An IoT application development using IoTSuite

Saurabh Chauhan, Pankesh Patel
ABB Corporate Research, Bangalore, India

Abstract
Application development in the Internet of Things (IoT) is challenging because it involves dealing with issues that attribute to different life-cycle phases. First, the application logic has to be analyzed and then separated into a set of distributed tasks for an underlying network. Then, the tasks have to be implemented for the specific hardware. Moreover, we take different IoT applications and present development of these applications using IoTSuite.

In this paper, we introduce a design and implementation of ToolSuite, a suite of tools, for reducing burden of each stage of IoT application development process. We take different class of IoT applications, largely found in the IoT literature, and demonstrate these IoT application development using IoTSuite. These applications have been tested on several IoT technologies such as Android, Raspberry PI, Arduino, and JavaSE-enabled devices, Messaging protocols such as MQTT, CoAP, WebSocket, Server technologies such as Node.js, Relational database such as MySQL, and Microsoft Azure Cloud services.

Keywords: Internet of Things, Wireless Sensor Networks, Ubiquitous/Pervasive Computing, Programming Framework, Toolkit, Domain-specific languages, Development Life-cycle

1. Introduction
Recent technological advances in computer and communication technology have been fueling a tremendous growth in a number of smart objects (or things) [19]. In the Internet of Things [1] these "things" acquire intelligence, thanks to the fact that they access information that has been aggregated by other things. For example, a building interacts with its residents and surrounding buildings in case of fire for safety and security of residents, offices adjust themselves automatically accordingly to user preferences while minimizing energy consumption, or traffic signals control in-flow of vehicles according to the current highway status [7, 1]. It is the goal of our work to enable the development of such applications. In the following, we discuss one of such applications.

1.1. Challenges and contributions
Application development in the IoT is challenging because stakeholders [2] have to address issues that are attributed to different life cycles phases, including development, deployment, and maintenance [4]. At the development phase, the application logic has to be analyzed and separated into a set of distributed tasks for the underlying network consisting of a large number of heterogeneous entities. Then, the tasks have to be implemented for the specific platform of a device. At the deployment phase, the application logic has to be deployed onto a large number...
of devices. Manual effort in above two phases for hundreds to thousands of heterogeneous devices is a time-consuming and error-prone process.

An important challenge that needs to be addressed in the IoT is to enable the rapid development of applications with minimal effort by stakeholders involved in the process [11, 13]. In our previous publications [15, 14], we have provided a complete tour of our IoT application development process, summarized in Section 2. This paper goes beyond it. In particular, it describes implementation technologies, tools, language used and their rationales of choosing them [17, 6, 12]. Although various efforts exist in literature for making IoT application development easier, very few of them are publicly available for stakeholders to choose from. Given the usefulness of open source, our aim is to provide an opportunity to community for the creation of novel software engineering tools and the conduction of novel research for IoT application development.

The main contribution of this paper is an implementation of ToolSuite, a suite of tools, for reducing burden at different phases for IoT application development (detail in Section 3). Moreover, we take different class of IoT applications [16] and describe an application development process using IoTSuite.

Outline. The remainder of this paper is organized as follows: Section 2 presents the development framework that includes the proposed modeling languages and automation techniques. Section 3 presents the implementation of the development framework and describes implementation technologies, tools, and languages used. Section 4 describes step by step IoT application development process using IoTSuite. It also focus on different class of application to demonstrate application development using IoTSuite. Finally, in Section 5, we conclude this article and briefly mention some of future directions.

2. IoT application development process

This section presents our development framework that separates IoT 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 an IoT using our approach:

2.1. Domain Concern

This concern is related to concepts that are specific to a domain (e.g., building automation, transport) of an IoT. 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 1). 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.

2.2. Functional Concern

This concern is related to concepts that are specific to functionality of an IoT application. 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 application architecture. Referring the domain specification, the software designer specifies an application architecture using the Architecture Language (AL)- (Step 2 in Figure 1). It consists of specification of computational services and interaction among them. A com-
putational 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 application architecture consists of common operations and custom components that are specific to the application logic.

**Implementing application logic.** The compilation of an architecture specification generates an architecture framework (Step 3 in Figure 1). The architecture framework contains abstract classes, corresponding to each computational service, that hide interaction details with other software components and allow the application developer to focus only on application logic. The application developer implements only abstract methods of generated abstract classes, described in our work [9, p. 73]. We have integrated a framework for common operations. This further reduces the development effort for commonly found operations in IoT application and provides re-usability.

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 1). 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 [9, p. 75]. We have integrated existing open-source sensing framework\(3\) for Android devices. Moreover, we have implemented sensing and actuating framework for Raspberry Pi and storage framework for MongoDB, MySQL, and Microsoft AzureDB. So,

\[\text{http://www.funf.org/}\]
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 them using User Interaction Language (UIL) (Step 5 in Figure 1).

**Implementing user-interface code.** Leveraging the user interaction specification, the development framework generates a User Interface (UI) framework (step 6 in Figure 1). The UI framework contains a set of interfaces and concrete classes corresponding to resources defined in the user interaction specification. The concrete classes contain concrete methods for interacting with other software components. The user interface designer implements interfaces. These interfaces implement code that connects appropriate UI elements to concrete methods. For instance, a user initiates a command to heater by pressing UI element such as button that invokes a sendCommandToHeater() concrete 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 1). The deployment specification includes the details of each device as well as abstract interactors specified in the user interaction specification.

**Mapping.** The mapper 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 1). It maps computational service to a device. The current version of mapper algorithm [9] 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 1). 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. Components of IoTSuite

This section presents the implementation of the proposed IoT application development process discussed in Section 2. In particular, it describes implementation technologies, tools, language used and their rationales of choosing them. Figure 2 shows the various components at each phase of application development that stakeholders can use, described below.

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

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

Each component is described in detail in the following sections.

### 3.1. Editor

The editor provides supports for specifying high-level textual languages with the facilities of outline view, syntax coloring, code folding, error checking, rename re-factoring, and auto completion. The editor support is provided at different phases of IoT application development to help stakeholders illustrated in Figure 2: (1) editor for specifying a domain to aid the domain expert, (2) editor for specifying an architecture to aid the software designer, (3) editor for specifying an user interaction to aid the software designer and, (4) editor for specifying a deployment scenario to aid the network manager.

We take the editor for domain specification as an example to demonstrate an editor support provided by IoT-Suite, illustrated in Figure 3. The zone 1 shows the editor, where the domain expert writes a domain specification. The zone 2 shows the context menu, where the domain expert invokes the compiler for domain specification to generate a framework.

**Features of editor.** We use Xtext\(^4\) for a full fledged editor support, similar to work in 8. The Xtext is a framework for a development of domain-specific languages, and provides an editor with features such as syntax coloring.\(^5\)

\(^4\)http://www.eclipse.org/Xtext/

---

**Figure 2** – Overview of components in IoT-Suite.
error checking, auto completion, rename re-factoring, outline view, and code folding:

- We implemented Outline/Structure view feature, which is displayed on top most right side of the screen - (Refer Figure 4). It displays an outline of a file highlighting its structure. This is useful for quick navigation. As shown in Figure 4 vocab.mydsl file contains a large number of structures, sensors, and actuators, than from outline view by just clicking on particular structure (e.g., TempStruct) it navigates
to TempStruct definition in the vocab.mydsl. So, developers don’t need to look into an entire file.

- Using syntax coloring feature, keywords, comments, and other datatype elements are appeared in colored text. Here, resources, and tags are appeared in colored text as shown in Figure 5.

- 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, again click on plus sign. As shown in Figure 6, the code is folded for TempStruct, and BadgeStruct.

- The error checking feature guides developer if any error is there. An 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.

- The Auto completion is used to speed up writing text in specification files. In order to use auto completion feature, developer needs to press ctrl+space key at current cursor position, so it will provide suggestion. Here in BadgeReader definition, if developer writes Bad and press ctrl+space than it will suggest developer to write BadgeStruct (Refer Figure 7).

3.2. Compiler

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

Parser. It converts high-level specifications (domain, architecture, userinteraction, and deployment specification)
into data structures that can be used by the code generator. Apart from this core functionality, it also checks syntax of specifications and reports errors to stakeholders. The parser is implemented using ANTLR parser generator [8]. The ANTLR parser is a well-known parser generator that creates parser files from grammar descriptions. **Code generator.** Based on parser outputs, the code generator creates required files. It is composed of two sub-
modules: (1) core-module, (2) plug-in. The core-module manages a repository of plug-ins. Each plug-in is specific to a target implementation code. The target code could be in any programming language (e.g., Java, Python). Each plug-in is defined as template files, which the core-module uses to generate code. The key advantage of separating core-module and plug-in is that it simplifies an implementation of a new code generator for a target implementation.

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

We build two compilers to aid stakeholders shown in Figure 2. (1) compiler for a domain specification. It translates a domain specification and generates a domain framework, and a customized architecture and deployment grammar to aid stakeholders. (2) compiler for an architecture specification. It translates an architecture specification and generates an architecture framework to aid the application developer. The both generated frameworks are compatible with Eclipse IDE. For example, Figure 8 shows a generated architecture framework containing Java files (1 in Figure 8) in Eclipse IDE. 2 in Figure 8 shows a generated Java file in the architecture framework for a RoomController. Note that the generated framework contains abstract method (3 in Figure 8), which are implemented by the application developer using Eclipse IDE.

3.3. Mapper

The mapper produces a mapping from a set of computational services to a set of devices. Figure 9 illustrates the architecture of the mapper component. This component parses a deployment and architecture specification. The parser converts high-level specifications into appropriate data structures that can be used by the mapping algorithm. The mapping algorithm maps computational services described in the architecture specification to devices described in the deployment specification and pro-
duces mapping decisions into appropriate data structures. The code generator consumes the data structures and generates mapping files that can be used by the linker component.

In our current implementation, this module randomly maps computational services to a set of devices. However, due to generality of our framework, more sophisticated mapping algorithm can be plugged into the mapper component.

3.4. Linker

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

The current version of the linker generates packages for Android, Node.js, and JavaSE platform. Figure 10 illustrates packages for Android devices (1 in Figure 10), JavaSE target devices (2 in Figure 10), and Node.js devices (3 in Figure 10) and imported into Eclipse IDE. In order to execute code, these packages still need to be compiled by a device-level compiler designed for a target platform.

3.5. Runtime system

The main responsibility of the runtime system is a distributed execution of IoT applications [10]. It is divided into three parts: (1) middleware: It runs on each individual device and provides a support for executing distributed tasks. (2) wrapper: It plugs packages, generated by the linker module, and middleware. (3) support library: It separates packages, produced by the linker component, and underlying middleware by providing interfaces that are implemented by each wrapper. The integration of a new middleware into IoTSuite consists of an implementation of the following interfaces in the wrapper:

- publish(). It is an interface for publishing data from a sender. The definition of this interface contains: an event name (e.g., temperature), event data (e.g., a temperature value, Celsius), and publisher’s information such as location of a sender.

- subscribe(). It is an interface for receiving event notifications. An interest of events is expressed by sending a subscription request, which contains: a event name (e.g., temperature), information for filtering events such as regions of interest (e.g., a RoomAvgTemp component wants to receive events only from a current room), and subscriber’s information.

- command(). It is an interface for triggering an action of an actuator. A command contains: a command name (e.g., switch-on Heater), command parameters (e.g., set temperature of Heater to 30°C), and a sender’s information.

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

![Figure 9 – Architecture of the mapper component in IoTSuite.](image)
The current implementation of IoTSuite uses the MQTT middleware. It enables interactions among Android devices, Node.js-enabled devices, and JavaSE-enabled devices. The current wrapper implementation for the MQTT middleware is available at URL [6].

3.6. Eclipse plug-in

We have integrated the above mentioned components as Eclipse plug-in to provide end-to-end support for IoT application development. Figure 11 illustrates use of our plug-in at various phases of IoT application development:

1. Vocab.mydsl (① in Figure 11) – using which the domain expert can describe and compile a domain specification of an application domain.
2. Arch.mydsl (② in Figure 11) – using which the software designer can describe and compile an architecture specification of an application.
3. Userinteraction.mydsl (③ in Figure 11) – using which the software designer can describe and compile an userinteraction specification of an application.

[6] https://github.com/pankeshlinux/ToolSuite
4. Deploy.mydsl (4 in Figure 11) – using which the network manager can describe a deployment specification of a target domain and invoke the mapping component. The network manager can combines and packs code generated by various stages of compilation into packages that can be deployed on devices.

4. A step-by-step applications development using IoTSuite

This section describes each step of IoT application development process using IoTSuite. Application development using IoTSuite is a multi-step process and focuses on design, implement, and deployment phases to develop IoT applications. We take different class of application (SCC (Sense-Compute-Control), End-User-Interaction, and Data Visualization) as an example to demonstrate application development using IoTSuite.

**Specifying high-level specifications.** It includes specifications of domain specification (includes concepts that are specific to domain of IoT), architecture specification (includes concepts that are specific to functionality of IoT), userinteraction specification (defines what interaction are required by an application), and deployment specification - (describes a device and its properties in a target deployment). Stakeholders specify high-level specifications using IoTSuite-Eclipse-Plugin[7].

4.1. Personalized HVAC application:

A home consists of several rooms, each one is instrumented with several heterogeneous entities for providing resident’s comfort. 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 device (Heater) are updated as shown in Figure 12. Developers need to follow following steps to develop above discussed application using IoTSuite.

Figure 11 – Eclipse plug-in for IoT application development.
Specifying Domain specification. Developer specifies domain specification using IoTSuite as shown in Listing 1. Each resource is characterized by types of information it generates or consumes. A set of information is defined using the `structs` keyword (Listing 1 line 1). For instance, a BadgeReader (lines 10-12) may generate a badgeDetected and badgeDisappeared. This information is defined as BadgeStruct and its two fields (lines 6-7). A Heater is set according to a user’s temperature preference illustrated in Listing 1 lines 14-16. The SetTemp action takes a user’s temperature preference shown in line 15. A set of storage is declared using the `storages` keyword (Listing 1 line 17). A retrieval from the storage requires a parameter, specified using the `accessed-by` keyword (Listing 1 line 19).

For instance, a user’s profile is accessed from storage by a badgeID (Listing 1 lines 19-20).

```plaintext
1 structs:
2   TempStruct
3     tempValue : double;
4     unitOfMeasurement : String;
5   BadgeStruct
6     badgeID: String;
7     badgeEvent: String;
8 resources:
9    tags:
10       BadgeReader
11         generate badgeDetected: BadgeStruct;
12         generate badgeDisappeared: BadgeStruct;
13       actuators:
14         Heater
15           action SetTemp(setTemp:TempStruct);
16           action Off();
17       storages:
18         ProfileDB
19         generate profile: TempStruct accessed-by
           badgeID:String;
```

Listing 1 – Code snippet of domain spec.

Compilation of domain specification. To compile domain specification (vocab.mydsl file), Right click on the vocab.mydsl file (Step 1) and click on Compile Vocab option (Step 2) as shown in Figure 13.

Architecture specification. Developer specifies architecture specification using IoTSuite as shown in Listing 2. It is described as a set of computational services. It consists of two types of computational services: (1) Common component specifies common operations (e.g., average, count, sum) in the application logic. (2) Custom specifies an application specific logic. For instance, the Proximity component is a custom component that coordinates events from BadgeReader with the content from ProfileDB as shown in Figure 12. Each computational service is described by a set of inputs and outputs. For instance, the Proximity consumes badgeDetected and badgeDisappeared (Listing 2 lines 13), request to ProfileDB to access user’s profile (Listing 2 line 15) and generates temp-
Pref (Listing 2 line 7). Command is issued by a computational service to trigger an action. For instance, the RoomController issues a SetTemp command (Listing 2 line 10) with a setTemp as an argument to Heater (Listing 1 lines 14-15).

```java
computationalServices:
  Custom:
    Proximity
      consume badgeDetected from BadgeReader;
      consume badgeDisappeared from BadgeReader;
      request profile to ProfileDB;
      generate tempPref: TempStruct;
    RoomController
      consume tempPref from Proximity;
      command SetTemp(setTemp) to Heater;
      command Off() to Heater;
```

Listing 2 – A code snippet of architecture spec.

Compilation of architecture specification. To compile architecture specification (arch.mydsl file), Right click on the arch.mydsl file (Step 1) and click on Compile Arch option (Step 2) as shown in Figure 14.

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 15.

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 16.

Implement application logic. The application logic project contains a generated framework that hides the low-level details from a developer and allows the developer to focus on the application logic. The developer writes application specific logic in the logic package. Listing 3 and 4 show generated Java programming framework from the architecture specification, defined in Listing 2. To implement application logic of Proximity component, developers have to implement the generated abstract methods...
The Proximity component receives the badgeDetected and badgeDisappeared events from the BadgeReader and coordinates with ProfileDB component for user-temperature preference defined in Listing 2, lines 3-6. To implement this logic, three methods (onNewbadgeDetected, onNewbadgeDisappeared, and onNewprofileReceived) have to be implemented as shown in Listing 3. It requests for ProfileDB for user’s preference using onNewbadgeDetected method, turn off Heater using onNewbadgeDisappeared method, and generate tempPref using onNewprofileReceived method which is consumed by the RoomController defined in Listing 2, line 9. In similar ways, developer implements application logic of RoomController component using onNewtempPref method. It issues a SetTemp and Off commands to Heater using onNewtempPref method as illustrated in Listing 4.

```java
package logic;
import android.content.Context;
import iotsuite.semanticmodel.*;
import framework.*;

public class LogicProximity extends Proximity {
    public LogicProximity(PubSubMiddleware pubSubM, Device deviceInfo, Object ui, Context myContext) {
        super(pubSubM, deviceInfo);
    }

    @Override
    public void onNewbadgeDetected(BadgeStruct arg) {
        // Request profileDB for his preference
        getprofile(arg.getbadgeID());
    }

    @Override
    public void onNewbadgeDisappeared(BadgeStruct arg) {
        // Person is leaving the room. Turn off the heater.
        TempStruct tempStruct = new TempStruct(-100, "C");
        settempPref(tempStruct);
    }
}
```
public void onNewProfileReceived(TempStruct arg) {
    TempStruct tempStruct = new TempStruct(arg.getTempValue(), "C");
    // Set temperature to RoomController
    setTempPref(tempStruct);
}

Listing 3 – The implementation of the Java abstract class Proximity written by the developer.

package logic;

import iotsuite.pubsubmiddleware.PubSubMiddleware;
import android.content.Context;
import iotsuite.semanticmodel.*;
import framework.*;

public class LogicRoomController extends RoomController {
    public LogicRoomController(PubSubMiddleware pubSubM, Device deviceInfo, Object ui, Context myContext) {
        super(pubSubM, deviceInfo);
    }

    @Override
    public void onNewTempPref(TempStruct arg) {
        if (arg.getTempValue() == -100) {
            // It means that person is leaving the room
            Off(); // Set the Off() command
        } else {
            // If the person is entering to the room
            // set temperature according to his preference
            double tempValue = arg.getTempValue();
            TempStruct tempStruct = new TempStruct(tempValue, arg.getUnitOfMeasurement());
            SetTemp(tempStruct);
        }
    }
}
Listing 4 – The implementation of the Java abstract class RoomController written by the developer.

Deployment specification. Developer specifies deployment specification using IoTSuite as shown in Listing 5. It includes properties such as location that defines where a device is deployed, resources define 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 5 shows a code snippet to illustrate these concepts. BadgeReaderMgmtDevice is located in room#1 (line 4), BadgeReader is attached with the device (line 6) and the device driver code for these two component is in NodeJS (line 5), mqtt runtime system is installed on BadgeReaderMgmtDevice (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, DatabaseSrvDevice (line 8) runs ProfileDB (line 12) component implemented in MySQL database (line 14) as illustrated in Listing 5.

```
devices:
  BadgeReaderMgmtDevice:
    location:
      Room: 1;
    platform: NodeJS;
    resources: BadgeReader;
    protocol: mqtt;
  DatabaseSrvDevice:
    location:
      Room: 1;
    platform: JavaSE;
    resources: ProfileDB;
    protocol: mqtt;
    database: MySQL;
  ResourceMgmtDevice1:
```
Compilation of deployment spec. Right click on deploy.mysdl file (Step 1) and selecting Compile Deploy (Step 2) as shown in Figure 17 generates a deployment packages.

Deployment of generated packages. The output of compilation of deployment specification produce a set of platform specific project/packages as shown in Figure 18 for devices, specified in the deployment specification (Refer Listing 5). These projects compiled by device specific compiler designed for the target platform. The generated packages integrate the run-time system.

4.2. Fire Detection Application

A home consists of several rooms, each one is instrumented with several heterogeneous entities for providing resident’s safety. 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 as shown in Figure 19.

Specifying Domain specification. Developer specifies domain specification using IoTSuite as shown in Listing 6. Each resource is characterized by types of information it generates or consumes. A set of information is defined using the structs keyword (Listing 6, line 1). A set of sensors is declared using the sensors keyword (Listing 6, line 9). Each sensor produces one or more sensor measurements along with the data-types specified in the data structure (Listing 6, lines 2-7), declared using the generate keyword. A periodic sensor samples results every d seconds for duration k seconds. For instance, a temperature sensor generates a tempMeasurement (Listing 6, line 12) of TempStruct type (Listing 6, lines 2-4). It samples data every 1 second for next 6 minutes (Listing 6, line 13). An event-driven sensor produces data when the event condition is met. For instance, a smoke sensor generates smokeMeasurement when smokeValue > 650 PPM (Listing 6, lines 16-17). A set of actuators is declared using the actuators keyword (Listing 6, line 18). Each actuator has one or more actions declared using the action key-
word. For instance, an Alarm may have one action (e.g., on), illustrated in Listing 6 line 20.

```java
1 structs:
2 TempStruct
3   tempValue: double;
4 struct SmokeStruct
5   smokeValue: double;
6   unitOfMeasurement: String;
7   resources:
```
sensors:
  periodicSensors:
    TemperatureSensor
generate tempMeasurement: TempStruct;
sample period 1000 for 6000000;
eventDrivenSensors:
  SmokeDetector
generate smokeMeasurement: SmokeStruct;
onCondition smokeValue > 650 PPM;
actuators:
  Alarm
  action On();

Listing 6 – Code snippet of domain spec.

Compilation of domain specification. To compile domain specification (vocab.mydsl file), Right click on the vocab.mydsl file (Step 1) and click on Compile Vocab option (Step 2) as shown in Figure 13.

Architecture specification. Developer specifies architecture specification using IoTSuite as shown in Listing 7. It is described as a set of computational services. It consists of two types of computational services: (1) Common component specifies common operations (e.g., average, count, sum) in the application logic. For instance, RoomAvgTemp component consumes 5 temperature measurements (Listing 7, lines 4-5), apply average by sample operation (Listing 7, line 5), and generates roomAvgTempMeasurement (Listing 7, line 6). (2) Custom component specifies an application specific logic. Additionally, each computational service is described by a set of inputs and outputs. For instance, the FireController consumes smokeValue (Listing 7, line 13), issues On and FireNotify (with fireNotify as argument) commands (Lines 14-15).

Listing 7 – A code snippet of architecture spec.

Compilation of architecture specification. To compile architecture specification (arch.mydsl file), Right click on the arch.mydsl file (Step 1) and click on Compile Arch option (Step 2) as shown in Figure 14.

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 15.

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 16.

Implement application logic. The application logic project contains a generated framework that hides low-level details from a developer and allows the developer to focus on the application logic. The developer writes application specific logic in logic package. Listing 8 and 9 show generated Java programming framework from the architecture specification, defined in Listing 7. To implement application logic of FireState, developers have to implement the generated abstract methods illustrated in Listing 8. The FireState component consumes the roomAvgTempMeasurement from RoomAvgTemp, smokeMeasurement from SmokeDetector and generates smokeValue defined in Listing 7. To implement this logic, two methods onNewsmokeMeasurement, and onNewroomAvgTempMeasurement have to implemented as shown in Listing 8. It reads the smokeMeasurement using onNewsmokeMeasurement method, roomAvgTempMeasurement using onNewroomAvgTempMeasurement method, and sets smokeValue if smokeValue and avgtempValue are greater than threshold value. In
similar ways, developer implements application logic of FireController component using onNewtempPref method. It issues On command to alarm and fireNotify command to EndUserApp using onNewsmokeValue method as illustrated in Listing 9 defined in Listing 7 lines 14-16.

```java
package logic;

import iotsuite.pubsubmiddleware.PubSubMiddleware;
import android.content.Context;
import iotsuite.semanticmodel.*;
import framework.*;

public class LogicFireState extends FireState {
    double avgtempValue, smokeMeasurement;
    public LogicFireState(PubSubMiddleware pubSubM, Device deviceInfo, Object ui, Context myContext) {
        super(pubSubM, deviceInfo);
    }

    @Override
    public void onNewsmokeMeasurement(SmokeStruct arg) {
        // Read smokeMeasurement from SmokeDetector
        smokeMeasurement = arg.getsmokeValue();
        // If smokeValue and avgtempValue are greater than threshold means fire
        if (smokeMeasurement > 650 && avgtempValue > 55) {
            SmokeStruct smokeStruct = new SmokeStruct(650, "PPM");
            setsmokeValue(smokeStruct);
        }
    }

    @Override
    public void onNewroomAvgTempMeasurement(TempStruct arg) {
        // Read roomAvgTempMeasurement from RoomAvgTemp
        avgtempValue = arg.gettempValue();
    }
}
```

Listing 8 – The implementation of the Java abstract class FireState written by the developer.

```java
package logic;

import java.sql.Timestamp;
import iotsuite.pubsubmiddleware.PubSubMiddleware;
import android.content.Context;
import iotsuite.semanticmodel.*;
import framework.*;

public class LogicFireController extends FireController {
    public LogicFireController(PubSubMiddleware pubSubM, Device deviceInfo, Object ui, Context myContext) {
        super(pubSubM, deviceInfo);
    }

    @Override
    public void onNewsmokeValue(SmokeStruct arg) {
        // Read smokeValue from FireState and fire command to Alarm and EndUserApp
        FireNotify(new FireStateStruct("Fire has been occurred",
                new Timestamp(new java.util.Date().
                    getTime()).toString()));
    }
}
```

Listing 9 – The implementation of the Java abstract class FireController written by the developer.

User interaction specification. Developer specifies user interaction specification using IoTSuite as shown in Listing 10. It 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. Notify 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 10 line 8). The application notifies users with the fire information specified in the data structure (Listing 10 lines 24).

```java
1 structs:
2 FireStateStruct
3 fireValue: String;
```
Compilation of userinteraction specification. To compile userinteraction specification (userinteraction.mydsl file-e), Right click on the userinteraction.mydsl file (Step 1) and click on Compile UserInteraction option (Step 2) as shown in Figure 20.

Listing 10 – Code snippet of user-interaction spec.

Compilation of deployment spec. Right click on deploy.mydsl file (Step 1) and selecting Compile Deploy (Step 2) as shown in Figure 17 generates a deployment packages.

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 15.

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 21.

Implementing user-interface code. In this step, developer implements user-interface code generated by compilation of user-interaction specification. Developer implements user-interface code in deviceImpl package [9, p. 15-16]. The implementation of user-interface code involves the use of drag-and-drop functionality provided by formwidget using activity_main.xml as shown in Figure 22. The developer connects this interaction with generated
framework in the AndroidEndUserApp.java file. Listing 12 shows how developer connects UI element with generated framework in deviceImpl package.

```java
package deviceImpl;
import logic.*;
import framework.*;
import android.content.Context;
import android.app.Activity;
import iotsuite.android.localmiddleware.IDataListener;
import iotsuite.android.localmiddleware.PubSubsSensingFramework;
public class AndroidEndUserApp implements IEndUserApp, IDataListener {
    public static PubSubsSensingFramework pubSubSensingFramework;
    private Context appContext;
    public static Activity appActivity;
    public static String txtDisplay;
    public AndroidEndUserApp(Context context, LogicEndUserApp obj) {
        this.appContext = context;
        appActivity = (Activity) appContext;
        pubSubSensingFramework = PubSubsSensingFramework.getInstance();
        pubSubSensingFramework.registerForSensorData(
            this, "fireNotifyNotify");
    }
    @Override
    public void onDataReceived(String eventName, Object data) {
        // Developer connects UI element with generated framework
        if (eventName.equals("fireNotifyNotify")) {
            FireStateStruct fireData = (FireStateStruct) data;
            txtDisplay = fireData.getfireValue();
        }
    }
    public void sendDataToUI(String eventName, Object data) {
        // Developer sends data to UI elements
    }
}
```

Figure 20 – Compilation of userinteraction spec.
Deployment of generated packages. The output of compilation of deployment specification produce a set of platform specific project/packages as shown in Figure 23 for devices, specified in the deployment specification (Refer Listing 11). These projects compiled by device specific compiler designed for the target platform. The generated packages integrate the run-time system.

4.3. Smart Home Monitoring Application

A home consists of several rooms, each one is instrumented with several heterogeneous entities for providing resident’s awareness. To provide a resident’s awareness, the system generates the current environment status on dashboard (e.g., humidity, temperature, outside temperature by interacting with external web services) as shown in Figure 24.

Listing 12 — The implementation of the generated AndroidEndUserApp class written by the developer.

Specifying Domain specification. Developer specifies domain specification using IoTSuite as shown in Listing 13. Each resource is characterized by types of information it generates or consumes. A set of information is defined using the structs keyword (Listing 13 line 1). A set of sensors is declared using the sensors keyword (Listing 13 line 9). Each sensor produces one or more sensor measurements along with the data-types specified in the data structure (Listing 13 lines 2-7), declared using the generate keyword. A periodic sensor samples results every d seconds for duration k seconds. For instance, a temperature sensor generates a tempMeasurement (Listing 13 line 12) of TempStruct type (Listing 13 lines 2-4). It samples data every 1 second for next 6 minutes (Listing 13 line 13). A request-based sensor responds its results only if it is requested. For instance, the YahooWeatherService

Figure 21 — Locate user-interface project
provides temperature value of a location given by a `locationID` (Listing 13 lines 17-19).

```java
structs:
TempStruct
tempValue : double;
unitOfMeasurement : String;
HumidityStruct
humidityValue : double;
unitOfMeasurement : String;
resources:
sensors:
periodicSensors:
TemperatureSensor
generate tempMeasurement: TempStruct;
sample period 1000 for 6000000;
```

Figure 22 – Implementation of user-interface code

Figure 23 – Packages for target devices specified in the deployment spec.
Figure 24 – Dataflow of Data Visualization Application

Compilation of domain specification. To compile domain specification (vocab.mydsl file), Right click on the vocab.mydsl file (Step 1) and click on Compile Vocab option (Step 2) as shown in Figure 13.

Architecture specification. Developer specifies architecture specification using IoTSuite as shown in Listing 14. It is described as a set of computational services. It consists of two types of computational services: (1) Common component specifies common operations (e.g., average, count, sum) in the application logic. (2) Custom specifies an application-specific logic. Additionally, each computational service is described by a set of inputs and outputs. For instance, the DataVisualizer consumes tempMeasurement, humidityMeasurement, and weatherMeasurement (Listing 14 lines 16), issues a DisplaySensorMeasurement command with a sensorMeasurement as an argument to Dashboard (line 7).

Listing 13 – Code snippet of domain spec.

Listing 14 – A code snippet of architecture spec.

Compilation of architecture specification. To compile architecture specification (arch.mydsl file), Right click on the arch.mydsl file (Step 3) and click on Compile Arch option (Step 4) as shown in Figure 14.

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 15.

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 16.

Implement application logic. The application logic project contains a generated framework that hides low-level details from a developer and allows the developer to focus on the application logic. The developer writes application specific logic in logic package. Listing 15 shows generated Java programming framework from the architecture specification, defined in Listing 14.

To implement application logic of DataVisualizer,
developers have to implement the generated abstract methods illustrated in Listing 15. The DataVisualizer component consumes the tempMeasurement, humidityMeasurement, weatherMeasurement and issues DisplaySensorMeasurement command to Dashboard defined in Listing 14, lines 4-7. To implement this logic, three methods (onNewweatherMeasurement, onNewhumidityMeasurement, and onNewtempMeasurement) have to be implemented as shown in Listing 15. It reads tempMeasurement using onNewtempMeasurement method, humidityMeasurement using onNewhumidityMeasurement method, and weatherMeasurement using onNewweatherMeasurement method illustrated in Listing 15. Additionally, it issues a DisplaySensorMeasurement command to Dashboard using onNewweatherMeasurement method as shown in Listing 15.

Listing 15 – The implementation of the Java abstract class DataVisualizer written by the developer.

Userinteraction specification. Developer specifies user-interaction specification using IoTSuite as shown in Listing 16. It 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. Notify denotes information flow from an application to a user. For instance, an application notifies a user by providing temperature, humidity and outside temperature value by interacting with YahooWeatherService. It is declared using the notify keyword (Listing 16, line 9). The application notifies users with the visualization information specified in the data structure (Listing 16, lines 29).
from DataVisualizer;

Listing 16 – Code snippet of user-interaction spec.

Compilation of userinteraction specification. To compile userinteraction specification (userinteraction.mydsl file), Right click on the userinteraction.mydsl file (Step 1) and click on Compile UserInteraction option (Step 2) as shown in Figure [20]

Deployment specification. Developer specifies deployment specification using IoTSuite as shown in Listing [17]. It includes properties such as location that defines where a device is deployed, resources define 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 [5] shows a code snippet to illustrate these concepts. SensorMgmtDevice is located in room#1 (line 4), TemperatureSensor and HumiditySensor are attached with the device (line 6) and the device driver code for these two component is in NodeJS (line 5), mqtt runtime system is installed on SensorMgmtDevice (line 7).

1 devices:
2 SensorMgmtDevice:
3 location:
4 Room: 1 ;
5 platform: NodeJS;
6 resources: TemperatureSensor, HumiditySensor;
7 protocol: mqtt ;
8 ResourceMgmtDevice1:
9 location:
10 Room: 1 ;
11 platform: JavaSE;
12 resources: ;
13 protocol: mqtt ;
14 EndUserDevice:
15 location:
16 Room: 1 ;
17 platform: NodeJS;
18 resources: Dashboard;
19 protocol: mqtt ;

Listing 17 – Code snippet of deployment spec.

Compilation of deployment spec. Right click on deploy.mydsl file (Step 1) and selecting Compile Deploy (Step 2) as shown in Figure [17] generates a deployment packages.

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 [15].

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 [21].

Implementing user-interface code. In this step, developer implements user-interface code generated by compilation of user-interaction specification. Developer implements user-interface code in deviceImpl package [9, p. 15-16]. To reduce development efforts to implement functionality of Dashboard, we are generating code with default template.

Deployment of generated packages. The output of compilation of deployment specification produce a set of platform specific project/packages as shown in Figure [25] for devices, specified in the deployment specification (Refer Listing [17]). These projects compiled by device specific compiler designed for the target platform. The generated packages integrate the run-time system.

5. Conclusion and Future work

Since the main goal of this research is to make IoT application development easy for stakeholders, we believe that our IoT application development process should be supported by tools to be applicable in an effective way. Therefore, this paper introduces a design and implementation of IoTSuite, a suite of tools, for reducing burden of each phase of IoT application development process. Moreover, we take different class of IoT applications, largely found in the IoT literature, and demonstrate these IoT
application development using IoTSuite. These applications have been tested on several IoT technologies such as Android, Raspberry PI, Arduino, and JavaSE-enabled devices, Messaging protocols such as MQTT, CoAP, WebSocket, Server technologies such as Node.js, Relational database such as MySQL, and Microsoft Azure Cloud services.

It consists of the following components to aid stakeholders: (1) We integrate a customized editor support for specifying high-level textual specification with the facilities of syntax coloring and syntax error reporting. (2) A compiler parses the high-level specifications and supports the application development phase by producing a programming framework that reduces development effort. This compiler knows at run-time which code generation plug-ins are installed and generates code in a target implementation language. (3) A deployment module is supported by mapping and linking techniques. They together support the deployment phase by producing device-specific code to result in a distributed system collaboratively hosted by individual devices. (4) A runtime system leverages existing middleware platforms and generates glue code to customize them with respect to needs of IoT applications. This mechanism increases the possibility of executing applications on different middleware.

6. Future work

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

Testing support for IoT application development. Our near term future work will be to provide support for the testing phase. A key advantage of testing is that it emulates the execution of an application before deployment so as to identify possible conflicts, thus reducing application debugging effort. The support will be provided by integrating an open source simulator in IoTSuite. This simulator will enable transparent testing of IoT applications in a simulated physical environment. Moreover, we expect to enable the simulation of a hybrid environment, combining both real and physical entities. Currently, we are investigating open source simulators for IoT applications. We see Siafu8 as a possible candidate due to its open source and thorough documentation.

Mapping algorithms cognizant of heterogeneity. We will provide rich abstractions to express both the properties of the devices (e.g., processing and storage capacity, networks it is attached to, as well as monetary cost of hosting a computational service), as well as the requirements

---

**Figure 25** – Packages for target devices specified in the deployment spec.

---

8 [http://siafusimulator.org/](http://siafusimulator.org/)
from stakeholders regarding the preferred placement of the computational services of the applications. These will then be used to guide the design of algorithms for efficient mapping (and possibly migration) of computational services on devices.

*Bibliography

[1] Atzori, L., Iera, A., Morabito, G., 2010. The Internet of Things: A Survey. Computer Networks 54, 2787–2805.

[2] Ballard, E., Consel, C., N.Kaoua, B., Sauzéon, H., 2013. A case for human-driven software development. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2486788.2486970

[3] Bertran, B., Bruneau, J., Cassou, D., Loriant, N., Ballard, E., Consel, C., 2012. DiaSuite: a tool suite to develop sense/compute/control applications. Science of Computer Programming.

[4] Bischoff, U., Kortuem, G., 2007. Life cycle support for sensor network applications. In: Proceedings of the 2nd international workshop on Middleware for sensor networks. ACM, pp. 1–6.

[5] Chauhan, S., Patel, P., Delicato, F. C., Chaudhary, S., 2016. A development framework for programming cyber-physical systems. In: Proceedings of the 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems. SEsCPS ’16. ACM, New York, NY, USA, pp. 47–53. URL http://doi.acm.org/10.1145/2897035.2897039

[6] Chauhan, S., Patel, P., Sureka, A., Delicato, F. C., Chaudhary, S., April 2016. Demonstration abstract: Iotsuite - a framework to design, implement, and deploy iot applications. In: 2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), pp. 1–2.

[7] de Saint-Exupery, A., 2009. Internet of things, strategic research roadmap.

[8] Parr, T., 2007. The definitive ANTLR reference: Building domain-specific languages. Pragmatic Bookshelf.

[9] Patel, P., Cassou, D., 2015. Enabling high-level application development for the internet of things. Journal of Systems and Software 103, 62 – 84. URL http://www.sciencedirect.com/science/article/pii/S0164121215000187

[10] Patel, P., Jardosh, S., Chaudhary, S., Ranjan, P., Sept 2009. Context aware middleware architecture for wireless sensor network. In: Services Computing, 2009. SCC ’09. IEEE International Conference on, pp. 532–535.

[11] Patel, P., Kattepur, A., Cassou, D., Bouloukakis, G., Feb. 2013. Evaluating the Ease of Application Development for the Internet of Things. Technical report.

URL https://hal.inria.fr/hal-00788356

[12] Patel, P., Kaulgud, V., Chandra, P., Kumar, A., Dec 2015. Building enterprise-grade internet of things applications. In: 2015 Asia-Pacific Software Engineering Conference (APSEC). pp. 4–5.

[13] Patel, P., Luo, T., Bellur, U., 2016. Evaluating a development framework for engineering internet of things applications. CoRR abs/1606.02119. URL http://arxiv.org/abs/1606.02119

[14] Patel, P., Morin, B., Chaudhary, S., 2014. A model-driven development framework for developing sense-compute-control applications. In: Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation. MoSEMiNA 2014. ACM, New York, NY, USA, pp. 52–61.

[15] Patel, P., Pathak, A., Cassou, D., Issarny, V., Apr. 2013. Enabling High-Level Application Development in the Internet of Things. In: 4th International Conference on Sensor Systems and Software. Lucca, Italy.

[16] Patel, P., Pathak, A., Teixeira, T., Issarny, V., 2011. Towards application development for the Internet of Things. In: Proceedings of the 8th Middleware Doctoral Symposium. ACM, p. 5.

[17] Soukaras, D., Patel, P., Song, H., Chaudhary, S., January 2015. IoTSuite : A ToolSuite for Prototyping Internet of Things Applications. The 4th International Workshop on Computing and Networking for Internet of Things (ComNet-IoT), co-located with 16th International Conference on Distributed Computing and Networking (ICDCN), 2015.

[18] Taylor, R., Medvidovic, N., Dashofy, E., 2009. Software architecture: foundations, theory, and practice. Wiley Publishing.

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