UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Software Engineering Curriculum

Pätris Halapuu

Design and Realization of a Sensor-aware Task List Handler for Adaptive Processes in Cyber-Physical Environments

Masters Thesis (30 ECTS)

Supervisors:
Fabrizio Maria Maggi, PhD, University of Tartu
Andrea Marrella, PhD, Sapienza - Universitá di Roma
Massimo Mecella, PhD, Sapienza - Universitá di Roma

Tartu 2015

Design and Realization of a Sensor-aware Task List Handler for Adaptive Processes in Cyber-Physical Environments

Abstract:

Process Management Systems (PMSs) are more and more used to support highly dynamic situations and cooperative processes. Some domains have great diversity of environment variables that can change during the process and therefore affect the workflow in a way that process can not be successfully carried out. Such can be emergency management, health care and other domains involving in most cases in-field actors. In those domains, the frequency and variety of unexpected changes is really high compared to classical business domains that current Business Process Management (BPM) solutions can handle. In 2011, a model and an initial proof-of-concept prototype of SmartPM (Smart Process Management) was introduced in Sapienza - Universitá di Roma that is able to automatically cope with unplanned changes. The continuous screening of the real-world factors is suggested for such domains. A cyber-physical system can be created to automate the screening via *physical-to-digital bridge*. This *bridge* can be a set of tools consisting of sensors, mobile devices and translation layer to extract and feed the real-world information to the digital system. Challenge arises when transferring the information from sensors to the system as the system works with discrete values, but the information gathered by the sensors is continuous in most cases. To target this problem, a concrete solution is proposed and implemented by the author. This thesis explains the architecture and implementation of the sensor-aware task list handler and the web tool approach that was created to solve the discretization challenge of the real-world values. It is also explained how the adaptive PMS, SmartPM, was further developed and updated as the contribution of this thesis.

Keywords:

Adaptive process management system, SmartPM, discretization of real-world objects, cyber-physical system

Sensori-teadliku ülesannete juhtimise süsteemi disain ja realisatsioon kohanduvatele protsessidele küber-füüsilises keskkonnas

Lühikokkuvõte:

Protsesside juhtimise süsteemid leiavad aina enam kasutust toetamaks muutlike situatsioone ja koostööd nõudvaid protsesse. Mõned valdkonnad on väga muutlikud oma keskkonna poolest, võides muutuda protsessi jooksul ja seega mõjutada töövoogu moel, mil protsessiga pole enam võimalik jätkata. Sellistes valdkondades tegelevad näiteks hädaabi, päästekomandod, kiirabi ja teised. Taolised meeskonnad koosnevad üldjuhul vastavalt tegevuskohale opereerivatest osalejatest. Nendes valdkondades on oodamatute sündmuste sagedus ja erinevus väga suur võrreldes tavapäraste äriprotsessidega mida praegused äriprotsesside juhtimise lahendused hallata suudavad. 2011. aastal tutvustati Rooma Sapienza Ülikoolis esialgset SmartPM (Tark Protsesside Juhtija) konseptsiooni tõestavat prototüüpi ja mudelit mis suudab automaatselt kohanduda planeerimata muutustega. Pidev reaalmaailma muutujate jälgimine on vajalik taolistes valdkondades. Küber-füüsilise süsteemi loomine aitab seda automatiseerida, luues *füüsilisest-digitaalseks silla*. See sild võib olla tööriistade kogum mis koosneb sensoritest, mobiilsetest seadmetest ja tõlkivast kihist et võtta reaalmaailmast informatsioon ja muuta see digitaalsele süsteemile mõistetavaks. Probleem tekib sensoritelt tuleva informatsiooni tõlkimisel kuna digitaalne süsteem töötleb ainult diskreetseid väärtuseid, aga sensoritelt tulev informatsioon on üldjuhul pidev. Selle probleemi lahendamiseks pakkus autor välja ja implementeeris konkreetse lahenduse. Käesolev töö tutvustab lähemalt sensori-teadliku ülesannete juhtijat ja veebi tööriista (mis loodi lahendamaks reaalmaailma väärtuste diskretiseermise probleemi) arhitektuuri ja implementatsiooni. Samuti seletatakse kuidas käesoleva töö tulemusena täiendati ja uuendati kohanevat protsesside juhtimise süsteemi, SmartPMi.

Võtmesõnad:

Kohanduvate protsesside juhtimise süsteem, SmartPM, reaalmaailma objektide diskretiseerimine, küber-füüsiline süsteem

Contents

List of Figures

Chapter 1

Introduction

Process Management Systems (PMSs) are more and more used in different domains by many organizations and companies. Those systems in many cases have to support highly dynamic situations and cooperative business processes. Domains like emergency management and health care, for example, have a lot of variables which can change, that are hard to predict and therefore in case of occurrence can prevent the process from being successfully finished. An engine that can adapt processes during run-time was proposed and a first version of it developed in 2011 by de Leoni et al. [2] in Sapienza - Universitá di Roma. This breakthrough enables to manage processes in changing environments that require adaptation. On the other hand, *cyber-physical systems* have become more integrated and used in everyday life (e.g. in smart houses, smart factories, personalized healthcare etc.) as sensors, microcontrollers, PCs, smartphones and other technologies are being in rapid development and becoming more affordable. These developments have made it possible to create a *cyber-physical* environment and build an adaptive process management system where task handling can be automated using the information gathered by the sensors.

1.1 Motivation

Information and Communication Technologies (ICTs) are being integrated into our everyday environment, making the *cyber-physical systems* becoming a reality. A cyber-physical system (CPS) is a system of interconnected and collaborating computational elements controlling physical components that provide real world entities (e.g. people, machines, robots, agents, etc.) with a wide range of innovative applications and services [3]. That is especially driven by rapid developments in smartphones, microcontrolles, sensors, wireless technologies etc. CPSs are designed to support and facilitate collaboration among people and software services on complex tasks. On the other side, the Business Process Management (BPM) discipline has gained an increasing importance in describing complex correlations between distributed systems and offers a powerful representation of collaborative activities [4].

1.2 Problem

The current maturity of Process Management Systems (PMSs) can lead to the application of *process-oriented* approaches in new challenging *cyber-physical domains* beyond business computing, such as personalized healthcare, emergency management, factories of the future and home automation [5]. Such domains are characterized by the presence of a CPS coordinating heterogeneous ICT components with a large variety of architectures, sensors, actuators, computing and communication capabilities, and involving real world entities that perform complex tasks in the "physical" real world to achieve a common goal. In this context, a PMS is used to manage the life cycle of the collaborative processes that coordinate the services offered by the CPS to the real world entities. Moreover, to guarantee a better control over the interaction that PMS has with the real world, it is required to continuously collect contextual information from the specific cyber-physical domain it is employed in. The physical world, however, is not entirely predictable. CPSs do not necessarily operate in a controlled environment, and their collaborative processes must be robust to unexpected conditions and adaptable to exceptions and external exogenous events [6].

1.3 Objective

When adapting a process in a cyber-physical domain, the role of the data perspective becomes fundamental. Data, including information processed by process tasks as well as contextual information, is the main driver for triggering process adaptation, as focusing on the control flow perspective only would be insufficient. Therefore, the screening of real-world objects performed by the physical sensors disseminated in the real world (which play the role of main interfaces towards real-world information) must be taken into consideration when planning, executing and adapting a process in cyber-physical domains. This emphasizes the problem of *how representing digitally real-world objects*, i.e., the problem of making the PMSs aware of the physical world, which is typically continuous, through a *physical-to-digital bridge* that transforms the knowledge extracted from real-world objects in its digital counterpart. This problem is even more important in the case of SmartPM, which exploits automated AI-based techniques to provide self-adaptation features. In fact, it is well known [7, 8] that AI techniques work more efficiently only if they are able to "reason" on a discrete knowledge of the world.

1.4 Contributions

Adaptive process management system called SmartPM was firstly prototyped in 2011 as proof-of-concept to demonstrate the real-time processes self-adaptiveness in domains outside classical business processes. This thesis improves this work by updating the system to ensure proper communication and data exchange with real devices. The author of this thesis developed the prototype from a simulation based application to an actual tool able to work with in everyday processes and environments. The platform was extended with Android application that follows a plug-in architecture. Plug-in approach was considered and realized to increase the usability of the system and make it dynamic for new automatic task types. Automatic task types are the ones in the SmartPM system that get their information, data automatically filled from a sensor or a service. The creation of the new plugins for Android application was documented and explained. A server based middleware was created to solve the communication issue between the SmartPM engine and Android task list handlers. To create the cyber-physical environment, Intel Galileo and Arduino microcontrollers were studied (Ardunio Uno rev3). A setup of different sensors was built with Arduino and firmware created to communicate with Android task list handler over the Bluetooth. Discretization challenge was faced when defining the domain theory for automatic sensor data types and different solutions were studied. The problem was targeted by creating a web tool solution that was integrated with the SmartPM domain theory building application.

1.5 Organization of Thesis

Chapter 2: introduces the concept of Smart Process Management system that was initially developed in 2011. It explains the prototyped system and the communication protocol that the system uses and has to be followed. In this chapter, a cyber-physical environment concept is also explained.

Chapter 3: explains the updated structure and new layers of the SmartPM system which represents one of the contributions of the thesis.

Chapter 4: targets the discretizing challenge in the SmartPM system. The author proposes a concrete solution to solve the problem. After that, the implemented solution is explained based on software layer that maps the continuous values to discrete ones according to the defined rules. The web tool concept and structure are explained which is used to create the discretization rules. The location and Arduino web tools are explained in detail to demonstrate the concrete approach.

Chapter 5: describes the task list handler Android application created for SmartPM. It explains the structure of the Android application and the plugin approached realized in the application. Three plugins are described more in detail. The arduino setup to create the cyber-physical environment is also described in this chapter.

Chapter 6: presents an use case for the system. This chapter also explains the user validation of the SmartPM task list handler, that was carried out based on the use case. Results of the questionnaire, filled out by the users, are analyzed and discussed.

Chapter 7: concludes the thesis by discussing the technologies and techniques used to achieve the goals of this thesis. Possible future works are brought out based on the users feedback and authors opinion.

Chapter 2

Background

Process management systems have become more complex and advanced, being used in large diversity of domains. Process adaptivity at run-time expands the horizon even further, making it possible to use process management systems in vastly changing situations. The concept of the SmartPM enables the process adaptation at run-time. The monitoring of the environment is important in some domains where adaptive process management can be used. To automate the information gathering from the environment, different sensors and microcontrollers can be used, creating a *cyber-physical environment*.

2.1 SmartPM

SmartPM is a self-adaptive PMS which combines several practices: process execution monitoring; unanticipated exception detection (without requiring an explicit definition of exception handlers); and automated exception handling and resolution strategies on the basis of well-established Artificial Intelligence (AI) techniques, including the Situation Calculus [7], IndiGolog [9] and classical planning [8]. The adaptation mechanisms provided by SmartPM allows to deviate at run-time from the execution path prescribed by the original process without altering its process model, a feature that makes SmartPM particularly suitable for managing complex processes in cyber-physical domains.

2.1.1 Communication protocol

SmartPM IndiGolog engine follows a certain communication protocol for notifying actors and initiating tasks. A described order and logic must be followed to ensure a proper work of the engine and development of a process. The protocol is explained in detail to provide better understanding of the work- and taskflow of the whole system. The message flow of the task is illustrated in Figure 2.1

When SmartPM engine creates a new task, it sends out an "assign" message like:

"assign(actorName,[workitem(taskName,taskId,[taskInput],[expectedOutput])])"

where "assign" is a fixed keyword to notify that a new task is assigned; "actorName" is the name of the actor who has to carry out the task - it can contain only 1 element; "workitem" is a fixed keyword to encapsule the task itself; "taskName" is the name of the task that is being initiated (i.e. "go", "measure", "temperature" etc.) - it can contain only 1 element; "taskId" is the identification of the task to keep track of them in the engine - it can contain only one element `id_n` where n is a natural number; "taskInput" is the set of the input arguments of the task - it can contain 0 or more elements; "expectedOutput" is the expected result of the task - it can contain 0 or more elements. When multiple elements are allowed, for example in case of "expectedOutput", then the elements must be separated by comma.

After receiving the "assign" message on the actors' device and on starting the task, the following message must be sent back to the engine:

"readyToStart(actorName,taskId,taskName)"

where "readyToStart" is a fixed keyword for letting the engine know that the message has been received and the actor is ready to start the task.

The engine then confirms that the task can be started and sends the start message:

"start(actorName,taskId,TaskName,[taskInput],[expectedOutput])"

where "start" is a reserved keyword to acknowledge that the corresponding task is being carried out by certain actor.

The task list handler application sends a finish message to the engine when the task is finished:

"finishedTask(actorName,taskId,taskName,[results])"

where "finishedTask" is a fixed keyword for notifying that the task is finished and "results" is the actual result(s) of the task - it can contain 1 or more elements.

The engine then sends out a message to acknowledge the completion of the task:

"ackCompl(actorName,taskId,taskName)"

where "ackCompl" is a reserved keyword to acknowledge the completion of the task.

As the final step of the uninterrupted message flow a release message is sent to acknowledge that the actor is available for the next task:

release(actorName,[workitem(taskName,taskId,[taskInput],[expectedOutput])])

where "release" is a fixed keyword for notifying that the actor is available for the next task.

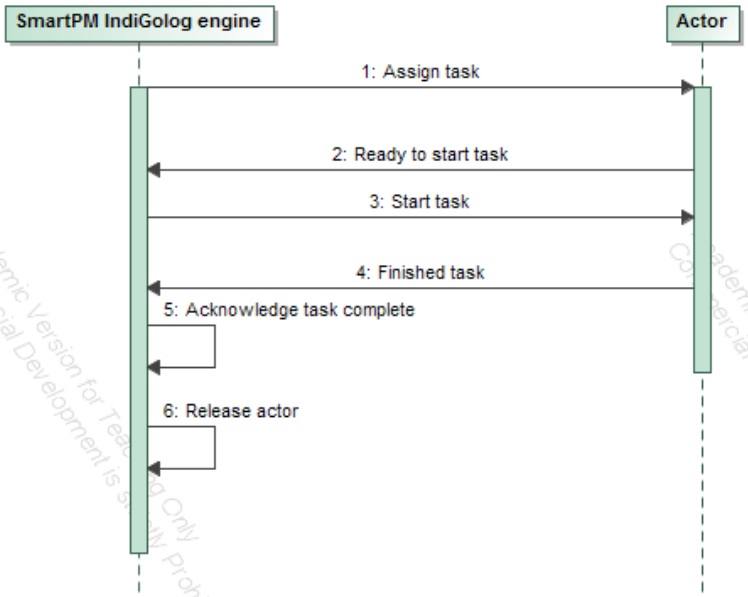


Figure 2.1: Communication protocol, uninterrupted task life cycle.

At any point at run-time, an exogenous event can happen that will trigger the adaptation in the SmartPM engine. Adaptation will also be triggered when the result of a task is not as expected. In those cases, the following adaptation message is sent out by the engine:

adaptStart

which is also a reserved keyword in the domain to acknowledge the devices that a new solution is being calculated. A new solution calculation is a process in the SmartPM engine that is ran by the IndiGolog logical part that finds the shortest path to resolve a situation. For example if the process expects some boolean value task result as "true" but instead it is "false", then solution calculation returns the set of tasks that help to resolve that inconsistency.

When the adaptation has been finished, a notification message is sent:

adaptFinish

which is also a reserved keyword to acknowledge that the adaptation process has been finished.

2.2 Cyber-physical environment

In the last decade, the developments in technologies (mobile, microcontrolles, sensors, wireless, etc.) have led to the point where cyber-physical systems are not fiction anymore but affordable and part of the everyday life. Essential part of the cyber-physical systems is the cyber-physical layer.

The *cyber-physical layer* consists mainly of two classes of physical components: sensors (such as GPS receivers, RFID chips, 3D scanners, cameras, air quality sensors, etc.) that collect data from the physical environment by monitoring real-world objects and actuators (robotic arms, 3D printers, electric pistons, etc.), whose effects affect the state of the physical environment. The cyber-physical layer is also in charge of providing a physical-to-digital interface, which is used to transform *raw* data collected by the sensors into machine-readable events, and to convert *high-level* commands sent by the upper layers into *raw* instructions readable by the actuators. It is important to underline that the cyber-physical layer does not provide any intelligent mechanism neither to clean, analyse or correlate data, nor to compose high-level commands into more complex ones; such tasks are in charge of the uppers layer. In the cyber-physical layer, hardware components, firmware and low level software (mainly devoted to data acquisition) are the main technological ingredients.

Several Do-It-Yourself (DIY) hardware projects have gained popularity during the last decade. Some examples are Arduino[1], Raspberry Pi[2], Intel Galileo[3]. Using any of those proven platforms it is possible to set up a cyber-physical environment. For this thesis, different platforms were considered and Arduino was chosen as it is well documented, has a large community, suitable specifications and has set standards for a large variety of components and sensors.

2.3 Summary

The chapter starts by introducing the main concepts of SmartPM and then continues to explain more in detail the communication protocol of the SmartPM engine. The communication protocol is a crucial part of the whole system as the task list handler logic and the server middleware are highly dependent on it. To have the full understanding of the task message sequence, the protocol must be understood. Another

[1] `http://arduino.cc/en/Main/arduinoBoardUno`

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

[3] `http://www.intel.com/content/www/us/en/do-it-yourself/galileo-maker-quark-board.html`

section in this chapter introduced the cyber-physical environment definition. Following that, it is briefly described the hardware solutions that can be used to create such cyber-physical environments.

Chapter 3

SmartPM system

As one of the objective of this thesis, SmartPM system was re-designed and further developed. Previously the system consisted of three parts - SmartPM engine, designer application and domain simulator realized in Java Swing. As a contribution of this thesis, the SmartPM engine and the designer application were updated and a web server part and an Android task list handler with Arduino sensors were added.

3.1 Structure

SmartPM was updated and currently consists of four parts with different layers.

The kernel of the system is the SmartPM engine that deals with the taskflow logic (coordinates the task flow, sends out task messages as introduced in previous chapter and processes results, calculates the shortest path during run-time to recover the normal process flow defined by the designer when adaptation is needed). The engine is written in a logical programming language called IndiGolog, therefore it can only do calculations in a defined domain. When a process is started, it sends out task messages to the actors according to defined process model. In case of an unexpected task result, the engine calculates the recovery plan (the shortest path to recover the normal process flow defined by the designer). When the recovery plan is found, the engine starts to send tasks to the corresponding actors. The engine is set up and runs on an Ubuntu virtual machine and communicates with actors through a defined port, in this case the port 5555 was used. Through that, a communication channel is established with the web server using SSH reverse tunnel to send and receive messages. The engine was updated to work more efficiently, reducing the number of adaptation-related messages by unifying the message content. Such messages are for example "adaptStart" and "adaptFinish". Previously the engine sent out those messages for every actor, for example six actors meant sending out six "adaptStart" messages. The updated engine sends "adaptStart" and "adaptFinish" message out only once per occurrence, no matter how many actors, to avoid the communication overhead with the web server. The web server middleware then handles the message and forwards it to all the affected actors.

On top of the SmartPM engine is the graphical designer application (Figure 3.1) where user can define the domain theory for the engine (actors, their capabilities,

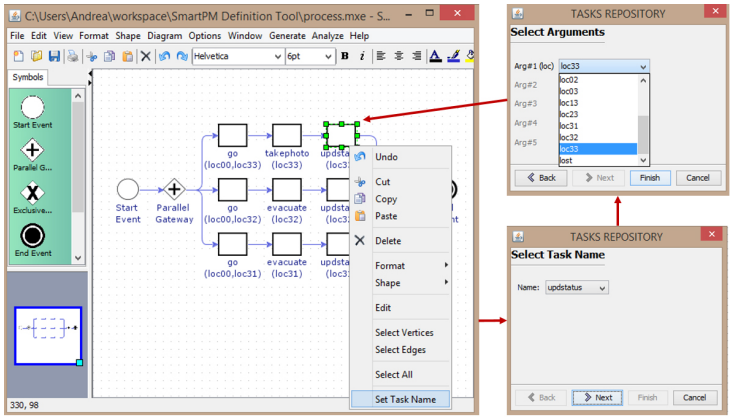


Figure 3.1: Designer view.

data types and data values - e.g. location, temperature, battery level etc.), create the processes and run it. The designer application was updated with a new feature - the web tool feature. During the design time, the designer can invoke the online web tools to have data listing generated for certain data type. Alternatively, the designer can define the domain theory manually by typing the values to the designer application. The designer application was also updated to upload the domain theory knowledge file (also known as process schema) to the web server. The domain theory knowledge file is generated by the designer application every time a process is started. The file is in xsd format and contains the data types and values defined during the process design time. Based on the domain theory knowledge file, the web server takes care of sending correct messages to the task list handlers. This improvement helps to reduce communication load between the engine and the web server as only minimal task information messages (as described previously in the section 2.1.1 Communication protocol) are sufficient for integrity.

A web server was added to the structure that enables the communication between task list handlers and the SmartPM engine. To ensure lightweight and fast communication, the Google Cloud Messaging (GCM) service is invoked in the web server scripts. The main function of the web server is to be the middleware at runtime between the SmartPM engine and actors devices. From the virtual machine that hosts the SmartPM engine, SSH reverse tunnel is initiated on port 5555 to the web server. The SmartPM engine sends messages to that port, then the web server parses those messages into the right format for the task list handler, generates and stores the tasks as XMLs. As a next step, a database lookup is made to get the GCM registration ID and a GCM service request is made to send the task to the corresponding actors device. It handles all the communication according to the protocol between the actors' devices and the SmartPM engine. After the submission of the task, the web server

again parses the message to the right format for the SmartPM engine and sends it to port 5555. The web server also holds a set of tools for discretizing values used during the design time, called the web tools.

The task list handler is an Android application for version 4.0 and greater. The task list handler application registers an user as an actor to the web servers' database and then the device is ready to receive push notifications from the GCM service. Push notifications contain information about the task for the relevant actor. On the Android device, a form is generated according to the task and displayed to the user. Forms can contain automatically and manually filled fields. In case of automatically filled field, a sensor is being used for gathering data - either built-in (i.e. GPS, gyroscope, microphone, etc.) or external (i.e. Arduino, Intel Galileo, Rawsberry Pi, etc.). For example when a task requires the location of an actor, the GPS plugin can be used to get the position of the device automatically from the built-in GPS sensor. The coordinates are then converted at run-time by the task list handler from continuous values (latitude and longitude) to discrete value (e.g. *loc00*). After the task is finished, the result of the task is sent to the web server for processing. The web server formats the result and forwards it to the SmartPM engine.

The Arduino microcontroller is used to gather different environmental data, for example humidity level, temperature, VOC gas level, HCHO level etc. A bluetooth board is added to the setup to enable wireless communication between the task list handler and the Arduino board. The instructions of how to set up the whole system can be found in the Appendix B.

3.2 Summary

This chapter makes an overview of the updated SmartPM system and discusses the developments done as contribution of this thesis. Each part of the initial system is brought out and the improvements relevant to the part are explained. As the last part of this chapter, new additions to the system are introduced.

Chapter 4

Discretizing challange

The SmartPM engine is written in a logical programming language called IndiGolog. For this reason, the domain in which the calculations are made must be defined and values must be discrete. In real world, measurable values are continuous. Problem arises when automating processes, therefore mapping from continuous to discrete values must be done. To target this problem, a web tool approach is introduced and realized.

4.1 Web tool

When designing a business process in SmartPM, the designer has to define the domain, the types, and the values of data. The knowledge about the domain is essential for the whole SmartPM system as based on that information the process is built and calculations in the SmartPM engine are made. One possibility to define the domain is to enter all the data manually. That might be time consuming and typing errors can be made. For minimizing those problems, a web tool approach was used.

While defining the domain, the designer can choose to add and invoke different web tools to make adding data types and values take less effort (Figure 4.1). When the URL of the web tool is invoked in the designer application, a web page is opened with the default browser. In the web tool, the designer can mark the real life continuous values on a graphical element (i.e. map, slider, thermometer, etc.) and then change the discrete values names if necessary. When ready, the designer can press a button to create and store an XML of the defined rules. The URL of the generated XML is then displayed. The designer can then copy the address of the rules and paste it to the designer application. Let's say an user wants to add location data of 10 times 10 grid to the system. That means manually he or she would have to type in 100 unique data values to the location type in the designer application. As an alternative, the designer can invoke the location web tool (Figure 4.2) and create 10 times 10 grid with a few clicks. The location web tool then generates the data based on the areas marked visually on the map. The web tool checks if the values are correct and then returns an XML with the data. The designer application knows how to parse that XML into suitable format and store it as domain information.

Each web tool must produce discretization rules - an XML with the main tag including

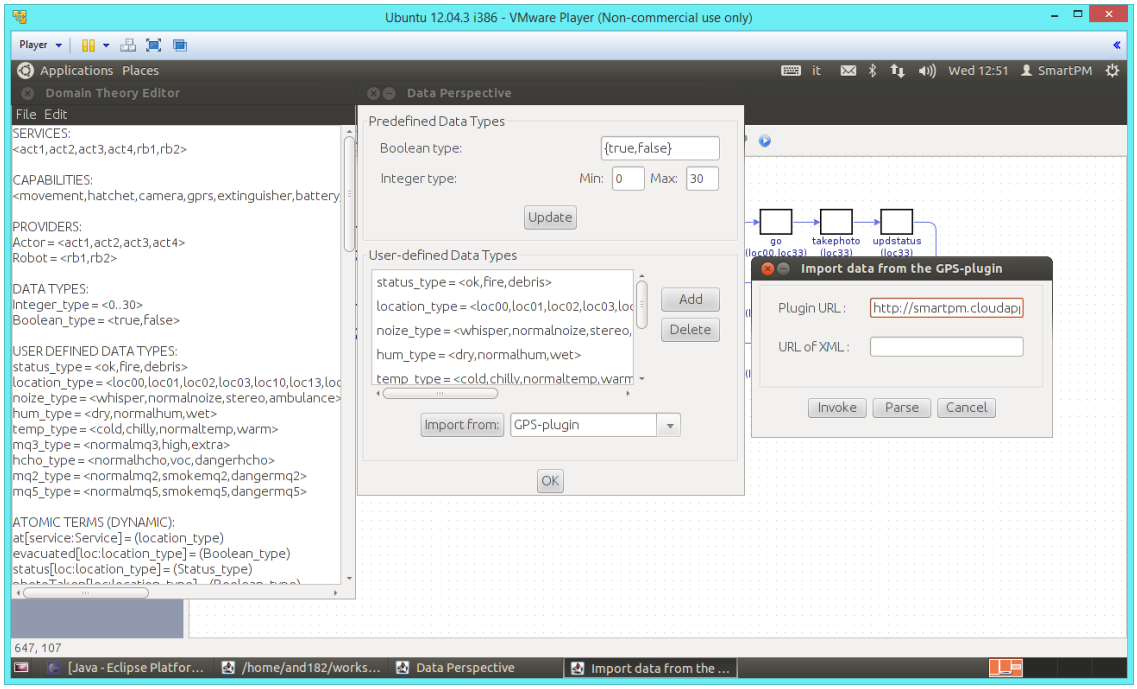


Figure 4.1: The designer tool - add data.

the keyword *_type* for letting the designer application know during the parse time about the new data type. The following tag *lib* is relevant for the Android application to acknowledge the plugins' name and URL. The following tags' names must be *data_value*, that contains the mapping information of this data value. Also every *data_value* tag must have an attribute *value* defined - it marks the discrete value. The *mappingRule* attributes are used for discretizing the real world values. The number of mapping rules in a data_value tag are dependant on the data dimensions and the task list handler Android library that uses those rules to discretize the values - for example the location data_value has four rules that define one area (top left and bottom right latitude and longitude values) but temperature data_value has two rules to define one temperature range (minimum and maximum values). These discretization rules are used by the task list handler to translate the incoming sensor data from continuous to discrete value. A sample structure of the rules XML is following:

```
1   <data_type>
2       <lib
3           name="plugIn name"
4           url="http://Android.Plugin.URL"/>
5       <data_value
6           mappingRule1=""
```

```
7           ...
8           mappingRulen=""
9           value="discreteValue"/>
10      <data_value ... />
11      ...
12  </data_type>
```

The designer application is able to parse new data types and values that are structured in that way. Being guided by that, the new web tools that produce rules XMLs can be created that are compatible with the SmartPM designer application. The URL of the rules XML file is added to the data type field in the process schema. A sample location type in the process schema is the following:

```
1   <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    ...
4    <xs:simpleType name="location_type"
5                   url="http://domain.com/rules.xml">
6     <xs:restriction base="xs:string">
7      <xs:enumeration value="discreteValue1"/>
8      ...
9      <xs:enumeration value="discreteValueN"/>
10    </xs:restriction>
11   </xs:simpleType>
12  </xs:schema>
```

The task XML creation script on the web server side takes the URL from the process schema and adds it to the task field. When the URL is set, the Android task list handler task form generator knows that this fields' type is automatic and the discretization rules can be found from the pointed XML (explained more in detail in chapter 5). Continuous sensor value is converted to discrete value on the Android task list handler at run time using the same rules XML.

Currently, three web tools have been created to demonstrate the approach - a location tool, a noise tool and an Arduino tool. The location tool and the Arduino tool are explained more in detail as one of them has two dimensional data and the other one has one dimensional data. The noise tool is similar to the Arduino tool, having one dimensional data.

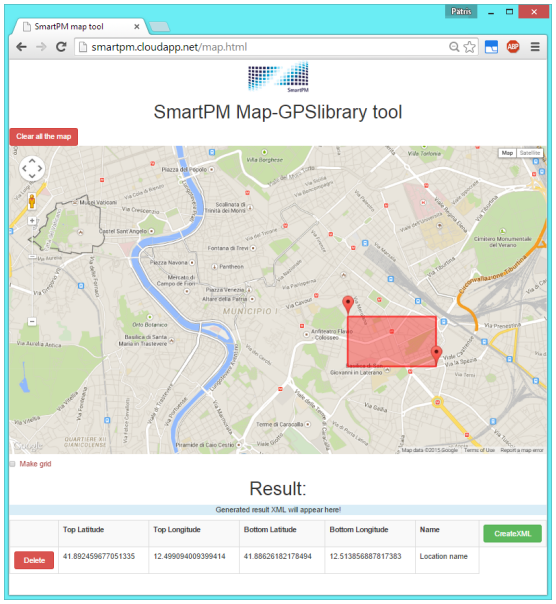


Figure 4.2: Location web tool.

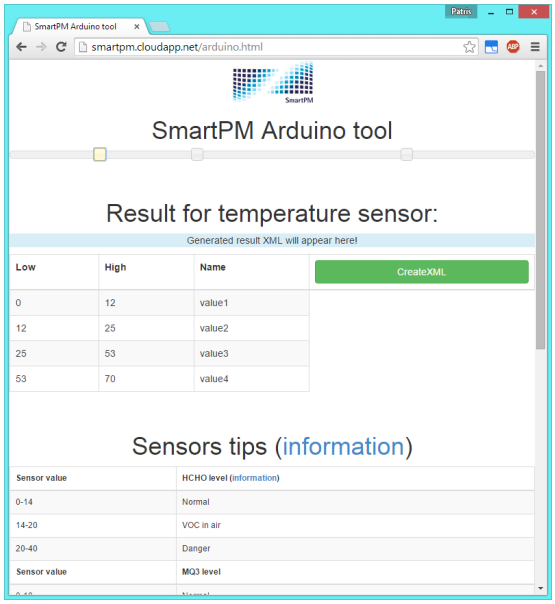


Figure 4.3: Arduino web tool.

4.1.1 Location web tool

The designer can take advantage of the location web tool whenever there is a process that is dependant on the physical location that can be acquired with the GPS. Using the location web tool, the designer can set discrete data values to the areas defined by the GPS simply by clicking on the map in the location web tool (Figure 4.2). The location web tool uses the Google Maps API and provides two different means to divide the marked area: divide it into a grid or have it as a singleton. The number of areas that the user can mark on the map is not limited. For marking the area, the user has to mark the top left corner by clicking on the corresponding location on the map and the bottom right corner (in that exact order) of the desired area on the map. In case of making a grid, the user is asked for the number of rows and columns that the marked area has to be divided into to create the grid. A table of the marked areas values is dynamically generated at the bottom of the webpage to show the results. The generated rules XML of the location web tool looks like:

```
1  <location_type>
2      <lib
3          name="GPS lib"
4          url="http://smartpm.cloudapp.net/SmartPM_libGPS.apk"/>
5      <data_value
6          topLat="41.894" topLon="12.498"
```

```
7           botLat="41.892" botLon="12.502" value="loc00"/>
8       <data_value ... />
9       ...
10  </location_type>
```

In case of location data, the sensor information is two dimensional - it has a latitude and a longitude. To discretize the location value the users' current location is compared against the defined areas coordinates in the XML. Four comparisons must be made to make sure that the users' current location is in the defined area. The formula to check that is the following:

```
1   (currentLatitude < topLat) && (currentLatitude > botLat) &&
2       (currentLongitude > topLon) && (currentLongitude < botLon)
```

This check is done in the Android task list handler against every *data_value* or until a match is found. When a match is found, the discrete *value* is returned.

4.1.2 Arduino web tool

The Arduino web tool is useful when the designer needs to define a process where Arduino sensors are used. For example if the process requires environment temperature information, the designer can set the temperature ranges in the web tool simply by moving the sliders on the slider bar and writing the desired discrete values to the corresponding fields (Figure 4.3). The Arduino web tool has been generalized for handling multiple different sensors data discretization rules generation in the same tool. Most Arduino sensors provide one dimensional data, therefore those sensors were considered for the tool. For visualizing the ranges of continuous values, a slide bar was used. The Arduino web tool first asks for the sensor type the user wants to create discretization rules for. After that, the number of sliders is asked to create the number of different ranges the user needs. According to the inserted information the slider bar is created. The user can then move the sliders and name the discrete values in the dynamic table under the slider bar. The generated XML of the Arduino web tool for temperature sensors looks like:

```
1   <temp_type>
2       <lib
3           name="arduino lib" keyword="temp"
4           url="http://smartpm.cloudapp.net/SmartPM_libHumid.apk"/>
5       <data_value
6           low="0" high="10" value="value1"/>
7       <data_value ... />
```

```
8       ...
9   </temp_type>
```

An extra attribute is taken into use in the *lib* tag called *keyword*. It specifies which Arduino sensor is used for the task list handler application. The *keyword* must be exactly the same as the variable name given for this sensor by Arduino.

4.2 Summary

This chapter proposes a solution for the discretization challenge that is in cyber-physical systems - how to map physical world continuous values to discrete computable values for the logical engine. The proposed solution, called web tool, was implemented for three different data types and described in detail in this chapter.

Chapter 5

Task list handler

The SmartPM task list handler is a mobile application for Android devices from version 4.0. It is an essential tool for actors to exchange data with the SmartPM engine - to receive tasks and send back results. The task list handler has a modular architecture with a plugin approach - this is for automating data collection from sensors for the tasks.

5.1 Architecture

The SmartPM task list handler application can be divided into three components - Google Cloud Messaging handler; task form generator; plugins manager.

5.1.1 Google Cloud Messaging handler

The communication from the SmartPM engine to the task list handler is established via Google Cloud Messaging (GCM) service. It sends lightweight push notifications to the device to notify about a new task or the status of the process. To handle the push notifications coming from the GCM service, a device must first be registered with the actors' name. For that, a simple form (Figure 5.1) is displayed to the user when the application is ran for the first time. The actors name and the GCM reg. id. strings of the devices are stored into the web server. When the user wants to log out, again the GCM service is invoked and the user is removed from the web server database. All the user names must be unique - this requirement comes from the SmartPM engine. After logging in, the user is kept logged in on the device and is directed to the task view (Figure 5.2). The Google Cloud Messaging handler parses the incoming push notifications. When a push notification with new task is received, then the task name and the status are updated accordingly (Figure 5.3). A button to start the task is visualized at the bottom of the view. A task form is generated after the task push notification is received and the actor has started the task with a button click. The task push notification must have the following format:

```
1   taskName|Task name here;URL|http://smartpm.cloudapp.net/task.xml
```

where *taskName* is a fixed keyword for marking the task name and after the pipe sign (|) is the defined task name. Semicolon separates variables. *URL* is a fixed variable

name for marking the URL of the task XML. *http://smartpm.cloudapp.net/task.xml* is the URL of the task XML. The other messages that control the process - *start*, *pause*, *resume* - are parsed by the server into the following format and sent via GCM service to the devices:

```
1  taskName|start;
2  taskName|pause;
3  taskName|resume;
```

The *start* message implies that the SmartPM engine is ready to receive the result of the task from the actor, therefore the *stop* button is enabled in the application for submitting the results. The *pause* message notifies the actors that something has caused the process adaptation and therefore are put on hold. That means that the tasks are paused and results can not be sent out to the SmartPM engine until the *resume* message is received or a new task has been assigned to the actor. *stop* and *resume* messages are sent to all the actors simultaneously.

5.1.2 Task form generator

The task XML is then parsed from the defined URL and the form of the task is generated. The task XML must have the following structure:

```
1  <xmlgui>
2      <form id="taskId" name="taskName"
3          actor="actorName" submitTo="serverResponseURL">
4          <field name="fieldId" label="fieldLabel"
5              type="fieldType" required="requiredBoolean"
6              options="fieldOptions" autoLib="pluginURL"
7              rules="discretizationRules"/>
8          <field ... />
9          ...
10     </form>
11 </xmlgui>
```

where:

- *xmlgui* tag marks the whole task XML to be parsed.
- *form* tag nests the whole form and includes attributes:
  - *id* - stands for task id, its value is the same as the *taskId* sent out by the SmartPM engine;

  - *name* - stands for the task name, its value is the same as the *taskName* sent out by the SmartPM engine;
  - *actorName* - stands for the actor name who the task is assigned to, its value is the same as the *actorName* sent out by the SmartPM engine;
  - *submitTo* - stands for the URL address that the response must be sent back to, its value is the URL of the PHP script that receives and parses the response.

- *field* tag marks one field in the form of the task. All the described fields must be present in every task. If the attribute has no value, then it is left as an empty string. The *field* tag has seven different attributes that are necessary to define a concrete field for the Android task list handler.
  - *name* attribute stands for the name of the field that is identified by the task form generator, therefore the value must be unique and never empty.
  - *label* attribute stands for the label that is displayed to the user, its value can be an empty string.
  - *type* attribute stands for the field type. Five different field types have been defined for the application. The value of the *fieldType* must be one of the following:
    * *text* - in that case, a textbox is created and displayed to be filled out;
    * *numeric* - in that case again a textbox is created and displayed but the user can only input numbers;
    * *choice* - in that case a dropdown box is created and displayed to the user;
    * *boolean* - in that case a checkbox is created and displayed to the user;
    * *auto* - in that case a label is created that displays the value of the automatic field.
  - *required* attribute defines if the field is required or not, *requiredBoolean* value can be either *Y* for true or *N* for false.
  - *options* attribute is a non-empty string only when *type="choice"* - *fieldOptions* is a string including the elements of the dropdown box. The elements must be separated by the pipe sign (|).
  - *autoLib* attribute stands for the plugin URL - *pluginURL* is a non-empty string only if the type is *auto* - then the value is the URL where the

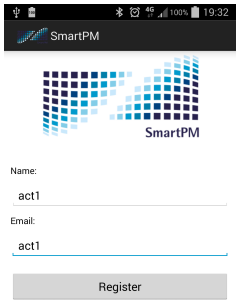


Figure 5.1: Registration view.

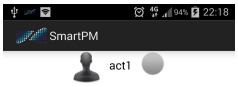


Figure 5.2: Task view, no task.

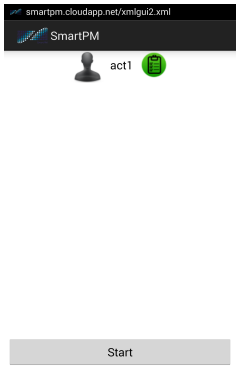


Figure 5.3: Task view, new task.

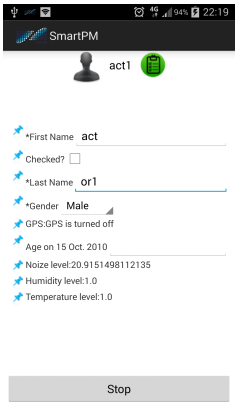


Figure 5.4: Task view, new task started.

application can download the plugin from to use the required sensor.

– *rules* attribute stands for the discretization rules for the plugin and is a non-empty string only when the *type* is *"auto"* - then the value is the URL to the rules XML file.

Each *field* is parsed into an object according to the type *type*, for each *type* there is a corresponding class with methods for dealing with the object. Figure 5.4 illustrates the sample form displayed to the user.

5.1.3 Plugins manager

Plugins are managed mainly by one class. When *field* is automatic (*auto*) type, then at runtime, the application checks if the plugin defined in the task XML (filename

value of attribute *autoLib*) already exists in the location *getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS) + "/SmartPM/"*. If the file is not in the directory, it is downloaded from the defined URL to that location on the device. The class *MyClass* of the plugin is then loaded and a new instance is created. The plugin approach is described more in detail in next section.

5.2 Plugin approach

Automating the data collection for the task can be done using different sensors. In this thesis sensors are divided into two categories - built-in sensors of the Android task list handler device (GPS, microphone, camera etc.) and external sensors that are connected through the Arduino microcontroller (VOC gas sensors, temperature sensor, humidity sensor, etc.). To allow using different kinds of sensors, a plugin approach was realized.

The task list handler application loads plugins at runtime. Therefore, new plugins can be created and added to the domain theory and workflow according to the need without changing anything in the task list handler applications' code. Plugins must implement an interface called *LibInterface*, defined in the SmartPM task list handler for Android. The most important method that a plugin must override is *useMyLib* that can access the application context, update the corresponding *TextView* with the collected data and passes the URL of the discretization rules XML file for parsing and converting the values accordingly in the plugin.

```
1  public interface LibInterface {
2      public String useMyLib(Context context,
3          TextView mAutoLabel, String rules);
4      public String getName();
5      public String getType();
6  }
```

After starting the task, the form is being parsed by the application to display it to the user. When the automatic field is being parsed, the application checks for the plugin from the device storage and if it is not present, then it downloads it. A new instance of the main class of the plugin - *MyClass* is created and the application context, the label that is updated with automatic data and the discretization rules URL are passed to that class object.

```
1  final DexClassLoader classloader =
2      new DexClassLoader(libPath + fileName,
3      tmpDir.getAbsolutePath(), null,
```

```
4       this.getClass().getClassLoader());
5   final Class<Object> classToLoad =
6       (Class<Object>) classloader.loadClass("ut.ee.SmartPM.lib.MyClass");
7   LibInterface obj = (LibInterface) classToLoad.newInstance();
8   obj.useMyLib(mContext, mAutoLabel, rules);
```

Class *MyClass* must be in the package *ut.ee.SmartPM.lib*. On the plugin side, the *rules* XML URL is parsed and the rules objects are created. For example the temperature rules class in the plugin is:

```
1   package ut.ee.SmartPM.lib;
2   public class RulesObject<Low, High, Name> {
3       private Low low;
4       private High high;
5       private Name n;
6       public RulesObject(Low low, High high, Name n){
7           this.low = low;
8           this.high = high;
9           this.n = n;
10      }
11      public Low getLow(){ return low; }
12      public High getHigh(){ return high; }
13      public Name getName(){ return n; }
14      public void setLow(Low low){ this.low = low; }
15      public void setHigh(High high){ this.high = high; }
16      public void setName(Name n){ this.n = n; }
17  }
```

After the rules objects are created, the plugin enables the sensor and the incoming data is discretized by comparing the rules objects and the data provided by the sensor. To elaborate it, GPS and microphone examples are explained.

5.2.1 GPS and microphone plugins description

Several plugins have been created for the task list handler to demonstrate the plugin approach using sensors that are built-in the mobile devices. The GPS capability of the mobile device allows for automatically filling out the current location of the device. Using the microphone, it is possible to automatically get the current noise level nearby the device. This section explains how these plugins work.

When a task requires the location of the actor, in the task XML, in the location field

it, is possible to mark it as an automatic field and give the URL of the GPS plugin. An example automatic GPS field is:

```
1  <field
2      name="gps" label="GPS:" type="auto"
3      required="N" options=""
4      autoLib="http://smartpm.cloudapp.net/SmartPM_libGPS.apk"
5      rules="http://smartpm.cloudapp.net/blankgpsrules.xml"/>
```

The GPS plugin is then downloaded to the Download/SmartPM folder of the device from the URL if the plugin is not already there. The typical plugin is relatively lightweight, around 300KB in size. While the task is being carried out, the GPS location field label is updated by the plugin. There are four classes in the GPS library *ParseXML*, *RulesObject*, *CurrentLocationListener*, *MyClass*. *ParseXML* class parses the rules from the rules XML file (that is defined in the task XML) and creates new instances of *RulesObject* as explained in previous section. In this case, *RulesObject* is a simple class that has getters and setters for the GPS rules (top and bottom latitude and longitude values, discrete name value matching the location name in the SmartPM designer). An example of rules XML is the following:

```
1  <location_type>
2  <lib
3      name="GPS lib"
4      url="http://smartpm.cloudapp.net/SmartPM_libGPS.apk"/>
5  <data_value
6      topLat="41.894" topLon="12.499"
7      botLat="41.884" botLon="12.517" value="rome"/>
8  </location_type>
```

CurrentLocationListener implements *LocationListener*, gets latitude and longitude coordinates from the GPS sensor, matches the continuous values to discrete values according to *RulesObject* and updates the *TextView* with the value. *MyClass* is the most important class - it implements *LibInterface*, calls out *ParseXML*, starts *LocationManager* and calls out *CurrentLocationListener* passing it the mapping rules.

The noise plugin works in a similar way like the GPS plugin. However instead of *CurrenLocationListener*, it has a class *DetectNoise*. *ParseXML* works in the same way, *RulesObject* is adapted according to the dimensions of the data getters and setters for high, low and name values as pointed out in the previous section. *DetectNoise* class starts *MediaRecorder* and calculates the decibels of the input. *MyClass* implements *LibInterface*, calls out *ParseXML*, calls out *DetectNoise*, maps the values to the discretized value and updates the *TextView* with the value.

5.2.2 Arduino plugin description

Different readings from the environment can be gathered with different Arduino sensors gas levels, temperature, humidity etc. For connecting Arduino with the Android task list handler, Bluetooth is used. On the Arduino part, the Arduino Uno board, the Bluetooth shield and sensors are connected. A Bluetooth connection is configured and enabled and on successful connection, all the sensors data is being sent:

```
1   ...
2   void loop()
3   {
4     float humidity = TH02.ReadHumidity();
5     blueToothSerial.print("hum=");
6     blueToothSerial.println(humidity);
7
8     float temper = TH02.ReadTemperature();
9     blueToothSerial.print("temp=");
10    blueToothSerial.println(temper);
11
12    int hchoSensorValue=analogRead(A0);
13    float hchovol=(hchoSensorValue*4.95/1023)*10;
14    blueToothSerial.print("hcho=");
15    blueToothSerial.println(hchovol);
16    ...
17  }
18  ...
```

On the Android task list handler part, the plugin for communicating with Arduino is based on the Bluetooth communication. Arduino is programmed to send all the sensors data to the Bluetooth, therefore the filtering of the necessary sensor data acquired by the task has to be done on the Android task list handler. The key-value pairs of all the sensors data are stored and updated in the shared preferences[1] of the task list handler application. This ensures that one task can simultaneously automatically use different sensors data coming from Arduino. If there are more than one automatic field in the same task that use the same Arduino plugin, then the plugin checks if it is already in use and receiving data or not (as Bluetooth allows only one socket at a time). If not, then the connection is established and data is received and saved to the shared preferences. If the Arduino plugin is already in use (sensors data is updated in the shared preferences), then the sensor data is read from the shared preferences. The rules XML defines which sensor data is used by the *keyword*

[1] `http://developer.android.com/reference/android/content/SharedPreferences.html`

value. For example the following rules XML defines *keyword="temp"* meaning that these discretization rules apply to the Arduino temperature sensor data:

```
1  <temp_type>
2  <lib name="arduino lib" keyword="temp"
3      url="http://smartpm.cloudapp.net/SmartPM_libHumid.apk"/>
4  <data_value low="0" high="10" value="cold"/>
5  ...
6  </temp_type>
```

The task list handler Arduino plugin has three classes *RulesObject*, *ParseXML* and *MyClass*. *RulesObject* and *ParseXML* have the same functionality as described previously for GPS and noise plugins. *MyClass* calls out *ParseXML*, checks if the plugin is already being used, or not. If not, then configures and establishes the Bluetooth connection, starts listening for the Bluetooth input stream. As a final step, *MyClass* maps the continuous values to the discrete values and updates the corresponding *Textview* field with the discretized values.

5.3 Arduino sensors

The sensor-aware cyber-physical environment is created by automating the environmental data collection for tasks using the Arduino electronics platform. To create the sensor-aware cyber-physical environment, the following setup was constructed: Arduino Uno R3 board[2], Seeed Studio Bluetooth shield V2.0[3], Grove HCHO sensor[4], Grove MQ2, MQ3 and MQ5 gas sensors[5], Grove temperature and humidity sensor[6]. Grove MQ9 gas sensor was considered also for the setup but due to the limited number of analog pins that are used by the Grove sensors it was left out.

Grove HCHO sensor measures the volatile organic compound (VOC) gas concentration in the air, for example toluene, benzene, methanal etc. Grove MQ2 sensor measures combustible gas and smoke concentration in the air. Grove MQ3 sensor measures alcohol vapour concentration in the air. Grove MQ5 sensor measures natural gas, town gas, liquified petroleum gas (also known as propane or butane), etc. concentration in the air. Grove MQ9 sensor measures carbon monoxide, coal gas, liquefied gas concentration in the air.

[2] http://arduino.cc/en/Main/arduinoBoardUno

[3] http://www.seeedstudio.com/wiki/Bluetooth_Shield_V2.0

[4] http://www.seeedstudio.com/wiki/Grove_-_HCHO_Sensor

[5] http://www.seeedstudio.com/wiki/Grove_-_Gas_Sensor

[6] http://www.seeedstudio.com/wiki/Grove_-_Tempture&Humidity_Sensor_(High-Accuracy_&Mini)_v1.0

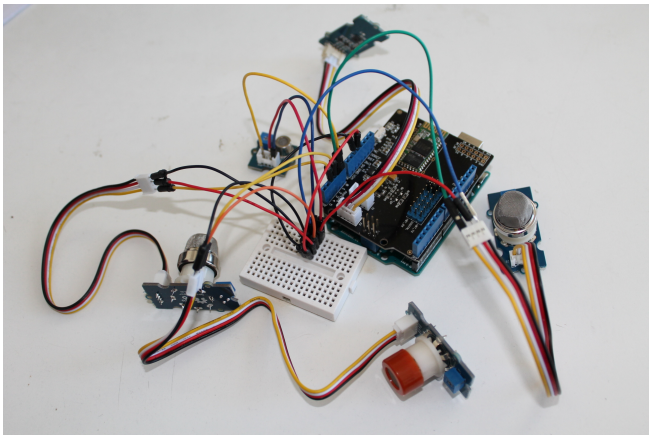


Figure 5.5: Arduino sensors.

The described sensors setup is practical in many use cases when there might be fire hazard, including gas leakages from buildings, vehicles, gas pipes etc. Some of the sensors need pre-heating, according to data sheets up to 24h, to show the correct values. During the system tests it came out that actually it takes up to 15 minutes of pre-heating to already get accurate results. Pre-heating means having the sensor working for some time with defined voltage for the calibration.

The described Arduino setup is relatively energy greedy, which means that for powering it, the typical one 9V battery is not sufficient. The setup operates with 5V, using averagely 550mA of power. To have the setup working non-stop for the full average 8h working day, the battery pack was reconsidered and remodeled into 8 pieces AA battery pack that can provide this kind of power for non-stop for 10 hours, assuming one 1.5V AA battery is around 2Ah.

5.4 Summary

This chapter describes the proposed task list handler for the SmartPM system and the implementation of this mobile application for Android operating system. The plugin approach is described alongside with other important features of the task list handler. For demonstrating the approach, three implemented plugins are described in detail. As automating tasks in cyber-physical environment requires hardware side as well, an Arduino set is introduced.

Chapter 6

User validation

User validation was carried out to evaluate the usability of the SmartPM task list handler. A scenario for six participants was created. After playing through the whole scenario, users were asked to fill out the questionnaire which was based on the USE questionnaire [10].

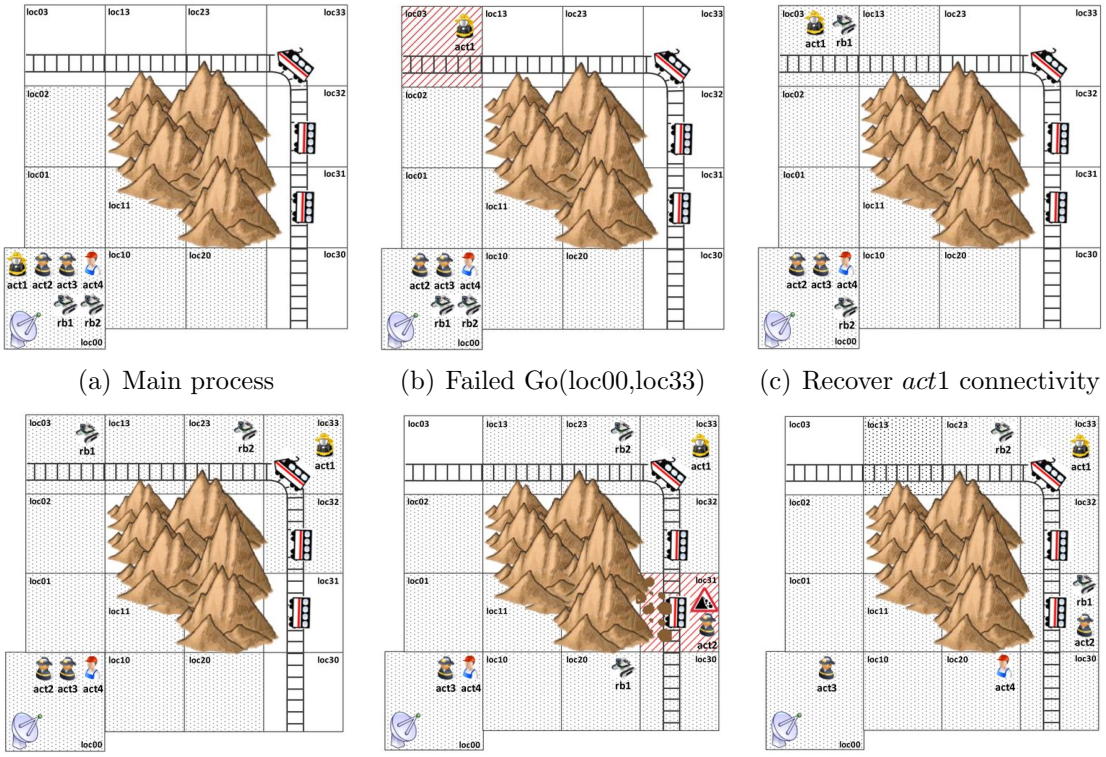
6.1 Scenario

As an application scenario, the *emergency management* domain was considered. Let there be a team of six actors in a disaster location to assist potential victims. A cyber-physical system at hand is composed of mobile devices, robots, wireless communication technologies, Arduino sensor systems and process management system SmartPM. A response plan is encoded as a process and executed by a process management system and task list handler deployed on mobile devices, helping to coordinate the activities carried out by the team. The following case study involves an improved disaster management inspired by the WORKPAD project[1].

The disaster area is divided into a four times four grid-type map where a train has come off the rails. The train is composed of a locomotive and two coaches (located at loc33, loc32 and loc31 respectively). The situation is described in Figure 6.1(a).

The goal of the incident response plan is first to take air measurements to make sure that there are no dangerous gases in the air. When the air measurements are done, the team starts to evacuate people and take pictures of the disaster to evaluate possible damages to the locomotive. The team consist of four human actors (act1, act2, act3, act4) and two robots actors (rb1, rb2), all starting from the location loc00. All the actors are equipped with Android mobile devices with task list handler installed and configured. Each actor has specific capabilities - act1 is able to extinguish fire, take pictures and measure gas levels in the environment with Arduino sensors; act2 has also an Arduino sensors kit to measure the environment gas levels and can evacuate people; act3 can evacuate people; act4 can fix robots; rb1 and rb2 are designed to remove debris and provide fast wireless Internet connection. To carry out the response plan, all actors must have Internet connection. Fast wireless Internet connection is provided by a fixed beacon in the loc00. The dotted squares on Figure 6.1(a) represent

[1] `http://www.dis.uniroma1.it/~workpad`



(a) Main process (b) Failed Go(loc00,loc33) (c) Recover *act*1 connectivity

(d) Successful Go(loc03,loc33) (e) Recover *act*2 connectivity (f) Final stage

Figure 6.1: A train derailment situation; area and context of the intervention. [1]

areas that are covered by the fast wireless Internet connection provided by the beacon. The other squares have mobile Internet coverage, good enough to send and receive simple messages for the task list handler, although due to low bandwidth there might be some delay. To eliminate possible delay due to low connectivity, a robot is sent to restore better Internet quality when needed. Each robot can move in the area, but has to be always connected to the main network. This is guaranteed if the intersection between the squares covered by the main network and the squares covered by the robot connection is not empty.

Based on the given information, it is possible to define and configure a concrete incident response plan for the scenario using the BPMN [11] modelling language as shown in Figure 6.2.

The created process is composed of three sets of parallel branches. The first set is composed of two parallel branches with tasks instructing actors to measure noise level and the temperature. The second set is also composed of two parallel branches with tasks instructing actors to take air quality measurements with the Arduino set. The

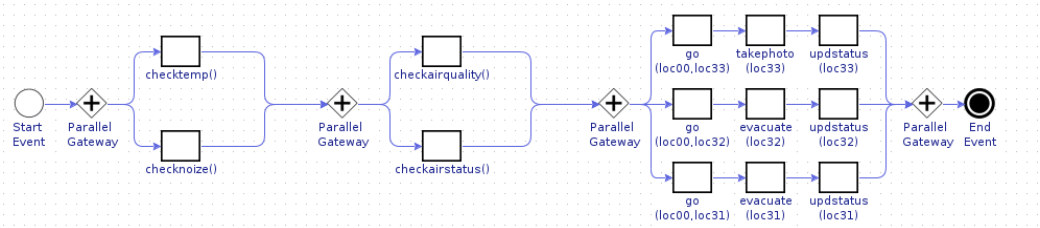


Figure 6.2: Case study - main process.

third set is composed of three parallel branches with tasks asking to evacuate people and take pictures to later assess the accident.

The environment is highly dynamic, therefore there is a wide range of exceptions that can occur. Because of that, there is not a clear anticipated correlation between a change in the context and a change in the process. For example when actor act1 is sent to the locomotives' location loc33 but instead he or she reaches loc03. This makes him or her located at a different position than the desired one and out of the fast wireless Internet range (Figure 6.1(b)). As all participants need to be connected to fast wireless Internet connection to execute the process, the PMS has to first find a recovery procedure to provide act1 with fast wireless Internet connection, and then find a way to re-align the process.

The SmartPM engine finds the following recovery solution - send one robot to loc03 (Figure 6.1(c)) in order to re-establish the Internet connection to actor act1, then instruct the second robot to go to location loc23 in order to extend the network range to cover the locomotive's location loc33. Finally, actor act1 is asked to go to location loc33 again (Figure 6.1(d)). The corresponding updated process part that required adaptation is shown in Figure 6.3, with the encircled section being the recovery (adaptation) procedure.

After the recovery procedure has been done, the original process is resumed to its normal flow. For example actor act2 can be sent to location loc31. However, even if act2 completes its task as expected (Figure 6.1(e)), a further exception is thrown. In fact, act2 is out of the fast wireless Internet connectivity range and, again, the SmartPM engine sends a task to the first robot to move to location loc20 in order to re-establish fast wireless Internet connection to actor act2 (top of Figure 6.1(c)). Now actor act2 can start evacuating people from loc31.

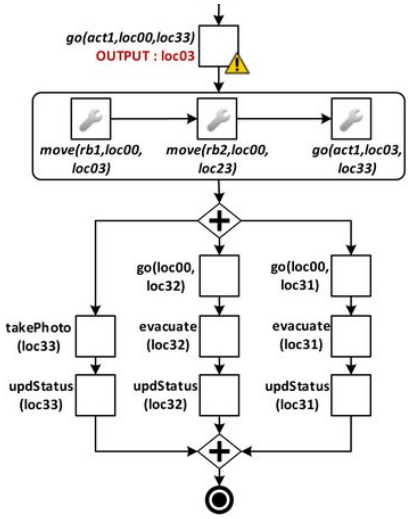


Figure 6.3: Case study - adapted process part. [1]

6.2 Results

Six people took part to the user validation. The domain was explained to the test subjects and the equipment was handed out (Android devices with task list handler installed, Arduino sets) to act out the scenario. The scenario simulation took place at Tartu Toomemäe. Afterwards they were asked to fill out a questionnaire to evaluate the task list handler. The questionnaire consisted of four topics - usefulness, ease of use, ease of learning, satisfaction - with 19 statements in total with one extra field to write any other suggestions and thoughts. The users were asked to evaluate the statements on the seven-point Likert rating scale where one was "totaly disagree" and seven "totaly agree". The questionnaire and the results are in the Appendix C.

Overall, the users found that the task list handler "is useful" scoring average 5.5 points out of 7, helping them to "be more effective" scoring average 4.7. One user did not agree with that point explaining that people in rescue teams are experienced and do not need to be told what to do, it only slows them down to wait for an order. Based on that feedback the author suggests to carry out further tests and user validations to measure and compare team effectiveness with and without the SmartPM task list handlers.

All the users found that the application was "easy to use" scoring average 6.7 points. The user friendliness was rated averagely six out of seven, where seven was "Totally agree" with the statement "It is user friendly". The lowest rating given by users in section "Ease of use" was 4 points, meaning that users think the task list handler

"requires the fewest steps possible to accomplish what is needed" scoring average 6.7, "using it is effortless" scoring average 6.3, "it is flexible" scoring average 6.5 and they "could use it successfully every time" scoring average 6.0. Everybody rated the statement "I can use it without instructions" with 7.

The section "Ease of learning" was the highest averagely rated section in the questionnaire. The users found that they "quickly became skillful with it" scoring average 6.8, "it was easy to learn to use it" scoring average 6.8, "it was easy to remember how to use it" scoring average 6.8 and they "learnt to use it quickly" scoring average 7.0.

Last but not least, the user satisfaction was targeted with the questionnaire. The results showed that the users are "satisfied with the application" scoring average 6.0 as it "works the way they want" scoring average 6.3 and they find it "pleasant to use" scoring average 6.5. This section also had high scores, having no points below 5.

All of the users had different roles in the simulation. To analyze users evaluation, the ANOVA statistical test was done. Test results (Appendix D) show that there was a statistically significant difference between users as determined by one-way ANOVA ($F(5,108)=4.359$, $p=0.001$). Users who acted out the rb1 and rb2 gave similar scores to the statements, having both 6.68 mean. Users who played act1 and act2 had to operate with the Arduino set. Their ratings given to the statements were relatively diverse, means being accordingly 6.47 and 5.79. That diversion of the mean score might be due to having different experience with the Arduino set. Act3 and act4 did not have the Arduino set and their roles were different, yet their means differed significantly being accordingly 6.42 and 5.74.

In conclusion, the users participating to the evaluation were satisfied with the task list handler as they found it rather useful and easy to learn and use. Based on the users' comments there is still some room for improvements of the task list handler. They pointed out that the user interface could be improved and other communication methods with Arduino could be considered as establishing the connection with the Arduino set was not always successful. Also they pointed out that an interactive map could be added to the application.

6.3 Summary

A scenario, inspired by the WORKPAD project was created to test the system and evaluate the task list handler. The scenario included a team of 6 different actors who had to solve a train derailment situation. They had to measure different gas levels, move to different locations, provide fast wireless Internet connection, take pictures

and rescue people. 6 users played through the scenario and afterwards rated the task list handler, filling out a questionnaire which was based on the USE questionnaire. Base on their feedback, the task list handler is useful and users are satisfied with it as it it easy to learn and use.

Chapter 7

Conclusions and future work

7.1 Conclusion

The goal of this thesis was to update and further develop the SmartPM system and to add a task list handler mobile application to it. In the process, a discretization problem was faced. A concrete solution to it was proposed and realized using the web tool approach. A cyber-physical environment for automating the processes was created, using an Arduino setup which was specially designed and built to fit the needs of the use case. The Android task list handler was designed and created following a plug-in architecture to enable different sensors access in the future. The task list handler was then evaluated by users who played through a sample scenario and then filled out a questionnaire. Users were satisfied with the application as they found the task list handler useful and easy to use and learn.

A number of different technologies and and techniques were tackled throughout this thesis, keeping in mind the good practices of software engineering [12] and development [13] - iterative development, requirements management, component architecture, visual modelling, quality verification, source control usage, etc. The main focus was on the Android task list handler [1] which was written in Java. Plug-ins [2] [3] [4] to extend the application and the SmartPM designer application [5] updates were also realized in Java. Network and communication techniques were studied to create the communication tunnel from the SmartPM engine to the task list handler. Reverse SSH tunnel was considered as the best solution for the system at hand. The server side [6] for hosting the web tools and communication handling was set-up to the Microsoft Azure cloud service, running on Ubuntu virtual machine. The server side was written mainly in PHP, Javascript and HTML. For the communication, the Google Cloud Messaging service was used to send task list handlers the push notifications. The cyber-physical environment was created using Arduino hardware and Arduino programming language [7]. The created design and realization of a sensor-aware task

[1] https://github.com/p2tris/SmartPM
[2] https://github.com/p2tris/SmartPM_arduinoHumidLib
[3] https://github.com/p2tris/SmartPM_Noizelib
[4] https://github.com/p2tris/SmartPM_GPSlib
[5] https://github.com/p2tris/SmartPMsuite
[6] https://github.com/p2tris/SmartPM/tree/master/Server
[7] https://github.com/p2tris/SmartPM_arduinoHumidLib/tree/master/arduino

list handler for adaptive processes in cyber-physical environments proved to be useful and functional through a user evaluation.

7.2 Future work

It is advised to add some features to the application to make it more convenient for the users. Considering location-critical domains, an interactive in-application map could be a useful feature as pointed out by the users. Also tasks history is considered as a nice-to-have feature for any kind of task list handler application. The user interface can also be improved to give better user experience. Bluetooth communication with Arduino is not very stable and energy efficient as it turned out, therefore the author suggests to improve the created cyber-physical system and investigate other alternatives.

The author also suggests some improvements for the SmartPM engine and the overall systems' architecture. The SmartPM engine is advised to be improved to handle simultaneous tasks handling during the adaptation. The structure of the system is also suggested to be reconsidered to improve the communication flow - SmartPM designer and engine could be fully transferred to the same Ubuntu server where the web tools and the GCM communication scripts are. It would lose the need for the SSH reverse tunneling, making the system faster and more secure.

Overall, the created task list handler and the updated system is functional and proved to be ready for using in real situations. Nevertheless it is advised to carry out more thorough testing of the task list handler and the SmartPM system as a whole.

Bibliography

[1] A. Marrella, M. Mecella, and S. Sardiña, "Smartpm: An adaptive process management system through situation calculus, indigolog, and classical planning," in *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*, 2014.

[2] A. Marrella, M. Mecella, S. Sardina, and P. Tucceri, "Smartpm: Automated adaptation of dynamic processes," in *12th International Conference on Service Oriented Computing (ICSOC 2014)*, 2014.

[3] E. A. Lee, "Cyber physical systems: Design challenges," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pp. 363–369, IEEE, 2008.

[4] M. Weske, *Business process management: concepts, languages, architectures*. Springer, 2012.

[5] C. Di Ciccio, A. Marrella, and A. Russo, "Knowledge-intensive processes: Characteristics, requirements and analysis of contemporary approaches," *Journal on Data Semantics*, pp. 1–29, 2014.

[6] M. Reichert and B. Weber, *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies*. Springer, 2012.

[7] R. Reiter, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, September 2001.

[8] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[9] G. De Giacomo, Y. Lespérance, H. Levesque, and S. Sardina, "Indigolog: A high-level programming language for embedded reasoning agents," in *Multi-Agent Programming*, pp. 31–72, Springer US, 2009.

[10] A. M. Lund, "Measuring usability with the use questionnaire." `http://www.stcsig.org/usability/newsletter/0110_measuring_with_use.html`, 2001.

[11] T. Allweyer, *BPMN 2.0: introduction to the standard for business process modeling*. BoD–Books on Demand, 2010.

[12] C. Jones, *Software Engineering Best Practices*. New York, NY, USA: McGraw-Hill, Inc., 1 ed., 2010.

[13] S. McConnell, *Code complete*. Microsoft press, 2004.

Appendix A

License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Pätris Halapuu** (date of birth: 31.07.1991),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
   (a) reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
   (b) make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright, of my thesis

   Design and Realization of a Sensor-aware Task List Handler for Adaptive Processes in Cyber-Physical Environments,

   supervised by Massimo Mecella, Andrea Marrella, Fabrizio Maria Maggi
2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **20.05.2015**

Appendix B

Setting up the system

Several steps must be done to properly set up and use the SmartPM system. In this section it is explained how to achieve that.

Prerequisites: Android version >4.0 device per actor; web server with public IP address and SmartPM web server side system configured for actors management and tasks creation and message handling; virtual machine with SmartPM engine and designer set up, domain theory and workflow defined - domain theory and workflow creation in the SmartPM designer is not tackled in this part; Arduino sensors system wired and powered.

B.0.1 Setting up devices

1. Install the last version of Android task list handler
2. Make sure device is connected to the Internet
3. Start the application
4. Insert username for the device (must be equal to the actor name in designer)
5. On success, actor is ready for tasks
6. Make sure Bluetooth and GPS are enabled

B.0.2 Setting up Arduino

1. Connect all the sensors - White wire is not used by sensors (except Temperature-Humidity sensor), Red is 5V, Black is GND, yellow is to Analog (A*) for communication:
   (a) Temperature-Humidity to the socket directly on the Bluetooth shield (as uses 2 pins - A4 and A5 for working)
   (b) HCHO to A0
   (c) MQ-2 to A1
   (d) MQ-3 to A2

(e) MQ-5 to A3

(f) MQ-9 does not fit but can be used if Temperature-Humidity sensor is removed and MQ-9 connected to A4 (otherwise it will give result with value around 380)

2. Gas sensors need preheating which means that they must work around 15 minutes before they start giving correct values
3. Connect the board to the suitable power source - laptop, tablet, battery pack etc.
4. Arduino is ready for connection via Bluetooth
5. Depending on the Android device, pairing is popped up for the user or in the context menu (where you get all the notifications by all the applications) or in the current activity
6. Pairing code is 1234

B.0.3 Connecting Virtual Machine with web server for communication

1. Make sure the virtual machine is running and you have the Internet connection
2. Designer must not be running the scenario
3. Open Terminal
4. Create reverse SSH tunnel to the server on port 5555 (the one that is used by IndiGolog engine for communication):

```
1        ssh -R 5555:localhost:5555 username@ipaddress
```

5. Socket must be established (server does not give error messages but is logged in)
6. Run the process in designer - on success, actors start receiving the tasks

Appendix C

Questionnaire and results

Following is the questionnaire, based on the USE questionnaire, that was modified and given to the users for the task list handler evaluation. Users were asked to rate arguments on a scale from 1 to 7 where 1 was "Totally disagree" and 7 "Totally agree". The arguments are followed by the given results accordingly.

	U1	U2	U3	U4	U5	U6
1. It helps me be more effective.	6	5	4	2	5	6
2. It is useful.	6	5	6	4	5	7
3. It does everything I would expect it to do.	6	5	5	4	5	6
4. It is easy to use.	7	6	7	6	7	7
5. It is user friendly.	7	5	6	6	7	5
6. It requires the fewest steps possible to accomplish what I want to do with it.	7	6	7	6	7	7
7. It is flexible.	7	5	6	7	7	7
8. Using it is effortless.	7	6	7	4	7	7
9. I can use it without written instructions.	7	7	7	7	7	7
10. I don't notice any inconsistencies as I use it.	7	5	7	7	7	7
11. I can recover from mistakes quickly and easily.	6	6	7	6	7	7
12. I can use it successfully every time.	6	4	7	5	7	7
13. I learned to use it quickly.	7	7	7	7	7	7
14. I easily remember how to use it.	6	7	7	7	7	7
15. It is easy to learn to use it.	6	7	7	7	7	7
16. I quickly became skillful with it.	7	7	7	7	7	6
17. I am satisfied with it.	6	6	5	5	7	7
18. It works the way I want it to work.	6	6	6	6	7	7
19. It is pleasant to use.	6	7	7	6	7	6

Other remarks, opinion:
Add map, consider other Arduino connection ways than Bluetooth - unstable
Nice and simple
It could look nicer, but other than that it is good

Appendix D

ANOVA test

The results of a ANOVA statistical test

Source of Variation	Sum of Squares	d.f.	Mean Squares	F
between	17.44	5	3.488	4.359
error	86.42	108	0.8002	
total	103.9	113		

The probability of this result, assuming the null hypothesis, is 0.001

Act1: Number of items= 19
6.00 6.00 6.00 6.00 6.00 6.00 6.00 6.00 6.00 6.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00

Mean = 6.47
95% confidence interval for Mean: 6.067 thru 6.880
Standard Deviation = 0.513
Hi = 7.00 Low = 6.00
Median = 6.00
Average Absolute Deviation from Median = 0.474

Act2: Number of items= 19
4.00 5.00 5.00 5.00 5.00 5.00 5.00 5.00 6.00 6.00 6.00 6.00 6.00 6.00 7.00 7.00 7.00 7.00 7.00

Mean = 5.79
95% confidence interval for Mean: 5.383 thru 6.196
Standard Deviation = 0.918
Hi = 7.00 Low = 4.00
Median = 6.00
Average Absolute Deviation from Median = 0.737

Act3: Number of items= 19
4.00 5.00 5.00 6.00 6.00 6.00 6.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00

Mean = 6.42
95% confidence interval for Mean: 6.014 thru 6.828
Standard Deviation = 0.902
Hi = 7.00 Low = 4.00
Median = 7.00
Average Absolute Deviation from Median = 0.579

Act4: Number of items= 19
2.00 4.00 4.00 4.00 5.00 5.00 6.00 6.00 6.00 6.00 6.00 6.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00

Mean = 5.74
95% confidence interval for Mean: 5.330 thru 6.144
Standard Deviation = 1.41
Hi = 7.00 Low = 2.00
Median = 6.00
Average Absolute Deviation from Median = 1.00

Rb1: Number of items= 19
5.00 5.00 5.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00

Mean = 6.68
95% confidence interval for Mean: 6.277 thru 7.091
Standard Deviation = 0.749
Hi = 7.00 Low = 5.00
Median = 7.00
Average Absolute Deviation from Median = 0.316

Rb2: Number of items= 19
5.00 6.00 6.00 6.00 6.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00 7.00

Mean = 6.68
95% confidence interval for Mean: 6.277 thru 7.091
Standard Deviation = 0.582
Hi = 7.00 Low = 5.00
Median = 7.00
Average Absolute Deviation from Median = 0.316