Enabling High-Level Application Development in the Internet of Things

Pankesh Patel[1], Animesh Pathak[1], Damien Cassou[2], and Valérie Issarny[1]

[1] Inria Paris-Rocquencourt, France
[2] Inria Lille-Nord Europe, France
firstname.lastname@inria.fr

Abstract. The sensor networking field is evolving into the Internet of Things (IoT), owing in large part to the increased availability of consumer sensing devices, including modern smart phones. However, application development in the IoT still remains challenging, since it involves dealing with several related issues, such as lack of proper identification of roles of various stakeholders, as well as lack of suitable (high-level) abstractions to address the large scale and heterogeneity in IoT systems.

Although the software engineering community has proposed several approaches to address the above in the general case, existing approaches for IoT application development only cover limited subsets of above mentioned challenges. In this paper, we propose a multi-stage model-driven approach for IoT application development based on a precise definition of the role to be played by each stakeholder involved in the process – domain expert, application designer, application developer, device developer, and network manager. The abstractions provided to each stakeholder are further customized using the inputs provided in the earlier stages by other stakeholders. We have also implemented code-generation and task-mapping techniques to support our approach. Our initial evaluation based on two realistic scenarios shows that the use of our techniques/framework succeeds in improving productivity in the IoT application development process.

Keywords: Internet of Things, Sensor networks, High-level programming, Application development, Computing abstractions.

1 Introduction

With the increased availability of consumer sensing devices including modern smart phones, the domain of networked sensing has evolved into the Internet of Things (IoT) [2], where applications exhibit traditional Wireless Sensor and Actuator Network (WSAN) features such as large number of sensing and actuation devices, coupled with increased device heterogeneity and integration with database servers, etc. [1]. To illustrate the characteristics of IoT applications, we consider the following office environment management application.

M. Zuniga and G. Dini (Eds): S-Cube 2013, LNICST 122, pp. 111–126, 2013.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2013

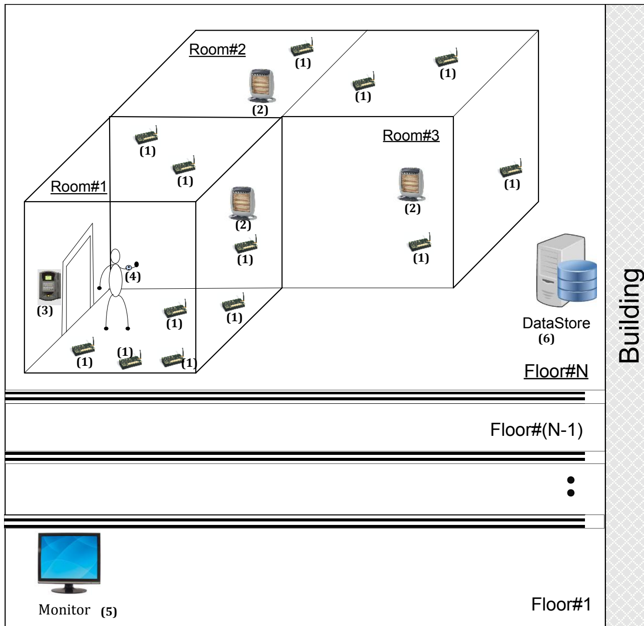

Fig. 1. Multi-floored building with deployed devices with (1) Temperature sensor, (2) Heater, (3) Badge reader, (4) Badge, (5) Monitor, and (6) DataStore

1.1 Illustrative Application: Office Environment Management

An office might consist of several buildings, with each building in turn consisting of one or more floors, each with several rooms, each instrumented with a *large number* of *heterogeneous* devices with attached sensors, actuators, and data storage devices (Figure 1). The system aims to regulate appropriate temperature for worker productivity and comfort, and provides general information such as average temperature of the building.

The temperature in each room of the building is regulated by a sense-compute-actuate loop executing among the temperature sensors and heaters of the room to maintain an appropriate room temperature. Additionally, average temperature values are computed at floor and building levels to be displayed on monitors at the building entrance and control station. When a user enters or leaves a room, a badge reader detects this event, and queries a central employee database for the user's preferences. Based on the response, the threshold used by the room's devices is updated.

1.2 Challenges and Contributions

An important challenge that remains to be addressed in the IoT is the *ease of application development*. There is a growing awareness of this problem in the research community, and several approaches have been proposed for both

WSANs [12, 17, 21] and pervasive computing systems [4, 7, 8]. While the main challenge in WSANs is the extremely *large scale* of the systems (hundreds to thousands of largely similar nodes), the primary concern in the pervasive computing field has been the *heterogeneity* of devices. Since IoT applications include both these aspects, application development in such a system raises the following research challenges:

– **Lack of Division of Roles.** Most application development approaches proposed for IoT assume only *one* role—the developer/programmer—to be played by the individuals developing the IoT application. This is in clear conflict with the varied set of skills required during the IoT application development process including domain expertise, distributed system knowledge, and low-level hardware knowledge, and therefore hinders rapid application development, a challenge recognized by recent works such as [5].
– **Inadequately Customized Abstractions.** While there are some approaches based on domain-specific languages [20, 23] for such applications, domain-specific development approaches consider all IoT applications to belong to the *same domain*. We believe, however, that further customization of abstractions is needed at the level of the *application domain*, and thus the one-language-to-program-them-all approach needs to be revisited.
– **Heterogeneity of Target Devices.** IoT systems are heterogeneous both in terms of the implementations of the sensors/actuators to be used, as well as in terms of interaction modes (e.g., command, request-response, and publish-subscribe). It should not be the developer's responsibility to handle this heterogeneity, since ideally the *same* application should execute on entirely different deployments (e.g., the same smart building application on different offices with different devices).
– **Scale.** Applications in the IoT may execute on systems consisting of hundreds to thousands of devices. Requiring the ability of reasoning at such levels of scale is impractical in general. Consequently, there is a need for adequate abstractions to allow the stakeholders involved to express their requirements in a compact manner regardless of the scale of the final systems.

Existing software engineering approaches for IoT application development [4, 7, 17, 21–23] only cover a limited subset of the above-mentioned challenges. This paper, on the contrary, proposes an integrated approach for all these challenges and provides a comprehensive coverage of the IoT application development process. Our work aims to address the above challenges by making the following contributions:

– **Identification of Roles in the Development Process.** Leveraging the conceptual model of the IoT from our previous work [19], we identify the precise role of each stakeholder involved in the development of IoT applications, thus promoting a suitable division of labor among them.
– **A Multi-stage Model-driven Approach for IoT Application Development.** In order to support the different stages of application development, we propose a multi-stage model-driven approach that links the actions of

each stakeholders. Our approach (discussed in Section 2) defines a precise sequence of steps to follow, thus smoothening the IoT application development process.

- **Modeling Languages.** Our multi-stage process is complemented by a set of parametrized languages, (detailed in Section 3) for each type of stakeholder. Their grammars are generated based on the inputs from other stakeholders from earlier in the process, and include abstractions to hide the scale- and heterogeneity-related complexity.
- **Code Generation and Task-mapping.** Our multi-stage model-driven approach is supported by code generation and task-mapping techniques. The two techniques together provide automation in the application development process, which significantly reduces development effort (Section 4). While the former aids in specifying the logic of the software components in the IoT application, the latter supports the application deployment phase by producing device-specific code to result in a distributed software system collaboratively hosted by individual devices.

We present the related work in Section 5. We summarize our contribution so far and discuss our future work in Section 6.

2 Multi-stage Model-driven Approach for IoT Application Development

As stated above, traditional WSAN/IoT application development assumes that the individuals involved in application development 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 the 4+1 view model of software architecture [14], collaboration model for smart spaces [5], and tool-based methodology for pervasive computing [4], we propose to divide the responsibilities of the stakeholders in the IoT application development process into five distinct roles —domain expert, software designer, application developer, device developer, and network manager— whose skills and responsibilities are stated in Table 1. Note that although these roles have been discussed in the Software Engineering literature in general, e.g., domain expert and software designer in [24], and application developer, device developer and network manager in [4], their clear identification for IoT application is largely missing.

Based on the roles defined in Table 1, we propose the following multi-stage model-driven development process (detailed in Figure 2) that consists of the following steps:

Domain Vocabulary Specification. Since several IoT applications might be developed for the same application domain, the first step of our approach consists of the specification of the domain vocabulary by the *domain expert* stakeholder (step ① in Figure 2). The vocabulary includes the specification of entities (i.e.,

Table 1. Roles in the IoT Application Development Process

Role	Skills	Responsibilities
Domain Expert	Understands domain concepts, including the data types produced by the sensors and consumed by actuators, as well as how the system is divided into regions.	Specify the vocabulary to be used by all applications in the domain.
Software Designer	Software architecture concepts, including the proper use of interaction pattern such as publish-subscribe and request-response for use in the application.	Define the structure of the IoT application by specifying the software components and their producer/consumer relationships.
Application Developer	Skilled in algorithm design and use of programming languages.	Develop the internal code for the computational services, and controllers in the application.
Device Developer	Deep understanding of the inputs/outputs, and protocols of the individual devices.	Write drivers for the sensors and actuators 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.

sensor, actuator, and storage service) that are responsible for interacting with the physical world. The entities are specified in high-level manner to abstract their different implementations, thus abstracting heterogeneity. It also includes the definition of partitions that the system is divided into to support scalable operations within the IoT system.

Application Architecture Specification. Using the concepts defined in the vocabulary, the *software designer* specifies the architecture of an application (step ② in Figure 2), consisting of the details of the computational and controller component, as well as how they interact with other software components using different interaction paradigms. To address scalable operations within IoT system, our language offers scope constructs to identify the particular region whose data a software component is interested in.

Application Logic Implementation. Our approach leverages both vocabulary and architecture specification to support the *application developer*. To describe the application logic of each software component, the application developer is provided a customized programming framework, pre-configured according

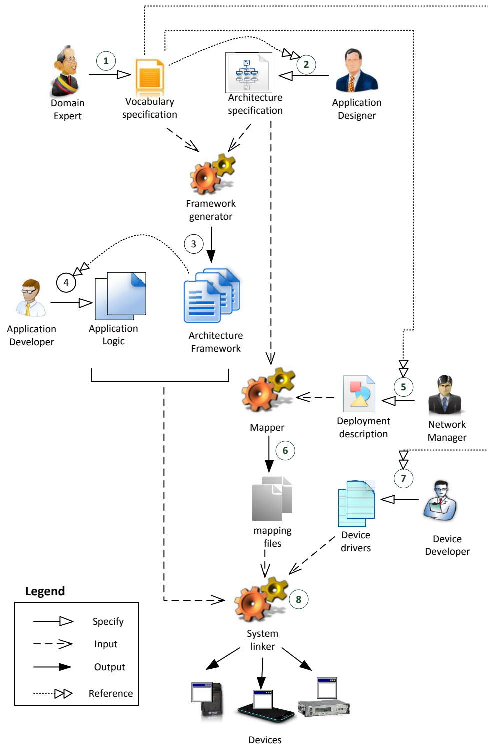

Fig. 2. IoT Application Development using Our Approach

to the architecture specification of an application, an approach similar to the one discussed in [3] (step ③ in Figure 2). The programming framework contains abstract classes[1] for each software component that hide low-level communication details such as the communication paradigm used, and allow the developer to focus only on the code that implements the logic of that software component (step ④ in Figure 2).

Target Deployment Specification. The same IoT application should be deployable on different target networks (e.g., the same inventory tracking application can be deployed in different warehouses). In our approach, for each new target network where the application is to be deployed, the respective network manager specifies (step ⑤ in Figure 2) the details of the devices in his system, using the concepts defined previously in the vocabulary.

Deployment Code Generation. To generate the final code to be deployed on each device, this stage consists of two core sub-stages: *Mapping* and *Linking*.

– **Mapping.** The mapper produces a mapping from the set of instantiated software components to the set of devices (step ⑥ in Figure 2). It takes as

[1] We assume that the developer uses an object-oriented language.

input the set of *instantiation rules* of software components from architecture specification and the *set of devices* defined in the deployment description. The mapper decides the specific device where each software component will be deployed.

– **Linking.** The linker combines the information generated by the various stages of the compilation into the actual code to be deployed on the real devices. It merges the generated code, the code provided by the *application developer*, the mappings produced in the mapping stage, and device specific drivers provided by *device developer* (step ⑦ in Figure 2). The output of this stage is a final executable code for each device in the deployment (step ⑧ in Figure 2).

Note that our approach currently does not provide support for testing and debugging, which is part of our future work. Support for iterative development methodologies is implicit in our work since, for example, the domain expert can release a new version of the vocabulary that can then be used to minimally change the architecture.

3 Detail of Our Approach

Based on our analysis of the roles played by the various stakeholders in the stages described above, we have designed a set of customizable languages, each named after *Srijan*, the Sanskrit word for “creation”. In this section, we provide the details of above mentioned stages with the set of languages using the application introduced in Section 1. Details such as the grammars of the languages can be found in [18].

3.1 Domain Vocabulary Specification

The Srijan Vocabulary Language (SVL) is designed to enable the *domain expert* to describe the vocabulary of a domain. The language offers the following constructs to specify the following concepts that interact with physical world:

– *regions:* These define the various labels that can be used to specify the (logical) locations of the devices and scopes from which the software components will produce/consume data. For example, HVAC[2] applications reason in terms of *room* and *floors*, while smart city applications can be best expressed in terms of *city blocks*. This construct is declared using the `regions` keyword (Listing 1.1, lines 1-4).
– *data structures:* Each entity is characterized by the types of information it generates and consumes. This information is defined using the `struct` keyword (Listing 1.1, lines 5-11).

[2] Heating, Ventilation, Air-Conditioning.

– *abilities:* These define the abilities of each node that might be attached to various types of sensors, actuators, or storages along with the inputs/output data type. Each sensor and its ability is declared using the `sensors` and `generate` keywords respectively (Listing 1.1, lines 13-15). An actuator and its actions are declared using the `actuators` and `action` keywords respectively. An action of the actuator may take one or more inputs, specified as *parameters* of an action (Listing 1.1, lines 16-18). A storage is declared using `storages` keyword (Listing 1.1, lines 19-21). Retrieval of data from storage service requires a *parameter*. Such a parameter is specified by `accessed-by` keyword.

```
1 regions:
2   Building : integer;
3   Floor : integer ;
4   Room : integer;
5 structs :
6   BadgeDetectedStruct
7     badgeID : string;
8     timeStamp : long;
9   TempStruct
10    tempValue : double;
11    unitOfMeasurement: string;
12 abilities:
13   sensors:
14     BadgeReader
15       generate badgeDetected:BadgeDetectedStruct;
16   actuators:
17     Heater
18         action SetTemp(setTemp:TempStruct);
19   storages:
20     ProfileDB
21       generate profile : TempStruct accessed-by badgeID :
             string;
```

Listing 1.1. Code snippet of office environment management vocabulary specification using SVL. Keywords are printed in ***bold***.

The region labels and data structures defined using SVL in the vocabulary are used to parameterize the grammar of the Srijan Architecture Language (SAL), and can be exploited by tools to provide support such as code completion to the application designer, discussed next.

3.2 Application Architecture Specification

Based on the domain vocabulary, the grammar of SAL is parameterized to enable the *application designer* to design an application. Specifically, the sensors, actuators, storage, and regions defined in the application domain's vocabulary become the possible set of values for certain attributes in the SAL (see underlined words in Listing 1.2). The resulting language offers constructs to specify the following concepts:

– *software components:* SAL provides abstractions for describing two classes of software components that are relevant to the architecture of IoT application. The *computational service* is fueled by sensor(s), storage, or other computational services. It transforms data to be consumed by other software components in accordance with the application needs. The *controller* takes data from other software components as input and takes decisions that are carried out by invoking actuators.
– *interactions among components:* SAL defines the following data-flow between each type of software component.
  - For each controller, the data types it consumes and the actuator to which it issues commands is expressed using the `consume` and `command` keywords respectively.
  - For each computational service, the data types it consumes, produces is expressed using `consume` and `generate` keyword respectively. The accesses from storage services is specfied using `request` keyword.
– *instantiation rule and data interest of components:* To address scalable operations within IoT applications, SAL offers the *scope* construct. Inspired by the work in [16], this serves to provide a logical grouping among the devices in the deployment. The scope constructs in SAL affect (1) component placement on the real devices (defines using `in-region` keyword), which is used to govern the placement of software components on the real devices; and (2) data exchange among software components (define using `hops` keyword), which is used to annotate `consume` or `command` keyword with data interest regardless of its physical location. Component placement and data interest jointly define the scopes from which data should be gathered.

We illustrate SAL by examining a code snippet describing part of the architecture of the office environment management application (similar to one discussed in [4]) (Listing 1.2). This code snippet revolves around the actions of the `Proximity` component (Listing 1.2, lines 3-7), which coordinates events from the `BadgeReader` with the content of `ProfileDB` storage service. To do so, `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` that is expressed using the `request` keyword (Listing 1.2, line 6). Input data is declared using the `consume` keyword that takes source name and data interest of component from logical region (Listing 1.2, line 5). The declaration of `hops:0:room` indicates that the component is interested in consuming badge events of the current room. The `Proximity` component is in charge of managing badge events of *room*. Therefore, we need `Proximity` service to be partitioned per room using `in-region:room` (Listing 1.2, line 7).

The output of the `Proximity` and `RoomAvgTemp` are consumed by the `RegulateTemp` component, declared using the `controller` keyword (Listing 1.2, lines 13-17). This component is responsible for taking decisions that are carried out by invoking commands on the *Heater*, declared using the `command` keyword (Listing 1.2, line 16).

```
1 softwarecomponents:
2   computationalService:
3     Proximity
4       generate tempPref : UserTempPrefStruct;
5       consume badgeDetected from hops:0: Room;
6       request profile;
7       in-region: Room;
8     RoomAvgTemp
9       generate roomAvgTempMeasurement:TempStruct;
10      consume tempMeasurement from hops:0: Room ;
11      in-region: Room;
12  controller:
13    RegulateTemp
14      consume roomAvgTempMeasurement from hops:0: Room;
15      consume tempPref from hops:0: Room;
16      command SetTemp(setTemp) to hops:0: Room;
17      in-region: Room;
```

Listing 1.2. Code snippet of office environment management architecture specification using SAL. Keywords derived from vocabulary are printed underlined, while language keywords are printed in ***bold***.

3.3 Application Logic Implementation

Leveraging both the vocabulary and architecture specification, we generate a programming framework to aid the application developer in development process. We note two key advantages of this programming framework: (1) *Handling large scale*: The generated framework contains code that defines the data interest of a software component, which enables scalable operation. (2) *Ease of application development*: The generated framework contains significant amount of glue code that takes care of the interfacing between hardware and software components. The generated framework raises the level of abstractions by providing the application developer with suitable operations for specifying the application logic.

The generated framework contains *abstract classes* corresponding to architecture and vocabulary specification, an approach similar to the one discussed in [3]. The abstract classes include two types of methods: (1) *concrete methods* to interact with other components transparently, without dealing with the low-level interaction details; and (2) *abstract methods* that the *application developer* implements to specify the application logic.

For each input declared by a component, an abstract method is generated for receiving data. This abstact method is then implemented by the application developer in the subclass. For example, from the `badgeDetected` input of `Proximity` declaration in the architecture specification (Listing 1.2, lines 3-7), `onNewbadgeDetected()` abstract method is generated. This method is implemented by the application developer. Listing 1.3 illustrates the implementation of `onNewbadgeDetected()`. It updates the user's temperature preference and sets it using `settempPref()` method.

```
1 public class SimpleProximity extends Proximity {
2
3     public void onNewbadgeDetected(BadgeDetectedStruct arg)
          {
4
5         UserTempPrefStruct userTempPref = new
              UserTempPrefStruct(
6         arg.gettempValue(), arg.getunitOfMeasurement());
7
8     settempPref(userTempPref);
9   }
10 }
```

Listing 1.3. A concrete implementation of Java abstract class `Proximity`. This implementation is written by the application developer.

3.4 Target Deployment Specification

Given a vocabulary, the Srijan Deployment Language (SDL) is customized to enable the *network manager* to specify the details of each node in the system, including its placement (in terms of values of the region labels defined in the vocabulary), and abilities (a subset of those defined in the vocabulary). We illustrate SDL by examining the deployment specification of our office environment management (Listing 1.4). This snippet describes a device called `TemperatureMgmt-Device-1` with an attached `TemperatureSensor` and `Heater`, situated in `building` 15, `floor` 11, `room` 1.

```
1 devices:
2   TemperatureMgmt-Device-1 :
3     region :
4       Building : 15 ;
5       Floor : 11;
6       Room : 1;
7     abilities : TemperatureSensor, Heater;
8   ...
```

Listing 1.4. Code snippet of office environment management deployment specification using SDL. Keywords from vocabulary are printed underlined, while language keywords are printed in ***bold***.

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 decoupling between this description and the application's code provided by our approach, the same description file can be re-used to create node-level code for all IoT applications based on the same domain vocabulary deployed on a given network.

4 Evaluation

This section evaluates our approach and shows how it reduces the *development effort* of IoT applications. We measure development effort through the number of lines of code (LoC) written by the authors. For this evaluation, we implemented two representative IoT applications of two different domains with our approach: (1) an **office environment management** application (as described in Section 1) and (2) a **fire management** application, which aims to detect fire in house and housing community (collection of houses). In the latter application, fire is detected by analyzing data from smoke and temperature sensors. When a fire occurs, the application triggers sprinklers and unlocks doors to allow residents to evacuate the house. Additionally, residents of the house and of the whole neighborhood are informed through a set of alarms and warning lights. Table 2 summarizes the kinds of components used by each application. For validation, we deployed both applications on a set of nodes running on top of a middleware/simulator written in Java.

Table 2. List of components of two representative applications

Component Type	Office Environment Management	Fire Management
Sensing	TemperatureSensor BadgeReader	TemperatureSensor SmokeDetector
Actuating	Heater Monitor	Door Alarm SprinklerSystem Warning Light
Storage	ProfileDB	*none*
Computational	RoomAvgTemp FloorAvgTemp BuildingAvgTemp Proximity	HouseAvgTempComputation HouseFireComputation HcFireComputation
Controller	RegulateTemp ManageTemp	HouseFireController HcFireController

Development Effort.. Our measurements (using the Metrics[3] 1.3.6 Eclipse plug-in) reveal that more than 81% of total number of lines of code is generated in both applications (see Table 3).

Code Coverage.. The measure of LoC is only useful if the generated code is actually executed. Similar to the approach in [4], we measured the coverage of the generated programming framework and handwritten application logic (see

[3] `http://metrics.sourceforge.net`

Table 4) using the EclEmma[4] Eclipse plug-in. Our measures show that more than 90% of generated code is actually executed, the 10 other percents 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 the development effort by generating useful code.

Table 3. Lines of code in application development process

Application Name	Handwritten (Lines of Code)				Generated (Lines of Code)			$\frac{generated}{handwritten+generated}$
	Vocab Spec.	Arch. Spec.	Network Spec.	App. Logic	Partial App. Logic	Mapping code	Generated Framework	
Office Env. Mgmt.	30	36	49	169	139	470	638	81.45%
Fire Mgmt.	28	35	41	125	144	336	575	82.16%

Table 4. Code coverage of handwritten and generated code

Application	Handwritten code	Generated code
Office Env. Mgmt.	95.8 %	90.7 %
Fire Mgmt.	95.4 %	93.3 %

5 Related Work

Many software engineering approaches have been proposed to simplify the development of Pervasive computing systems, WSAN, and IoT applications. The existing work can broadly be divided into two categories: (1) node-level programming and (2) system-level programming.

Node-level programming refers to a process in which programmers are directly concerned with the operation of each node in a system. Programmers typically write such an application in a general-purpose programming language (often Java or C) and target a particular middleware API or node-level service [6, 9, 25]. Even if this approach scales for a large number of similar nodes, it is impractical for large scale heterogeneous systems as can be found in IoT systems.

System-level programming refers to a process in which programmers describe how the system will behave as a whole, regardless of how the system will get deployed on particular nodes. The techniques investigated in the literature

[4] `http://www.eclemma.org/`

towards IoT application development can be further categorized as: (1) *library- or toolkit-based* and (2) *model driven*.

Library- or toolkit-based approaches. These typically offer abstractions and services to help developers implement their applications in a general-purpose programming language. Gaia (with its Olympus toolkit) [22], context-toolkit [7], AURA [10], and one.world [11] are notable examples. They reduce development effort by hiding many of the low level details (such as network communication) inherent in IoT development. Nevertheless, these approaches have a steep learning curve as they tend to become more complex with time. Moreover, a significant amount of glue code needs to be written to adapt the approach to the requirements of the application. In contrast, our languages are small and each is dedicated to a particular stakeholder's skills and knowledge.

Model-driven (MDE) approaches. MDE techniques have been proposed to limit the burden of developing IoT applications. In such an approach, applications are specified using high-level and abstract models and then given as input to code generators which produce low-level code as output. For instance, PervML [23] allows developers to specify pervasive systems at a high-level of abstraction through a set of models (in UML). Nevertheless, such approaches typically require expertise in the modeling language which stakeholders might not be willing to acquire.

A few vocabulary-inspired approaches [4, 8] have been proposed in the pervasive computing domain. Representative of these approaches is DiaSuite [4], a tool-based framework that allows stakeholders to define a vocabulary of entities dedicated to an application (i.e., sensor, actuator), thus abstracting over their heterogeneity. Compared to DiaSuite, our work offers concepts (such as regions) dedicated to describing large scale applications. These concepts are of utmost importance in IoT. Additionally, we offer automated deployment of the application on all devices using a mapping technique.

Numerous approaches [12, 15, 17, 21] have been proposed to address the large scale challenge of the sensor network community. However, the clear separation of roles among the various stakeholders of the application development process, as well as the focus on heterogeneity among the constituent devices has been largely missing from WSAN macroprogramming research so far.

6 Conclusion and Future Work

To address the challenges faced during the development of IoT applications, in this paper we presented a multi-stage model driven approach for IoT application development, founded on a clear identification of the skills and responsibilities of the various stakeholders involved in the process. Notable in our approach is the use of customized modeling languages tuned to each stakeholder's specific task and the application domain, where the abstractions available to one stakeholder are generated from the information provided by other stakeholders in previous stages. Our approach is complemented by code generation and task-mapping techniques which lead to the node-level code to be deployed on the constituent

devices. Our initial evaluation based on two representative scenarios shows that the use of our techniques improves productivity in the IoT application development process.

In our work so far, we have made progress toward providing support to all the stakeholders in the IoT application development process and have prepared a foundation for our future work. With this foundation in place, our future work will involve three complementary aspects: (1) We will provide richer abstractions to express the properties of the devices (e.g., processing and storage capacity, as well as mobility properties). These will then be used to guide the design of algorithms for efficient mapping of software components on devices. (2) We intend to include features to enable the stakeholders in common development tasks such as testing, and allowing for iterative application development based on evolution of requirements. (3) The evaluation presented in Section 4 is preliminary. We plan to conduct an empirical evaluation based on a well-defined experimental methodology [13]. In particular, we will explore the aspects of *reusability* and *expressiveness*.

Acknowledgments.. This work was supported in part by the European Commission FP7 NESSoS project and the ANR Murphy project. The authors are grateful to the reviewers for their helpful comments.

References

1. Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. Computer Networks 54(15), 2787–2805 (2010)
2. CASAGRAS EU project final report (2009), `http://www.rfidglobal.eu/userfiles/documents/FinalReport.pdf`
3. Cassou, D., Bertran, B., Loriant, N., Consel, C.: A Generative Programming Approach to Developing Pervasive Computing Systems. In: GPCE 2009: Proceedings of the 8th International Conference on Generative Programming and Component Engineering (2009)
4. Cassou, D., Bruneau, J., Consel, C., Balland, E.: Towards a tool-based development methodology for pervasive computing applications. IEEE Transactions on Software Engineering (2011)
5. Chen, C., Helal, S., de Deugd, S., Smith, A., Chang, C.: Toward a collaboration model for smart spaces. In: 2012 Third International Workshop on Software Engineering for Sensor Network Applications (SESENA), pp. 37–42. IEEE (2012)
6. Costa, P., Mottola, L., Murphy, A.L., Picco, G.P.: Programming wireless sensor networks with the teeny lime middleware. In: Cerqueira, R., Campbell, R.H. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 429–449. Springer, Heidelberg (2007)
7. Dey, A., Abowd, G., Salber, D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction 16(2-4), 97–166 (2001)
8. Drey, Z., Mercadal, J., Consel, C.: A taxonomy-driven approach to visually prototyping pervasive computing applications. In: Taha, W.M. (ed.) DSL 2009. LNCS, vol. 5658, pp. 78–99. Springer, Heidelberg (2009)
9. Frank, C., Römer, K.: Algorithms for generic role assignment in wireless sensor networks. In: Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems, pp. 230–242. ACM (2005)

10. Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste, P.: Project aura: Toward distraction-free pervasive computing. IEEE Pervasive Computing 1(2), 22–31 (2002)
11. Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., Wetherall, D.: System support for pervasive applications. ACM Transactions on Computer Systems (TOCS) 22(4), 421–486 (2004)
12. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using *kairos*. In: Prasanna, V.K., Iyengar, S.S., Spirakis, P.G., Welsh, M. (eds.) DCOSS 2005. LNCS, vol. 3560, pp. 126–140. Springer, Heidelberg (2005)
13. Kitchenham, B., Pickard, L., Pfleeger, S.: Case studies for method and tool evaluation. IEEE Software 12(4), 52–62 (1995)
14. Kruchten, P.: The 4+ 1 view model of architecture. IEEE Software 12(6), 42–50 (1995)
15. Luo, L., Abdelzaher, T., He, T., Stankovic, J.: Envirosuite: An environmentally immersive programming framework for sensor networks. ACM Transactions on Embedded Computing Systems (TECS) 5(3), 543–576 (2006)
16. Mottola, L., Pathak, A., Bakshi, A., Prasanna, V.K., Picco, G.P.: Enabling scope-based interactions in sensor network macroprogramming. In: IEEE Internatonal Conference on Mobile Ad Hoc and Sensor Systems, MASS 2007, pp. 1–9 (October 2007)
17. Mottola, L., Picco, G.: Programming wireless sensor networks: Fundamental concepts and state of the art. ACM Computing Surveys (CSUR) 43(3), 19 (2011)
18. Patel, P.: Enabling High-Level Application Development in the Internet of Things. Techreport (July 2012), `http://hal.inria.fr/hal-00732094`
19. Patel, P., Pathak, A., Teixeira, T., Issarny, V.: Towards application development for the internet of things. In: Proceedings of the 8th Middleware Doctoral Symposium. ACM (2011)
20. Pathak, A., Mottola, L., Bakshi, A., Prasanna, V., Picco, G.: Expressing sensor network interaction patterns using data-driven macroprogramming. In: Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2007, pp. 255–260. IEEE (2007)
21. Pathak, A., Prasanna, V.K.: High-Level Application Development for Sensor Networks: Data-Driven Approach. In: Nikoletseas, S., Rolim, J.D. (eds.) Theoretical Aspects of Distributed Computing in Sensor Networks, Monographs in Theoretical Computer Science. An EATCS Series, pp. 865–891. Springer, Heidelberg (2011)
22. Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R., Mickunas, M.: Olympus: A high-level programming model for pervasive computing environments. In: Third IEEE International Conference on Pervasive Computing and Communications, PerCom 2005, pp. 7–16. IEEE (2005)
23. Serral, E., Valderas, P., Pelechano, V.: Towards the model driven development of context-aware pervasive systems. Pervasive and Mobile Computing 6(2), 254–280 (2010)
24. Taylor, R., Medvidovic, N., Dashofy, E.: Software architecture: foundations, theory, and practice. Wiley (2009)
25. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services, pp. 99–110. ACM (2004)