\mathcal{R}_{region}$) where computational services can be placed. $\mathcal{C}_{hops}$ is a set of regions ($\mathcal{R}_{region}$) where computational services receive data. In the following, we describe these concepts in detail.

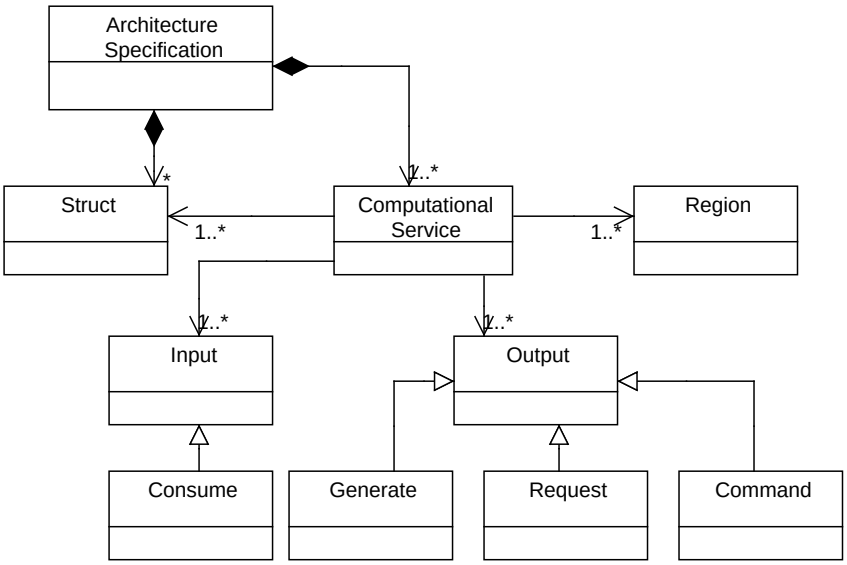

Figure III.4 – Class diagram of functionality-specific concepts

consume **($\mathcal{C}_{consume}$) and** ***generate*** **($\mathcal{C}_{generate}$).** These two concepts together define publish/-

subscribe interaction mode that provides subscribers with the ability to express their interest in an event, generated by a publisher, that matches their registered interest. A computational service represents the publish and subscribe using *generate* and *consume* concept respectively. We describe these two concepts in details as follows:

- ***consume***: It represents a set of subscriptions (or consumes) expressed by computational services to get event notifications generated by sensors ($\mathcal{S}_{generate}$) defined in the vocabulary specification or other computational services ($\mathcal{C}_{generate}$) defined in the architecture specification. Thus, $\mathcal{C}_{consume}$ can be $\mathcal{C}_{generate} \cup \mathcal{S}_{generate}$. A consume ($c \in \mathcal{C}_{consume}$) of a computational service is expressed using the `consume` keyword. The computational service expresses its interest by an event name. For instance, a computational service `RoomAvgTemp`, which calculates an average temperature of a room, subscribes its interest by expressing event name `tempMeasurement` illustrated in Listing III.2, line 9.
- ***generate***: It represents a set of publications (or generates) that are produced by computational services. A generate ($g \in \mathcal{C}_{generate}$) of a computational service is expressed using the `generate` keyword. The computational service transforms data to be consumed by other computational services in accordance with the application needs. For instance, the computational service `RoomAvgTemp` consumes temperature measurements (i.e., `tempMeasurement`), calculates an average temperature of a room, and generates `roomAvgTempMeasurement` (Listing III.2, lines 7-9) that is used by `RoomController` service (Listing III.2, lines 11-12).

request **($\mathcal{C}_{request}$).** It is a set of requests, issued by computational services, to retrieve data from storages ($\mathcal{R}_{storage}$) defined in the vocabulary specification. A request is a one-to-one synchronous interaction with a return values. In order to fetch data, a requester sends a request message containing an access parameter to a responder. The responder receives and processes the request message, ultimately returns an appropriate message as a response. An access ($rq \in \mathcal{C}_{request}$) of the computational service is specified using `request` keyword. For instance, a computational service `Proximity` (Listing III.2, line 5), which wants to access user's profile data, sends a request message containing profile information as an access parameter to a storage `ProfileDB` (Listing III.1, line 24).

command **($\mathcal{C}_{command}$).** It is a set of commands, issued by a computational service to trigger actions provided by actuators ($\mathcal{R}_{actuator}$) or user interfaces ($\mathcal{R}_{ui}$). So, it can be a subset of $\mathcal{A}_{action} \cup \mathcal{U}_{action}$. The software designer can pass arguments to a command depend on action signature provided by actuators or user interfaces. Moreover, he specifies a scope of command, which specifies a region where commands are issued. A command is specified using the `com-`

`mand` keyword. An example of command invocation is given in line 14 of Listing III.2. The room controller service (i.e., `roomController`), which regulates temperature, issues a `SetTemp` command with a preferred temperature as an argument (i.e., `settemp`) to heaters (Listing III.1, line 21).

in-region **($\mathcal{C}_{in-region}$) and *hops*** **($\mathcal{C}_{hops}$).** To facilitate the scalable operations within an IoT application, devices should be grouped to form a cluster based on their spatial relationship [Shen et al., 2001] (e.g.,"devices are in room#1"). The grouping could be recursively applied to form a hierarchy of clusters. Within a cluster, a computational service is placed to receive and process data from its cluster of interest. Figure III.5 shows this concept for more clarity. The temperature data is first routed to a local average temperature service (i.e., `RoomAvgTemp`), deployed in per room, then later per floor (i.e., `FloorAvgTemp`), and then ultimately routed to building average temperature service (i.e., `BuildingAvgTemp`).

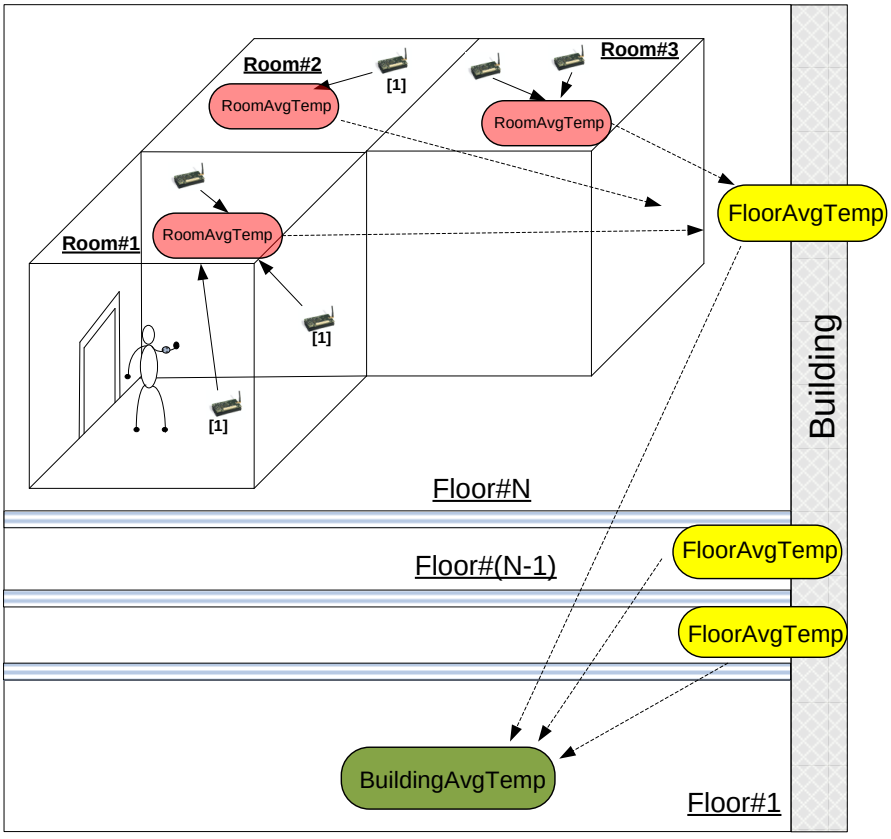

Figure III.5 – Clustering in the smart building application. The device with temperature sensor is numbered as [1].

SAL offers *scope* constructs to define both the service placement ($\mathcal{C}_{in-region}$) and its data interest ($\mathcal{C}_{hops}$). The service placement (defined using the `in-region` keyword) is used to govern

a placement of computational service in a cluster. The service placement can be in regions defined in a vocabulary specification. So, it is a subset of $\mathcal{P}$.

The data interest of a computational service is used to define a cluster from which the computational service wants to receive data. The data interest can be in regions defined in the vocabulary specification. So, it is a subset of $\mathcal{P}$. It is defined using the `hops` keyword. The syntax of this keyword is `hops: radius: unit of radius`. Radius is an integer value. The unit of radius is a cluster value. For example, if a computational service `FloorAvgTemp` deployed on floor number 12 has a data interest `hops: i: Floor`, then it wants data from all floors starting from `12`-th floor to `(12+i)`-th floor, and all floors starting from `12`-th floor to `(12-i)`-th floor .

Figure III.6 shows the layered architecture of the smart building application. Computational services are fueled by sensing components. They process inputs data and take appropriate decisions by triggering actuators. We illustrate SAL by examining a code snippet in Listing III.2, which describes a part of Figure III.6. This code snippet revolves around the actions of the `Proximity` service (Listing III.2, lines 2-6), which coordinates events from the `BadgeReader` with the content of `ProfileDB` storage service. To do so, the `Proximity` composes information from two sources, one for badge events (i.e., badge detection), and one for requesting the user's temperature profile from `ProfileDB`, expressed using the `request` keyword (Listing III.2, line 5). Input data is declared using the `consume` keyword that takes source name and data interest of a computational service from logical region (Listing III.2, line 4). The declaration of `hops: 0: room` indicates that the computational service is interested in consuming badge events of the current room. The `Proximity` service is in charge of managing badge events of room. Therefore, we need `Proximity` service to be partitioned per room using `in-region: room` (Listing III.2, line 6). The outputs of the `Proximity` and `RoomAvgTemp` are consumed by the `RoomController` service (Listing III.2, lines 11-15). This service is responsible for taking decisions that are carried out by invoking commands declared using the `command` keyword (Listing III.2, line 14).

```
1 computationalServices:
2     Proximity
3       generate tempPref: UserTempPrefStruct;
4       consume  badgeDetected from hops:0: Room;
5       request  profile( badgeID);
6       in-region: Room;
7     RoomAvgTemp
8       generate roomAvgTempMeasurement:TempStruct;
```

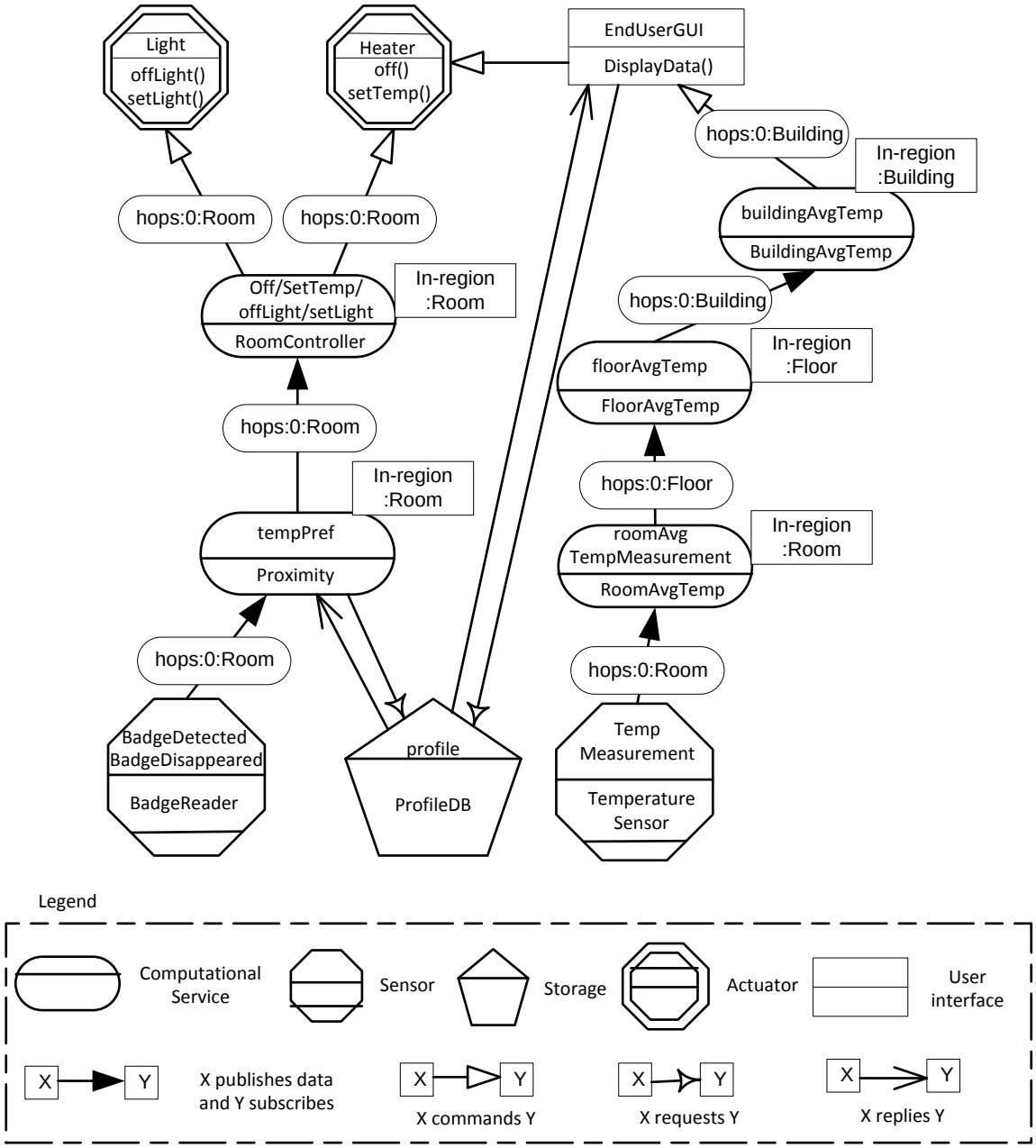

Figure III.6 – Layered architecture of the smart building application.

```
9       consume  tempMeasurement from hops:0: Room ;
10      in-region: Room;
11    RoomController
12      consume roomAvgTempMeasurement from hops:0: Room;
13      consume tempPref from hops:0: Room;
14      command  SetTemp( setTemp) to hops:0: Room;
```

```
15        in-region: Room;
```

Listing III.2 – A code snippet of the architecture specification for the smart building application using SAL. The language keywords are printed in blue, while the keywords derived from vocabulary are printed underlined. For the full listing of the architecture specification, see Appendix 2.

2.3.2 Implementing application logic

Leveraging the architecture specification, we generate a framework to aid the application developer. The generated framework contains abstract classes corresponding to the architecture specification. The abstract classes include two types of methods: (1) *concrete methods* to interact with other components transparently through the middleware and (2) *abstract methods* that allow the application developer to program the application logic. The application developer implements each abstract method of generated abstract class. The key advantage of this framework is that a framework structure remains uniform. Therefore, the application developer have to know only locations of abstract methods where they have to specify the application logic.

Abstract methods. For each input declared by a computational service, an abstract method is generated for receiving data. This abstract method is then implemented by the application developer. The class diagram in Figure III.7 illustrates this concept. This class diagram uses *italicized* text for the `Proximity` class, which represents an abstract class, and `onNewbadgeDetected()` that represents abstract method. Then, it is implemented in the `SimpleProximity` class.

Listing III.3 and III.4 show Java code corresponding to the class diagram illustrated in Figure III.7. From the `badgeDetected` input of the `Proximity` declaration in the architecture specification (Listing III.2, lines 2-6), the `onNewbadgeDetected()` abstract method is generated (Listing III.3, line 16). This method is implemented by the application developer. Listing III.4 illustrates the implementation of `onNewbadgeDetected()`. It updates a user's temperature preference and sets it using `settempPref()` method.

```
1 public abstract class  Proximity {
2     private String partitionAttribute = "Room";
3     public void notifyReceived(String eventName, Object arg) {
4         if (eventName.equals("badgeDetected")) {
5             onNewbadgeDetected((BadgeDetectedStruct) arg);
6         }
7     }
```

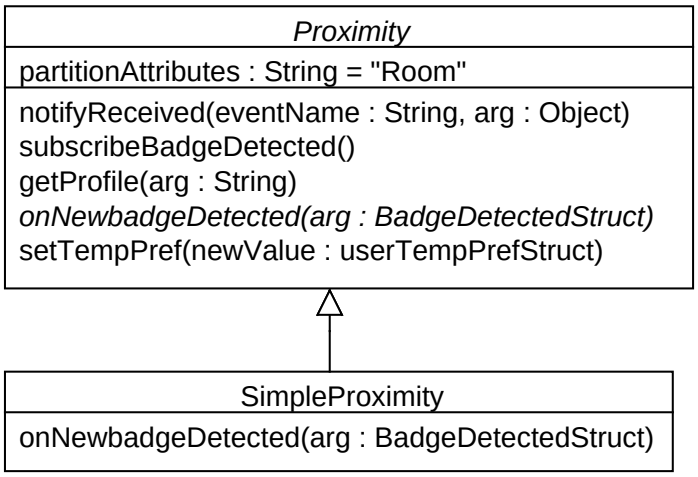

Figure III.7 – Class diagram represents (1) the abstract class `Proximity` with its abstract method `onNewbadgeDetected()` illustrated in *italicized* text, and (2) the concrete implementation of `onNewbadgeDetected()` method is the `SimpleProximity` class.

```
8     public void subscribebadgeDetected() {
9       Region regionInfo = getSubscriptionRequest(
10          partitionAttribute, getRegionLabels(), getRegionIDs());
11        PubSubMiddleware.subscribe(this, "badgeDetected",
             regionInfo);
12    }
13    protected TempStruct getprofile(String arg) {
14        return (TempStruct) PubSubMiddleware.sendCommand("
             getprofile", arg, myDeviceInfo);
15  }
16    protected abstract void onNewbadgeDetected(
       BadgeDetectedStruct arg);
17    protected void settempPref(UserTempPrefStruct newValue) {
18        if (tempPref != newValue) {
19            tempPref = newValue;
20        PubSubMiddleware.publish("tempPref", newValue,
             myDeviceInfo);
21    }
22  }
```

```
23 }
```

Listing III.3 – The Java abstract class `Proximity` generated from the declaration `Proximity` in the architecture specification.

```
1 public class SimpleProximity extends Proximity {
2     public void onNewbadgeDetected(BadgeDetectedStruct arg) {
3         long timestamp = ((long) (System.currentTimeMillis())) *
              1000000;
4         UserTempPrefStruct userTempPref = new UserTempPrefStruct(
5       arg.gettempValue(), arg.getunitOfMeasurement(), timestamp);
6     settempPref(userTempPref);
7   }
8 }
```

Listing III.4 – The concrete implementation of the Java abstract class `Proximity` from Listing III.3, written by the application developer.

Concrete methods. The compilation of an architecture specification generates concrete methods to interact with other software component transparently. The generated concrete methods has the following two advantages:

1. **Abstracting heterogeneous interactions.** To abstract heterogeneous interactions among software components, a compiler generates concrete methods that takes care of heterogeneous interactions. For instance, a computational service processes input data and produces refined data to its consumers. The input data is either notified by other component (i.e., publish/subscribe) or requested (i.e., request/response) by the service itself. Then, outputs are published. The concrete methods for these interaction modes are generated in an architecture framework. The lines 2 to 6 of Listing III.2 illustrates these heterogeneous interactions. The `Proximity` service has two inputs: (1) It receives `badgeDetect` event (Listing III.2, line 4). Our framework generates the `subscribebadgeDetected()` method to subscribe `badgeDetected` event (Listing III.3, lines 8-12). Moreover, it generates the implementation of `notifyReceived()` method to receive the published events (Listing III.3, lines 3-7). (2) It requests `profile` data (Listing III.2, line 5). A `sendcommand()` method is generated to request data from other components (Listing III.3, lines 13-15).
2. **Abstracting large scale.** To address the scalable operations, a computational service annotates (1) its inputs with data interest, and (2) its placement in the region. Service placement and data interest jointly define a scope of a computational service to gather data.

A generated architecture framework contains code that defines both data interest and its placement. For example, to get the `badgeDetected` event notification from the `BadgeReader` (Listing III.2, line 4), the `subscribebadgeDetected()` method (Listing III.3, lines 8-12) is generated in the `Proximity` class. This method defines the data interest of a service from where it receives data. The value of `partitionAttribute` (Listing III.3, line 2), which comes from the architecture specification (Listing III.2, line 6), defines the scope of receiving data. The above constructs are empowered by our choice of middleware, which is a variation of the one presented in [Mottola et al., 2007], and enables delivery of data across logical scopes.

2.4 Specifying deployment concern

This concern describes information about a target deployment containing various attributes of devices (such as location, type, attached resources) and locations where computational services are executed in a deployment, described using SDL (discussed in Section 2.4.1). In order to map computational services to devices, we present a mapping technique that produces a mapping from a set of computational services to a set of devices (discussed in Section 2.4.2).

2.4.1 Srijan deployment language (SDL)

Figure III.8 illustrates deployment-specific concepts (defined in the conceptual model Figure III.1), specified using SDL. It includes device properties (such as name, type), regions where devices are placed, and resources that are hosted by devices. The resources ($\mathcal{R}$) and regions ($\mathcal{P}$) defined in a vocabulary become a set of values for certain attributes in SDL (see the underlined words in Listing III.5). Appendix 1 presents a customized SDL grammar with respect to the building automation domain. SDL can be described as $\mathcal{T}_v = (D)$. D represents a set of devices. A device ($d \in D$) can be defined as $(\mathcal{D}_{region}, \mathcal{D}_{resource}, \mathcal{D}_{type}, \mathcal{D}_{mobile})$. D_{region} represents a set of device placements in terms of regions defined in a vocabulary. $D_{resource}$ is a subset of resources defined in a vocabulary. D_{type} represents a set of device type (e.g., JavaSE device, Android device) that is used to pick an appropriate device driver from a device driver repository. D_{mobile} represents a set of two boolean values (true or false). The true value indicates a location of a device is not fixed, while the false value shows a fixed location. Listing III.5 illustrates a deployment specification of the smart building application. This snippet describes a device called `TemperatureMgmt-Device-1` with an attached `TemperatureSensor` and `Heater`, situated in `building` 15, `floor` 11, `room` 1, it is JavaSE enabled and non-mobile device.

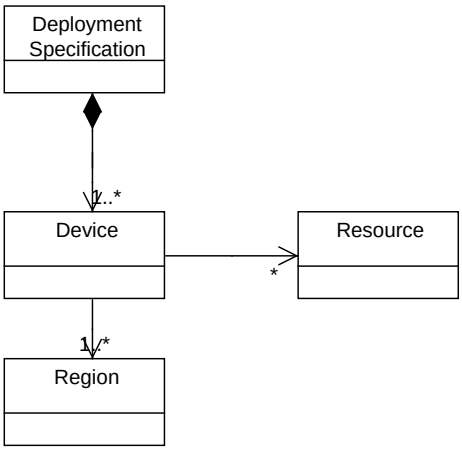

Figure III.8 – Class diagram of deployment-specific concepts

Note that although individual listing of each device's attributes appears tedious, *i*) we envision that this information can be extracted from inventory logs that are maintained for devices purchased and installed in systems, and *ii)* thanks to the separation between the deployment and functional concern in our approach, the same deployment specification can be re-used across IoT applications of a given application domain.

```
devices:
  TemperatureMgmt-Device-1:
    region:
       Building: 15 ;
       Floor: 11;
       Room: 1;
    resources: TemperatureSensor, Heater;
    type: JavaSE;
    mobile: false;
  ...
```

Listing III.5 – Code snippet of a deployment specification for the building automation domain using SDL. The language keywords are printed in `blue`, while the keywords derived from a vocabulary are printed underlined. For a full Listing of a deployment specification, see Appendix 2.

2.4.2 Mapping

This section presents our mapping algorithm that decides devices for a placement of computational services. It takes inputs as (1) a list of devices D defined in a deployment specifi-

cation (see listing III.5) and (2) a list of computational services C defined in an architecture specification (see listing III.2). It produces a mapping of computational services to a set of devices.

We presents the mapping algorithm (see Algorithm 1) that comprises two steps. The first step (lines 4-9) constructs the two key-value data structures from a deployment specification. These two data structures are used in the second step. The second step (lines 10-20) selects devices randomly and allocates computational services to the selected devices [6]. In order to give more clarity to readers, we describes these two steps in detail below.

The first step (Algorithm 1, lines 4-9) constructs two key-value data structures $regionMap$ and $deviceListByRegionValue$ from D. The $regionMap$ (line 6) is a key-value data structure where $regionName$ (e.g., `Building`, `Floor`, `Room` in the listing III.5) is a key and $regionValue$ (e.g., `15`, `11`, `1` in the listing III.5) is a value. The $deviceListByRegionValue$ (line 7) is a key-value data structure where $regionValue$ is a key and $device$ (e.g., `TemperatureMgmt-Device-1` in the listing III.5) is a value. Once, these two data structures are constructed, we use them for the second step (lines 10-20).

The second step (Algorithm 1, lines 10-20) selects a device and allocates computational services to the selected device. To perform this task, the line 10 retrieves all keys (in our example `Building`, `Floor`, `Room`) of $regionMap$ using $getKeySet()$ function. For each computational service (e.g., `Proximity`, `RoomAvgTemp`, `RoomController` in listing III.2), the selected key from the $regionMap$ is compared with a partition value of a computational component (line 12). If the value match, the next step (lines 13-17) selects a device randomly and allocates a computational service to the selected device.

Computational complexity. The first step (Algorithm 1, lines 4-9) takes $O(mr)$ times, where m is a number of devices and r is a number of region pairs in each device specification. The second step (Algorithm 1, lines 10-20) takes $O(nks)$ times, where n is a number of region names (e.g., building, floor, room for the building automation domain) defined in a vocabulary, k is a number of computational services defined in an architecture specification, and s is a number of region values specified in a deployment specification. Thus, total computational complexity of the mapping algorithm is $O(mr + nks)$.

6. A mapping algorithm cognizant of heterogeneity, associated with devices of a target deployment, is a part of our future work. See future work for detail.

Algorithm 1 Mapping Algorithm

Input: List D of m numbers of devices, List C of k numbers computational services
Output: List *mappingOutput* of m numbers that contains assignment of C to D

```
1: Initialize regionMap key-value pair data structure
2: Initialize deviceListByRegionValue key-value pair data structure
3: Initialize mappingOutput key-value pair data structure
4: for all device in D do
5:    for all pairs (regionName, regionValue) in device do
6:       regionMap[regionName] ← regionValue // construct regionMap with regionName
         as key and assign regionValue as Value
7:       deviceListByRegionValue[regionValue] ← device
8:    end for
9: end for
10: for all regionName in regionMap.getKeySet() do
11:    for all computationalservice in C do
12:       if computationalservice.partitionValue() = regionName then
13:          for all regionValue in regionMap.getValueSet(regionName) do
14:             deviceList ← deviceListByRegionValue.getValueSet(regionValue)
15:             selectedDevice ← selectRandomDeviceFromList(deviceList)
16:             mappingOutput[selectedDevice] ← computationalservice
17:          end for
18:       end if
19:    end for
20: end for
21: return mappingOutput
```

2.5 Specifying platform concern

This concern describes software components that act as a translator between a hardware device and an application. Because these components are operating system-specific, the device developer implements them by hand. To aid the device developer, we generate a vocabulary framework to implement platform-specific device drivers. In the following section, we describe it in more detail.

2.5.1 Implementing device drivers

Leveraging the vocabulary specification, our system generates a vocabulary framework to aid the device developer. The vocabulary framework contains *concrete classes* and *interfaces* corresponding to resources defined in a vocabulary. A concrete class contains concrete methods for interacting with other software components and platform-specific device drivers. The

interfaces are implemented by the device developer to write platform-specific device drivers. In order to enable interactions between concrete class and platform-specific device driver, we adopt the factory design pattern [Gamma et al., 1995]. This pattern provides an interface for a concrete class to obtain an instance of different platform-specific device driver implementations without having to know what implementation the concrete class obtains. Since the platform-specific device driver implementation can be updated without necessitating any changes in code of concrete class, the factory pattern has advantages of encapsulation and code reuse. We illustrate this concept in the following paragraph with a `BadgeReader` example.

The class diagram in Figure III.9 illustrates the concrete class `BadgeReader`, the interface `IBadgeReader`, and the associations between them through the factory class `BadgeReaderFactory`. The two abstract methods of the `IBadgeReader` interface (Listing III.8, lines 1-4) are implemented in the `AndroidBadgeReader` class (Listing III.9, lines 1-10). The platform-specific implementation is accessed through the `BadgeReaderFactory` class (Listing III.7, lines 1-10). The `BadgeReaderFactory` class returns an instance of platform-specific implementations according to request by the concrete method `registerBadgeReader()` in the `BadgeReader` class (Listing III.6, lines 12-15). In the following, we describe this class diagram with code snippet.

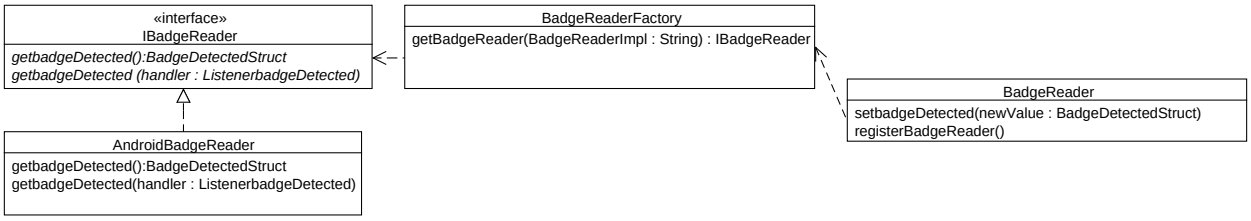

Figure III.9 – Class diagram representing (1) the interface `IBadgeReader` and the implementation of two abstract methods in the `AndroidBadgeReader` class, (2) the concrete class `BadgeReader` that refers the `AndroidBadgeReader` through the `BadgeReaderFactory` factory class.

Concrete class. For each resource declared in a vocabulary specification, a concrete class is generated. This class contains concrete methods for interacting with other components transparently (similar to discussed in Section 2.3.2) and for interacting with platform-specific implementations. For example, the `BadgeReader` (Listing III.6, lines 1-16) class is generated from the `BadgeReader` declaration (Listing III.1, lines 14-15). The generated class contains the `registerBadgeReader()` method (Listing III.6, lines 12-15). This method first obtains a reference of one (in our example Android) of platform-specific implementations, then uses that reference to create an object of that device-specific type (Listing III.6, line 13). This reference is used to disseminate badgedetect event (Listing III.6, lines 6-11).

```
1  public class BadgeReader {
2      protected void setbadgeDetected(BadgeDetectedStruct newValue)
           {
3              ...
4          PubSubMiddleware.publish("badgeDetected", newValue,
               DeviceInfo);
5      }
6      badgeDetected badgeDetectEvent = new badgeDetected() {
7          public void onNewbadgeDetected(BadgeDetectSt resp) {
8              BadgeDetectSt sBadgeDetectSt = new BadgeDetectSt(resp
                   .getbadgeID(), resp.gettimeStamp());
9              publishbadgeDetectedEvent(sBadgeDetectSt);
10         }
11     };
12     protected void registerBadgeReader(){
13         IBadgeReader objBadgeReader = BadgeReaderFactory.
               getBadgeReader("Android");
14         objBadgeReader.getbadgeDetected(badgeDetectEvent);
15     }
16 }
```

Listing III.6 – The Java `BadgeReader` class generated from the `BadgeReader` declaration in the vocabulary specification.

Interfaces. For each resource declared in a vocabulary specification, interfaces are generated. Each interface contains synchronous and asynchronous abstract methods corresponding to a resource declaration. These methods are implemented by the device developer to write device-specific drivers. For example, our development system generates a vocabulary framework that contains the interface `IBadgeReader` (Listing III.8, lines 1-4) corresponding to the `BadgeReader` (Listing III.1, lines 14-17) declaration in the vocabulary specification. The device developer programs Android-specific implementations in the `AndroidBadgeReader` class by implementing the methods `getbadgeDetected()` and `getbadgeDetected(handler)` of the generated interface `IBaderReader` (Listing III.9, lines 1-10).

```
1  public class BadgeReaderFactory {
2      public static IBadgeReader getBadgeReader(String
           nameBadgeReader) {
```

```
3
4       if(nameBadgeReader.equals("Android"))
5           return new AndroidBadgeReader();
6
7        if (nameBadgeReader.equals("PC"))
8        return new PCBadgeReader();
9      }
10 }
```

Listing III.7 – The Java `BadgeReaderFactory` class.

```
1 public interface IBadgeReader {
2   public  BadgeDetectedStruct getbadgeDetected();
3   public  void getbadgeDetected(ListenerbadgeDetected handler);
4 }
```

Listing III.8 – The Java interface `IBadgeReader` generated from the `BadgeReader` declaration in the vocabulary specification.

```
1 public class AndroidBadgeReader implements IBadgeReader {
2   @Override
3   public BadgeDetectedStruct getbadgeDetected() {
4         // The device developer implements platform-specific code
              here
5   }
6   @Override
7   public void getbadgeDetected(ListenerbadgeDetected handler) {
8         // The device developer implements platform-specific code
              here
9   }
10 }
```

Listing III.9 – The device developer writes Android-specific device driver of a badge reader by implementing the `IBadgeReader` interface.

2.6 Handling evolution

Evolution is an important aspect in IoT application development where resources and computational services are added, removed, or extended. To deal with these changes, we separate

IoT application development into different concerns and allows an iterative development for these concerns. We now review main evolution cases in each development concern and how our approach handles them.

2.6.1 Evolution in functional concern

Evolution could be addition, removal, or extension of computational services. To deal with them, we adopt an iterative development approach, similar to the work in [Cassou et al., 2012], illustrated in Figure III.10. A change in an architecture specification requires recompilation of it. The recompilation generates a new architecture framework and preserves the previously written application logic. This requires changes in the existing application logic implementations. The application developer commits changes manually. Moreover, the changed architecture specification is compiled by the mapper to generate new mapping files that replaces old mapping files.

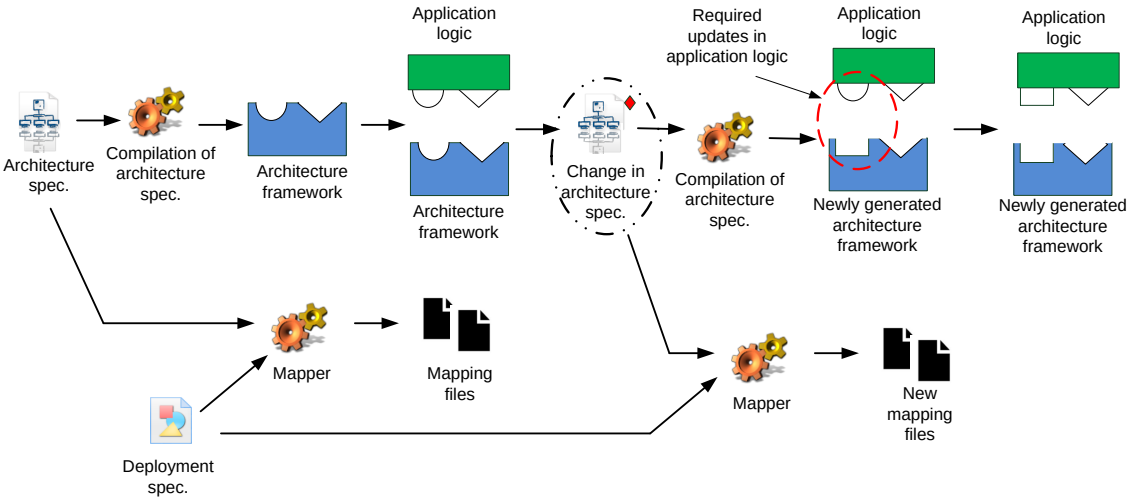

Figure III.10 – Handling evolution in the functional concern

We now review main evolution cases in the functional concern.

Changing functionality. It refers to a change in behaviors of an application. For example, while an application might be initially defined to switch on an air-conditioner when a temperature of room is greater than $30°C$, a new functionality might be to open a window. This case requires to write a new architecture specification and application logic.

Adding a new computational service. It refers to the addition of a new computational service in an architecture specification. The application developer implements the application logic of the newly added computational services.

Removing a computational service. It refers to the removal of an existing computation service from an architecture specification. The application developer has to manually remove application logic files of the removed computational service.

Adding a new input source. A new input of a computational service, represented as `consume` keyword, can be added. The application developer implements a generated abstract method corresponding to a new input in application logic files.

Removing a input source. An input can be removed from a computational service. In this case, the abstract method that implements the application logic becomes dead in application logic files. The IDE automatically reports errors. The application developer has to remove this dead abstract method manually.

Removing an output or command. An output (`generate` keyword) or command (`command` keyword) can be removed from an architecture specification. In this case code, which deals with output or command, becomes dead in application logic files. The application developer has to manually remove dead code.

2.6.2 Evolution in deployment concern

The evolution could be change in a deployment (e.g., the smart building application is deployed on different buildings with different deployment scenarios.), changes in a distribution of devices (e.g., device is transferred from a floor#12 to floor#14), addition or removal of devices in a deployment.

Figure III.11 illustrates our approach to cope with the above evolution cases in the deployment concern. Initially, the mapping files are generated by the mapper module. In case of changes, the network manager does not require any code changes beyond a deployment specification. Then, re-mapping of devices and computational services generates new mapping files that replace old mapping files.

2.6.3 Evolution in platform concern

Evolution in the platform concern refers to a change in device drivers. An application may be updated when new platform-specific software features of device drivers are available. For instance, a new Android API for a location sensor is available that senses a device location more accurately. Figure III.12 illustrates our approach to cope with these evolutions. Initially, an individual device-specific code is produced from the functional, deployment, and platform concern by the linker module. In case of addition or removal of device drivers, the device

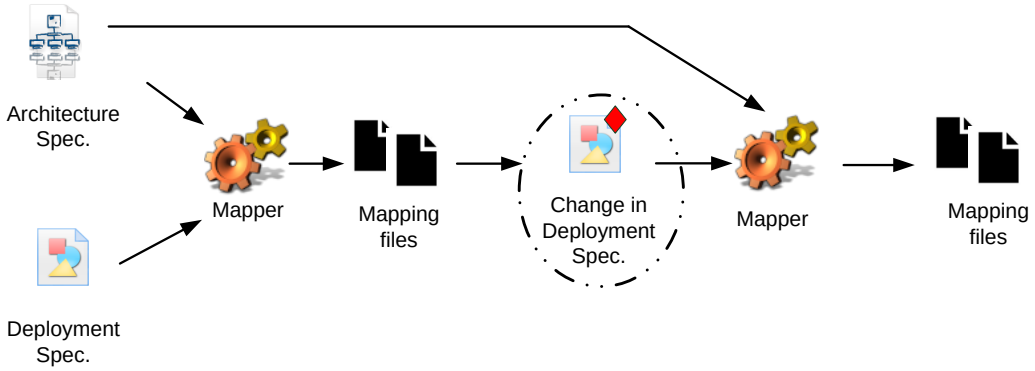

Figure III.11 – Handling evolution in the deployment concern

developer does not need to make code changes beyond device drivers. Then, re-linking with other concerns generates device-specific code.

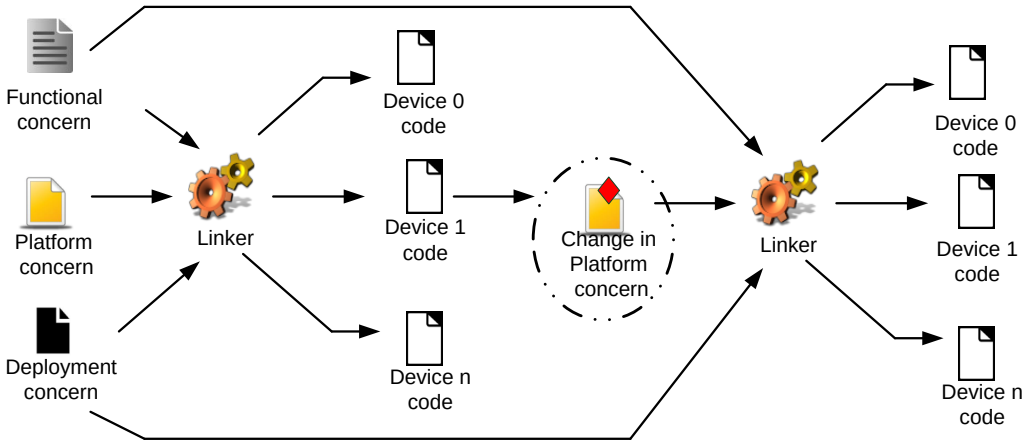

Figure III.12 – Handling evolution in the platform concern

2.6.4 Evolution in domain concern

Evolution in this concern could be addition, removal, or extension of resources. To cope with them, our approach allows an iterative development illustrated in Figure III.13. The compilation of a vocabulary specification generates three artifacts: (1) a vocabulary framework that aids the device developer to write device drivers, (2) a customized architecture grammar according to a vocabulary specification that aids the software designer to write an architecture specification,

and (3) a customized deployment grammar according to a vocabulary specification that aids the network manager to write a deployment specification. A change in the vocabulary specification requires recompilation of it. The re-compilation generates a new vocabulary framework and preserves previously written device drivers. This requires changes in the existing device driver implementations. The device developer has to commit changes manually. Moreover, the re-compilation generates a new customized architecture and deployment grammar, which replace old grammar files.

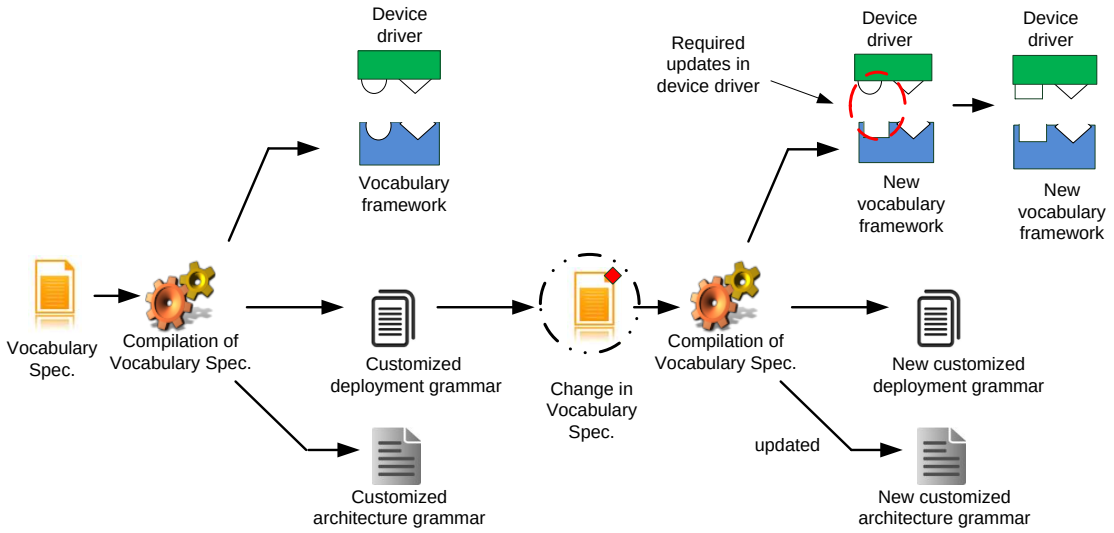

Figure III.13 – Handling evolution in the domain concern

We now review main evolution cases in the domain concern.

Adding new resources. New resources can be added in a vocabulary specification. This requires re-compilation of a vocabulary specification. This results into availability of new resources to a deployment and architecture grammar. Moreover, a vocabulary framework is updated. The device developer requires an implementation of new device drivers on top of the generated vocabulary framework.

Removing resources. This requires re-compilation of a vocabulary specification. The code in device drivers associated with removed resources becomes dead. The device developer has to manually remove source files associated with resources.

Extending resources. Existing resources in a vocabulary specification can be extended with additional functionality. This extension does not require any changes in device drivers besides implementing new functionality on top of a generated vocabulary framework. The extended functionality of resources are now available to a architecture and deployment grammar for further use.

3 Chapter summary

This chapter presents our approach for IoT application development. It separates IoT application development into different concerns and integrates a set of high-level modeling languages to specify them. This approach is supported by automation techniques at different phases of IoT application development and allows an iterative development to handle evolutions in different concerns. Our approach thus addresses the challenges discussed in Chapter I in the following manner :

Lack of division of roles. Our approach identifies roles of each stakeholder and separates them according to their skills. The clear identification of expectations and specialized skills of each stakeholder helps them to play their part effectively, thus promoting a suitable division of work among stakeholders involved in IoT application development.

Heterogeneity. SAL and SVL provide abstractions to specify different types of devices, as well as heterogeneous interaction modes in a high-level manner. Further, high-level specifications written using SAL and SVL are compiled to a programming framework that (1) abstracts heterogeneous interactions among software components and (2) aids the device developers to write code for different platform-specific implementations.

Scale. SAL allows the software designer to express his requirements in a compact manner regardless of the scale of a system. Moreover, it offers scope constructs to facilitate scalable operations within an application. They reduce scale by enabling hierarchical clustering in an application. To do so, these constructs group devices to form a cluster based on their spatial relationship (e.g., "devices are in room#1"). Within a cluster, a cluster head is placed to receive and process data from its cluster of interest. The grouping could be recursively applied to form a hierarchy of clusters. The scale issue is thus handled, thanks to the use of a middleware that supports logical scopes and regions.

Different life cycle phases. Our approach is supported by code generation, task-mapping, and linking techniques. These techniques together provide automation at different life cycle phases. At the development phase, the code generator produces (1) an architecture framework that allows the application developer to focus on the application logic by producing code that hide low-level interaction details and (2) a vocabulary framework to aid the device developer to implement platform-specific device drivers. At the deployment phase, the mapping and linking together produce device-specific code to result in a distributed software system collaboratively hosted by individual devices. To support maintenance phase, our approach separates IoT application development into different concerns and allows an iterative development, supported by the automation techniques.

Lack of special-purpose modeling languages. Our approach separates the domain concern from other concerns and customizes a architecture specification (functional concern) and deployment specification (deployment concern) with respect to domain concepts defined in a vocabulary specification. The advantage of this customization is that domain-specific knowledge is made available to stakeholders. This guides stakeholders for describing specifications with respect to an application domain.

Chapter IV

SrijanSuite: implementation of our approach

In Chapter III, we have proposed a development framework. Since the main goal of this research is to make IoT application development easy for stakeholders, we believe that our development framework should be supported by tools to be applicable it in an effective way. This chapter presents an implementation of the development framework, realized as a suite of tools. We call it as SrijanSuite system.

This chapter is structured as follows. Section 1 provides an overview of SrijanSuite system that consists of five components: editor, compiler, mapper, linker, and runtime system. These components are individually discussed in Section 2.1 (editor), Section 2.2 (compiler), Section 2.3 (mapper), Section 2.4 (linker), and Section 2.5 (runtime system). Section 3 presents Eclipse plug-in that integrates these components. Finally, this chapter is concluded with a summary in Section 4.

1 System overview

Figure IV.1 shows the various components of SrijanSuite that stakeholders can use. This system consists of the following components.

1. **Editor**: It helps stakeholders to write high-level specifications, including vocabulary, architecture, and deployment specification.
2. **Compiler**: It parses the high-level specifications and translates them into the code that can be used by other components in the system.
3. **Mapper**: It maps computational services described in an architecture specification to devices listed in an deployment specification.
4. **Linker**: It combines and packs code generated by various stages of compilation into packages that can be deployed on devices.

5. **Runtime system**: It is responsible for a distributed execution of an application.

We present the above mentioned components in detail in the following sections.

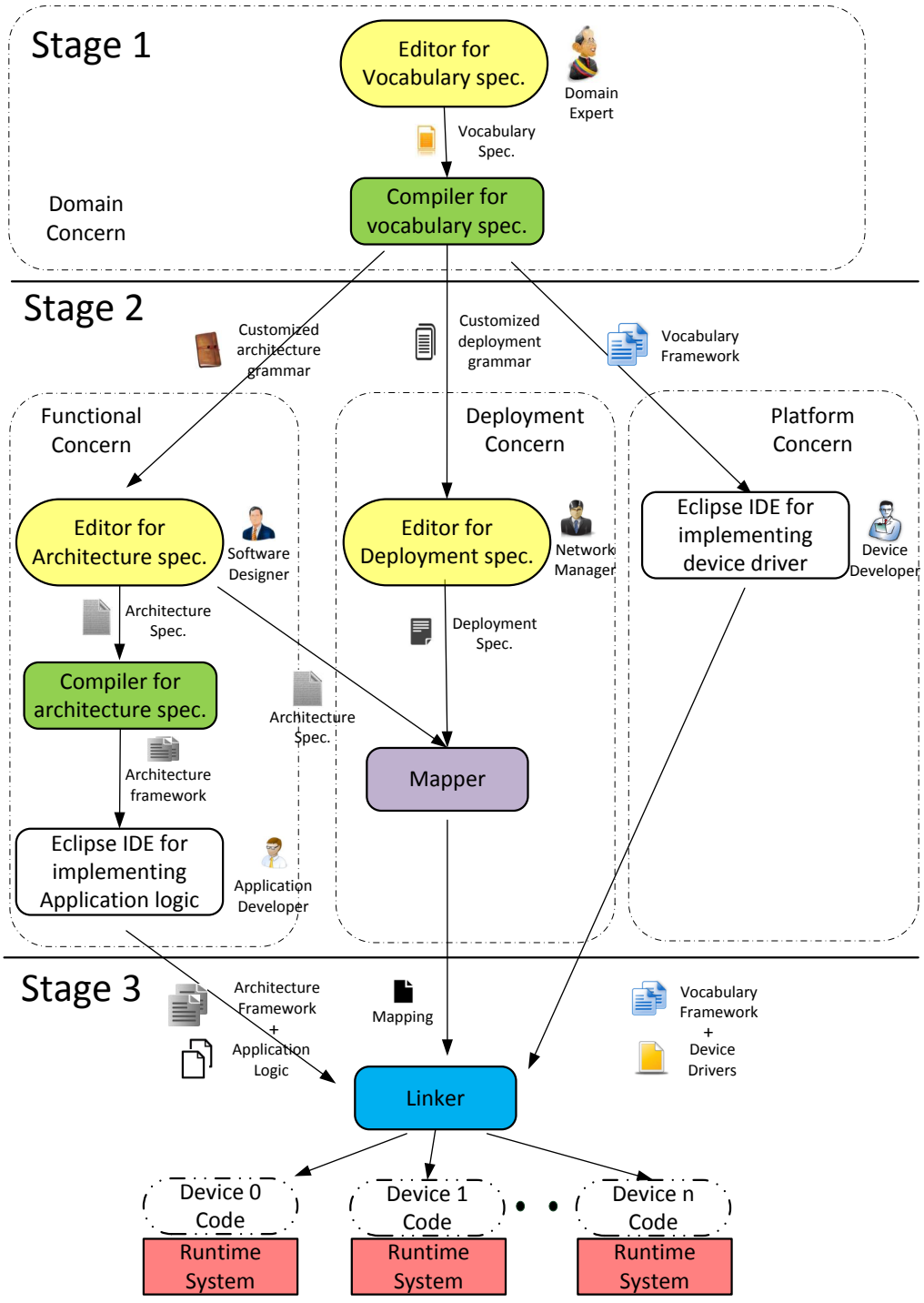

Figure IV.1 – Overview of components in SrijanSuite.

2 System components

This section discusses each of SrijanSuite components in detail.

2.1 Editor

The SrijanSuite editor provides supports for specifying high-level textual languages with the facilities of syntax coloring and syntax error reporting. The editor support is provided at differ phases of IoT application development to help stakeholders illustrated in Figure IV.1: (1) editor for a vocabulary specification to aid the domain expert, (2) editor for an architecture specification to aid the software designer, and (3) editor for a deployment specification to aid the network manager.

We take the editor for vocabulary specification as an example to demonstrate an editor support provided by SrijanSuite. Figure IV.2 illustrates the editor for writing a vocabulary specification. The zone ① shows the editor, where the domain expert writes vocabulary. The zone ② shows the menu bar, where the domain expert invokes the compiler for vocabulary to generate a vocabulary framework, a customized architecture and deployment grammar. We use Xtext [1] for a full fledged editor support, similar to work in [Bertran et al., 2012]. The Xtext is a framework for a development of domain-specific languages, and provides an editor with syntax coloring by writing Xtext grammar.

2.2 Compiler

The SrijanSuite compiler parses high-level specifications and translates them into code that can be used by other components in the system. This component is composed of two modules: (1) parser. It converts high-level specifications into data structures that can be used by the code generator. (2) code generator. It uses outputs of the parser and produces files in a target code. In the following, each of these modules are discussed.

Parser. It converts high-level specifications (vocabulary, architecture, and deployment specification) into data structures that can be used by the code generator. The conceptual model shown in Figure III.1 directly corresponds to data structures from the high-level specifications. Apart from this core functionality, it also checks syntax of specifications and reports errors to stakeholders. The SrijanSuite parser is implemented using ANTLR parser generator [Parr, 2007]. The ANTLR parser is a well-known parser generator that creates parser files from grammar descriptions. Appendix 1 shows three grammar files for parsing a vocabulary, architecture,

1. http://www.eclipse.org/Xtext/

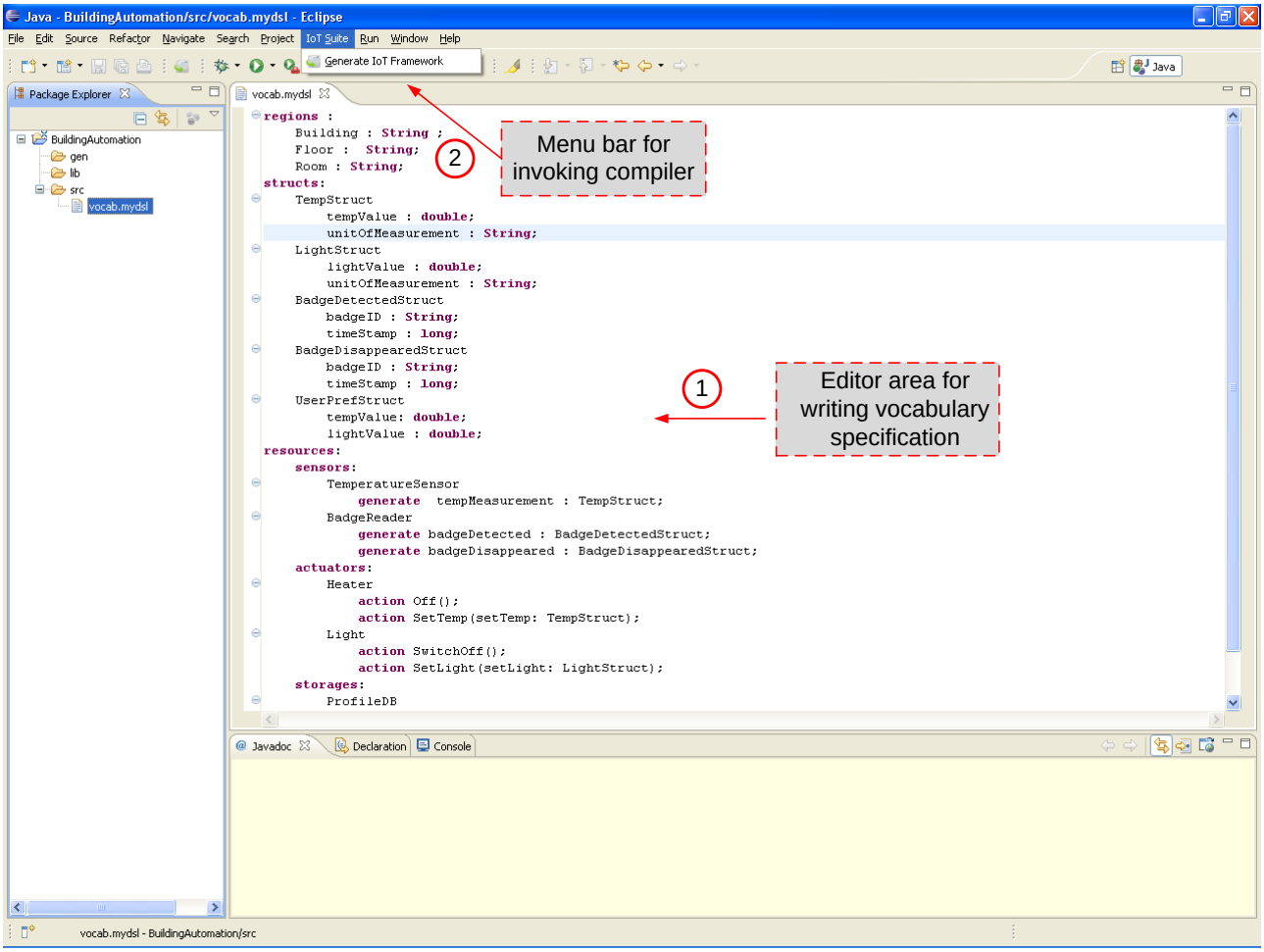

Figure IV.2 – SrijanSuite editor support for writing vocabulary specification.

and deployment specification. The whole parser including the implementation of conceptual model is written in 777 lines of code. The lines of code is measured using EclEmma 2.2.1 plug-in [2] that counts actual Java statement as lines of code and does not consider blank lines or lines with comments.

Code generator. Based on parser outputs, the code generator creates the required files. It is composed of two sub-modules: (1) core-module, (2) plug-in. The core-module manages a repository of plug-ins. Each plug-in is specific to a target implementation code. The target code could be in any programming language (e.g., Java, Python). Each plug-in is defined as template files, which the core-module uses to generate code. The key advantage of separating core-module and plug-in is that it simplifies an implementation of a new code generator for a target implementation.

2. http://www.eclemma.org/

The SrijanSuite core-module is 1221 lines of Java SE 1.6. code. The plug-ins are implemented using StringTemplate Engine,[3] a Java template engine for generating source code or any other formatted text output. In our prototype implementation, the target code is in the Java programming language compatible with Eclipse IDE. However, the code generator is flexible to generate code in any object-oriented programming language, thanks to the architecture of the SrijanSuite code generator that separates core-module and plug-ins.

We build two compilers to aid stakeholders shown in Figure IV.1. (1) compiler for a vocabulary specification. It translates a vocabulary specification and generates a vocabulary framework, and a customized architecture and deployment grammar to aid stakeholders. (2) compiler for an architecture specification. It translates an architecture specification and generates an architecture framework to aid the application developer. The both generated frameworks are compatible with Eclipse IDE. For example, Figure IV.3 shows a generated architecture framework containing Java files (① in Figure IV.3) in Eclipse IDE. ② in Figure IV.3 shows a generated Java file in the architecture framework for a `RoomAvgTemp`. Note that the generated framework contains abstracts methods (③ in Figure IV.3), which are implemented by the application developer using Eclipse IDE shown in Figure IV.4.

2.3 Mapper

The mapper produces a mapping from a set of computational services to a set of devices. Figure IV.5 illustrates the architecture of the mapper component. This component parses a deployment and architecture specification. The parser converts high-level specifications into appropriate data structures that can be used by the mapping algorithm. The mapping algorithm maps computational services described in the architecture specification to devices described in the deployment specification and produces mapping decisions into appropriate data structures. The code generator consumes the data structures and generates mapping files that can be used by the linker component.

In our current implementation, this module randomly maps computational services to a set of devices according to the algorithm presented in Section 2.4.2 of Chapter III. However, due to generality of our framework, more sophisticated mapping algorithm can be plugged into the mapper component. The mapping functionality is implemented in 193 lines of Java SE 1.6, measured using EclEmma 2.2.1 plug-in for Eclipse IDE.

3. http://www.stringtemplate.org/

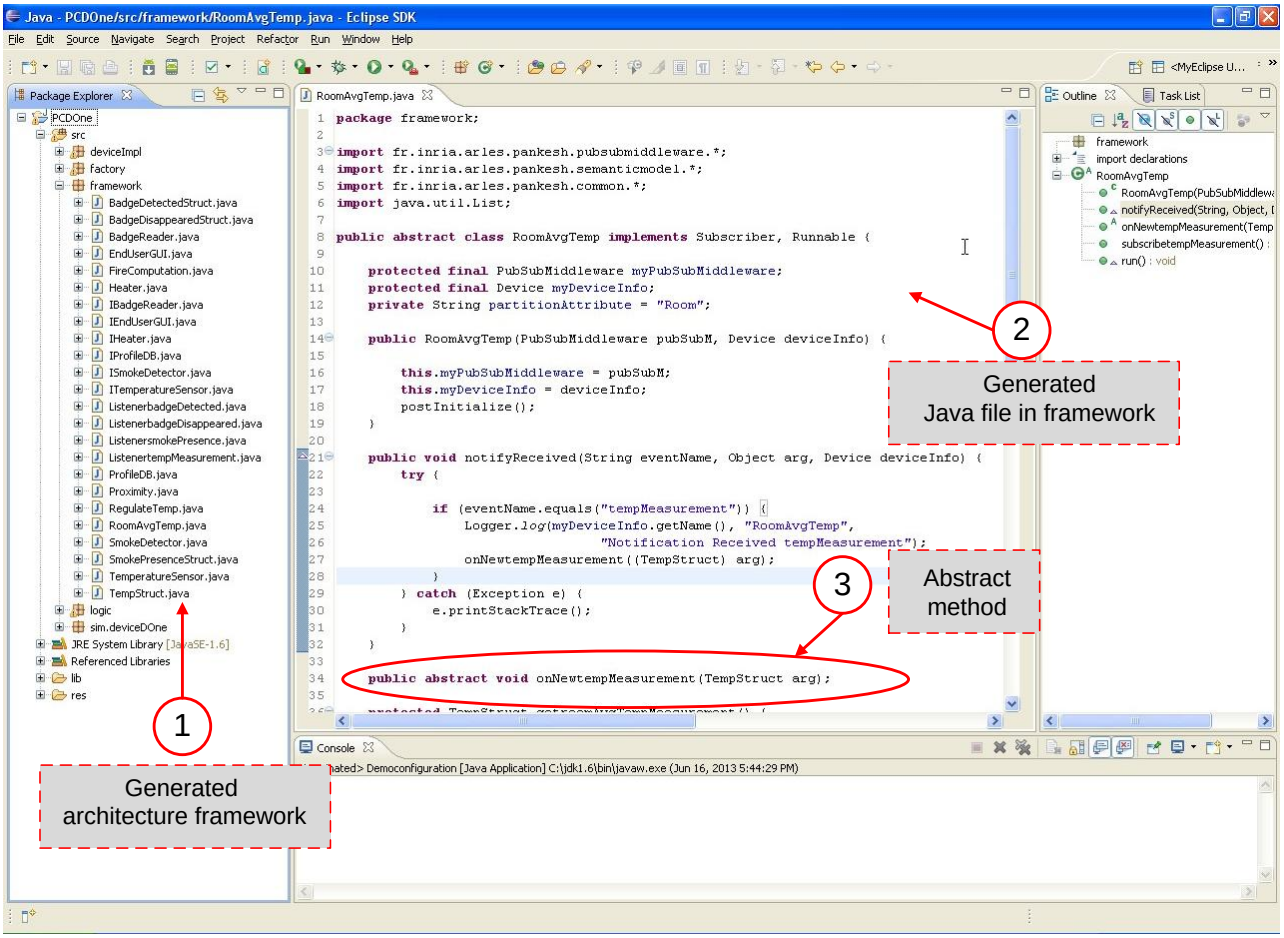

Figure IV.3 – Generated architecture framework in Eclipse

2.4 Linker

The linker combines and packs code generated by various stages of compilation into packages that can be deployed on devices. It merges a generated architecture framework, application logic, mapping code, device drivers, and vocabulary framework. This component supports the deployment phase by producing device-specific code to result in a distributed software system collaboratively hosted by individual devices.

The current version of the SrijanSuite linker generates Java source packages for Android and Java SE platform. Figure IV.6 illustrates packages for Java SE target devices (① in Figure IV.6) and Android devices (② in Figure IV.6) imported into Eclipse IDE. In order to execute code, these packages still need to be compiled by a device-level compiler designed for a target platform. The linking functionality is implemented in 94 lines of Java SE 1.6, measured using EclEmma 2.2.1 plug-in for Eclipse IDE.

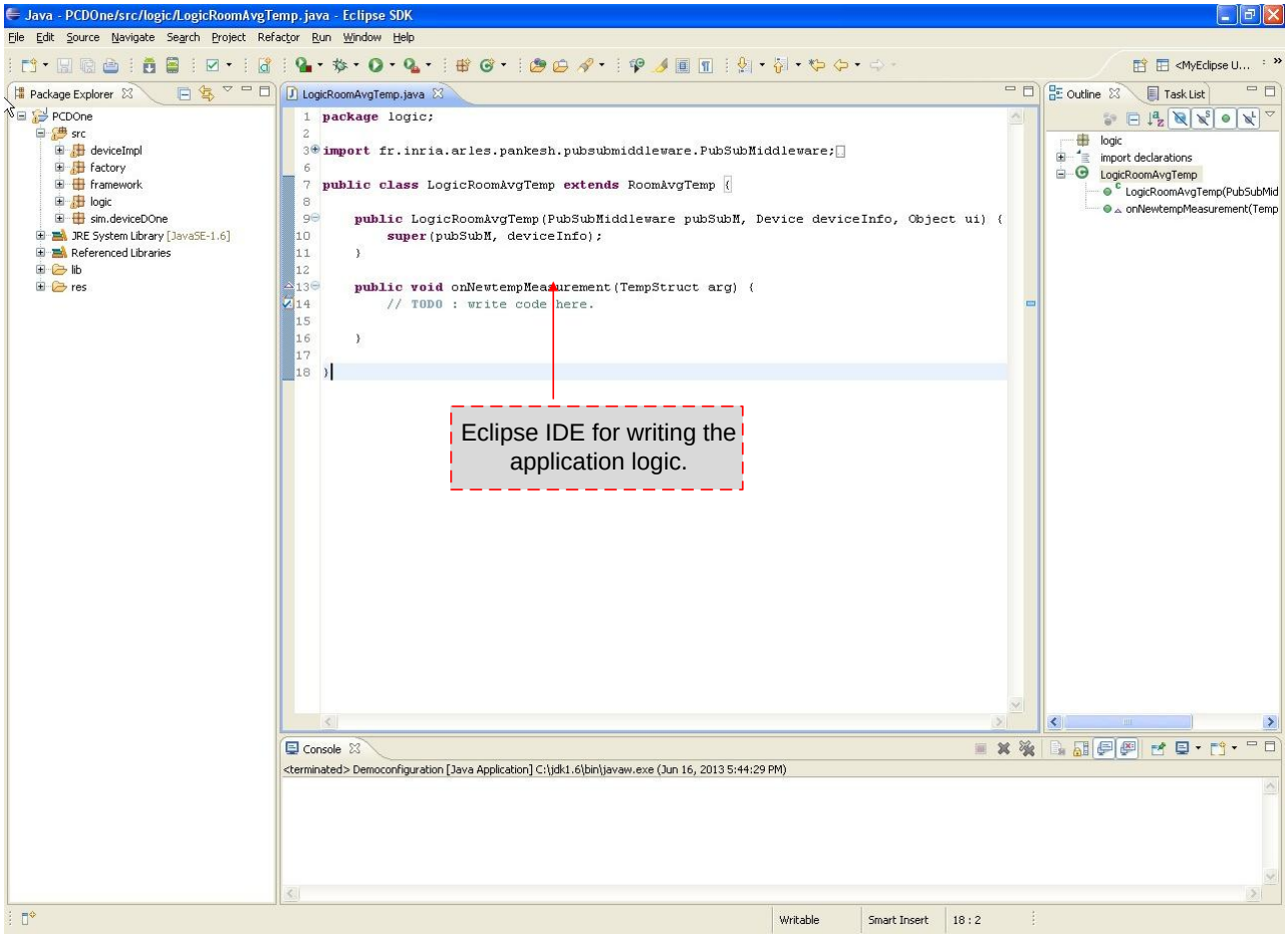

Figure IV.4 – Implementation of the application logic in Eclipse

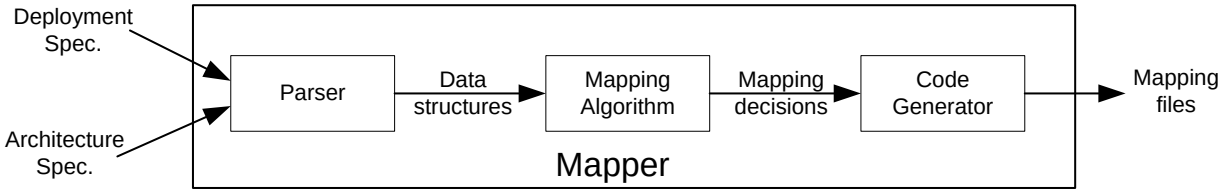

Figure IV.5 – Architecture of the mapper component in SrijanSuite.

2.5 Runtime system

The main responsibility of the SrijanSuite runtime system is a distributed execution of IoT applications. It is divided into three parts: (1) middleware, running on each device and provides a support for executing distributed tasks. (2) wrapper, plugging into packages, generated by the linker module, and middleware. (3) support library, separating packages produced by the linker component and underlying middleware by providing interfaces that are implemented by each

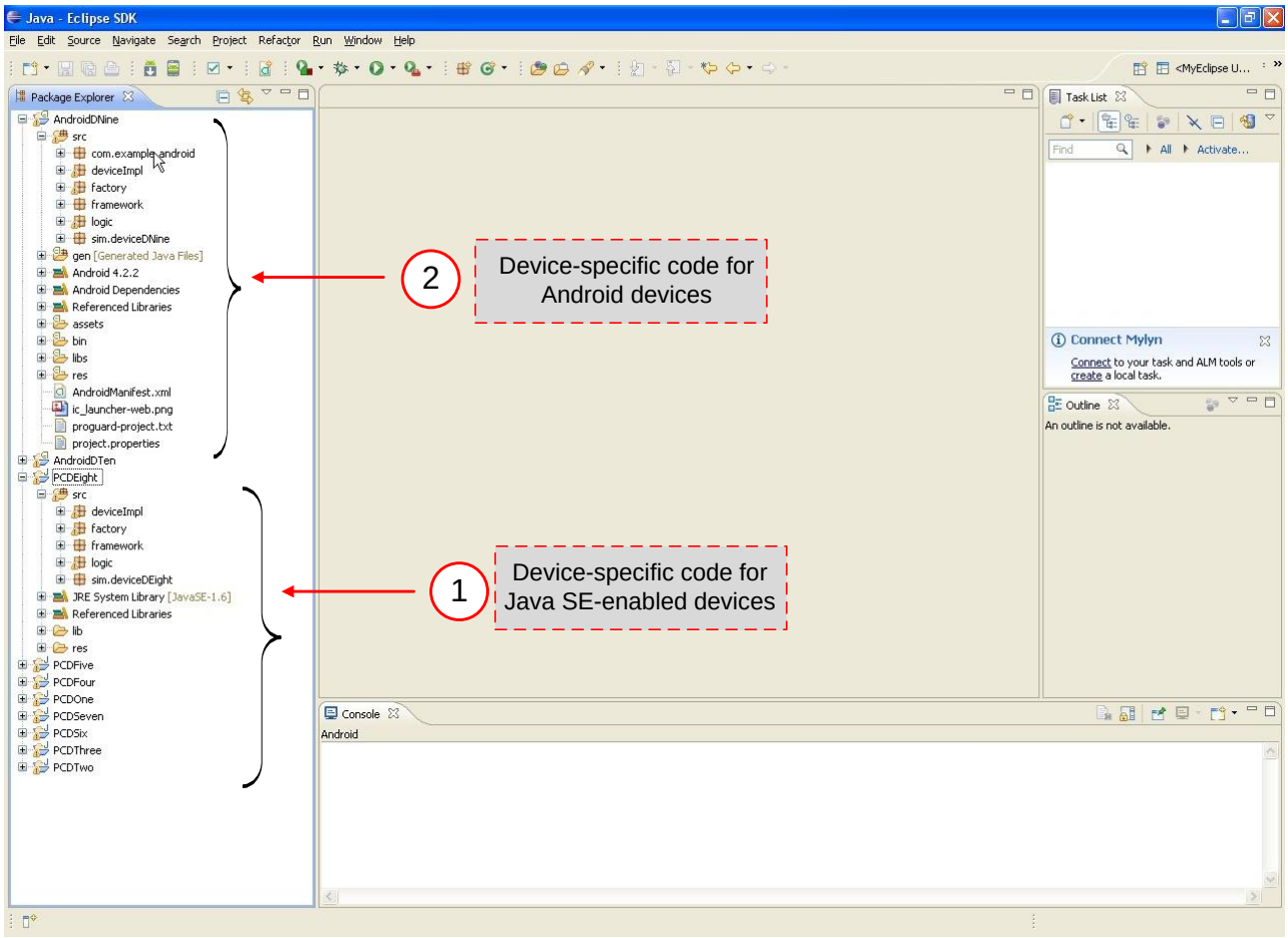

Figure IV.6 – Packages for target devices in Eclipse

wrapper. The integration of a new middleware into SrijanSuite consists of an implementation of the following interfaces in the wrapper:

- `publish()`. It is an interface for publishing data from a sender. The definition of this interface contains: an event name (e.g., temperature), event data (e.g., a temperature value, Celsius), and publisher's information such as location.
- `subscribe()`. It is an interface for receiving event notifications. An interest of events is expressed by sending a subscription request, which contains: a event name (e.g., temperature), information for filtering events such as regions of interest (e.g., a RoomAvgTemp component wants to receive events only from a current room), and subscriber's information.
- `command()`. It is an interface for triggering an action of an actuator. A command contains: a command name (e.g., switch-on heater), command parameters (e.g., set temperature of heater to 30°C), and a sender's information.

– `request-response()`. It is an interface for requesting data from a requester. In reply, a receiver sends a response. A request contains a request name (e.g., give profile information), request parameters (e.g., give profile of person with identification 12), and information about the requester.

The current implementation of SrijanSuite uses the iBICOOP middleware [Bennaceur et al., 2009]. It enables interactions among Android devices and Java SE enabled devices. Both the wrapper and support library are implemented in 191 and 110 lines of Java SE 1.6. code respectively. The current wrapper implementation for the iBICOOP middleware is available at https://github.com/pankeshlinux/ToolSuite.

3 Eclipse plug-in

We have integrated the system components as Eclipse plug-in to provide end-to-end support for IoT application development. The current prototype is available at http://web-tutorials.gr/iotsuite/download. Figure IV.7 illustrates use of our plug-in at various phases of IoT application development:

1. IoT domain project (① in Figure IV.7) – using which the domain expert can describe and compile a vocabulary specification of an application domain.
2. IoT architecture project (② in Figure IV.7) – using which the software designer can describe and compiler an architecture specification of an application.
3. IoT deployment project (③ in Figure IV.7) – using which the network manager can describe a deployment specification of a target domain and invoke the mapping component.
4. IoT Linking project (④ in Figure IV.7) – using which the network manager can combines and packs code generated by various stages of compilation into packages that can be deployed on devices.

4 Chapter summary

Since the main goal of this research is to make IoT application development easy for stakeholders, we believe that our development framework should be supported by tools to be applicable in an effective way. Therefore, this chapter presents an implementation of our development framework as a suite of tools, called as SrijanSuite. It consists of five components to aid stakeholders: (1) Editor: It helps stakeholders to write high-level specifications, including

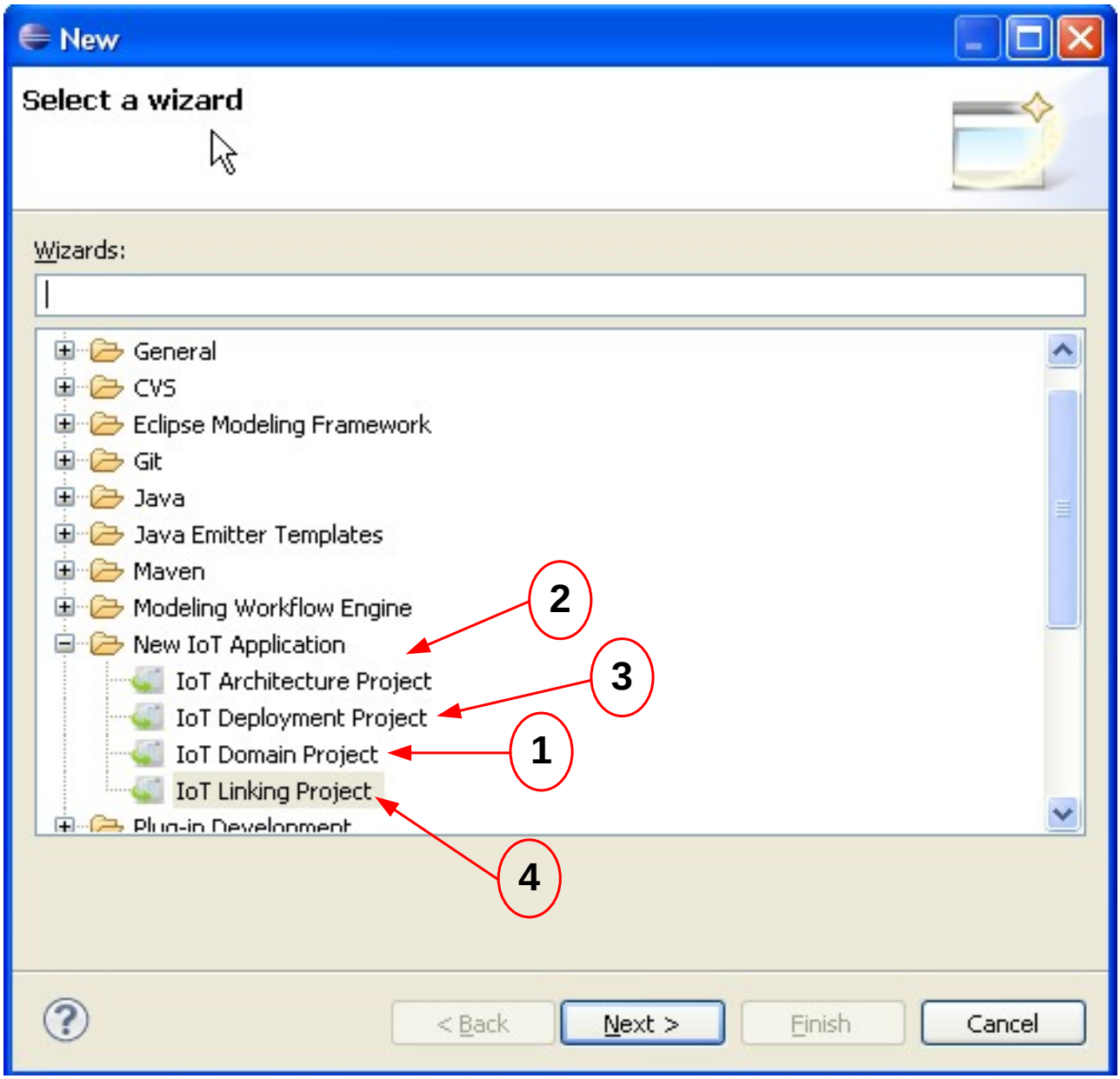

Figure IV.7 – Eclipse plug-in for IoT application development.

vocabulary, architecture, and deployment specification. (2) Compiler: It parses these high-level specifications and translates them into code that can be used by other components in the system. (3) Mapper: It maps computational services describes in the architecture specification to devices in the deployment specification. (4) Linker: It combines and packs code, generated by various stages, into packages that can be deployed on devices. (5) Runtime system: It is responsible for a distributed execution of an application. All these five components are integrated

into SrijanSuite system. The system is provided as Eclipse plug-in that integrates the support for different phases of IoT application development.

Chapter V

Evaluation

The main aim of this thesis is to devise an approach that would make the IoT application development process easy for stakeholders. In order to achieve this task at hand, we identified the application development challenges discussed in Chapter I. Then, we proposed an approach that addresses these challenges presented in Chapter III.

The goal of this chapter is to describe how well the proposed approach addresses our aim in a quantitative manner. Unfortunately, the goal is very vague because quality measures are not well-defined and they do not provide a clear procedural method to evaluate development approaches in general. We explore three aspects that are vital for the productivity of stakeholders, following the pattern in [Cassou et al., 2012]:

- **Development effort**: It indicates effort required to create a new application.
- **Reusability**: It indicates reuse of specifications and implementations across applications.
- **Code quality**: It measures the quality of generated framework. A framework with good code quality can be evolved and maintained easily, thus improving productivity.

The above set of measures is not exhaustive. However, they reflect principal quantitative advantages that our approach provides to stakeholders involved in IoT application development.

This chapter is structured as follows: Section 1 describes metrics to measure the productivity in a quantitative manner. Section 2 discusses two IoT applications used for the evaluation. Section 3 evaluates our approach on development effort to develop a new application. Section 4 demonstrates reusability of our approach. Section 5 evaluates code quality of generated framework. Finally, this chapter concludes with a summary in Section 6.

1 Evaluation metrics

Many publications that propose new programming frameworks or development environments evaluate their system in *lines of code* [Mottola and Picco, 2011, p. 50]. Hence, we consider it

as a metric for measuring development effort and demonstrate reusability of our approach. While the lines of code metric is useful, a number of limitations have been noted. It depends heavily on programming languages, styles, and stakeholder's skills. Moreover, it does not capture difficulties of maintaining code. In view of this, we believe that *code quality* must be captured as it is vital for code maintenance. In order to measure code quality, we use some of the standard and well-known metrics: Cyclomatic complexity [McCabe, 1976] and Code coverage, detailed in Section 5. We believe that these standard metrics together with lines of code metric would help to get a better picture of the productivity provided by our approach.

2 Applications for evaluation

To evaluate our approach we consider two representative IoT applications: (1) the smart building application described in Chapter I and (2) a fire detection application, which aims to detect fire by analyzing data from smoke and temperature sensors. When fire occurs, residences are notified on their smart phones by an installed application. Additionally, residents of the building and neighborhood are informed through a set of alarms. Figure V.1 shows a layered architecture of the fire detection application. A fire state is computed based on a current average temperature value and smoke presence by a local fire state service (i.e., `roomFireState`) deployed per room, then a state is sent to a service (i.e., `floorFireState`) deployed per floor, and finally a computational service (i.e., `buildingFireController`) decides whether alarms should be activated and users should be notified or not. Table V.1 summarizes the types of components used by each application.

3 Development effort

In order to measure effort to develop an application, we evaluate our approach on the following: (1) automation, evaluating a percentage of a total number of lines of code generated by our approach and (2) large scale, evaluating effort to develop an application involving a large number of devices using our approach.

Evaluating automation. The primary aim is to evaluate automation provided by our approach. More specifically, we answer this question: *how complete is the code automatically generated by our approach ?* To answer this question, we have implemented two IoT applications discussed in Section 2 using our approach. These applications are implemented independently. We did not reuse specifications and implementations of one application in other application.

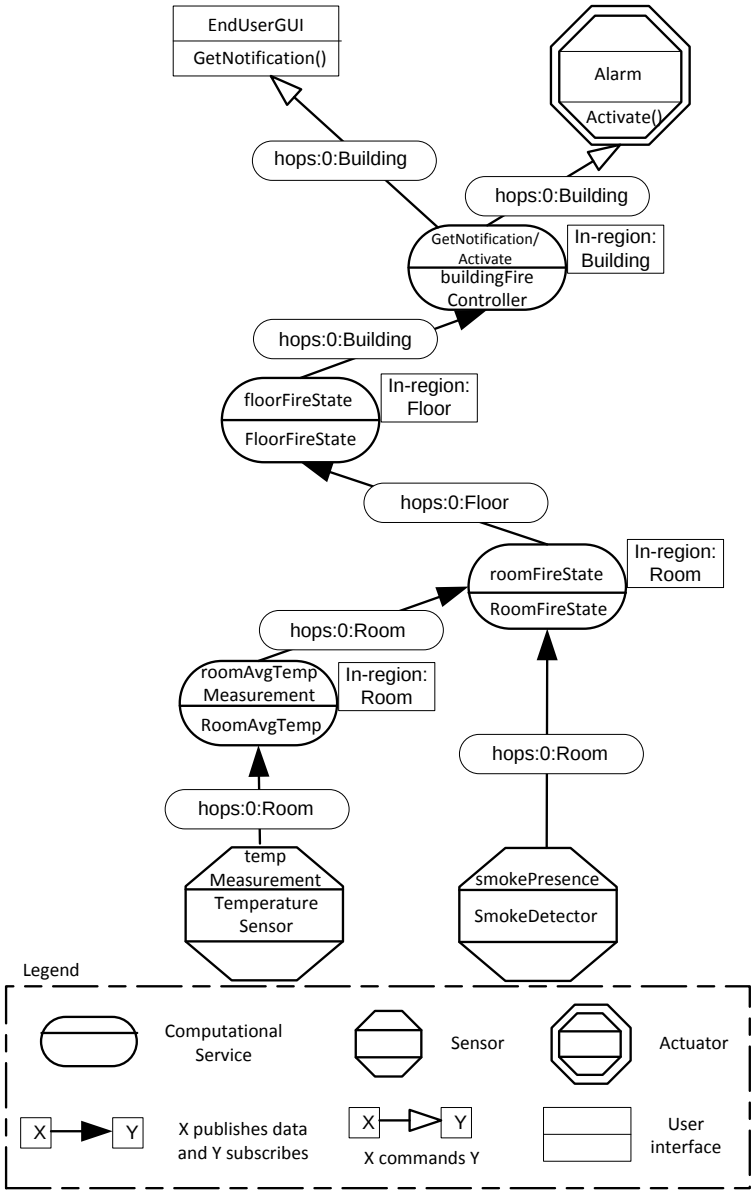

Figure V.1 – Layered architecture of the fire detection application.

We deployed the two applications on 10 simulated devices running on top of a middleware that simulates network on a single PC dedicated to the evaluation.

We measured development effort using Eclipse EclEmma 2.2.1 plug-in [1]. This tool counts actual Java statement as lines of code and does not consider blank lines or lines with comments. Our measurements reveal that more than 82% of the total number of lines of code is generated in two applications (see Table V.2).

1. http://www.eclemma.org/

Component Type	Smart building	Fire detection
Sensing	TemperatureSensor BadgeReader	TemperatureSensor SmokeDetector
Actuating	Heater Light	Alarm -
Storage	ProfileDB	none
Computational	RoomAvgTemp Proximity FloorAvgTemp BuildingAvgTemp RoomController	RoomAvgTemp RoomFireState FloorFireState BuildingFireController -
End-user	DisplayData ProfileRequest ControlHeater	GetNotification - -

Table V.1 – List of components of smart building and fire detection applications

Application Name	Handwritten (lines of code)					Generated (lines of code)			$\frac{generated}{handwritten+generated}$
	Vocab Spec.	Arch. Spec.	Deploy. Spec. (devices=10)	Device driver	App. logic	Mapping code	Archi. fram.	Vocab. fram.	
Smart building	41	28	81	98	131	561	408	757	81.99%
Fire detection	27	21	81	53	72	528	292	476	83.61%

Table V.2 – Lines of code in smart building and fire detection applications

Evaluating development effort for a large number of devices. The above experiment was conducted for 10 simulated devices. It does not demonstrate development effort using our approach for a large number of devices. Therefore, the primary aim of this experiment is to evaluate effort to develop an IoT application involving a large number of devices.

In order to achieve the above aim, we have developed the smart building application on a set of simulated device running on top of the middleware dedicated to the evaluation. The assessments were conducted over an increasing number of devices. The first development effort assessment was conducted on 10 devices instrumented with heterogeneous sensors, actuators, storages, and user interfaces. In the next subsequent assessments, we kept increasing the number of devices equipped with sensors and actuators. In each assessment, we have measured lines of code to specify vocabulary, architecture, and deployment, application logic, and device drivers. Table V.3 illustrates the assessment results containing a number of devices involved in the experiment and hand-written lines of code to develop the smart building application.

Number of devices	Handwritten (lines of code)				
	Vocab Spec.	Arch. Spec.	Deploy. Spec.	Device driver	App. logic
10	41	28	81	98	131
34	41	28	273	98	131
50	41	28	401	98	131
62	41	28	497	98	131
86	41	28	689	98	131
110	41	28	881	98	131
200	41	28	1601	98	131
300	41	28	2401	98	131
350	41	28	2801	98	131
500	41	28	4001	98	131

Table V.3 – Number of devices involved in the development effort assessment and hand-written lines of code to develop the smart building application.

In Table V.3, we have noted the following two observations and their reasons:

1. As the number of devices increases, lines of code for vocabulary and architecture specification, device drivers, and application logic remain constant for a deployment consisting a large number of devices. The reason is that our approach provides the ability to specify an application at a global level rather than individual nodes.
2. As the number of devices increases, lines of code for a deployment specification increase. The reason is that the network manager specifies each device individually in the deployment specification. This is a limitation of SDL. Our future work will be to investigate how a deployment specification can be expressed in a concise and flexible way for a network with a large number of devices. We believe that the use of regular expressions is a possible technique to address this problem.

4 Reusability

This section demonstrates reusability of specifications and implementations across applications of a same application domain in our approach. We consider two evolutionary cases and demonstrate development effort to handle them: (1) a change in the target deployment. It

demonstrates reusability of specifications and implementations when an existing application is deployed to other deployment (detailed in Section 4.1). (2) evolution in the functionality. It demonstrates reusability of specifications and implementations when functionality of an application is changed (detailed in Section 4.2).

4.1 Change in target deployment

The primary aim of this section is to answer the question: *How effective is our approach in case of a change in a target deployment?* To answer this question, we consider user's requirement of developing an application on different deployments. To illustrate it, we have chosen three hypothetical target deployments. We have developed the smart building application discussed in Chapter I on them and measured effort to develop this application.

Deployment scenarios. The smart building application is deployed at site `I`. One day, director of site `G` and site `S` visit the site `I`. They are very impressed with the smart building application deployed at site `I`. They both want to deploy the same smart building application in their sites. Given that we are discussing the building automation domain, we assume that all these sites describe their buildings in terms of room, floor, and building and they have the same types of devices.

We show a deployment scenario at site `I` in Table V.4. Total 10 devices are deployed at this site with 3 floors. We consider the following aspects to show how deployment scenarios at site `G` and site `S` differ from deployment scenario at site `I`.

- **Deployment scenario at site `G`.** It differs from `I` in terms of the distribution of devices. For instance, in the deployment `I`, 4 devices are deployed in `Floor: 11`, while in the deployment `G`, 6 devices are deployed in `Floor: 11` illustrated in Table V.5.
- **Deployment scenario at site `S`.** It differs from `I` in terms of the attached resources with devices. For instance, in deployment `I`, heater and temperature sensor are attached to different devices, while in the deployment `S` heater and temperature sensor are attached to same devices deployed in `Room: 0` and `Room: 2`, illustrated in Table V.6.

Development effort for deployment scenarios. To measure the effort for developing the same application for different deployments, we have simulated the smart building application on a set of simulated devices running on top of the middleware dedicated to evaluation. Initially, when the smart building application is developed using our approach at site `I`, we have written vocabulary, deployment, and architecture specification, device drivers, and deployment specification for `I` from scratch. The first row of Table V.7 shows development effort for developing the smart building application. However, reusability of vocabulary and architecture

	Floor: 11				Floor: 12				Floor: 13	
	Room: 0		Room: 1		Room: 2		Room: 3		Room: 4	
Resources	Device 1	Device 2	Device 3	Device 4	Device 5	Device 6	Device 7	Device 8	Device 9	Device 10
Temperature sensor	1	1	1	1	1	1	1	1		
Badge reader	1		1		1		1			
Heater		1		1		1		1		
Light		1		1		1		1		
Storage									1	
EndUser GUI										1

Table V.4 – Deployment scenario at site `I`.

	Floor: 11						Floor: 12		Floor: 13	
	Room: 0		Room: 1		Room: 2		Room: 3		Room: 4	
Resources	Device 1	Device 2	Device 3	Device 4	Device 5	Device 6	Device 7	Device 8	Device 9	Device 10
Temperature sensor	1	1	1	1	1	1	1	1		
Badge reader	1		1		1		1			
Heater		1		1		1		1		
Light		1		1		1		1		
Storage									1	
EndUser GUI										1

Table V.5 – Deployment scenario at site `G`

specification, application logic, device drivers become apparent when we have deployed the smart building application at site `G` and site `S`. To develop subsequent applications, we only need to specify target deployments. The second and third row of Table V.7 show the drastic reduction in development effort for deployment `G` and `S`.

We conclude that the primary reason of drastic reduction of development effort in next two sites using our approach is separation of concerns. Our approach separates IoT application development into well-defined concerns. Therefore, stakeholders achieve high reusability of specifications and implementations across applications of a same application domain, thus reducing development effort.

	Floor: 11				Floor: 12				Floor: 13	
	Room: 0		Room: 1		Room: 2		Room: 3		Room: 4	
Resources	Device 1	Device 2	Device 3	Device 4	Device 5	Device 6	Device 7	Device 8	Device 9	Device 10
Temperature sensor	1	1	1	1	1	1	1	1		
Badge reader		1	1			1	1			
Heater	1			1	1			1		
Light		1		1		1		1		
Storage									1	
EndUser GUI										1

Table V.6 – Deployment scenario at site `S`

Domain	Deployments	Handwritten lines of code					
		Vocab. spec.	Arch. spec.	Deploy. spec.	App. logic	Device driver	Development effort
Building Automation	Site `I`	41	28	81	131	98	379
	Site `G`	0	0	81	0	0	81
	Site `S`	0	0	81	0	0	81

Table V.7 – Development effort to deploy the smart building application on different sites

4.2 Evolution in functionality

The primary aim of this section is to answer the question: *How effective is our approach when there is a requirement of developing a new functionality?* To answer this question, we consider user's requirement of adding a new functionality into an existing deployment. To illustrate it, we have developed the two applications of the building automation domain described in Section 2. We have deployed them on a set of simulated devices running on top of a simple middleware dedicated to the evaluation. Initially, when the smart building application was developed using our approach, we have written vocabulary and deployment specification, device drivers, an architecture specification of the smart building and its application logic from scratch. The first row of Table V.8 shows development effort for the smart building application. However, reusability of the vocabulary and deployment specification, and device drivers become apparent when we develop the fire detection application. To develop the fire detection application, we only need to specify an architecture specification and application logic. The second row of Table V.8 shows the drastic reduction in development effort for the fire detection application.

We conclude that the primary reason of drastic reduction of development effort in the fire detection application using our approach is separation of concerns. Our approach separates IoT application development into well-defined concerns. Therefore, stakeholders achieve high reusability of specifications and implementations across applications of a same application domain, thus reducing development effort.

Domain	Applications	Handwritten lines of code					
		Vocab. spec.	Arch. spec.	Deploy. spec.	App. logic	Device driver	Development effort
Building Automation	Smart building	41	28	81	131	98	379
	Fire detection	0	21	0	72	0	93

Table V.8 – Development effort to program two applications of the building automation domain

5 Code quality

The *lines of code* metric measures development effort. However, it does not capture the quality of the generated framework. A program with good quality can be maintained easily. Code quality is thus critical for the cost of a software system as maintenance accounts for more than 85% of the total cost of the application [Erlikh, 2000]. In order to measure code quality of our generated framework, we use two code metrics: (1) *Code Coverage*, measuring usefulness of code (detailed in Section 5.1) and (2) *Cyclomatic Complexity*, measuring structural complexity of code (detailed in Section 5.2).

5.1 Code coverage

The measure of lines of code is only useful if the generated code is actually executed. We measured code coverage of the generated programming frameworks of two applications (see Table V.9) using the EclEmma [2] Eclipse plug-in. Our measures show that more than 90% of generated code is actually executed, the other portion being error-handling code for errors that did not happen during the experiment. This high value indicates that most of the execution is spent in generated code and that, indeed, our approach reduces development effort by generating useful code.

2. http://www.eclemma.org/

Application	Generated code
Smart building	92.22%
Fire detection	90.38%

Table V.9 – Code coverage of generated programming frameworks

5.2 Cyclomatic complexity

The more complex the structure of code is, the harder it is to test or maintain, and more likely to contains bugs. So, it is important to measure structural complexity of code. One of methods to measure structural complexity is to calculate the number of different execution paths in a flow of the program. More specifically, code with a single path (i.e., no conditional statements or no loop statements) has a minimum possible cyclomatic complexity number (CCN) value. The CCN is increased with conditional, loop statements, and other code features. It is a well-known measure proposed by McCabe to guide the testing process and gives a minimum number of test cases for a program. McCabe notes that the CCN in the 1 to 10 range implies high testability and the cost and effort to maintain the code is less. CCN greater than 10 indicates poor testability, which hampers code maintenance.

We used Metrics [3] 1.3.6 Eclipse plug-in to measure code quality of our target IoT applications. We measured code complexity of a generated framework that contains mapping, architecture, and vocabulary frameworks. Table V.10 illustrates our measurements. Our measurements reveal that code complexity of the generated framework in both applications remains lower than 2. These results demonstrate that our generated programming framework guides stakeholders through a well-structured and easy to test code.

Application	CCN of generated code
Smart building	1.45
Fire detection	1.56

Table V.10 – Code complexity of generated programming framework

6 Chapter summary

This chapter evaluates our approach and shows its ability to facilitate stakeholders in IoT application development. The goal of this evaluation is to demonstrate how our approach makes

3. http://metrics.sourceforge.net

IoT application development easy for stakeholders. Unfortunately, this goal is very vague because quality measures are not well-defined, and they do not provide a clear procedural method to evaluate development approaches in general. Therefore, in order to evaluate our work in a quantitative manner we have established three measures: (1) development effort, evaluating effort to develop a new IoT application, (2) reusability, evaluating reuse of specifications and implementations across applications of a same application domain. (3) code quality, attributing quality of a generated programming framework. In the following, we summarize experiments for these measures and their results:

1. To measure effort required to develop a new IoT application, we evaluate our approach on two aspects:
   - *Automation*. The experiment results reveal that our approach generates a significant percentage of total lines of code. Thus, it provides a high-level of automation in IoT application development.
   - *Large scale*. The experiment results show that lines of code that need to be written for vocabulary and architecture specification, device drivers, and application logic remain constant (i.e., independent of a number of devices in an application). Thus, our approach reduces the effort needed to develop IoT applications involving a large number of devices.
2. To measure effectiveness of our approach to handle evolutions, we consider some evolutionary scenarios and demonstrate development effort to handle them. Our experiment results shows that our approach shows high reusability of specifications and implementations across applications of a same application domain, thus improving the productivity in IoT application development.
3. We measure code quality of generated frameworks for two applications. We use the standard and well-known metrics of code coverage and cyclomatic complexity. Our measurement reveals that code coverage of generated frameworks is high, which indicates our approach actually generates a useful code. The low cyclomatic complexity of the generated frameworks demonstrates that it guides stakeholders through a well-structured and easy to test code.

In conclusion, our evaluation reflect principal quantitative advantages that our approach provides to stakeholders involved in IoT application development.

Chapter VI

Conclusion and future work

1 Summary

Application development in the IoT involves dealing with a wide range of related issues such as lack of separation of concerns, and lack of high-level of abstractions to address both the large scale and heterogeneity. Moreover, stakeholders involved in application development have to address issues that can be attributed to different life-cycles phases when developing applications. First, the application logic has to be analyzed and then separated into a set of distributed tasks for an underlying network. Then, the tasks have to be implemented for the specific hardware. Apart from handling these issues, they have to deal with other aspects of life-cycle such as changes in application requirements and deployed devices.

We note that although the software engineering, MDD, WSN, and pervasive computing community have discussed and analyzed similar challenges in the general case, this has not been applied to the case of IoT in particular. Therefore, this work proposes a new approach that utilizes advantages and promising features of the existing works to develop a comprehensive integrated approach while focusing on ease of IoT application development.

This thesis presents a development methodology for IoT application development, based on techniques presented in the domains of sensor network macroprogramming and model-driven development. It separates IoT application development into different concerns and integrates a set of high-level languages to specify them. This methodology is supported by automation techniques at different phases of IoT application development. Our evaluation based on two realistic IoT applications shows that our approach generates a significant percentage of the total application code, drastically reduces development effort for IoT applications involving a large number of devices, shows high reusability of specifications and implementations across applications, and generates a high quality of code that can be maintained and tested easily. Thus, our approach improves the productivity of stakeholders involved in IoT application development.

2 Future work

This thesis addresses the challenges, presented by the steps involved in IoT application development, and prepares a foundation for our future research work. Our future work will proceed in the following complementary directions, discussed below.

Mapping algorithms cognizant of heterogeneity. While the notion of region labels is able to reasonably tackle the issue of scale at an abstraction level, the problem of heterogeneity among the devices still remains. We will provide rich abstractions to express both the properties of the devices (e.g., processing and storage capacity, networks it is attached to, as well as monetary cost of hosting a computational service), as well as the requirements from stakeholders regarding the preferred placement of the computational services of the applications. These will then be used to guide the design of algorithms for efficient mapping (and possibly migration) of computational services on devices.

Developing concise notion for SDL. In the current version of SDL, the network manager is forced to specify the detail of each device individually. This approach works reasonably well in a target deployment with a small number of devices. However, it may be time-consuming and error-prone for a target deployment consisting of hundreds to thousands of devices. Our future work will be to investigate how the deployment specification can be expressed in a concise and flexible way for a network with a large number of device. We believe that the use of regular expressions is a possible technique to address this problem.

Testing support for IoT application development. Our near term future work will be to provide support for the testing phase. A key advantage of testing is that it emulates the execution of an application before deployment so as to identify possible conflicts, thus reducing application debugging effort. The support will be provided by integrating an open source simulator in SrijanSuite. This simulator will enable transparent testing of IoT applications in a simulated physical environment. Moreover, we expect to enable the simulation of a hybrid environment, combining both real and physical entities. Currently, we are investigating open source simulators for IoT applications. We see Siafu [1] as a possible candidate due to its open source and thorough documentation.

Run-time adaptation in IoT applications. Even though our approach addresses the challenges posed by evolutionary changes in target deployments and application requirements, stakeholders have to still recompile the updated code. This is common practice in a single PC-based development environment, where recompilation is generally necessary to integrate changes. Ho-

1. http://siafusimulator.org/

wever, it would be very interesting to investigate how changes can be injected into the running application that would adapt itself accordingly. For instance, when a new device is added into the target deployment, an IoT application can autonomously include a new device and assign a task that contributes to the execution of the currently running application.

Integration of modeling language for user interfaces. To aid stakeholders, our approach integrates a three modeling languages: (1) SVL to describe the domain concern, (2) SAL to describe the functional concern, and (3) SDL to describe the deployment concern. However, currently the platform concerns such as details of the user interfaces are described as part of the domain vocabulary. We have come across a significant amount of work in the area of cross-platform DSLs [Guerrero-Garcia et al., 2009; Pohja, 2010; Souchon and Vanderdonckt, 2003]. These languages describe device properties at a higher level of abstractions than code used to write device drivers and code generator transform them into platform-specific code. For instance, many User Interface Description Languages (UIDLs) allow to specify the properties of User Interfaces into XML and the code generator transform the specification into a variety of heterogeneous platforms (e.g., PC-based UI, Android-based UI). Our future goal will be to integrate the existing DSLs into our methodology to further reduce development effort.

Appendices

1 Grammars of modeling languages

1.1 SVL grammar

```
vocabSpec :
    'regions'  ':' (region_def)+
    'structs' ':'  (struct_def)+
    'resources' ':' (abilities_def)+
;
region_def :
     CAPITALIZED_ID ':' dataType  ';'
;
struct_def:
    CAPITALIZED_ID
    (structField_def ';')+
;
structField_def:
  lc_id ':' dataType
;
abilities_def :
  'sensors' ':'   (sensor_def)+
  'actuators' ':' (actuator_def)+
  ('storages'  ':' ss_def)*
  ('userinterfaces' ':' gui_def)*
;
sensor_def:
    CAPITALIZED_ID
    (sensorMeasurement_def ';')+
;
sensorMeasurement_def :
    'generate' lc_id ':'  CAPITALIZED_ID
;
actuator_def:
   CAPITALIZED_ID
   (action_def ';')+
```

```
32 ;
33 action_def:
34     'action' CAPITALIZED_ID '(' (parameter_def)? ')'
35 ;
36 parameter_def :
37     lc_id ':'  CAPITALIZED_ID (',' parameter_def )?
38 ;
39 ss_def:
40   CAPITALIZED_ID
41     (storageDataAccess_def ';')+
42 ;
43 storageDataAccess_def :
44      storageGeneratedInfo_def  'accessed-by' storagedataIndex_def
45 ;
46 storageGeneratedInfo_def :
47     'generate' lc_id ':'  CAPITALIZED_ID
48 ;
49 storagedataIndex_def:
50         lc_id ':' dataType
51 ;
52 gui_def:
53   CAPITALIZED_ID
54    (gui_command_def ';')*
55    (action_def ';')*
56    (gui_request_def  ';')*
57 ;
58 gui_request_def :
59    'request' lc_id '(' gui_request_param_def ')'
60 ;
61 gui_request_param_def :
62       lc_id (',' gui_request_param_def)?
63 ;
64 gui_command_def :
65     'command'  CAPITALIZED_ID '(' (cntParameter_def)? ')'
66 ;
```

```
67 cntParameter_def :
68     lc_id (',' cntParameter_def )?
69 ;
70 lc_id: ID
71 ;
72 dataType:
73   primitiveType
74 ;
75 primitiveType:
76   (id='integer' | id='boolean' | id='string' |  id = 'double' |
       id = 'long')
77 ;
78 ID  : 'a'..'z'  ('a'..'z' | 'A'..'Z' )*
79 ;
80 INT : '0'..'9'('0'..'9')*
81 ;
82 CAPITALIZED_ID: 'A'..'Z' ('a'..'z' | 'A'..'Z' )*
83 ;
84 WS: ('\t' | ' ' | '\r' | '\n' | '\u000C')+ {$channel = HIDDEN;}
85 ;
```

1.2 Customized SAL grammar for building automation domain

```
1 archSpec :
2     ('structs' ':' struct_def)*
3     'softwarecomponents' ':' swcomponent_def
4 ;
5 struct_def:
6     CAPITALIZED_ID
7     (structField_def ';')*
8 ;
9 structField_def:
10   lc_id ':' dataType
11 ;
12 swcomponent_def :
```

```
13    'computationalServices' ':' (cs_def )+
14 ;
15 cs_def:
16   CAPITALIZED_ID
17     (csGeneratedInfo_def ';')*
18     (csConsumeInfo_def ';')+
19     (csRequest_def  ';')*
20     (csCommand_def ';')*
21      partition_def ';'
22 ;
23 csGeneratedInfo_def:
24     'generate' lc_id ':' struct_vocabulary
25 ;
26 csConsumeInfo_def:
27   'consume' generateInfo_vocabulary 'from'  'hops' ':' INT ':'
       region_vocabulary
28 ;
29 csRequest_def :
30    'request' generateInfo_vocabulary '(' requestParameter_def ')'
31 ;
32 csCommand_def :
33     'command'  action_vocabulary '(' (cntrlParameter_def)? ')'  '
         to'  'hops'  ':' INT ':' region_vocabulary
34 ;
35 cntrlParameter_def :
36     action_parameter  (',' cntrlParameter_def )?
37 ;
38 partition_def :
39     'in-region' ':' region_vocabulary
40 ;
41 generateInfo_vocabulary :
42      'tempMeasurement' | 'badgeDetected' | 'badgeDisappeared' |'
         profile'| 'smokePresence' |  lc_id
43 ;
44 struct_vocabulary :
```

```
45   'TempStruct' | 'BadgeDetectedStruct'| 'BadgeDisappearedStruct'
       | 'UserPrefStruct' | 'LightStruct' | 'SmokePresenceStruct' |
        'FireState' | CAPITALIZED_ID
46 ;
47 region_vocabulary :
48   'Building' | 'Floor' | 'Room'
49 ;
50 action_vocabulary :
51   'SetTemp' | 'Off' | 'Display' | 'OffLight' | 'SetLight' | '
       Activate' | 'DeActivate' | 'DisplayData' | 'GetNotification'
52 ;
53 action_parameter :
54     'setTemp' | 'setLight' | 'displayTemp' | 'fireState' | '
         badgeID'
55 ;
56 requestParameter_def :
57     request_parameter  (',' requestParameter_def )?
58 ;
59 request_parameter:
60       'badgeID' | lc_id
61 ;
62 lc_id: ID
63 ;
64 dataType:
65   primitiveType
66 ;
67 primitiveType:
68   (id='Integer' | id='Boolean' | id='String' |  id = 'double' |
       id = 'long' | id='boolean' )
69 ;
70 ID  : 'a'..'z'  ('a'..'z' | 'A'..'Z' )*
71 ;
72 INT : '0'..'9'('0'..'9')*
73 ;
74 CAPITALIZED_ID: 'A'..'Z' ('a'..'z' | 'A'..'Z' )*
```

```
75 ;
76 WS: ('\t' | ' ' | '\r' | '\n' | '\u000C')+ {$channel = HIDDEN;};
```

1.3 Customized SDL grammar for building automation domain

```
1  deploymentspec:
2        'devices' ':'
3         (device_def)+
4      ;
5  device_def:
6     deviceName = (ID|CAPITALIZED_ID) ':'
7      'region' ':' (location_def)+
8      'type' ':'  device_type ';'
9      'resources' ':' (resources_def)* ';'
10     'mobile'  ':' mobile_def ';'
11 ;
12 location_def :
13     region_vocabulary ':' INT ';'
14 ;
15 device_type :
16      CAPITALIZED_ID
17 ;
18 mobile_def :
19     'true' | 'false'
20 ;
21 resources_def :
22   'TemperatureSensor' | 'Heater' | 'ProfileDB' | 'BadgeReader' |
       'SmokeDetector' | 'Light' | 'Alarm' | 'EndUserGUI'
23 ;
24 region_vocabulary :
25   'Building' | 'Floor' | 'Room'
26 ;
27 ID   : 'a'..'z'  ('a'..'z' | 'A'..'Z' )* ('0'..'9')*
28    ;
29 INT : '0'..'9'('0'..'9')*
```

```
30 ;
31 CAPITALIZED_ID: 'A'..'Z' ('a'..'z' | 'A'..'Z' )* ('0'..'9')*
32 ;
33 WS: ('\t' | ' ' | '\r' | '\n' | '\u000C')+ {$channel = HIDDEN;};
```

2 Application domain: building automation

2.1 Building automation: vocabulary specification

```
regions :
  Building : integer ;
  Floor :  integer;
  Room : integer;
structs:
    TempStruct
        tempValue : double;
        unitOfMeasurement : string;
    LightStruct
        lightValue : double;
        unitOfMeasurement : string;
    BadgeDetectedStruct
        badgeID : string;
        timeStamp : long;
    BadgeDisappearedStruct
        badgeID : string;
        timeStamp : long;
    UserPrefStruct
        tempValue: double;
        lightValue : double;
    SmokePresenceStruct
        smokePresence : boolean;
        timeStamp : long;
    FireState
        firePresence    : boolean;
        timeStamp : long;
resources :
    sensors:
        TemperatureSensor
            generate  tempMeasurement : TempStruct;
        BadgeReader
```

```
32              generate badgeDetected : BadgeDetectedStruct;
33              generate badgeDisappeared : BadgeDisappearedStruct;
34          SmokeDetector
35              generate smokePresence : SmokePresenceStruct;
36      actuators:
37          Heater
38              action Off();
39              action SetTemp(setTemp: TempStruct);
40          Light
41              action OffLight();
42              action SetLight(setLight:LightStruct);
43          Alarm
44              action Activate();
45              action DeActivate();
46      storages:
47          ProfileDB
48              generate profile : UserPrefStruct  accessed-by
                    badgeID : string;
49      userinterfaces :
50          EndUserGUI
51              command Off();
52              command SetTemp(setTemp);
53              action DisplayData(displayTemp:TempStruct);
54              action GetNotification(fireState:FireState);
55              request profile(badgeID);
```

Listing 1 – Vocabulary Specification of the building automation. Keywords are printed in blue.

2.2 Smart building application: architecture specification

```
1 softwarecomponents:
2     computationalServices:
3         RoomAvgTemp
4             generate roomAvgTempMeasurement: TempStruct;
5             consume  tempMeasurement from hops:0: Room;
6             in-region: Room;
```

```
        FloorAvgTemp
            generate floorAvgTemp: TempStruct;
            consume roomAvgTempMeasurement from hops:0: Floor;
            in-region: Floor;
        Proximity
            generate tempPref: UserPrefStruct;
            consume  badgeDetected from hops:0: Room;
            consume  badgeDisappeared from hops:0: Room;
            request  profile( badgeID);
            in-region: Room;
        BuildingAvgTemp
            consume floorAvgTemp from hops:0: Building;
            command DisplayData( displayTemp) to hops:0:
               Building;
            in-region: Building;
        RoomController
            consume roomAvgTempMeasurement from hops:0: Room;
            consume tempPref from hops:0: Room ;
            command  Off() to hops:0: Room;
            command  SetTemp( setTemp) to hops:0: Room;
            command  OffLight() to hops:0: Room;
            command  SetLight( setLight) to hops:0: Room;
            in-region: Room;
```

Listing 2 – Architecture specification of the smart building application using SAL. Language keywords are printed in blue, while keywords derived from vocabulary are printed underlined.

2.3 Fire detection application: architecture specification

```
softwarecomponents:
  computationalServices:
    RoomAvgTemp
      generate roomAvgTempMeasurement :  TempStruct;
      consume  tempMeasurement from  hops : 0 :  Room;
      in-region :  Room;
    RoomFireState
        generate roomFireState :  FireState;
```

```
9           consume  smokePresence from hops:0: Room;
10        consume roomAvgTempMeasurement from hops : 0 :  Room;
11        in-region :  Room;
12      FloorFireState
13        generate floorFireState :  FireState;
14        consume roomFireState from hops:0: Floor;
15        in-region :  Floor;
16      BuildingFireController
17        consume floorFireState from hops :0: Building;
18        command  GetNotification( fireState) to hops :0:  Building;
19        command  Activate() to hops:0:  Building;
20        command  DeActivate() to hops:0: Building;
21        in-region :  Building;
```

Listing 3 – Architecture specification of the fire detection application using SAL. Language keywords are printed in blue, while keywords derived from vocabulary are printed underlined.

2.4 Building automation: deployment specification

```
1 devices:
2     D1 :
3         region :
4               Building : 15;
5               Floor : 1511;
6               Room : 15110;
7         type : JavaSE;
8         resources :  TemperatureSensor ,  BadgeReader ,
             SmokeDetector;
9         mobile : false;
10    D2:
11        region :
12              Building : 15;
13              Floor : 1511;
14              Room : 15110;
15        type : JavaSE;
16        resources :  TemperatureSensor ,  Heater ,  Light;
17        mobile : false;
```

```
18      D3 :
19          region :
20              Building : 15;
21              Floor : 1511;
22              Room : 15111;
23          type : JavaSE;
24          resources : TemperatureSensor, BadgeReader,
                SmokeDetector;
25          mobile : false;
26      D4:
27          region :
28              Building : 15;
29              Floor : 1511;
30              Room : 15111;
31          type : JavaSE;
32          resources : TemperatureSensor, Heater, Light;
33          mobile : false;
34      D5 :
35          region :
36              Building : 15;
37              Floor : 1512;
38              Room : 15122;
39          type : JavaSE;
40          resources : TemperatureSensor, BadgeReader,
                SmokeDetector;
41          mobile : false;
42      D6:
43          region :
44              Building : 15;
45              Floor : 1512;
46              Room : 15122;
47          type : JavaSE;
48          resources : TemperatureSensor, Heater, Light;
49          mobile : false;
50      D7 :
```

```
51          region :
52               Building : 15;
53               Floor : 1512;
54               Room : 15123;
55              type : JavaSE;
56              resources : TemperatureSensor, BadgeReader,
                   SmokeDetector;
57          mobile : false;
58      D8:
59          region :
60               Building : 15;
61               Floor : 1512;
62               Room : 15123;
63          type : JavaSE;
64          resources : TemperatureSensor, Heater, Light;
65          mobile : false;
66      D9 :
67          region :
68               Building : 15;
69               Floor : 1513;
70               Room : 15134;
71          type : JavaSE;
72          resources : ProfileDB, Alarm;
73          mobile : false;
74      D10:
75          region :
76               Building : 15;
77               Floor : 1513;
78               Room : 15134;
79          type : Android;
80          resources : EndUserGUI;
81          mobile : true;
```

Listing 4 – Target deployment specification in building using SDL. Language keywords are printed in blue, while keywords derived from vocabulary are printed underlined.

3 Publications

- **Pankesh Patel**, Animesh Pathak, Thiago Teixeira, Valerie Issarny, *Towards Application Development for the Internet of Things*, ACM/IFIP/USENIX 12th International Middleware doctoral symposium, 2011. **URL :** http://hal.inria.fr/hal-00711266/

- **Pankesh Patel**, Animesh Pathak, Damien Cassou, Valerie Issarny, *Enabling High-Level Application Development in the Internet of Things*, 4th International conference on Sensor Systems and Software, 2013. **URL :** http://hal.inria.fr/hal-00809438/

- **Pankesh Patel**, Animesh Pathak, Valerie Issarny, *Enabling High-level Application Development in the Internet of Things*, in preparation for submission to Journal of Systems and Software (JSS).

- **Pankesh Patel**, Dimitrios Soukaras, Georgios Bouloukakis, Animesh Pathak, Valerie Issarny, *A Toolkit to Prototype Internet of Things Applications*, in preparation for submission to IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS), 2014.

- **Pankesh Patel**, Ajay Kattepur, Damien Cassou, Georgios Bouloukakis, *Evaluating the Ease of Application development for the Internet of Things*, Tech-report, 2013. **URL :** http://hal.inria.fr/hal-00788366/

- **Pankesh Patel**, *Enabling High-level Application Development in the Internet of Things*, Tech-report, 2012. **URL :** http://hal.inria.fr/hal-00732094/

References

Andrews, G. R. 1991. Paradigms for process interaction in distributed programs. *ACM Comput. Surv. 23,* 1 (Mar.), 49–90. 6

Atzori, L., Iera, A., and Morabito, G. 2010. The Internet of Things : A survey. *Comput. Netw. 54,* 15 (Oct.), 2787–2805. 1

Avilés-López, E. and García-Macías, J. 2009. TinySOA : a service-oriented architecture for wireless sensor networks. *Service Oriented Computing and Applications 3,* 2 (June), 99–108. 18

Bakshi, A., Prasanna, V. K., Reich, J., and Larner, D. 2005. The abstract task graph : A methodology for architecture-independent programming of networked sensor systems. In *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*. USENIX Association, Berkeley, CA, USA, 19–24. 10

Bennaceur, A., Singh, P., Raverdy, P.-G., and Issarny, V. 2009. The iBICOOP middleware : Enablers and services for emerging pervasive computing environments. In *IEEE International Conference on Pervasive Computing and Communications*. 69

Berson, A. 1996. *Client/server architecture (2. ed.)*. McGraw-Hill. 6

Bertran, B., Bruneau, J., Cassou, D., Loriant, N., Balland, E., and Consel, C. 2012. DiaSuite : a Tool Suite To Develop Sense/Compute/Control Applications. *Science of Computer Programming, Fourth special issue on Experimental Software and Toolkits*. 63

Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., and Riboni, D. 2010. A survey of context modelling and reasoning techniques. *Pervasive Mob. Comput. 6,* 2 (Apr.), 161–180. 16

Bischoff, U. and Kortuem, G. 2006. Rulecaster : A macroprogramming system for sensor networks. In *Proceedings OOPSLA Workshop on Building Software for Sensor Networks*. 21

References

Bischoff, U. and Kortuem, G. 2007. Life cycle support for sensor network applications. In *Proceedings of the 2nd international workshop on Middleware for sensor networks*. ACM, 1–6. 8, 10, 21, 25

Blackstock, M. and Lea, R. 2012. WoTKit : a lightweight toolkit for the web of things. In *Proceedings of the Third International Workshop on the Web of Things*. ACM, 3. 17

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. 2000. Simple object access protocol (SOAP) 1.1. 19

Cassou, D., Bertran, B., Loriant, N., Consel, C., et al. 2009. A generative programming approach to developing pervasive computing systems. In *GPCE'09 : Proceedings of the 8th international conference on Generative programming and component engineering*. 137–146. 33

Cassou, D., Bruneau, J., Consel, C., and Balland, E. 2012. Toward a tool-based development methodology for pervasive computing applications. *IEEE Transactions on Software Engineering 38,* 6, 1445–1463. 21, 25, 29, 30, 54, 73

Castellani, A. P., Dissegna, M., Bui, N., and Zorzi, M. 2012. WebIoT : A web application framework for the internet of things. In *WCNC Workshops*. IEEE, 202–207. 17

Chen, C., Helal, S., de Deugd, S., Smith, A., and Chang, C. K. 2012. Toward a collaboration model for smart spaces. In *SESENA*. 37–42. 7, 29

Chen, H., Chou, P., Duri, S., Lei, H., and Reason, J. 2009. The design and implementation of a smart building control system. In *IEEE International Conference on e-Business Engineering. ICEBE'09.* IEEE, 255–262. 4

Cheong, E. and Liu, J. 2005. galsC : A Language for Event-Driven Embedded Systems. In *DATE* (2005-04-13). IEEE Computer Society, 1050–1055. 9

Chinnici, R., Moreau, J.-J., Ryman, A., and Weerawarana, S. 2007. Web services description language (WSDL) version 2.0 part 1 : Core language. *W3C Recommendation 26*. 19

Costa, P., Mottola, L., Murphy, A., and Picco, G. 2007. Programming wireless sensor networks with the teeny lime middleware. *Middleware 2007*, 429–449. 9

de Saint-Exupery, A. 2009. Internet of things, strategic research roadmap. Tech. rep. 2

Dey, A., Abowd, G., and Salber, D. 2001. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human–Computer Interaction 16*, 97–166. 16

Doddapaneni, K., Ever, E., Gemikonakli, O., Malavolta, I., Mostarda, L., and Muccini, H. 2012. A model-driven engineering framework for architecting and analysing wireless sensor networks. In *Third International Workshop on Software Engineering for Sensor Network Applications (SESENA)*. IEEE, 1–7. 25

Drey, Z., Mercadal, J., and Consel, C. 2009. A taxonomy-driven approach to visually prototyping pervasive computing applications. In *Domain-Specific Languages*. Springer, 78–99. 22

Duquennoy, S., Grimaud, G., and Vandewalle, J.-J. 2009. The Web of Things : interconnecting devices with high usability and performance. In *International Conference on Embedded Software and Systems (ICESS)*. 323–330. 17

Erlikh, L. 2000. Leveraging Legacy System Dollars for E-Business. *IT Professional 2,* 3 (May), 17–23. 81

Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A. 2003. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR) 35,* 2, 114–131. 6

Evans, D. 2011. The internet of things : How the next evolution of the internet is changing everything. *CISCO white paper*. 1

Fielding, R. T. 2000. Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California. 19

Fowler, M. 1996. *Analysis Patterns : Reusable Object Models*. Addison-Wesley Longman, Amsterdam. 25

France, R. and Rumpe, B. 2007. Model-driven development of complex software : A research roadmap. In *2007 Future of Software Engineering*. IEEE Computer Society, 37–54. 11

Frank, C. and Römer, K. 2005. Algorithms for generic role assignment in wireless sensor networks. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*. ACM, 230–242. 9

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. 51

Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesC language : A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. Vol. 38. 1–11. 9

Ghidini, G., Das, S. K., and Gupta, V. 2012. Fuseviz : A framework for web-based data fusion and visualization in smart environments. In *IEEE 9th International Conference on Mobile Adhoc and Sensor Systems (MASS)*. IEEE, 468–472. 17

References

Gibbons, P. B., Karp, B., Ke, Y., Nath, S., and Seshan, S. 2003. Irisnet : An architecture for a worldwide sensor web. *Pervasive Computing, IEEE 2,* 4, 22–33. 18

Guerrero-Garcia, J., Gonzalez-Calleros, J. M., Vanderdonckt, J., and Muoz-Arteaga, J. 2009. A theoretical survey of user interface description languages : Preliminary results. In *Web Congress. Latin American.* IEEE, 36–43. 87

Guinard, D., Trifa, V., and Wilde, E. 2010. A resource oriented architecture for the Web of Things. In *Internet of Things (IOT), 2010.* IEEE, 1–8. 17

Gummadi, R., Gnawali, O., and Govindan, R. 2005. Macro-programming wireless sensor networks using kairos. *Distributed Computing in Sensor Systems*, 466–466. 19

Gupta, V., Udupi, P., and Poursohi, A. 2010. Early lessons from building sensor. network : an open data exchange for the web of things. In *8th IEEE International Conference on Pervasive Computing and Communications Workshops.* 738–744. 17

Haller, S. 2010. The Things in the Internet of Things. *Poster at the (IoT 2010). Tokyo, Japan, November*. 5, 26

Harrison, C. and Donnelly, I. A. 2011. A theory of smart cities. In *Proceedings of the 55th Annual Meeting of the ISSS-2011.* Vol. 55. Hull, UK. 1

Haubensak, O. 2011. Smart cities and internet of things. In *Business Aspects of the Internet of Things, Seminar of Advanced Topics, ETH Zurich.* 33–39. 1

Henricksen, K. and Indulska, J. 2006. Developing context-aware pervasive computing applications : Models and approach. *Pervasive and Mobile Computing 2,* 1, 37–64. 16

Henricksen, K. and Robinson, R. 2006. A survey of middleware for sensor networks : state-of-the-art and future directions. In *Proceedings of the international workshop on Middleware for sensor networks.* ACM, 60–65. 15

Hnat, T. W., Sookoor, T. I., Hooimeijer, P., Weimer, W., and Whitehouse, K. 2008. Macrolab : a vector-based macroprogramming framework for cyber-physical systems. In *Proceedings of the 6th ACM conference on Embedded network sensor systems.* ACM, 225–238. 10, 20

Jaikaeo, C., Srisathapornphat, C., and Shen, C.-C. 2000. Querying and tasking in sensor networks. In *AeroSense 2000.* International Society for Optics and Photonics, 184–194. 18

Kruchten, P. 1995. The 4+1 View Model of Architecture. *IEEE Softw. 12,* 6 (Nov.), 42–50. 29

Kulkarni, V. and Reddy, S. 2003. Separation of Concerns in Model-Driven Development. *IEEE Software 20,* 5, 64–69. 11

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., et al. 2005. TinyOS : An operating system for sensor networks. *Ambient intelligence 35*. 16

Losilla, F., Vicente-Chicote, C., Álvarez, B., Iborra, A., and Sánchez, P. 2007. Wireless Sensor Network Application Development : An Architecture-Centric MDE Approach. In *ECSA*. Lecture Notes in Computer Science, vol. 4758. Springer, 179–194. 10

Luckenbach, T., Gober, P., Arbanowski, S., Kotsopoulos, A., and Kim, K. 2005. TinyREST-a protocol for integrating sensor networks into the internet. In *Proc. of REALWSN*. Citeseer. 19

Madden, S., Franklin, M., Hellerstein, J., and Hong, W. 2005. TinyDB : An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS) 30,* 1, 122–173. 17

McCabe, T. J. 1976. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*. ICSE '76. IEEE Computer Society Press, Los Alamitos, CA, USA, 407–. 74

Mellor, S. J., Clark, A. N., and Futagami, T. 2003. Guest editors' introduction : Model-driven development. *IEEE Softw. 20,* 5 (Sept.), 14–18. 11

Mohamed, N. and Al-Jaroodi, J. 2001. A survey on service-oriented middleware for wireless sensor networks. *Serv. Oriented Comput. Appl. 5,* 2 (June), 71–85. 18

Mottola, L., Pathak, A., Bakshi, A., Prasanna, V., and Picco, G. 2007. Enabling scope-based interactions in sensor network macroprogramming. In *IEEE Internatonal Conference on Mobile Adhoc and Sensor Systems*. IEEE, 1–9. 47

Mottola, L. and Picco, G. 2011. Programming wireless sensor networks : Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR) 43,* 3, 19. 15, 73

Naphade, M., Banavar, G., Harrison, C., Paraszczak, J., and Morris, R. 2011. Smarter Cities and Their Innovation Challenges. *Computer 44,* 6 (June), 32–39. 1

Nations, U. 2010. *World Urbanization Prospects : The 2009 Revision*. UN. 1

Newton, R., Morrisett, G., and Welsh, M. 2007. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*. ACM, 489–498. 10, 20

Nguyen, T. A. and Aiello, M. 2012. Energy intelligent buildings based on user activity : A survey. *Energy and Buildings 56,* 0, 244 – 257. 3

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15,* 12, 1053–1058. 11

References

Parr, T. 2007. *The Definitive ANTLR Reference : Building Domain-Specific Languages.* Pragmatic Bookshelf. 63

Pathak, A. and Gowda, M. K. 2009. Srijan : a graphical toolkit for sensor network macroprogramming. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* ACM, 301–302. 13

Pathak, A., Mottola, L., Bakshi, A., Prasanna, V., and Picco, G. 2007. A compilation framework for macroprogramming networked sensors. *Distributed Computing in Sensor Systems*, 189–204. 10

Pathak, A. and Prasanna, V. 2011. High-level application development for sensor networks : Data-driven approach. *Theoretical Aspects of Distributed Computing in Sensor Networks*, 865–891. 21

Perez-Lombard, L., Ortiz, J., and Pout, C. 2008. A review on buildings energy consumption information. *Energy and buildings 40,* 3, 394–398. 4

Picco, G. 2010. Software engineering and wireless sensor networks : happy marriage or consensual divorce ? In *Proceedings of the FSE/SDP workshop on Future of software engineering research.* ACM, 283–286. 7

Picek, R. and Strahonja, V. 2007. Model driven development-future or failure of software development. In *IIS.* Vol. 7. 407–413. 12

Pohja, M. 2010. Comparison of common XML-based web user interface languages. *Journal of Web Engineering 9,* 2, 95–115. 87

Priyantha, N. B., Kansal, A., Goraczko, M., and Zhao, F. 2008. Tiny web services : design and implementation of interoperable and evolvable sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems.* SenSys '08. ACM, New York, NY, USA, 253–266. 19

Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R., and Mickunas, M. 2005. Olympus : A high-level programming model for pervasive computing environments. In *Third IEEE International Conference on Pervasive Computing and Communications(PerCom).* 7–16. 16

Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K. 2002. Gaia : a middleware platform for active spaces. *ACM SIGMOBILE Mobile Computing and Communications Review 6,* 4, 65–67. 9, 16

Salber, D., Dey, A. K., and Abowd, G. D. 1999. The context toolkit : aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems : the CHI is the limit.* ACM, New York, NY, USA, 434–441. 16

Schmidt, D. C. 2006. Guest Editor's Introduction : Model-Driven Engineering. *IEEE Computer Society Available : http://www.cs.wustl.edu/~schmidt/GEI.pdf 39*, 25–31. 11

Serral, E., Valderas, P., and Pelechano, V. 2010. Towards the model driven development of context-aware pervasive systems. *Pervasive and Mobile Computing 6,* 2, 254–280. 8, 20

Shen, C., Srisathapornphat, C., and Jaikaeo, C. 2001. Sensor information networking architecture and applications. *Personal communications, IEEE 8,* 4, 52–59. 18, 41

Smith, D., Henretig, J., Pittenger, J., Bernard, R., Kofmehl, A., Levine, A., Falco, G., Schmidt, K., Granderson, J., and Piette, M. A. 2011. Energy-smart buildings : Demonstrating how information technology can cut energy use and costs of real estate portfolios. Technical report. 4

Solberg, A., Simmonds, D. M., Reddy, R., Ghosh, S., and France, R. B. 2005. Using aspect oriented techniques to support separation of concerns in model driven development. In *COMPSAC (1)*. IEEE Computer Society, 121–126. 11

Sommerville, I. 2010. *Software Engineering*, 9 ed. Addison-Wesley, Harlow, England. 13, 34

Souchon, N. and Vanderdonckt, J. 2003. A review of XML-compliant user interface description languages. In *Interactive Systems. Design, Specification, and Verification*. Springer, 377–391. 87

Sugihara, R. and Gupta, R. 2008. Programming models for sensor networks : A survey. *ACM Transactions on Sensor Networks (TOSN) 4,* 2, 8. 15

Taylor, R. N., Medvidovic, N., and Dashofy, E. M. 2009. *Software Architecture : Foundations, Theory, and Practice*. Wiley Publishing. 9, 15, 28, 30

Tubaishat, M. and Madria, S. 2003. Sensor networks : an overview. *Potentials, IEEE 22,* 2, 20–23. 26

Vasseur, J.-P. and Dunkels, A. 2010. *Interconnecting smart objects with IP : The next internet*. Morgan Kaufmann, San Francisco, CA, USA. 1, 3

Whitehouse, K., Sharp, C., Brewer, E., and Culler, D. 2004. Hood : a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM, 99–110. 9

Whitehouse, K., Zhao, F., and Liu, J. 2006. Semantic streams : a framework for composable semantic interpretation of sensor data. In *Proceedings of the Third European conference on Wireless Sensor Networks*. EWSN'06. Springer-Verlag, Berlin, Heidelberg, 5–20. 18

Yao, Y. and Gehrke, J. 2002. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec. 31,* 3 (Sept.), 9–18. 17