Programming Internet of Things, Service, and People (IoTSP) Applications

Submitted By:
Saurabh Chauhan,
Research Intern,
ABB Corporate Research, India

Mentor:
Dr. Pankesh Patel,
ABB Corporate Research, India

Acknowledgements

On the very outset of internship at ABB Corporate Research India, It is my radiant sentiment to place on record my best regards, deepest sense of gratitude to Pankesh Patel for his careful and precious guidance which are extremely valuable for my study both theoretically and practically. Without his active guidance, help, cooperation, and encouragement, I could not have made headway in the project.

I express my deepest thanks to Sudarsan, Ashish Sureka, Rahul, Prashanth and Anand for taking part in useful decision & giving necessary advices and guidance and arranged all facilities to make life easier. I choose this moment to acknowledge their contribution gratefully. I would like to express my deepest gratitude and special thanks to the Sudarsan who in spite of being extraordinarily busy with his duties, took time out to hear, guide and keep me on the correct path.

I extend my gratitude to ABB Corporate Research India for giving me this opportunity. I will strive to use gained skills and knowledge in the best possible way, and I will continue to work on their improvement, in order to attain desired career objectives.

Abstract

Application development for Internet of Things, Service, and People (IoTSP) is challenging because it involves dealing with the heterogeneity that exists both in Physical and Internet worlds. Second, stakeholders involved in the application development have to address issues pertaining to different life-cycles ranging from design, implementation to deployment. Given these, a critical challenge is to enable an application development for IoTSP applications with effectively and efficiently from various stakeholders.

Several approaches to tackling this challenge have been proposed in the fields of Wireless Sensor Networks (WSN) and Pervasive Computing, regarded as precursors to the modern day of IoTSP. However, existing approaches only cover limited subsets of the above mentioned challenges when applied to the IoTSP. In view of this, in this report, we have built upon our existing framework and evolved it into a framework for developing IoTSP applications, with substantial additions and enhancements in high-level modeling languages and their integration into the framework, and we present a comparative evaluation results with existing approaches. This provides the IoTSP community for further benchmarking. The evaluation is carried out on real devices exhibiting heterogeneity. Our experimental analysis and results demonstrate that our approach drastically reduces development effort for IoTSP applications compared to existing approaches.

Publications

- Saurabh Chauhan, Pankesh Patel, Flavia Delicato, and Sanjay Chaudhary *"A Development Framework for Programming Cyber-Physical Systems"*, at 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS), Co-located with 38th International Conference on Software Engineering (ICSE), 2016.
- Saurabh Chauhan, Pankesh Patel, Ashish Sureka, Flavia Delicato, and Sanjay Chaudhary *"A ToolSuite for Prototyping Internet of Things Applications"*, at 15th ACM-IEEE International Conference on Information Processing in Sensor Networks (IPSN), 2016

Implementation

- The code for experiments and eclipse plug-in available at `https://codebits.pl.abb.com/saurabh-internshhip-work`.

Contents

1 Introduction **8**
1.1 Application example: Smart Home . . . 8
1.2 IoTSP application development challenges . . . 8
1.3 State of the art: Programming Internet of Things, Service and People . . . 10
1.4 Contributions . . . 11

2 Different development concerns **12**
2.1 Domain Concern . . . 12
2.2 Functional Concern . . . 13
2.3 Platform Concern . . . 13
2.4 Deployment Concern . . . 14
2.5 Linking . . . 14

3 Specify Domain **15**

4 Specify Architecture **17**

5 Specify User-interactions **19**

6 Specify Deployment **20**

7 Application development using IoTSuite **21**
7.1 Create a IoTSuite Project . . . 21
7.1.1 Download IoTSuite-Eclipse-Plugin . . . 21
7.2 Open IoTSuite Eclipse . . . 21
7.3 Create a new IoTSuite Project . . . 22
7.3.1 Specifying high-level specifications . . . 24
7.4 Compilation of an IoTSuite Project . . . 28
7.4.1 Compilation of high-level specification . . . 28
7.5 Deployment of generated packages . . . 34

8 Evaluation **35**

9 Real time PT 100 sensor data visualization using RIO600 **37**

10 Conclusion and Future work **41**

List of Figures

1 A smart home with deployed devices with (1) Temperature sensor, (2) Heater, (3) Fire alarm, (4) Smoke Detector, (5) Badge reader, (6) Data storage, (7) Motion sensor, and (8) Smoke sensor. . . . 9
2 IoTSP application development: the overall process . . . 12
3 Dataflow of Smart Home Application. . . . 18
4 Open IoTSuite Eclipse . . . 22
5 Select default Workspace . . . 22
6 IoTSuite Project Creation Wizard (1) . . . 23
7 IoTSuite Project Creation Wizard (2) . . . 23
8 IoTSuite Project Structure . . . 24
9 IoTSuite editor feature: Outline view . . . 25
10 IoTSuite editor feature: Syntax coloring . . . 25
11 IoTSuite editor feature: Code folding . . . 26
12 IoTSuite editor features: Error checking & Auto completion . . . 26
13 Domain Specification . . . 27
14 Architecture Specification . . . 27
15 User-Interaction Specification . . . 28
16 Deployment Specification . . . 28
17 Compilation of Vocab specification . . . 29
18 Compilation of Architecture specification . . . 29
19 Import application logic package . . . 30
20 Locate application logic package . . . 30
21 Implement application logic package . . . 31
22 Compilation of User-Interaction specification . . . 31
23 Compilation of Deployment specification . . . 32
24 Import user-interface project . . . 32
25 Locate user-interface project . . . 33
26 Implement user-interface project . . . 33
27 Packages for target devices specified in the deployment specification . . . 34
28 Architecture of the system . . . 37
29 RIO600 parameter configuration for RTD module using PCM600 . . . 38
30 GOOSE communication configuration on RIO600 side . . . 38
31 Signal Matrix to subscribe for Analog Channel 1 . . . 39
32 Mapping GOOSE configuration in REF620 . . . 39
33 Displaying temperature value through web-interface of RIO600 . . . 40
34 Displaying Trend using Micro-SCADA . . . 40

List of Tables

1 Summary: existing approaches available for programming IoTSP application. 10
2 Smart home application implementation 35
3 Comparison of existing approaches: Lines of code required to develop the smart home application. **S** (Sensor), **A** (Actuator), **T** (Tag), **WS** (External Web Service), **EU** (End user Application), **ST** (Storage),**Comp.** (Computational Services). 36

1 Introduction

Internet of Things is composed of highly heterogeneous interconnections of elements from both the Physical as well as Internet worlds. IoTSP include WSN, RFID technologies, smart phones, smart appliances as well as the elements of the traditional Internet such as Web and database servers, exposing their functionalists as Web services. In the *Internet of Things*, things formed a network, things could be devices, vehicles, buildings, and other embedded objects that enable exchange of the data. A *thing* in the IoTSP is an entity which measures the *Entity of Interest*. For example, a smart watch is thing and measures the person's body temperature, heart rate etc. Here person's body temperature and heart rate are entity of interest for the entity smart watch. IoTSP applications will involve interactions among large numbers of devices, many of them directly interacting with their physical surroundings.

While IoTSP have shown a great potential of usage into several fields such as smart home, personal health, energy usage monitoring and others, the heterogeneity and diversity involved in IoTSP represent a difficult challenge to deal with. An important challenge that needs to be addressed is to enable rapid development of IoTSP applications with minimal effort by various stake holders[1] involved in the application development process. To address above issues, several challenges have already been addressed in the closely related fields of the Wireless Sensor Networks (WSNs) [9, p. 65]and pervasive computing [9, p. 65] to the modern day of IoTSP. The goal of our work [7] is to enable the development of applications for such complex systems. In the following, we discuss one of such application.

1.1 Application example: Smart Home

To illustrate characteristics of IoTSP, we consider smart home applications. A home consists of several rooms, each one is instrumented with several heterogeneous entities (physical devices) for providing residents' comfort, safety, and optimizing resources. Many applications can be developed on top of these devices, one of which we discuss below:

To accommodate a resident's preference in a room, a database is used to keep the profile of each resident, including his/her preferred temperature level. An RFID reader in the room detects the resident's entry and queries the database service. Based on this, the thresholds used by the room devices are updated. To ensure the safety of residents, a fire detection application is installed. It 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 and their neighbors are informed through a set of alarms. Moreover, the system generates the current environment status on dashboard (e.g., humidity, temperature, outside temperature by interacting with external web services) for the situation awareness.

1.2 IoTSP application development challenges

The development of application (as discussed in the previous section) is difficult because IoTSP exhibit the following challenges: *Heterogeneous entities*: An IoTSP may execute on a network consisting of different types of entities. For example, a home consists of entities encompassing, *sensors* (e.g., temperature sensor), *tags* (e.g., BadgeReader to read a user's badge), *actuators* (e.g.,

[1] We use the term **stakeholders** as used in software engineering to mean- people, who are involved in the application development. Examples of stakeholders are software engineer, developer, domain expert, technologist etc.

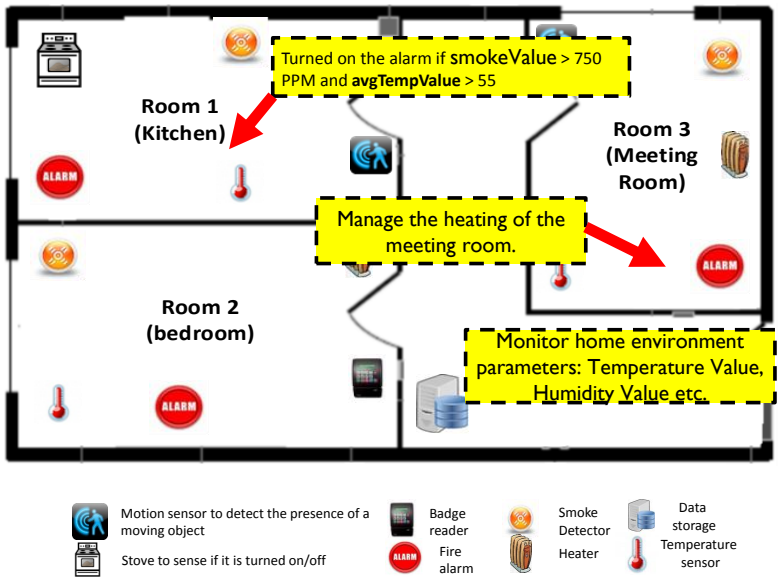

Figure 1: A smart home with deployed devices with (1) Temperature sensor, (2) Heater, (3) Fire alarm, (4) Smoke Detector, (5) Badge reader, (6) Data storage, (7) Motion sensor, and (8) Smoke sensor.

heater), *user interfaces* (e.g., user receives notification in case of fire), *storage* (e.g., profile storage to store users' data), and elements of the "traditional" Internet such as Web and database servers, exposing their functionality as *Web services* (e.g., Yahoo Weather service).

Heterogeneous interaction modes: The above mentioned entities exhibit different interaction modes such as *event-driven* (e.g., a smoke sensor fires when a smoke is detected), *request-response* (e.g., profile data is retrieved through request and response), *periodic* (e.g., temperature sensors sense data periodically), *command* [1] (e.g., a heater is commanded to regulate the temperature), and *notify* [2](e.g., a user is notified when the fire is detected).

Heterogeneous platforms: Unlike WSN, an IoTSP application may involve entities running on different platforms. For instance, a temperature sensor may be attached with Raspberry PI, smoke sensor runs on a resource constrained micro-controller with no OS (e.g., Arduino), an end-user application is deployed on Android Mobile OS,and a dashboard is implemented using JavaScript and HTML.

Different life cycle phases: Given the heterogeneity discussed above, Stakeholders have to address issues that are attributed to different life cycles [8], including *design*, *implementation*, and *deployment*. At the design phase, the application logic has to be analyzed and separated into a set of distributed tasks for the underlying network consisting of heterogeneous entities. Then, these tasks have to be implemented for a specific platform of a device. At the deployment phase, the application logic has to be deployed onto a network of devices. Manual effort in all above three phases for heterogeneous devices is a time-consuming and error-prone process.

Approach	Description	Examples	Benefits	Limitations
General Purpose Lang.	Developers think in terms of activities of individual devices & explicitly encode interactions with others in programming language.	Node.js, C, Python, Android	Development of efficient systems based on complete control over device.	• More development effort • Difficult to reuse & platform-dependent design.
WSN macro prog.	It provides an ability to specify an application at a global level rather than individual nodes.	Regiment [6], MacroLab	• Flexibility to write custom application logic. • Reduce development effort.	Largely targets for similar types of devices.
Cloud Platforms	• Devices are connected to cloud platforms through APIs or high-level (e.g., drag-and-drop) constructs. • Data from connected devices are collected or shared for their own applications. • they provide flexibility to write custom application logic in GPL such as JavaScript.	Node-RED, WoTKit [3]	• Reduce development efforts compared to GPLs. • Offers ease in application deployment and evolution.	• Platform-dependent design. • Scarifies direct node-to-node communication. • Restricts developers interms of functionality. • May not suitable for some critical applications.
Model-driven Dev.	It separates different concerns of a system at a certain level of abstractions and provides transformation engines to convert them to target platforms.	DiaSuite [4] , PervML [8]	Re-usable, Platform-indepe- ndent, Extensible design.	Long development time to build a MDD system.

Table 1: Summary: existing approaches available for programming IoTSP application.

1.3 State of the art: Programming Internet of Things, Service and People

To address the above mentioned challenges, several approaches have been proposed in the closely related fields of WSN, Pervasive Computing, and Software Engineering. These approaches are summarized in Table 1.

Currently, development of IoTSP 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 interactions with other devices. For example, they write a program that reads data from appropriate sensor devices, aggregates data pertaining to the some external events, decides where to send it and communicates with actuators if needed. For instance, stakeholders use **General-purpose Programming Languages** (GPLs) such as C, JavaScript, Android and target a particular middle-ware API to develop an application. The key advantage of such approach is that it allows the development of extremely efficient systems based on the complete control over individual devices. However, it is unwieldy for IoTSP application due to the inherent heterogeneity of systems.

To reduce development effort, an alternative approach is **WSN macro-programming**. It provides abstractions to specify high-level collaborative behaviors, while hiding low-level details such as message passing or state maintenance from stakeholders. Macro-programming is a viable approach compared to the general-purpose programming approaches. However, most of macro-programming systems largely aims to systems with similar types of nodes. Therefore, given heterogeneity this approach may not be viable. To address the heterogeneity issue and improve development effort over GPLs, the **Cloud-based platforms** have emerged. It reduces development efforts by providing cloud-based APIs to implement common functionality. As a secondary advantage, because of the application logic is centrally located in a cloud platform, it offers the ease deployment and evolution. However, some limitations of this approach are: first, stakeholders have to write code to implement

low-level details such as reading values from sensors and actuating actuators. This again leads to a platform-dependent design. Second, it restricts developers in-terms of functionality such as "in-network" aggregation or direct node-to-node communication locally. To address development effort and platform-dependent design issues, **Model-driven development** (MDD) [5] has been proposed. Several MDD frameworks [4, 8] have been proposed for programming Internet of Things. However, existing approaches only cover limited subsets of challenges when applied to the IoTSP such as support for development life-cycles and usual behaviors found in modern IoTSP – such as remote access to Web services, interaction by a human user with software, and database access. By following the MDD guidelines, many benefits can be achieved. For instance, by separating different concerns of a system at a certain level of abstractions and by providing transformation engines to convert these abstractions to target platforms lead to platform-independent design and improve productivity (*e.g.*, re-usability, extensibility) in the application development process.

1.4 Contributions

In view of the above, we have built upon our existing MDD development framework [7] and evolved it into a framework for IoTSP. The proposed development framework segregates IoTSP development concerns, provides a set of modeling languages to specify them and integrates automation techniques to parse these modeling languages. we present an enhanced and extended version of modeling languages and their integration into a development framework: (1) Domain language (DL) that models IoTSP characteristics such as describing heterogeneous entities such as tags, external third-party services, and different types of sensors. (2) Architecture Language (AL) to describe functionality of an application, (3) User Interaction Language (UIL) to model interaction between an application and a user, and (4) Deployment Language (DL) to describe deployment-specific features consisting information about a physical environment where devices are deployed.

We present a comparative evaluation results with existing approaches. This further provides the IoTSP community for benchmarking. The evaluation is carried out on *real devices* exhibiting IoTSP heterogeneity. The evaluation shows that the use of modeling languages reduces user-written lines of code by more than 70% in comparison to an implementation in GPLs and 30% compared to cloud-based approaches.

2 Different development concerns

This section presents our development framework that separates IoTSP application development into different concerns, namely *domain*, *platform*, *functional*, and *deployment*. It integrates a set of high-level modeling languages to specify such concerns. These languages are supported by automation techniques at various phases of application development process. Stakeholders carry out the following steps in order to develop a IoTSP application using our approach.

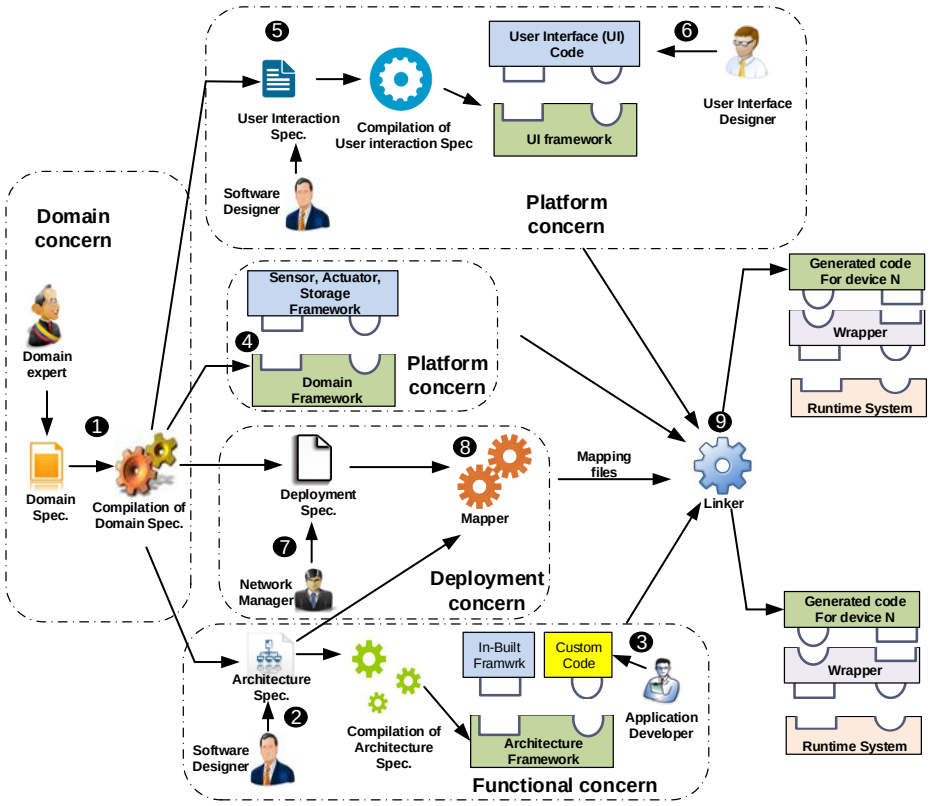

Figure 2: IoTSP application development: the overall process

2.1 Domain Concern

This concern is related to concepts that are specific to a domain (e.g., building automation, transport) of an IoTSP. The stakeholders task regarding such concern consists of the following step:

Specifying and compiling domain specification. The domain expert specifies a domain specification using the Domain Language (DL) (Step ① in Figure 2). The domain specification includes specification of resources, which are responsible for interacting with Entities of Interest (EoI). This includes *tags* (identify EoI), *sensors* (sense EoI), *actuators* (affect EoI), and *storage* (store information about EoI). In the domain specification, resources are specified in a high-level manner to abstract low-level details from the domain expert (detail in Section 3).

2.2 Functional Concern

This concern is related to concepts that are specific to functionality of an IoTSP. An example of a functionality is to open a window when an average temperature value of a room is greater than $30°C$. The stakeholders task regarding such concern consists of the following steps:

Specifying and compiling application architecture. Referring the domain specification, the software designer specifies an application architecture using the Architecture Language (AL)(Step ② in Figure 2). It consists of specification of computational services and interaction among them (detail in Section 4). A computational service is fueled by sensors and storage defined in the domain specification. They process inputs data and take appropriate decisions by triggering actuators defined in the domain specification. The architecture specification consists of two types of computational services: (1) *common* specifies common operations such as average, count, sum in the application logic, (2) *custom* specifies an application-specific logic (for instance, coordinating events from BadgeReader with the content of Profile data storage).

2.3 Platform Concern

This concern specifies the concepts that fall into computer programs that act as a translator between a hardware device and an application. The stakeholders task regarding such concern consists of the following steps:

Generating device drivers. The compilation of domain specification generates a domain framework (Step ④ in Figure 2). It contains *concrete classes* corresponding to concepts defined in the domain specification. The concrete classes contain concrete methods to interact with other software components and platform-specific device drivers, described in our work [7, p. 75]. We have integrated existing open-source sensing framework[2] for Android devices. Moreover, we have implemented sensing and actuating framework for Raspberry Pi and storage framework for MongoDB, MySQL, and Microsoft AzureDB. So, the device developers do not have to implement platform-specific sensor, actuator, and storage code.

Specifying user interactions. To define user interactions, we present a set of *abstract interactors*, similar to work [2], that denotes information exchange between an application and a user. The software designer specifies user interactions using User Interaction Language (UIL) (Step ⑤ in Figure 2). The UIL provides three abstract interactors: (1) *command* denotes information flow from a user to an application (e.g., controlling a heater according to a temperature preference), (2) *notify* denotes information flow from an application to a user (e.g., fire notification in case of emergency), (3) *request* denotes information flows round-trip between an application and a user, initiated from the user (e.g., requesting preference information to a database server). More detail is in Section 5. The Implementation of user-interaction code is divided into two parts: (1) compilation of user-interaction specification, and (2) Writing user-interaction code. These two steps are described below:

- **Compilation of user interaction spec**: Leveraging the user interaction specification, the development framework generates a User Interface (UI) framework to aid the user interface designer (step ⑥ in Figure 2). The UI framework contains a set of *interfaces* and *concrete classes* corresponding to resources defined in the user interaction specification. The concrete

[2] `http://www.funf.org/`

classes contain concrete methods for interacting with other software components. For instance, the compilation of *command* interactors generates `sendCommand()` method to send command to other component. Similar way, the compilation of *notify* interaction generates `notifyReceived()` method to receive notifications.

- **Writing user-interaction code**: Leveraging the UI framework, the user interface designer implements *interfaces*. These interfaces implements code that connects appropriate UI elements and concrete methods. For instance, a user initiates a command to heater by pressing UI elements such as `button` and `sendCommand()` method, or the application notifies a temperature value on `textlabel` through `notifyReceived()` method.

2.4 Deployment Concern

This concern is related to deployment-specific concepts that describe the information about a device and its properties placed in the target deployment. It consists of the following steps:

Specifying target deployment. Referring the domain specification, the network manager describes a deployment specification using the Deployment Language (DL) (Step ⑦ in Figure 2). The deployment specification includes the details of each device, resources hosted by each devices, and the type of device. Ideally, the IoTSP application can be deployed on different deployments. 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 a set of devices defined in the deployment specification and a set of computation components defined in the architecture specification(Step ⑧ in Figure 2). The mapper devices devices where each computational services will be deployed. The current version of mapper algorithm [7] selects devices randomly and allocates computational services to the selected devices.

2.5 Linking

The linker combines the code generated by various stages and creates packages that can be deployed on devices (Step ⑨ in Figure 2). It merges generated architecture framework, UI framework, domain framework, and mapping files. 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.

The final output of linker is composed of three parts: (1) a *runtime-system* runs on each individual device and provides a support for executing distributed tasks, (2) a *device-specific code* generated by the linker module, and (3) a *wrapper* separates generated code from the linker module and underlying runtime system by implementing interfaces.

3 Specify Domain

The Domain Language (DL) offers high-level constructs to specify the domain-specific concepts. We describe these constructs as follows:

Sensors A set of sensors is declared using the `sensors` keyword (Listing 1, line 6). Each sensor produces one or more sensor measurements along with the data-types specified in the data structure (Listing 1, lines 2-4), declared using the `generate` keyword (Listing 1, line 9). We categorize sensors into the following three types:

- **Periodic sensor:** It samples results every `d` seconds for a duration of `k` seconds. For instance, a temperature sensor generates a temperature measurement of `TempStruct` type (Listing 1, lines 2-4). It samples data every `1` second for next `6` minutes (Listing 1, line 10).
- **Event-driven sensor:** It produces data when the event condition is met. For instance, a smoke sensor generates `smokeMeasurement` when `smokeValue > 650 PPM` (Listing 1, lines 13-14).
- **Request-based sensor:** It responds its results only if it is requested. For instance, the `YahooWeatherService` provides temperature value of a location given by a `locationID` (Listing 1, lines 15-18).

Tag It is a physical object that can be applied to or incorporated into Entities of Interest for the purpose of identification. It is read through a reader. For instance, a `BadgeReader` read an user's badge and generates `badgeDetectedStruct` measurement (Listing 1, lines 20-21).

Actuators A set of actuators is declared using the `actuators` keyword (Listing 1, line 22). Each actuator has one or more actions declared using the `action` keyword. An action may take inputs specified as parameters (Listing 1, line 25). For instance, a heater may have two actions (e.g., switch off, set heater), illustrated in Listing 1, lines 23-25.

Storage A set of storage is declared using the `storages` keyword (Listing 1, line 26). A retrieval from the storage requires a parameter, specified using the `accessed-by` keyword (Listing 1, line 28). The data insertion into the storage is performed by the `action` keyword with parameter. For instance, a user's profile is accessed from storage by a `badgeID` and inserted by invoking an action (Listing 1, line 29).

Listing 1: Code snippet of domain spec.

```
1  structs:
2  TempStruct
3      tempValue : double;
4      unitOfMeasurement : String;
5  resources:
6  sensors:
7      periodicSensors:
8       TemperatureSensor
9          generate tempMeasurement : TempStruct;
10          sample period 1000 for 6000000;
11     eventDrivenSensors:
12      SmokeDetector
```

```
13          generate smokeMeasurement: SmokeStruct;
14          onCondition smokeValue>650PPM;
15      requestBasedSensors:
16          YahooWeatherService
17              generate weatherMeasurement: TempStruct accessed-by
18                                              locationID: String;
19  tags:
20      BadgeReader
21          generate badgeDetectedStruct: BadgeStruct;
22  actuators:
23      Heater
24          action Off();
25          action SetTemp(setTemp:TempStruct);
26  storages:
27      ProfileDB
28          generate profile: TempStruct accessed-by badgeID: String;
29          action InsertProfileData(profileData:ProfileStruct);
```

4 Specify Architecture

Referring the concepts defined in the domain specification, the software designer specifies an architecture specification. It is described as a set of computational services. It consists of two types of computational services: (1) *Common* components specify common operations (e.g., `average`, `count`, `sum`) in the application logic. For instance, `RoomAvgTemp` component consumes 5 temperature measurements (Listing 2, line 4), apply average by sample operation (Listing 2, line 5), and generates room average temperature measurements (Listing 2, line 6). (2) *Custom* specifies an application-specific logic. For instance, `Proximity` component is a custom component (in Figure 3) that coordinates events from `BadgeReader` with the content from `ProfileDB`.

Each computational service is described by a set of inputs and outputs. We describe them below:

Consume and Generate. They represent a set of subscriptions (or consume) and publications (or generate) expressed by a computational service. For instance, `RoomAvgTemp` consumes `tempMeasurement` (Listing 2, line 4), calculates an average temperature (Listing 2, line 5) and generates `roomAvgTempMeasurement` (Listing 2, line 6).

Request. It is a set of requests issued by a computational service to retrieve data. For instance, to access user's profile, `Proximity` (Listing 2, line 10) sends a request message containing profile information as an access parameter to a storage `ProfileDB` (Listing 1, lines 27-29).

Command. It is a set of commands, issued by a computational service to trigger actions. The software designer can pass arguments to a command depend on action signature. For instance, the `RoomController` issues a `SetTemp` command(Listing 2, line 15) with a `settemp` as an argument to `Heater` (Listing 1, lines 23-25).

Figure 3 shows a layered architecture of the application discussed in Section 1.1. Listing 2 describes a part of Figure 3. It revolves around the actions of the `Proximity` service (Listing 2, lines 8-11), 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, and one for requesting the user's temperature profile from `ProfileDB`. The output of the `Proximity` and `RoomAvgTemp` are consumed by the `RoomController` service (Listing 2, lines 13-14). This service is responsible for taking decisions that are carried out by invoking command (Listing 2, line 15).

Listing 2: A code snippet of architecture spec.

```
1  computationalServices:
2  Common:
3          RoomAvgTemp
4              consume tempMeasurement from TemperatureSensor;
5              COMPUTE(AVG_BY_SAMPLE,5);
6              generate roomAvgTempMeasurement:TempStruct;
7  Custom:
8          Proximity
9              consume badgeDetected from BadgeReader;
10             request profile(badgeID) to ProfileDB;
11             generate tempPref: UserTempPrefStruct;
12   RoomController
13             consume roomAvgTempMeasurement from RoomAvgTemp;
14             consume tempPref from Proximity;
```

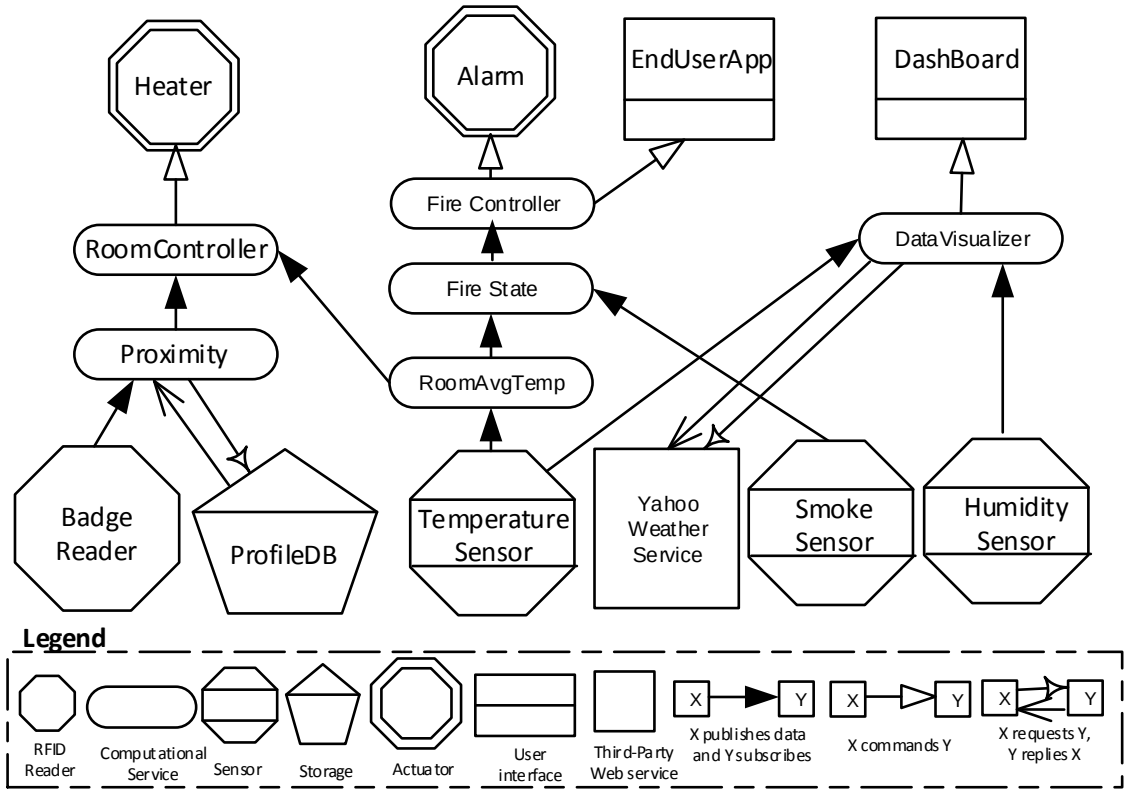

Figure 3: Dataflow of Smart Home Application.

```
15          command SetTemp(setTemp) to Heater;
```

5 Specify User-interactions

The user interactions specification defines *what* interactions are required by an application. We design a set of abstract *interactors* that denotes data exchange between an application and a user. The following are abstract interactors that are specified using User Interaction Language (UIL).

Command It denotes information flow from a user to an application. It is declared using the `command` keyword (Listing 3, lines 8-9). For instance, a user can control an actuator by triggering a `Off()` command (Listing 3, line 8). Command could be parametrized too. For instance, a user can set a temperature of heater by triggering a `SetTemp()` command (Listing 3, line 9).

Notify It denotes information flow from an application to a user. For instance, an application notifies a user in case of fire. It is declared using the `notify` keyword (Listing 3, lines 11-12). The application notifies users with the fire information specified in the data structure (Listing 3, lines 2-4).

Request It denotes information flow round-trip between an application and a user, but initiated from the user. For instance, a user can retrieve data by requesting to data storage. This is declared using the `request` keyword. The user requests `ProfileDB` for data (Listing 3, line 10) and the `ProfileDB` responses back with data to the user.

Listing 3: Code snippet of user interaction spec.

```
1  structs:
2  FireStateStruct
3   fireValue:String;
4   timeStamp:String;
5  resources :
6  userInteractions:
7   EndUserApp
8    command Off() to Heater;
9    command SetTemp(setTemp) to Heater;
10    request profile to ProfileDB;
11    notify FireNotify(fireNotify:FireStateStruct)
12                                        from FireController;
13   DashBoard
14    notify DisplaySensorMeasurement(sensorMeasurement:VisualizeStruct)
15                                        from DisplayController;
```

6 Specify Deployment

The deployment specification describes a device and its properties in a target deployment. It includes properties such as *location* that defines where a device is deployed, *resource* defines component(s) to be deployed on a device, *language-platform* is used to generate an appropriate package for a device, *protocol* specifies a run-time system installed on a device to interact with other devices. Listing 4 shows a small code snippet to illustrate these concepts. `TemperatureMgmt-Device-1` is located in room#1 (line 4), `TemperatureSensor` and `Heater` are attached with the device (line 5) and the device driver code for these two components is in NodeJS (line 6), MQTT runtime system is installed on `TemperatureMgmt-Device-1` device (line 7).

A storage device contains the **database** field that specifies the installed database. This field is used to select an appropriate storage driver. For instance, `DatabaseSrv-Device-2` (lines 8-10) runs `ProfileDB` component implemented in `MySQL` database.

When an application is deployed, the network manger decides what user interaction components need to be deployed on devices. The announcement of user interaction components in the deployment specification is now an integral part of the specification. This announcement is used to deploy the generated UI framework and UI code (Step ⑤ in Figure 2) on a device. Listing 4 illustrates a code snippet to describes this concept. `SmartPhone-Device-3` announces that `EndUserApp` (specified in user interaction specification) should be deployed on `Android` device (Lines 11-13).

Listing 4: Code snippet of deployment spec.

```
1  devices:
2      TemperatureMgmt-Device-1:
3          location:
4              Room: 1;
5          resources: TemperatureSensor, Heater;
6          language-platform: NodeJS;
7          protocol : mqtt;
8      DatabaseSrv-Device-2:
9          resources: ProfileDB;
10         database: MySQL;
11     SmartPhone-Device-3:
12         resources: EndUserApp;
13         language-platform: Android;
14         ...
```

7 Application development using IoTSuite

This section describes each step of Internet of Things, Service, and People (IoTSP) application development process using IoTSuite. Application development using IoTSuite focuses on design, implement, and deployment phases to develop IoTSP applications.

We take an example of smart home application (Refer Figure 3) to demonstrate application development using IoTSuite.

7.1 Create a IoTSuite Project

To develop a smart home application (Refer Figure 3) using IoTSuite, developers carry out the following steps.

7.1.1 Download IoTSuite-Eclipse-Plugin

- Download IoTSuite-Eclipse-Plugin [3].
- Go to Downloads folder.
- Extract downloaded IoTSuite-Eclipse-Plugin into C:\ drive.
- Go to C:\IoTSuite-Eclipse-Plugin\Template.
- Extract IoTSuite-TemplateV9-master.rar in to same directory (C:\IoTSuite-Eclipse-Plugin\Template).

7.2 Open IoTSuite Eclipse

- Go to C:\IoTSuite-Eclipse-Plugin.
- Right click on IoTSuite-Eclipse.exe as shown in Figure 4 (Step 1), and select Open option (Step 2).
- When IoTSuite-Eclipse is open, usually developers have to select workspace but here developers don't need to go to C:\IoTSuite-Eclipse-Plugin\Template in order to select workspace, it is already provided as default workspace.
- Just check on "Use this as the default and do not ask again" (Step 1) in and Click on OK button (Step 2) (Refer Figure 5).

[3] IoTSuite Eclipse Plugin is available at `https://codebits.pl.abb.com/saurabh-internshhip-work`

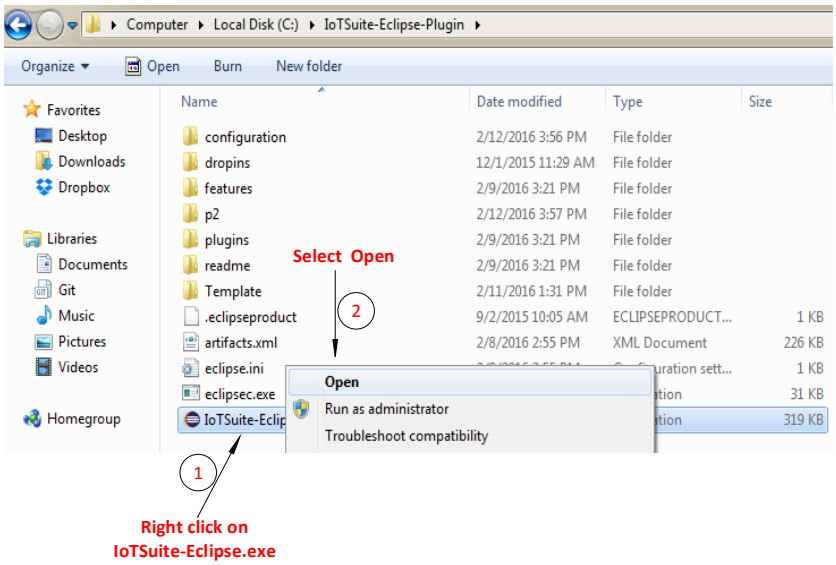

Figure 4: Open IoTSuite Eclipse

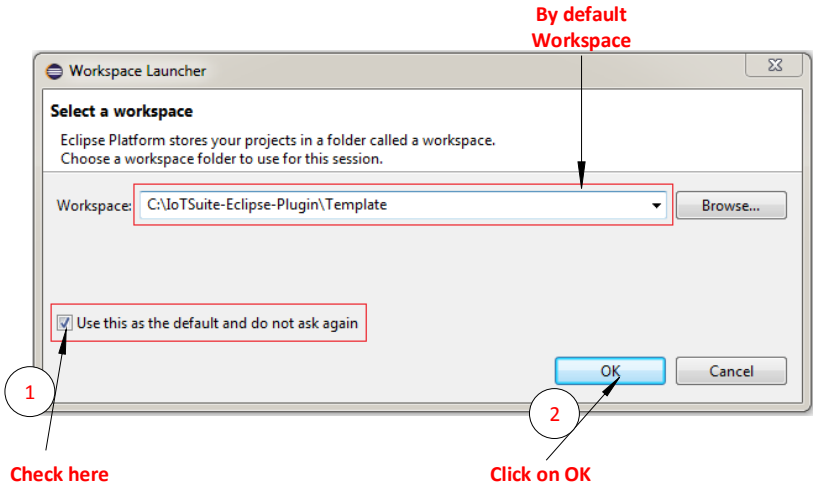

Figure 5: Select default Workspace

7.3 Create a new IoTSuite Project

- To create a new IoTSuite project using IoTSuite-Eclipse, click **File>New>Project..** , choose IoTSuite Project from the list (Refer Figure 6).

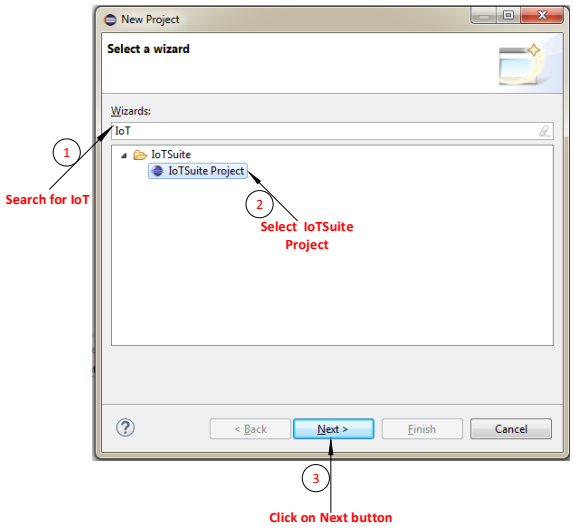

Figure 6: IoTSuite Project Creation Wizard (1)

- In the next screen, write the project name as IoTSuiteSpecification (Step 1), check Use default location (Step 2), and click on Finish button (Step 3) (Refer Figure 7).

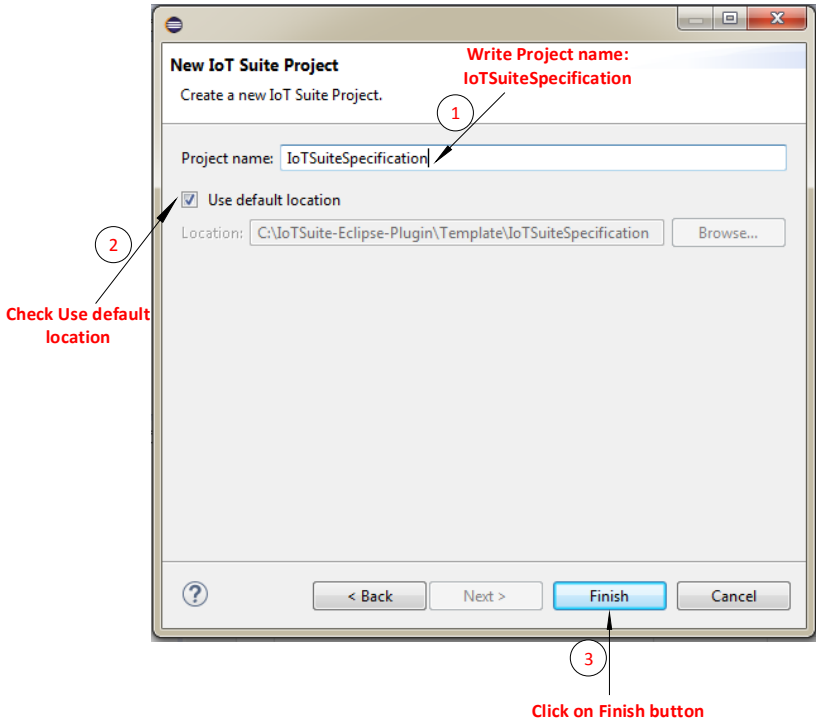

Figure 7: IoTSuite Project Creation Wizard (2)

As a result, new IoTSuite project is created, and the corresponding IoTSuite specification files will be opened in Xtext mode. By default, they have predefined contents in order to guide developers. The structure of created project is shown in Figure 8. Once the IoTSuiteProject is created, next step is to specify high-level specification (Refer Section 7.3.1).

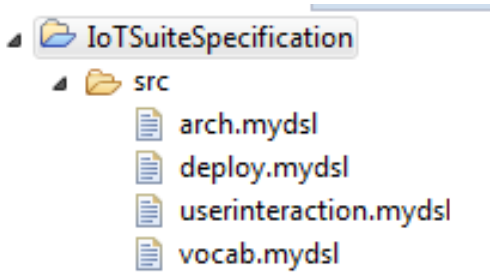

Figure 8: IoTSuite Project Structure

7.3.1 Specifying high-level specifications

- Specifying high-level specification using editor: To write these specifications, we present Xtext for a full fledged editor support with features such as syntax coloring, error checking, auto completion, rename re-factoring, outline view and code folding.
- We have implemented Outline/ Structure view feature which is displayed on top most right side of the screen (Refer Figure 9). It displays an outline of a file highlighting its structure. This is useful for quick navigation.
- As shown in Figure 9, in your vocab.mydsl file if you have a large number of structures, sensors, and actuators, than from outline by just clicking on particular structures (e.g. TempStruct) it navigates to TempStruct definition in the vocab.mydsl. So, developers don't need to lookup in entire file.
- Using syntax coloring feature, keywords are appeared in colored text (Refer Figure 10).
- Using code folding, developer can collapse parts of a file that are not important for current task. In order to implement code folding, click on dashed sign located in left most side of the editor. When developer clicked on dashed sign, it will fold code and sign is converted to plus. In order to unfold code, click again on plus sign. As shown in Figure 11, we have fold code for TempStruct, HumidityStruct, and BadgeStruct.
- The error checking feature guides developer if any error is there. General error in the file is marked automatically e.g., violation of the specified syntax or reference to undefined elements. Error checking indicates if anything is missing/wrong in particular specification file where Auto Completion helps developer to complete entire word/statement.
- Auto completion is used to speed up writing text in specification files. In order to use auto completion feature, developer need to press ctrl+space key at current cursor position, so it will

Figure 9: IoTSuite editor feature: Outline view

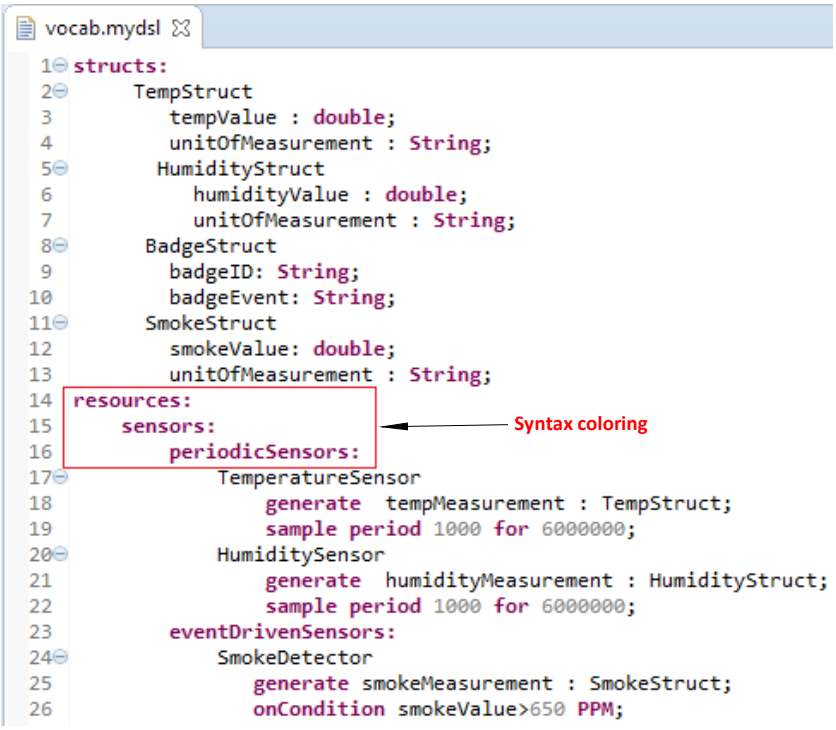

Figure 10: IoTSuite editor feature: Syntax coloring

provide suggestion. Here (Refer Figure 12), in TemperatureSensor definition if we write T and press ctrl+space than it will suggest us to write TempStruct.

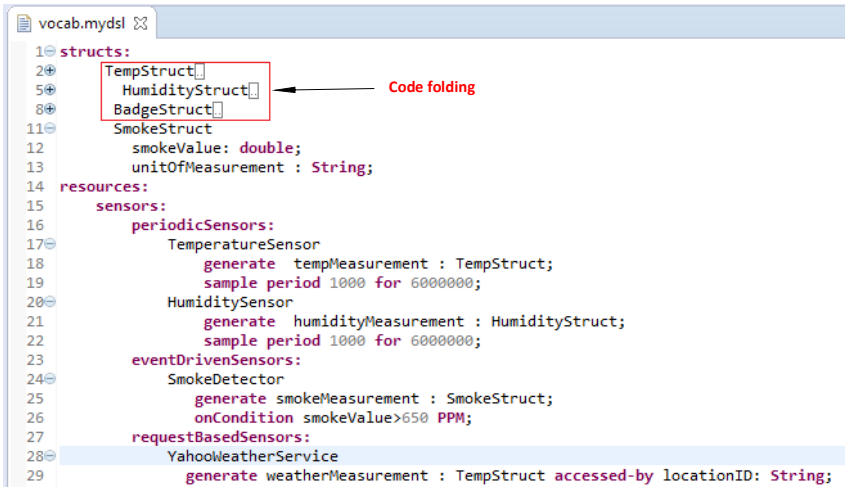

Figure 11: IoTSuite editor feature: Code folding

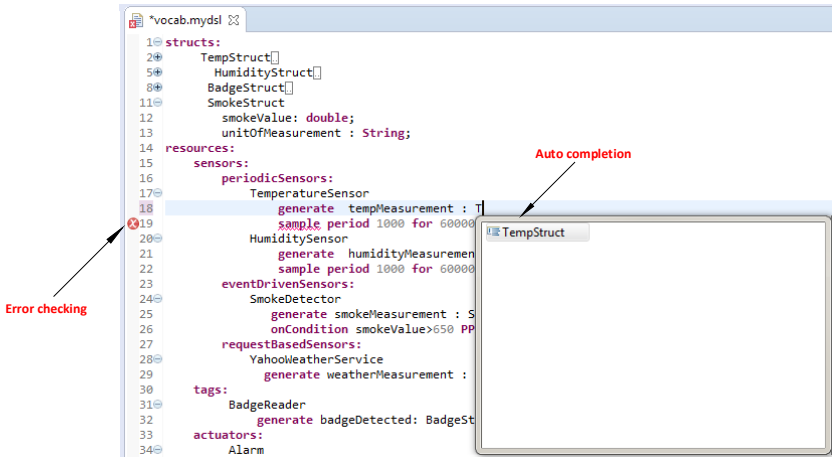

Figure 12: IoTSuite editor features: Error checking & Auto completion

– *Specifying domain specification*
Developer specifies domain specification using vocab.mydsl file. To do this, double click on *vocab.mydsl* (Step 1) and write domain specification using IoTSuite editor (Step 2) as shown in Figure 13.

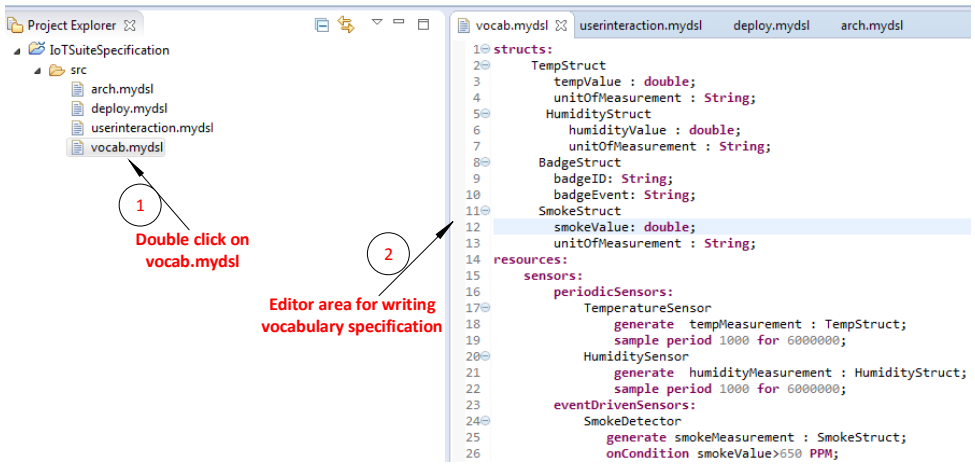

Figure 13: Domain Specification

– *Specifying arch specification*
Now, developer specifies architecture specification using arch.mydsl. To specify architecture specification, double click on *arch.mydsl* (Step 1) and write architecture specification using IoTSuite editor (Step 2) as shown in Figure 14.

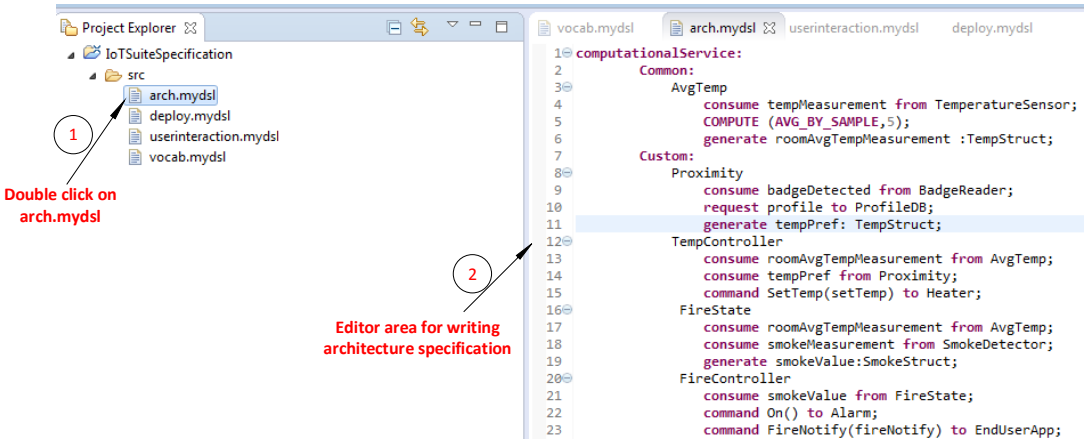

Figure 14: Architecture Specification

– *Specifying user-interaction specification* To specify user-interaction specification, developer double click on *userinteraction.mydsl*, and write user-interaction specification using IoTSuite editor (Step 2) as shown in Figure 15.

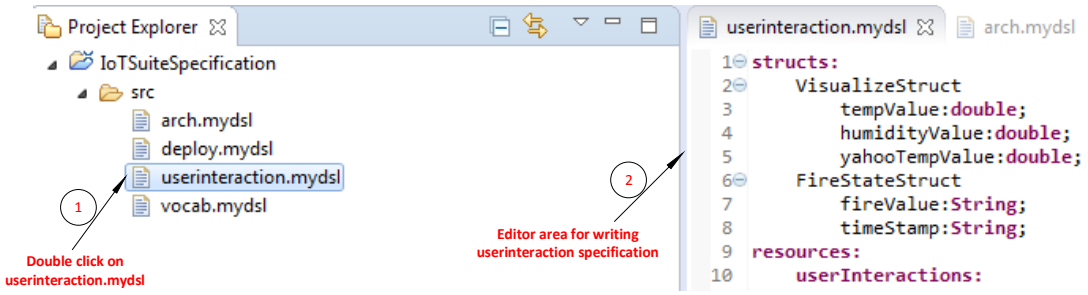

Figure 15: User-Interaction Specification

– *Specifying deployment specification*
Developer specifies deployment specification using deploy.mydsl. To do this, double click on *deploy.mydsl* and write deployment specification using IoTSuite editor (Step 2) as shown in Figure 16.

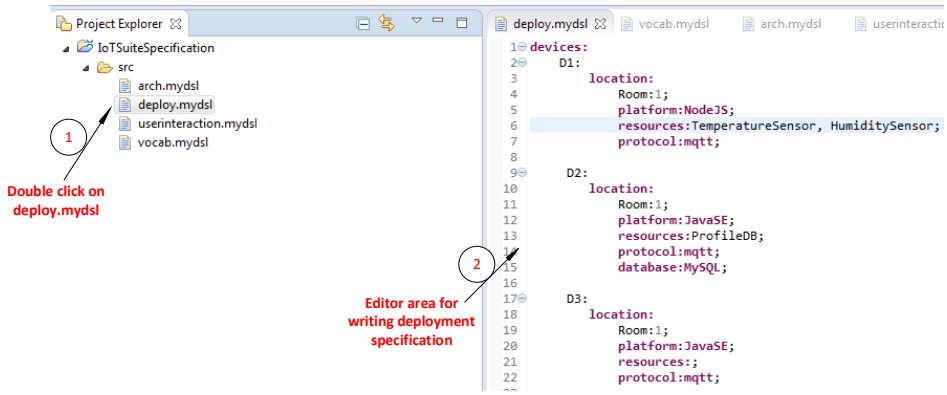

Figure 16: Deployment Specification

7.4 Compilation of an IoTSuite Project

Now, we need to compile the high-level specification written by developer in Section 7.3.1. To do so, we generate programming framework, by performing following Steps:

7.4.1 Compilation of high-level specification

- *Compilation of vocab specification*- Right click on vocab.mydsl file (Step 1) and selecting "Compile Vocab" (Step 2) (Refer Figure 17) generates a vocabulary framework.

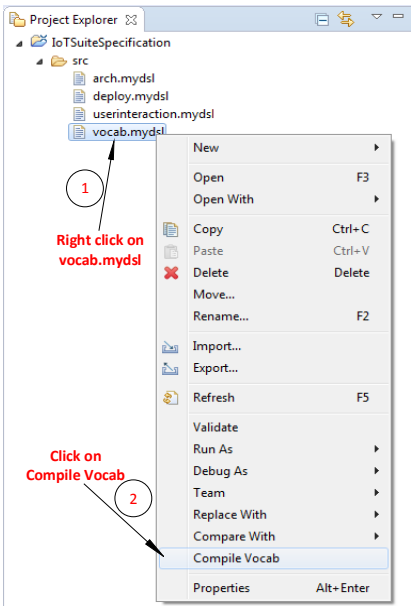

Figure 17: Compilation of Vocab specification

- *Compilation of architecture specification*- Right click on arch.mydsl file (Step 1) and selecting "Compile Arch" (Step 2) (Refer Figure 18) generates an architecture framework.

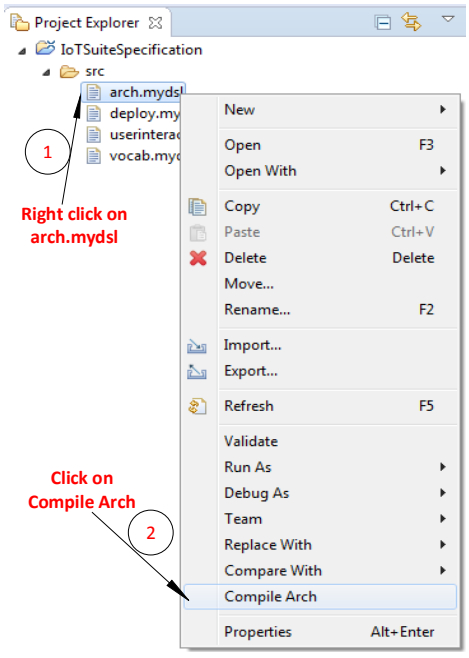

Figure 18: Compilation of Architecture specification

- *Import application logic package*- To import application logic package, click on File Menu (Step 1), and select Import option (Step 2) as shown in Figure 19.

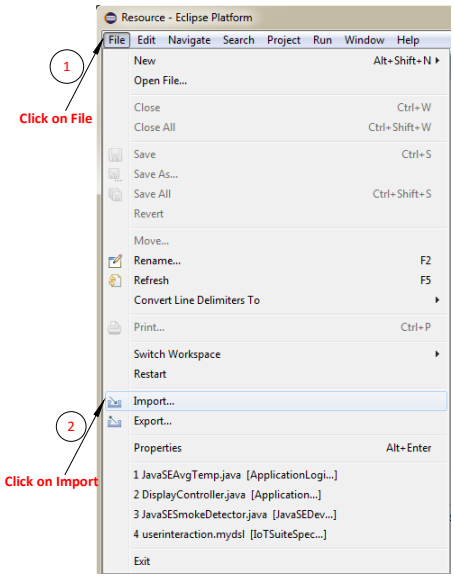

Figure 19: Import application logic package

- *Locate application logic package*- To locate application logic package, browse to Template path (Step 1), select application logic package (Step 2), and click on Finish button (Step 3) as shown in Figure 20.

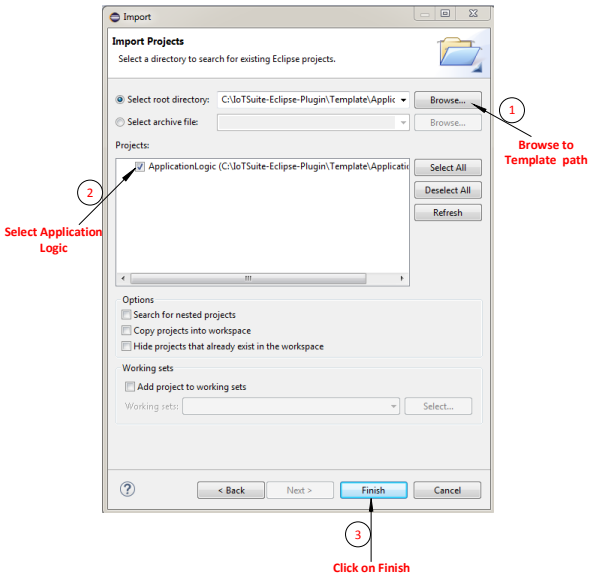

Figure 20: Locate application logic package

- *Implementing application logic*- To implement application logic refer [7, p. 12-13].

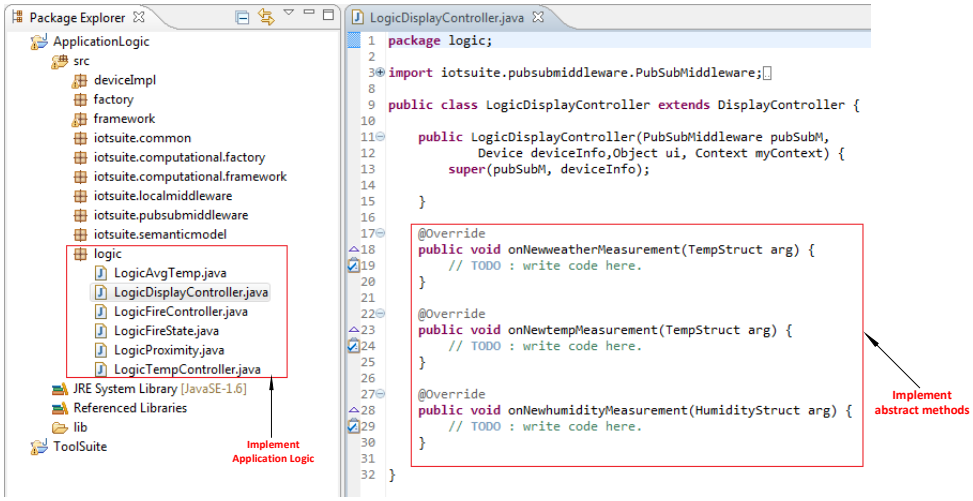

Figure 21: Implement application logic package

- *Compilation of user-interaction specification*- Right click on userinteraction.mydsl file and selecting "Compile UserInteraction" (Refer Figure 22) generates a User Interaction (UI) framework.

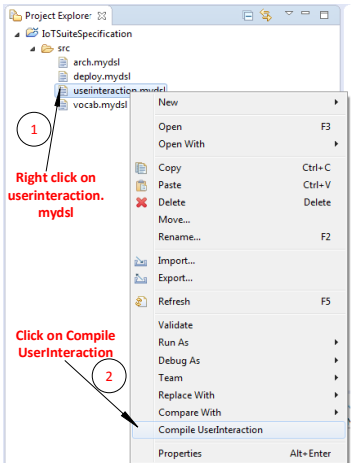

Figure 22: Compilation of User-Interaction specification

- *Compilation of deployment specification*- Right click on deploy.mydsl file and selecting "Compile Deploy" (Refer Figure 23) generates a deployment packages.

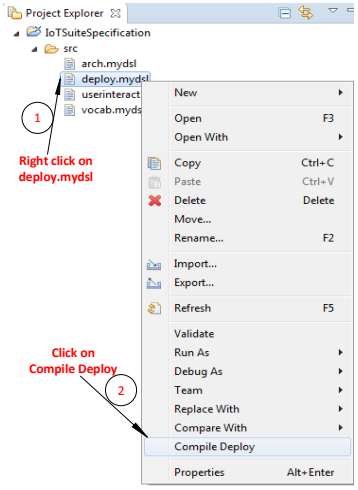

Figure 23: Compilation of Deployment specification

- *Import user-interface project* - To import user-interface project, click on File Menu (Step 1), and select Import option (Step 2) as shown in Figure 24.

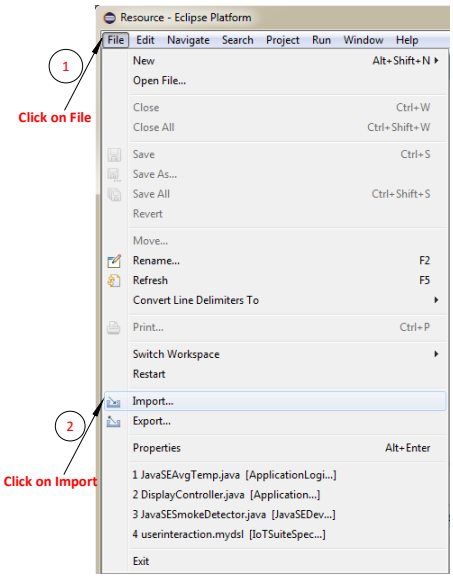

Figure 24: Import user-interface project

- *Locate user-interface project*- To locate user-interface project, browse to CodeForDeployment folder in Template path (Step 1), select project specified in the use-interaction specification (Step 2), and click on Finish button (Step 3) as shown in Figure 25.

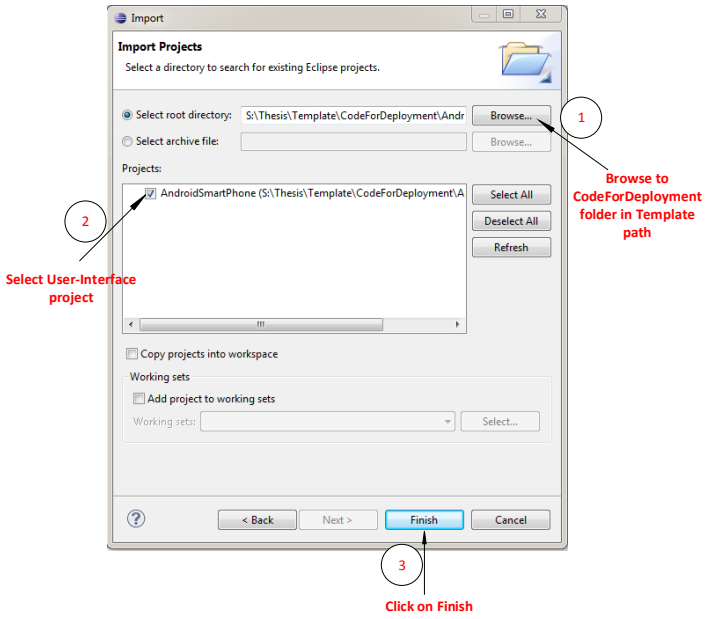

Figure 25: Locate user-interface project

- *Implementing user-interface project*- To implement user-interface project refer [7, p. 15-16].

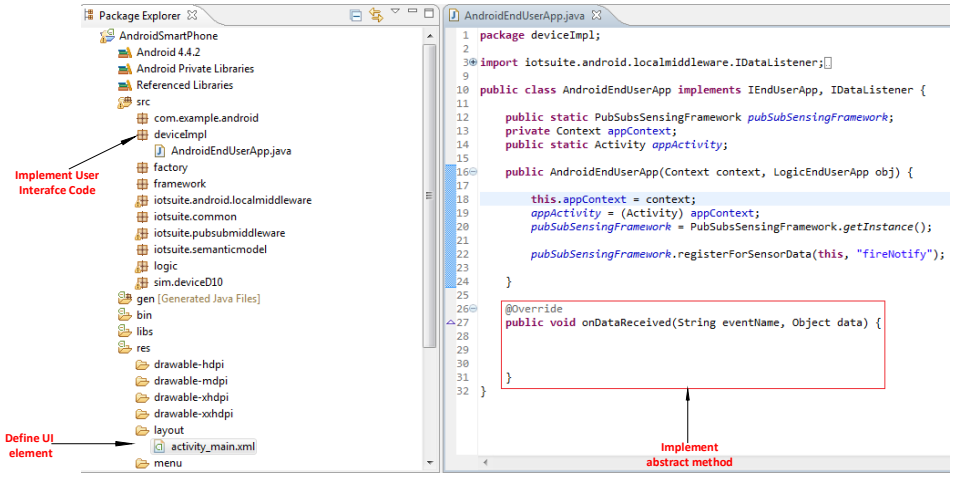

Figure 26: Implement user-interface project

7.5 Deployment of generated packages

- The output of previous Step (compilation of deployment specification) produce a set of platform specific project/packages (Figure 27) for devices, specified in the deployment specification (Refer Figure 16). These projects compiled by device-specific compiler designed for the target platform. The generated packages integrates the run-time system, it enables a distributed execution of IoTSP applications.

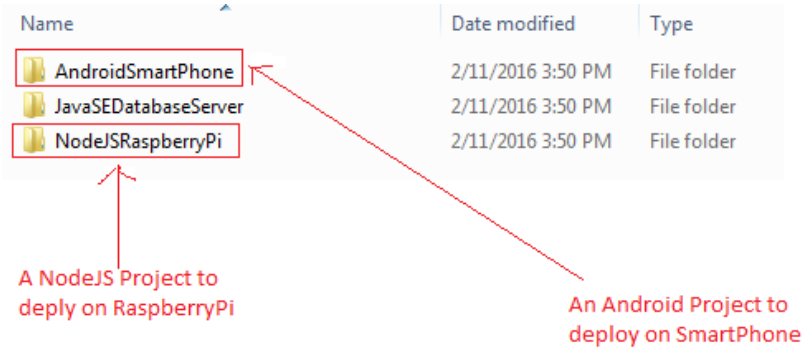

Figure 27: Packages for target devices specified in the deployment specification

8 Evaluation

The goal of this section is to describe how well the proposed approach eases the IoTSP application development compared to the existing approaches (discussed in Section 1.3). We implemented the smart home application (discussed in Section 1.1) using existing approaches.

Metrics for evaluation To evaluate the proposed approach, we employed the *lines of code* (LoC) as metrics to measure the development effort. We are aware that LoC is not a precise metrics and it depends on programming languages, styles and stakeholders' programming skills. However, it provides an approximate measurement of development effort. We measured development effort using Eclipse Metrics 1.3.6 plug-in[4]. This tool counts actual Java statement as LoC and does not consider blank lines or lines with comments.

Entities	**Component(model)**	**Interaction mode**	**Runs on**
Sensor	Temperature(AM2302) Humidity(AM2302) Smoke(MQ2)	Periodic Periodic Event	RaspberryPi RaspberryPi Arduino
Actuator	Heater(using LCD) Alarm(using buzzer)	Cmd Cmd	RaspberryPi RaspberryPi
Tag	BadgeReader(RFID-RC522)	Event	RaspberryPi
WebService	Yahoo Weather	Req./Resp.	Yahoo Server
End-user Interaction	EndUserApp DashBoard	Notify Notify	AndroidPhone Desktop
Storage	ProfileDB(MySQL)	Req./Resp.	Microsoft Cloud
Computation	Proximity & others	Event, Cmd, Req./Resp.	Desktop

Table 2: Smart home application implementation

Setup for experiments We selected the smart home application as a case study for the evaluation, implemented it with the following existing approaches, and compared them with our approach: (1) GPL: We have implemented application-specific functionality using general-purpose programming languages such as `Node.js`, `HTML`, `Android`, and `JavaScript`. (2) Cloud-based Platform: As a representative tool for cloud-based platforms, we selected Node-RED[5]. It is a widely popular visual tool for wiring together hardware devices, APIs and online services. It provides a flow editor where developers can drag-and-drop nodes and configure them using dialog editor to specify appropriate properties. To calculate the LoC using Node-RED, we counted each configuration specification as one line of code. We intentionally omitted WSN macro-programming approaches in the comparison because these approaches largely focus on similar types of devices and their main purpose largely is to sense data only. Therefore, WSN macroprogramming systems may not be suitable for evaluation, given IoTSP application execute on heterogeneous entities.

We set up a *real* execution environment that is made up of entities, exhibiting heterogeneity discussed in Section 1.2. Table 2 shows entities used to perform the evaluation. All three approaches have been implemented and tested on this setup.

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

[5] http://nodered.org/

Results Table 3 shows the LoC required to develop the case study using all three approaches. Using GPL, the stakeholders have to write more than twice number of LoC compared to Node-RED to implement the same application (57% effort reduction compared to GPL). The primary reason of effort reduction is that Node-RED provides high-level constructs that hides low-level details.

Approch.	**S**	**A**	**T**	**WS**	**EU**	**ST**	**Comp.**	**Total**
GPL	51	40	9	19	211	36	267	633
Node-RED	51*	40*	9*	0	39	14	118	271
MDD (IoTSuite)	40(Vocab. Spec.)+29(Arch. Spec.)+ 43(Deploy. Spec.)+14(User Interaction Spec.) +26(App. Logic code)+36(User Interface Code)							188

Table 3: Comparison of existing approaches: Lines of code required to develop the smart home application. **S** (Sensor), **A** (Actuator), **T** (Tag), **WS** (External Web Service), **EU** (End user Application), **ST** (Storage),**Comp.** (Computational Services).

Cloud-based platform is a viable approach compared to the GPL. It reduces the development effort by providing cloud-based APIs to implement common functionality. However, one of Node-RED drawbacks is its *node-centric* approach. The stakeholders have to write code to implement platform-specific functionality such as reading values from sensors, and actuating actuators. For Node-RED, nodes for sensors and actuators are contributed at public repository [6]. However, many nodes are not available in the library till May 19, 2016, marked as * in Table 3. This leads to a platform-dependent design and increases the development effort.

Table 3 shows that stakeholders can write the same application using our approach in 188 LoC (70% effort reduction compared to GPL and 30% effort reduction compared to Node-RED). The reason is that our approach provides the ability to specify an application at global-level rather than individual devices. For instance, the domain language provides abstractions to specify entities in platform-independent manner. The translation of this specification to platform-specific code is taken care by our approach. So, the stakeholders do not have to write platform-specific code while developing an application.

[6]http://flows.nodered.org/

9 Real time PT 100 sensor data visualization using RIO600

The goal of this experiment is to utilize the capability of ABB devices and tools to enable development of IoTSP application. To enable the real time (streaming) visualization, we used ABB devices such as RIO 600, REF 620 (IED) and ABB tools such as PCM600, Micro-SCADA. RIO600 is used to extent the capability of sensing/actuating entities and enable communication with other entities using GOOSE and Modbus communication protocol. Several units can connected to read or write analog/digital value. The RTD module is used to read analog value, DIM module is used to read digital value, and DOM module is used to write digital value.

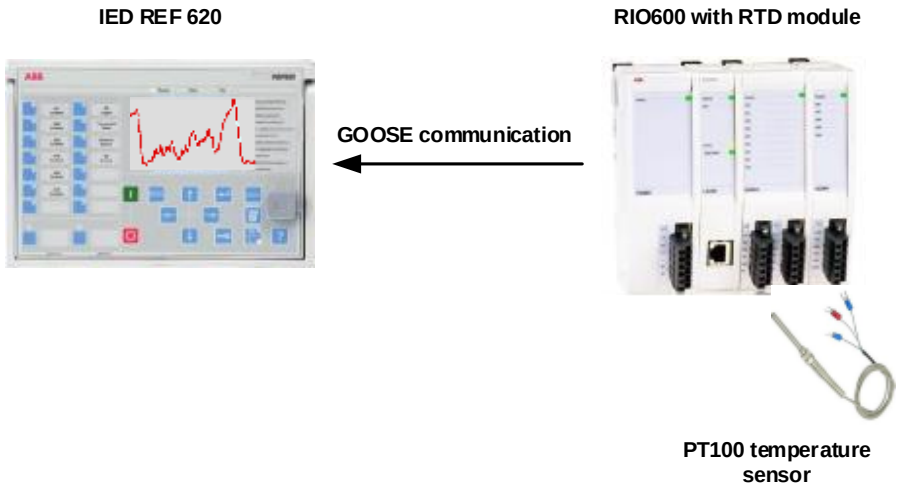

Figure 28: Architecture of the system

The architecture of the system is shown in Figure 28. The system consists of PT 100 temperature sensor, RIO600 with RTD module, and REF 620 (IED- Intelligent electronic device). PT 100 temperature sensor connected with RTD module (compatible with RTD module) is used to measure the temperature value between -40°C to +200°C. The RTD (analog module) reads the value of PT 100 temperature sensor. Once the PT 100 temperature sensor is connected to RTD module, stakeholders need to configure the parameter of RIO600 as shown in Figure 29. Referring Figure 29, first stakeholders have to create new project in PCM600, add RIO600, REF 620 IED used in the setup. Once the project is created stakeholders have to set the parameter configuration of RTD module. This configuration includes information about what channels are used (e.g., sensor is connected to which Channel), input mode (e.g., type of the sensor), conn type (whether it is a two wire or three wire connection), Value unit (whether sensor reads value in degree Celsius or Fahrenheit), and min-max value. Once this configuration is done, stakeholders have to write this configuration to RIO600 (this is required to update default configuration of RIO600).

The next step is to specify communication configuration. First, stakeholders need to identify publisher/subscriber available in the same network. Stakeholders configure such configuration using IEC 61850 configuration of RIO600 as shown in Figure 30. Referring Figure 30, stakeholders have to

enable options that what kind of value (analog or binary) RIO600 publish through GOOSE and who is the subscriber. Here RIO600 publish both analog and binary data and subscribe by LD0 (i.e. REF 620).

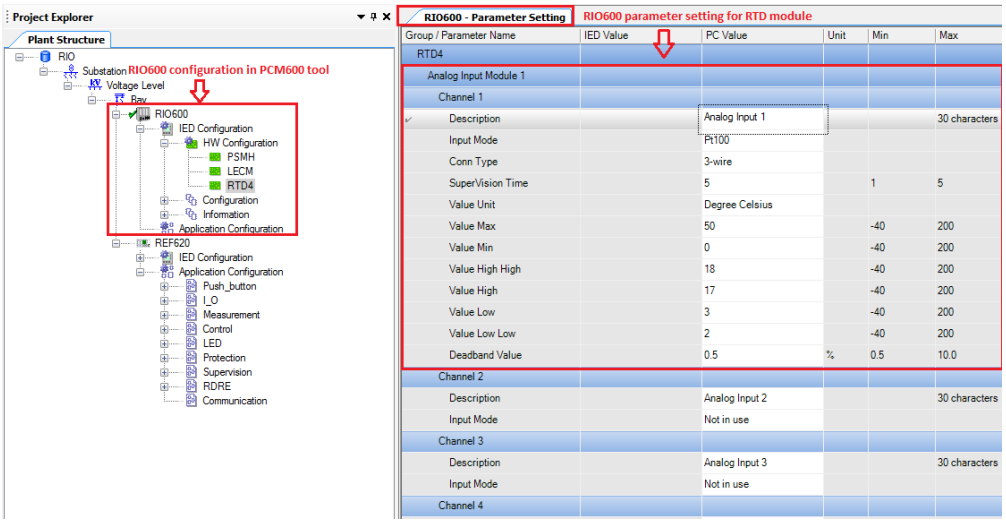

Figure 29: RIO600 parameter configuration for RTD module using PCM600

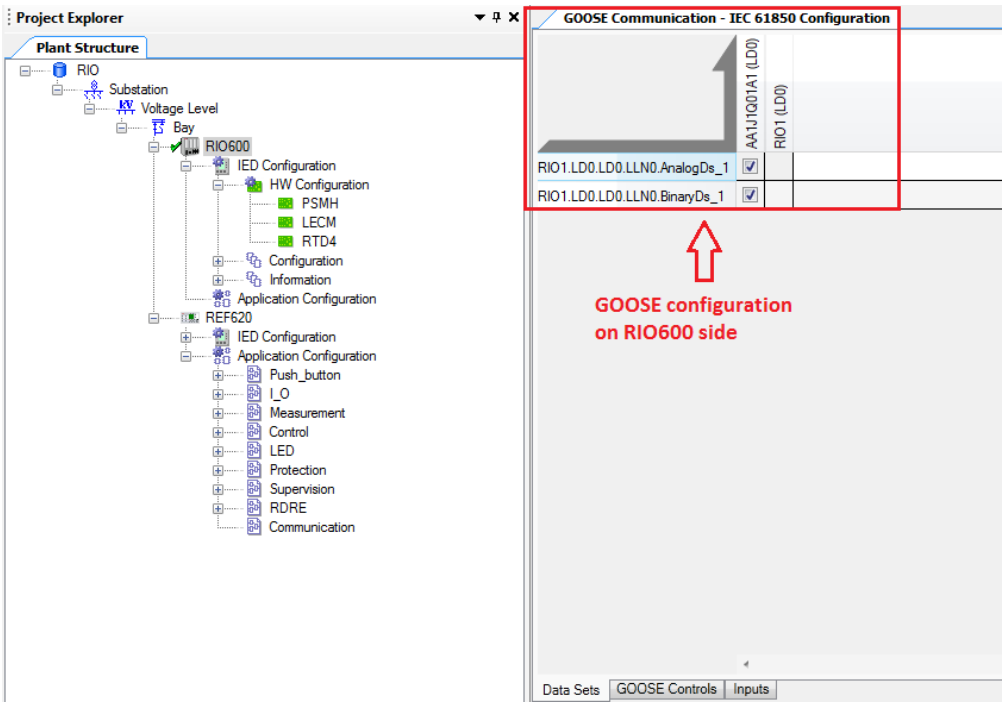

Figure 30: GOOSE communication configuration on RIO600 side

Once the GOOSE configuration is done on RIO600 side using PCM600, stakeholders have to configure signal matrix of REF620. Here they have to provide information about what channel

are they looking to subscribe for the data. This channel number should be mapped with publisher channel number as shown in Figure 29. Here, publisher (RIO600) publish data using Analog channel 1 (ref Figure 29) and subscriber (REF620) subscribe for the data using the same channel (i.e. Analog Channel 1, ref Figure 31).

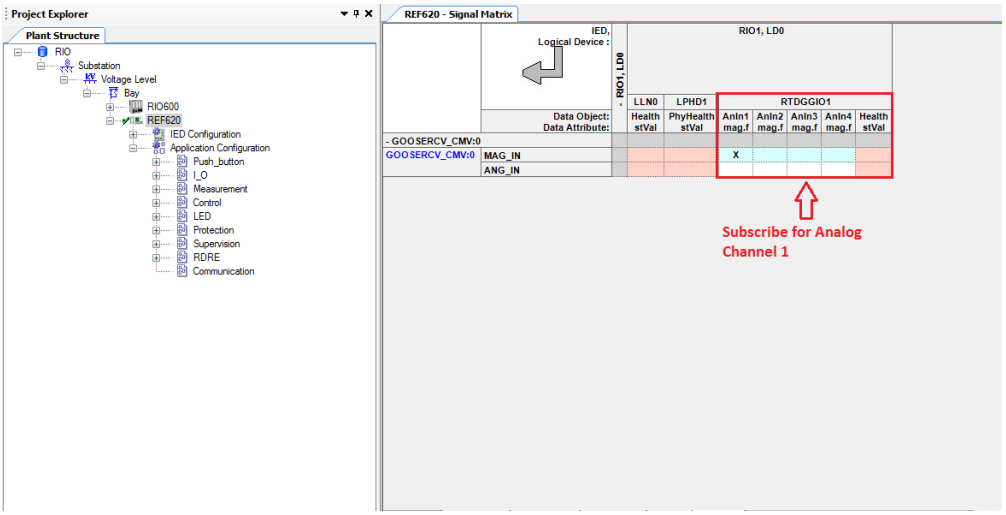

Figure 31: Signal Matrix to subscribe for Analog Channel 1

The stakeholders map the subscribe value using GOOSE communication to the subscriber input (REF620) as shown in Figure 32.

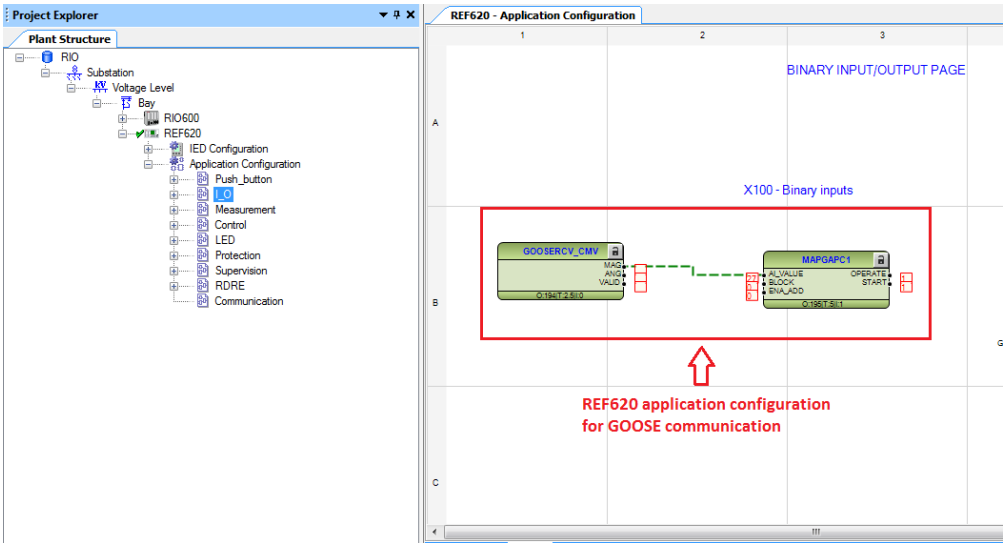

Figure 32: Mapping GOOSE configuration in REF620

Stakeholders visualize the data in form of trends or value. The RIO600 is equipped with web-interface, which display the real time temperature sensor value as shown in Figure 33. The Micro-SCADA (ABB measurement tool) is used to enable the functionality of real time visualization

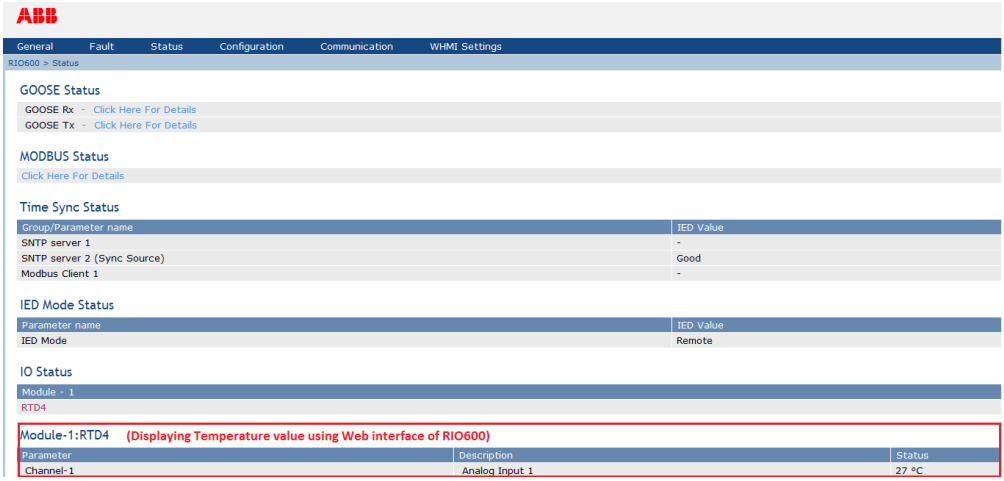

Figure 33: Displaying temperature value through web-interface of RIO600

of PT 100 temperature sensor. It allows the stakeholders to set axis legends, axis name, and also provide the summary of trend as shown in Figure 34. The Micro-SCADA also allows the stakeholders to export data to .CSV file or MySQL database which is used for further analysis.

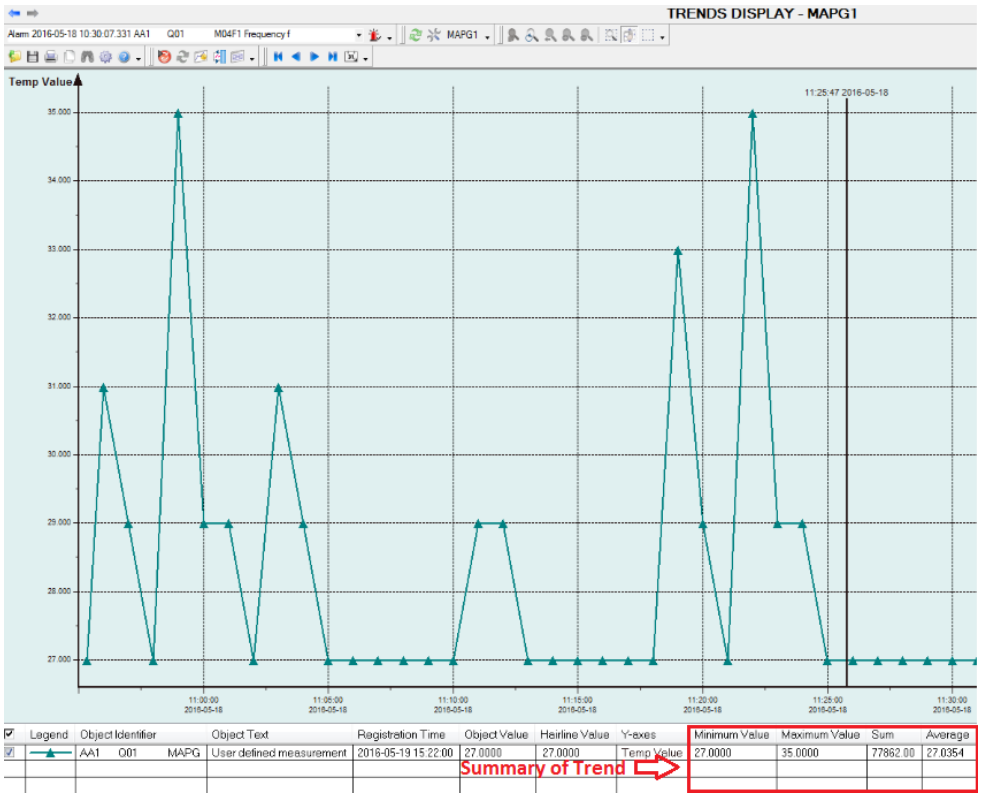

Figure 34: Displaying Trend using Micro-SCADA

10 Conclusion and Future work

We have built upon our existing framework and evolved it into a framework for developing IoTSP applications, with substantial additions and enhancements such as describing heterogeneous entities and end-user interactions. We present a comparative evaluation results with existing approaches. This provides the IoTSP community for further benchmarking. The evaluation is carried out on real devices exhibiting IoTSP heterogeneity. Our experimental analysis and results demonstrates that our approach drastically reduces development effort for IoTSP application development compared to existing approaches.

Our immediate future work will be to evaluate the usability of this development framework. This will help us to assess the potential for transferring it to industrial environment for IoTSP domain.

References

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

[2] E. Balland, C. Consel, B. N;Kaoua, and H. Sauzéon. A case for human-driven software development. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.

[3] M. Blackstock and R. Lea. Wotkit: A lightweight toolkit for web of things. In *Proceedings of the Third International Workshop on Web of Things*. ACM, 2012.

[4] D. Cassou, J. Bruneau, C. Consel, and E. Balland. Toward a tool-based development methodology for pervasive computing applications. *IEEE Transactions on Software Engineering*, 38(6):1445–1463, 2012.

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

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

[7] P. Patel and D. Cassou. Enabling high-level application development for the internet of things. *Journal of Systems and Software*, 103:62 – 84, 2015.

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

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