Faculty of Science, Technology and Communication

Principles and design of global proactive scenarios over a network of proactive engines

Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of Master in
Information and Computer Sciences

Author:
Gilles Neyens

Supervisor:
Prof. Denis Zampunieris

Reviewer:
Prof. Steffen Rothkugel

Advisor:
Remus Dobrican

September 2015

Declaration of Honour

I hereby declare that the work in this document is completely my own work. No part of this document is taken from other people's work without giving them credit. All references have been clearly listed in the bibliography.

Gilles Neyens

Abstract

This thesis examines Global Proactive Scenarios (GPaSs) in the context of proactive computing. GPaSs are Proactive Scenarios (PaSs) that dynamically collect information, provide strategies for cooperative reasoning and support collective decision making [1]. More precisely we will extract properties from GPaSs and define them. Based on these properties we will then create templates for GPaSs, which will help to facilitate and standardize the creation of future GPaSs. The applicability of these templates is showed through the design of GPaSs for three example applications and finally we will implement one of these applications as a proof of concept example to showcase the usage of the templates in the real world.

Acknowledgements

First of all, I would like to thank my supervisor Professor Denis Zampunieris for welcoming me in his team again and for the guidance and support during my thesis.

I would also like to express my thanks to Professor Steffen Rothkugel and Remus Dobrican for accepting to be the members of this thesis committee.

Furthermore, I would like to express my gratitude to Sandro Reis for his technical support during my thesis.

Finally, I express my special thanks to Marlene Müller, Sergio Marques Dias and Denis Shirnin for their moral support and for making the last years a great experience.

Contents

List of Figures

1 Introduction

Contents

This chapter will give the reader a short overview about the background and structure of this thesis. The reader will be given some information about important computer science paradigms related to this work, which will be explained in more detail in chapter 2. Afterwards he will be given a small summary of the work of this thesis and he will be introduced to the structure of this thesis.

1.1 Purpose of the thesis

The work of this thesis is based on proactive computing, more precisely on a Proactive Engine (PE) developed by Prof. Zampunieris and his team. This proactive engine was running on a server and not connected to any other engines. However, in a world where the usage of mobile phones overtook the usage of traditional desktop computers [2] new possibilities arise. While these mobile devices do not have the same computing power as a desktop computer they are omnipresent in the recent world and the user can take them with him wherever he goes. This opens the way for new applications based on the location of the user and based on collaboration between different devices. Therefore proactive engines should not only be able to enhance the user's experience on its traditional desktop computer but also on his mobile devices and by collaborating with other proactive engines. The technical part of such a connection was already developed for desktop computers and for smartphones [3, 4, 5]. However, as the proactive engines are rule-based, there is also a need for rule templates and design principles that will simplify and standardize the creation of Scenarios that enable collaboration between proactive engines. The idea of such scenarios called Global Proactive Scenarios (GPaSs) was already introduced previously [1]. In this thesis we will concentrate on the definition of properties for these scenarios and on the design of rule templates and design principles to make the GPaSs satisfy these properties.

1.2 Structure of the thesis

In the following chapters we will first have a more detailed look at the state of the art of the fields concerned by this thesis, including proactive systems, collaborative applications and ambient system intelligence. We will

also describe the framework that was used to create the proof of concept application of this thesis. In the following chapters we will then identify and define properties of GPaSs. We will then try to model these properties by giving either rule templates for satisfying a specific property or by giving design principles that can be followed in order to satisfy a property. Using these models we will then propose three example applications to show the applicability of the model and finally we will implement one of these applications as a proof of concept using the Proactive Engine for Mobile Devices (PEMD) framework for Android smartphones.

2 State of the art

Contents

2.1 Proactive systems

Proactive systems are systems that are based on the notion of proactive computing, initially defined by Tennenhouse [6, 7]. In proactive computing the human user is no longer the center of the interaction between humans and computers but takes the role of a supervisor, which watches over the actions executed by a Proactive System (PS). A PS thus does not necessary need explicit user input and can act on its own initiative [6] to other events such as the lack of user input. Therefore PSs need to be aware of their current context, extract the relevant information for their tasks from it and then react accordingly. [8] The idea of proactive computing lead to the development of a PE by Professor Zampunieris and his team, which was used since in a multitude of projects at the University of Luxembourg, mainly in the domains of E-Learning and cognitive science. [9, 10, 11]

2.1.1 Proactive engine

2.1.1.1 Meta-Scenarios, Scenarios and Rules

Before we will describe the PE we will first define the concepts of Meta-Scenarios, Scenarios and Rules. A Scenario is a set of rules that will be executed for a given event,a Rule is a part of a Scenario that executes specific actions if some conditions are met and a Meta-Scenario is a context-aware continuous never-ending rule. [12]

2.1.1.2 Algorithm and rule structure

The PE developed consists of a Rule-running system (RRS), which periodically executes so-called rules. This RRS is composed of two First in First

out (FIFO) queues called currentQueue and nextQueue. The currentQueue contains the rules that need to be executed at the current iteration of the RRS, while the nextQueue contains the rules that were generated during the current iteration. At the end of each iteration the rules from the nextQueue will be added to the currentQueue and the next Queue will be emptied.
A rule consists of any number of input parameters and five execution steps [13]. These five steps have each a different role in the execution of the rule.

1. Data acquisition
   During this step the rule gathers data that is important for its subsequent steps. This data is provided by the context manager of the proactive engine, which can obtain this data from different sources such as sensors or a simple database.

2. Activation guards
   The activation guards will perform checks based on the context information whether or not the conditions and actions part of the rule should be executed. If the checks are true, the activated variable of this rule will be set to true.

3. Conditions
   The objective of the conditions is to evaluate the context in greater detail than the activation guards. If all the conditions are met as well, the Actions part of the rule is unlocked.

4. Actions
   This part consists of a list of instructions that will be performed if the activation guards and condition tests are passed.

5. Rule generation
   The rule generation part will be executed independently whether the activation guards and condition checks were passed or not. I this section the rule creates other rules in the engine or in some cases just clones itself.

During an iteration of the RRS, each rule is executed one by one. The algorithm to execute a rule is presented in Figure 2.1. The data acquisition part of the rule is run first. If the data acquisition fails none of the other parts of the rule is executed. Upon successful data acquisition the activation guards part is executed and evaluated. If the tests pass, the conditions part is executed. If again all the tests of the conditions part pass the actions part of the rule is executed. Finally the rule generation part of the rule is executed independent of whether the activation guards or conditions tests were passed.

```
1.  repeat for each data acquisition request DA
    a.  perform DA
    b.  if error then
        raise exception on system manager console and go to step 7
        else
        create new local variable and initialize it with the
            result of DA
2.  create new local Boolean variable "activated" initialized to
    false
3.  repeat for each activation guard test AG
    a. evaluate AG
    b. if result == false then go to step 6
      else if AG == last activation guard test
      then activated = true
4.  repeat for each conditions test C
    a. evaluate C
    b. if result == false then go to step 6
5.  repeat for each action instruction A
    a. perform A
    b. if error then raise exception on system manager console and
         go to step 7
6.  repeat for each rule generation R
    a. perform R
    b. insert newly generated rule as the last rule of the system
7.  delete all local variables
8.  discard rule from the system
```

Figure 2.1: The algorithm to run a rule

2.2 Network of proactive engines (smartphone version)

In this section we present how PEs were connected, or more precisely how Proactive Engines for Mobile Devices (PEMDs) were enabled to communicate with each other. PEMDs, as the name says, are proactive engines running on mobile devices. In order to use these devices to their full potential a communication between these devices is needed. This communication is achieved by using Google Cloud Messaging (GCM) for Android-based mobile devices.[5] At the installation each device first needs to register with the GCM server and receives back a registration ID (Figure 2.2). This ID is then passed along with a username to our relay server and stored in the database of this server. After this registration process the device is ready to communicate with other devices. PEMDs communicate by sending rules to each other. These rules are encoded in JSON and on the receiving engine they get decoded again and added to the nextQueue of the RRS.

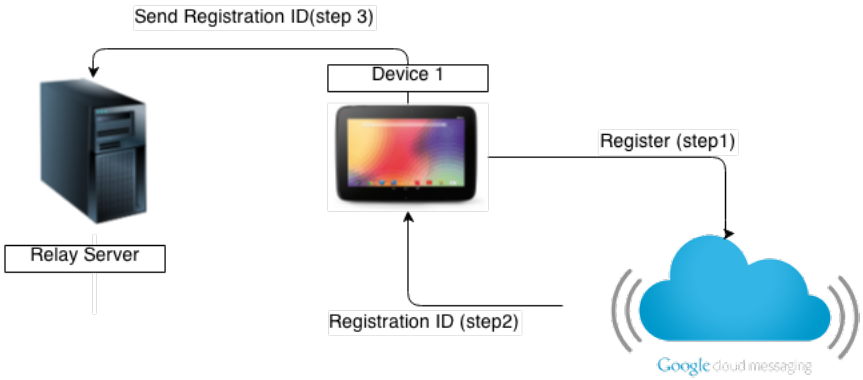

Figure 2.2: GCM registration

The communication itself is done in 6 steps like shown in Figure 2.3.

1. The device sends a rule to the relay server with the username of the device that should receive the rule and with its own registration ID.
2. The relay server looks up the registration ID of the other device in the database and pushes the message to the GCM cloud.
3. The GCM cloud delivers the message to the correct device.
4. The device that received the message sends back a confirmation to the sender device by first sending it to the relay server.
5. The relay server pushes the confirmation message to the GCM cloud.
6. The the GCM cloud delivers the confirmation message to the initial device.

The confirmation is used in order to avoid losing messages in the network. Each engine keeps track of the messages it sent and sends the message again if no confirmation message arrived. Also each engine keeps track of received messages in order to avoid receiving the same rule twice.

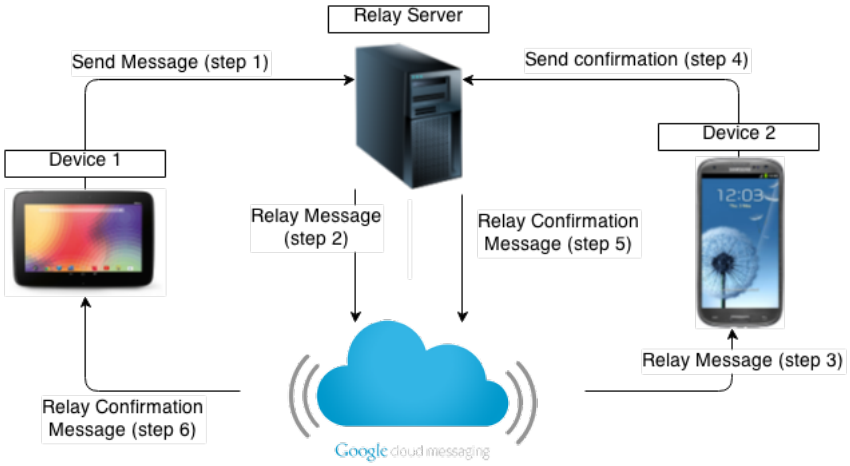

Figure 2.3: GCM communication

2.3 Global Proactive Scenarios

In the context of networks of proactive engines the concept of Global Proactive Scenarios was introduced [1]. GPaSs are an information sharing mechanism composed of proactive rules that have the following characteristics:detection of unexpected events (similar to a normal PaS), information collection from remote devices running a PE, strategy supply for cooperative reasoning and the support for collective decision making.
GPaSs are well suited for large-scale networks with a large number of devices running a PE. They support multiple information sharing strategies, which can be chosen based on the need of an application. Several of these strategies are introduced in this thesis under the form of rule templates.
GPaSs can be triggered in different ways. The two most common triggers are the detection of a foreseen event by a rule or simply user input. Once triggered the GPaSs will try to achieve their objective(s) by executing the rules they are composed of. Generally this happens by gathering information from remote PEs and then taking decisions based on this information.

2.4 Ambient intelligence

Ambient intelligence describes computer or simply electronic systems that sense the presence of people and react to their presence. The possibility for ambient intelligence is given by the recent emergence of different technologies such as the miniaturization of microprocessors, mobile phones or the embedment of computing power in every day objects.[14] Currently Google © is even working on interactive clothes. [15] Other technologies

such as RFID tags, Bluetooth or simple sensors allow the computing devices to detect the presence of a user (and of his smart phone/tablet etc.).

This omnipresence of computing devices opens the road for new types of applications like smart houses (Gator Tech Smart House [16] ,MavHome [17]). Other application areas include hospitals where the Lutheran General Hospital in Chicago has built a pavilion for children where they are entertained during their examination sessions through ambient intelligence [18], transportation where the driver can be warned from dangerous situations [19] or even in education where the Northwestern University created an intelligent classroom, which helps students during their lessons [20]. In the workplace an ambient system like the MOSES system [21] can help reminding the worker of its tasks left to do by keeping track of his actions and the objects on its work desk with an RFID tag.
In this thesis we use the concept of ambient intelligence to define one of the properties of GPaSs.

2.5 Collaborative applications

Collaborative applications, also called collaborative software, are applications that help human users to work together. This includes everything that improves the work flow, may it be a tool for simplifying the organization of meetings over different time zones, multi-user drawing tools like VideoDraw [22] or text editors like NTE [23]. Other collaborative systems like Group Kit [24] specialized in providing a framework for easily creating collaborative applications. In this thesis we will use a GPaS in order to implement an application that schedules a meeting between a group of users.

3 Properties of GPaS

Contents

In this chapter we are going to define the different properties that a Global Proactive Scenario has to fulfill in order to be considered as such. GPaSs need to fulfill the same properties as simple PaS. Additionally they need to satisfy a set of properties, which can be split into two subcategories: general properties and architecture specific properties. The general properties are those that every GPaS needs to fulfill while the architecture specific properties are those properties that depend on the architecture of the proactive systems.

3.1 Basic properties of PaS

3.1.1 Proactive

The proactive property of a PaS defines their ability to act on-behalf of the user and on their own initiative without the users explicit command.

3.1.2 Anticipating

Another important property of a PaS is its capability to correctly infer future events based on the current context and to delegate the task to an appropriate target scenario, which is capable of handling the situation.

3.2 General properties of GPaSs

In this section we will discuss the properties that a global proactive scenario will need to fulfill in addition to those required by a normal PAS and that are true for any global proactive scenario regardless of the architecture of the system. For GPaSa we will always differentiate between the proactive engine that started the scenario by activating the initial rule of the scenario (initiator) and the rest of the engines that participate in the GPaS.

3.2.1 Collaborative

The collaborative property is the most important property that separates a GPaS from a normal PaS as it ensures that different engines are working together to achieve a common goal. This property can be split into two types: active collaboration and reactive collaboration. A GPaS designed for active collaboration actively asks remote engines for information or to process data while a GPaS designed for reactive collaboration regularly receives information updates from remote engines.

3.2.1.1 Active collaboration

Active collaboration can be divided into 4 four steps: Communication, Processing (Non-Initiator), Coordination, Processing (Initiator). To satisfy active collaboration a GPaS has to consist of actions that perform the four steps mentioned. We will now take a closer look at the different steps and what exactly a GPaS would need to do in order to satisfy the necessary requirements.

- Communication
  This step simply requires the communication of different proactive engines during the execution of the GPaS. It is not important which technology is used to perform the communication (Bluetooth, WiFi, etc. ...).

- Processing (Non-Initiator)
  This step needs to be performed on the non- initiator engines. By processing we mean that the non-inititor engines have to produce some added value to the data/information sent by the initator before sending it back or before sending a response that was determined by the data received.

- Coordination
  The coordination step is specific to the initiator of the GPaS. The initiator needs to wait for the responses of the processing done by the other engines and also decides whether it is still useful to continue the execution of the main algorithm.

- Processing (Initiator)
  This step is also solely for the Initiator of the GPaS. This step is immediately situated after the coordination step. In this step the Initiator has to process the responses received by the the other engines and perform actions based on this processing.

3.2.1.2 Reactive collaboration

An Initiator does not exist in a GPaS with reactive collaboration as many engines regularly send updated information to the information gathering engine. Reactive collaboration for a GPaS can be defined in two steps:

- Receive regular information updates from remote engines
- Take appropriate actions based on this information

Active and reactive collaboration mainly differentiate on the trigger for the main actions taken by a GPaS. In active collaboration the GPaS decides based on local information that it needs additional information from remote engines in order to perform its actions while for reactive collaboration the GPaS decides to take actions based on information from other proactive engines.

3.2.2 Fault-tolerant

As GPaSs need to perform some sort of collaborative actions they need a communication between different proactive engines over certain types of networks. However, as devices are not always reachable due to network failure or simply because devices are not turned on, which is often the case for mobile devices, a GPaS needs to define a failure Scenario for these cases for **every** communication step that is performed where an answer is expected. This failure Scenario needs to take care of a few questions, varying based on the type of collaboration.

3.2.2.1 Active collaboration

- How long does the GPaS wait until every answer arrived?
- What actions does the GPaS take if not every answer arrived in time? Does it continue with its normal execution or does it end in an error state?

3.2.2.2 Reactive collaboration

- When not receiving information updates from a specific engine anymore, what is the time limit for error handling actions to be taken?
- What happens to the data previously received by the failing engine? Is it considered obsolete or may it still be used for future calculations and processing?

The answers to these questions vary greatly between different GPaS, but it is possible to have at least a rule template, which covers the core problem of timeout to error handling as we will discover in the next section.

3.3 Architecture specific properties

In this section we present different properties for specific architectures of proactive systems in the context of a GPaS, meaning that only actors involved in the GPaS are considered when classifying GPaSs in the different kinds of architectures and not all the actors connected to an engine. These properties are not mandatory for a GPaS to be considered global but in some cases they are mandatory in the sense that given the nature of a specific architecture of proactive engines the GPaS could not function properly if it does not meet these properties. Also these properties will allow us to define templates in the model section in order to simplify and standardize the design and creation of future GPaSs.

3.3.1 System-System

A System - System architecture for a GPaS is an architecture that only involves the proactive systems themselves during the execution of the GPaS. There is no need for user input and the GPaS is able to completely execute without any interaction with the user.

3.3.1.1 Continuous

The continuous property is a pretty straight-forward property. As in an architecture with only proactive engines without direct user input, the GPaS cannot be initiated by the user itself, the GPaS has to have an initial rule running at all times, which will then decide when to execute the rest of the GPaS.

3.3.2 System - User

A System - User architecture involves a proactive engine that execute rules that don't need to interact with the user and other PEs, which contain rules

that interact with the user, may it be by directly asking for input or by reading data that can be changed at any time trough a user interface. In this type of architecture we made the design choice that the system without user interaction will always be the initiator of the GPaS.

3.3.2.1 Continuous

Similarly to the Continuous property of the System-System architecture the GPaS has to have an initial rule running at all times on the engine without user interaction in order for the GPaS to function properly.

3.3.2.2 Ambient

This is an optional but useful property for GPaSs that involve one (or maybe more) PE(s) and many proactive engines connected with a human user. As described previously, an ambient system is an electronic environment that is sensitive and responsive to the presence of people. In order to satisfy this property, a GPaS thus has to react and carry out actions based on the presence of people. A few technologies can help satisfying this property, the most relevant one for GPaSs would be Bluetooth. There are many scenarios where the proactive engine needs to react to the proximity of users. One could imagine proactive systems in schools that will turn off the sound of the mobile phones of the students or, like the example given in one of the upcoming sections, negotiate an acceptable temperature for the house between the house owner and its guests.

3.3.3 User-User

In a User - User architecture every PE involved in the GPaS will have to, at some point, interact with the user, may it be by directly asking for input or by reading data that can be changed at any time trough a user interface. It is not possible for a PE to execute all the rules involved in the GPaS on their side without using input from the user. This architecture comes closest to traditional applications.

3.3.3.1 Initiated

In this type of architecture the user has to take action in order to start the GPaS by entering data in a user interface and thus 'initiating' the GPaS. While it would also be possible in some cases to have the GPaS constantly running and being triggered by changes to stored data on the device by the user, this type of architecture involves in most cases devices that run on battery (smartphones, smart watches, etc.) and thus we opted for the more battery saving option where the user himself has to start the GPaS.

3.3.3.2 Monitored

The execution of a GPaS under the user-user architecture is monitored by the user, meaning that it needs input from the user in critical and important situations in order to realize its task. Input from the user also can include preferences previously set by the user and fetched by the GPaS when needed.

4 Model

Contents

In this chapter we are going to present different templates for GPaS, which will facilitate and standardize the design of future GPaSs and in the same time make sure that the necessary properties are satisfied. Before we continue with the presentation of the different templates it is important to note that they are 'only' templates, meaning that they can be extended and modified at will (as long as this would not jeopardize the fulfillment of the mandatory properties) and even that different rules from different templates can be merged together as the templates are designed to provide a clearer picture of as how these templates satisfy different properties.

4.1 General properties of GPaS

4.1.1 Active collaboration

As described in the previous section, the GPaS has to perform 4 steps in order to satisfy the (active) collaborative property:

1. Communication
2. Processing (Non-Initiator)

3. Coordination

4. Processing (Initiator)

In order to achieve this property in a GPaS we will propose two different approaches, which both have their advantages and disadvantages and are best suited for different kinds of algorithms. The first approach consists of performing the collaboration with one engine at a time while the second approach will collaborate with all proactive engines involved simultaneously. We will now first give a schema for both approaches along with template rules written in pseudo-code and at the end we will discuss the advantages and disadvantages of these approaches in different situations. To note here is that these templates are meant to be used once per collaboration, meaning that if a GPaS needs to collaborate several times with other engines during its executions, these templates need to be applied at every collaboration step.

4.1.1.1 One by one

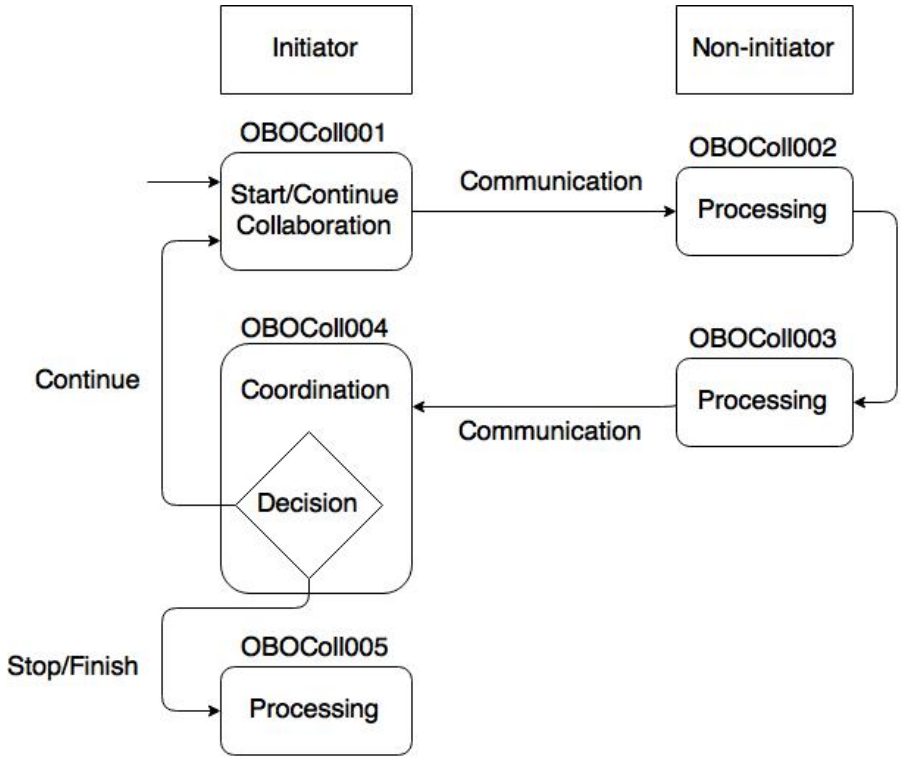

Figure 4.1: Schema for One by One collaboration

General description This approach will contact every participating engine one-by-one, wait for the result and then decide whether to continue the collaboration with the remaining engines or to continue or stop the algorithm (Figure 4.1).

OBOColl001 Start/Continue collaboration

Parameters:		
	receiverList	List of engines that will participate in the collaboration
	param_1 .. param_N	List of parameters needed for the algorithm
	result	Result or partial result of the algorithm

Description:

The job of this rule is to start or continue the collaboration by taking the first receiver in the receiverList, removing it and sending the needed information along with the rest of the receiverList to this first receiver.

OBOColl002 Processing (Non-Initiator)

Parameters:		
	initiatorId	ID of the engine that wants to have a reply
	receiverList	List of remaining engines that will participate in the collaboration
	param_1 .. param_N	List of parameters needed for the algorithm
	result	Result or partial result of the algorithm

Description:

The job of this rule is to receive the information needed for the algorithm, perform a form of processing based on this information and previous results and pass on the results to OBOColl003.

OBOColl003 Processing (Non-Initiator) ctd.

Parameters:		
	initiatorId	ID of the engine that wants to have a reply
	receiverList	List of remaining engines that will participate in the collaboration
	param_1 .. param_N	List of parameters needed for the algorithm
	result	Result or partial result of the algorithm

Description:

The job of this rule is to send the new result back to the Initiator engine. (It is possible to merge OBOColl002 and OBOColl003 into one rule)

OBOColl004 Coordination

Parameters:	receiverList	List of remaining engines that will participate in the collaboration
	param_1 .. param_N	List of parameters needed for the algorithm
	result	Result or partial result of the algorithm

Description:
The job of this rule is to perform the coordination step, meaning that it receives the partial results and decides about the future continuation of the algorithm. The continuation of the algorithm is influenced by the partial results and by the number of collaborating engines still in the list.

OBOColl005 Processing (Initiator)

Parameters:	result	Result or partial result of the algorithm

Description:
The job of this rule is to use the obtained result to either yield an error or to activate other target scenarios to perform appropriate actions.

Advantages

- Efficient
  If it becomes clear after the reception of a partial result that it is useless or not needed to continue the algorithm and contact the remaining engines, it is easy to stop the algorithm, which avoids unnecessary communication between engines, which in turn can increase the battery life of smaller devices.

Disadvantages

- Low time efficiency
  This approach needs to contact the collaborating engines one-by-one, which increases the time needed to complete the algorithm especially if there are two or more engines that are temporarily unavailable at the moment they are being contacted.

4.1.1.2 All at once

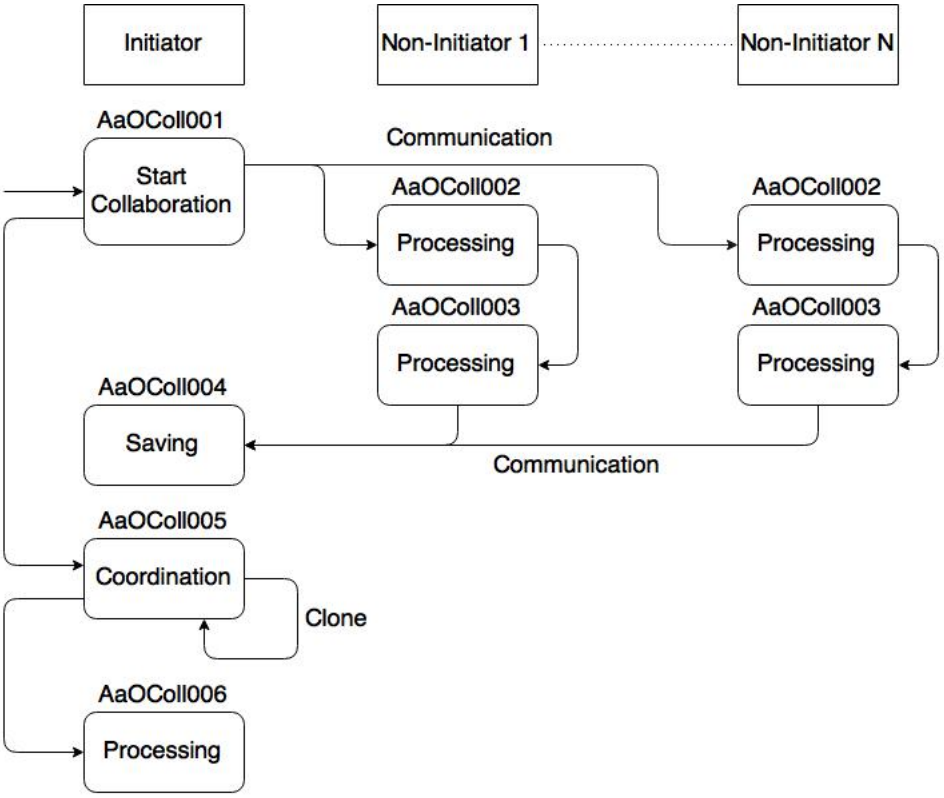

Figure 4.2: Schema for All at once collaboration

General description This approach contacts all the participating engines at once right at the beginning (Figure 4.2). A dedicated rule will then wait until all the answers arrived before continuing with the rest of the algorithm.

AaOColl001 Start collaboration

Parameters:		
	ScenarioID	ID of this Scenario instance
	receiverList	List of engines that will participate in the collaboration
	`param_1 .. param_N`	List of parameters needed for the algorithm

Description:

The job of this rule is to send the needed information to every engine in the list by creating a AaOColl002 rule on these engines. It will also create a AaOColl005 rule on the local engine.

AaOColl002 Processing (Non-Initiator)

Parameters:		
	ScenarioID	ID of this Scenario instance
	InitiatorId	Id of the engine that initiated the GPaS
	`param_1 .. param_N`	List of parameters needed for the algorithm

Description:

The job of this rule is to receive information, process it and pass on the result to rule AaOColl003.

AaOColl003 Processing (Non-Initiator) ctd.

Parameters:		
	ScenarioID	ID of this Scenario instance
	InitiatorId	Id of the engine that initiated the GPaS
	result	Result of the processing

Description:

The job of this rule is to send the result back to the initiator engine.

AaOColl004 Reception and Storage

Parameters:		
	ScenarioID	ID of this Scenario instance
	SenderID	Id of the engine that sent this answer
	result	Result of the processing

Description:

The job of this rule is to receive and store the result along with the ID of the sender and the Scenario ID in an appropriate Database table.

AaOColl005 Coordination

Parameters:		
	ScenarioID	ID of this Scenario instance
	expectedAnswers	Number of expected answers

Description:

The job of this rule is to check whether all answers for a particular scenario instance arrived. If this is the case this rule will combine the received results together and pass them on to rule AaOColl006.

AaOColl006 Processing (Initiator)

Parameters:		
	result	The result from the collaboration

Description:

The job of this rule is to perform further processing on the results from the collaboration.

Advantages

- Fast
  The fact that every engine is contacted right at the beginning of the algorithm makes this approach faster in average than the One-By-One approach, which needs to wait for the reply of the first engine before it can contact the second engine.

Disadvantages

- Extra Memory
  This approach needs to store the results in a Database table until all the results are received. This will consume more memory and the additional Database operations can have an impact on the performance of the proactive engines of less performant devices like smart phones.

4.1.2 Reactive Collaboration

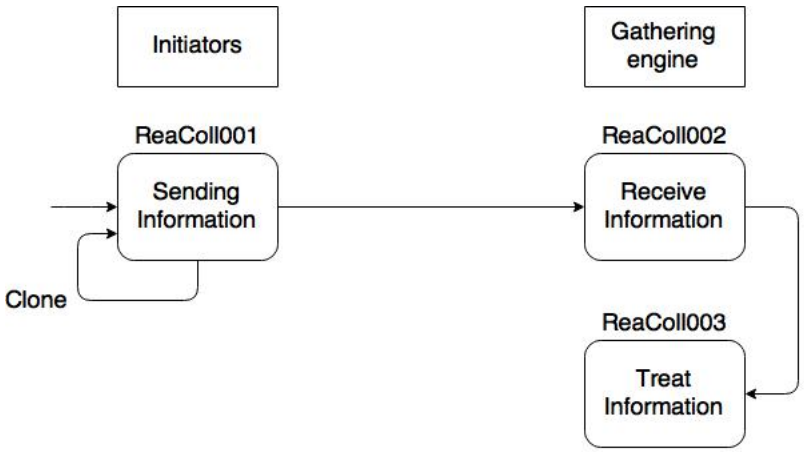

Figure 4.3: Schema for reactive collaboration

General description For reactive collaboration there is one engine that will receive information (right side of Figure 4.3) regularly from other engines (left side of Figure 4.3). Upon reception the data is treated and put into a Database for further use.

ReaColl001 Sending information

Parameters: None
Description:
This rule will send a ReaColl002 rule along with relevant data for the algorithm to the gathering engine at every execution. Then it will clone itself.

ReaColl002 Receive information

Parameters:	SenderID	Id of the device that sent this information
	param_1 .. param_N	List of parameters needed for the algorithm

Description:
This rule will receive information from other engines, save them in the database and create a ReaColl003 with the freshly received information.

ReaColl003 Treat information

Parameters:	SenderID	Id of the device that sent this information
	param_1 .. param_N	List of freshly received information

Description:
This rule will treat the freshly information received and take appropriate actions. The SenderID will be needed when comparing the new information to old stored information.

4.1.3 Fault-tolerant

While this section talks about how GPaSs can satisfy the fault-tolerant property, parts of it can also be used to solve a remaining problem of PAS, which is the fact that it was not possible to stop a specific scenario that was currently running. The following templates show how to satisfy the fault-tolerant property and are meant to be used in conjunction with the collaboration templates in order to handle possible failure scenarios.

4.1.3.1 Fault-tolerance for active collaboration

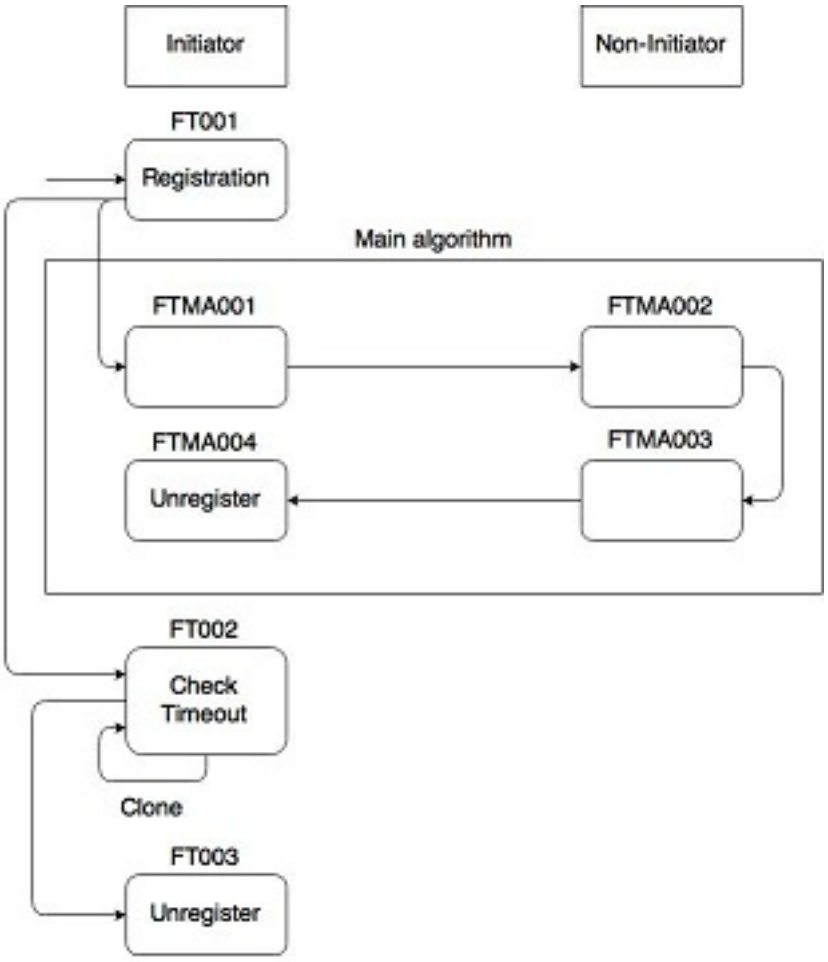

Figure 4.4: Fault-tolerance for active collaboration

General description The template for fault-tolerance for active collaboration is best wrapped around the collaboration templates, meaning that the rules of the collaborative templates should be included in the main algorithm (FTMA* rules in Figure 4.4). First the scenario is registered in the engine and the obtained ID is then used to interrupt the rules of this scenario in the case of a timeout. In this situation Rule FT003 then decides what actions should be taken, by continuing by executing the rest of the GPaS (case when enough data was received, or the GPaS does not expect answers from every single engine) or by launching a failure scenario to handle the situation. The database table for the registration can be seen in Figure 4.5, where the ID is the ID of a scenario instance, the scenario_type

is the type of a scenario, which can be used to terminate all instances of the same type of scenario and creation_time, which represents the time and date a scenario was registered. The type of a scenario is a string for the name of the scenario. Let us take as an example an application that negotiates a price for an object in an auction. The scenario type could then be called 'PriceNegotiationScenario'. There could be several negotiations going on at the same time. If one of the negotiation succeeded, it would be possible that the user ran out of money (or surpassed a limit he set himself), but the other negotiations would be still be running. However, as they are all running under the same scenario type, it is possible to just remove all of them from the database table, which will cause the checks of the different rules to fail and thus to stop every instance of this scenario type.

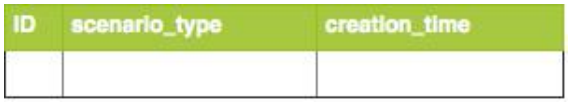

Figure 4.5: Database table for running instances of scenarios

FT001 Registration

Parameters: None
Description:
Upon activation, this rule registers a new instance of this scenario with the engine. It gets back an ID, which it will pass to FT002 and to the first rule of the main algorithm of the GPaS FTMA001. Additionally FT002 will be instantiated with the time limit for this instance of the scenario.

FT002 Check Timeout

Parameters:	ScenarioInstanceID	Id of this instance of the GPaS
	n	The frequency in seconds this rule will be executed
	last_executed	The time this rule was last executed
	timelimit	Number of seconds this scenario is allowed to run

Description:
This rule checks whether the instance of the scenario specified by the ScenarioInstanceID is still registered. If it not registered anymore (end of main algorithm) it will not clone itself. If it is still registered it will check if the time limit was exceeded for this instance of the scenario. If this is the case, this rule will create a FT003 rule with the ScenarioInstanceID. If not it will clone itself.

FT003 Unregister

Parameters: ScenarioInstanceID Id of this instance of the GPaS
Description:
This rule will unregister this particular instance of a scenario from the engine. This rule needs also to take care of the decision what happens to the data left by this instance of a scenario, if it needs to be cleaned up or if it is can still be used by future instances.

FTMA* Reception

Parameters: ScenarioInstanceID Id of this instance of the GPaS
Description:
The rules from the main algorithm of the GPaS all need to pass the ScenarioInstanceID as parameter to the next rules of the scenario. Every rule of the main algorithm on the intiator engine needs to check at every execution if the scenario is still registered. If the scenario instance is not registered anymore these rules will no longer perform any actions, not even clone themselves. The last rule of the main algorithm for this collaboration step needs to unregister the scenario instance.

4.1.3.2 Fault-tolerance for reactive collaboration

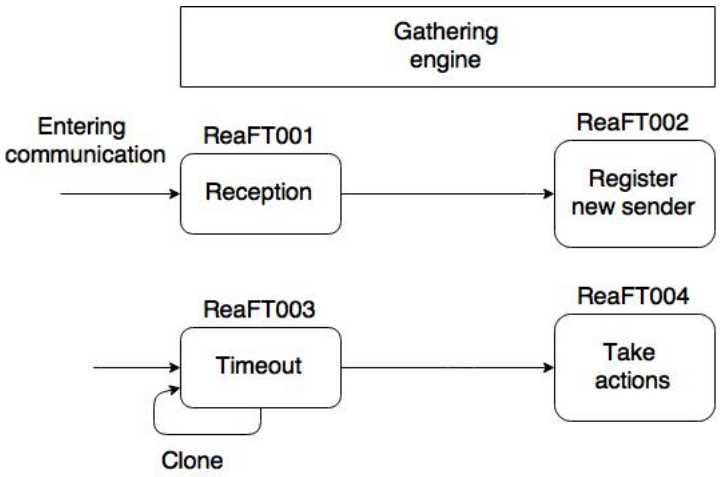

Figure 4.6: Fault-tolerance for reactive collaboration

General description Upon the reception of a message the GPaS needs to check whether the sending device is already registered on this engine or not (Figure 4.6). If it is not the case the device ID will be added to the database

column depicted in Figure 4.7, where last_received represents the time a message from a specific device was received and disabled is a boolean value, which tells the GPaS if data from this engine should be ignored for the moment. A time-out rule then periodically checks the last_received time for every device and in the case of a time-out it leaves the decision on the action that will be taken to the 'take actions' rule.

ID	last_received	Disabled

Figure 4.7: Database table for reactive fault tolerance

ReaFT001 Reception

Parameters: SenderID Id of the device that sent this rule
Description:
This rule checks whether the device that sent this rule already sent something in the past. If this is the case, this rule will update the last_received column of this device's ID in the database to the current time. If not, it will create a ReaFT002 rule. Note that this rule will be integrated in the receive information rule of the collaboration template.

ReaFT002 Register new sender

Parameters: SenderID Id of the new device
Description:
This rule will save the new device's ID in a database table, which contains all device IDs and sets disabled to false and last_received to the current time.

ReaFT003 Time-out

Parameters:	n	The frequency in seconds this rule will be executed
	last_executed	The time this rule was last executed
	timeout_limit	The time after which a device is considered to be timed-out

Description:
This rule will be executed every n seconds. At every execution it will check for every device whether the device timed-out by calculating last_received-current_time>timeout where last_received is the value stored in the database for the specific device. If a device is timed out this rule will create a ReaFT004 rule.

ReaFT004 Take actions

Parameters: SenderID Id of the device that timed out
Description:
This rule defines the actions that will be taken for the device that timed out. The standard action would be to set the device to disabled and ignore the old data received by this device. Another possible action could be trying to calculate new data based on the old data.

4.2 Architecture specific properties

4.2.1 System-System

4.2.1.1 Continuous

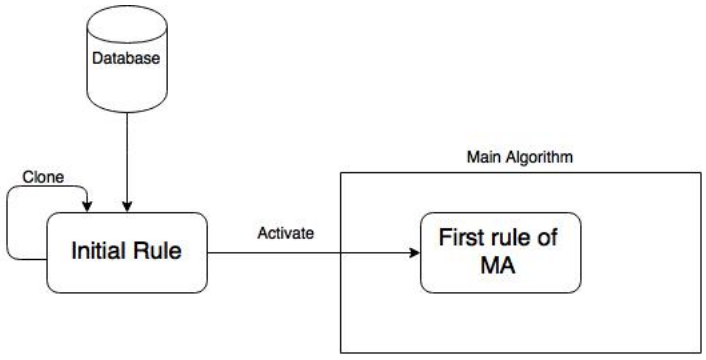

Figure 4.8: Continuous rule

Like shown in Figure 4.8, this property can be satisfied by having a rule initially in the Database that will clone itself forever. This rule will decide when the main part of the GPaS needs to be executed.

4.2.2 System-User

4.2.2.1 Continuous

The same approach as for System-System setup can also be used for the System-user setup in order to keep the algorithm running.

4.2.2.2 Ambient

Figure 4.9: Ambient intelligence in a proactive engine

Ambient intelligence can be achieved in a GPaS by having one rule that requests the list of near users (or proactive engines) through the Proximity Manager (Figure **??**). The Proximity Manager is a module of the proactive engine, which can use different technologies in order to detect near users and proactive engines. Note that the RFID technology is only suited for GPaS in the context of two NFC devices and not for simple passive RFID tags as the communicating devices all need a proactive engine running on them. The choice of technology depends on the situation.

4.2.3 User-User

4.2.3.1 Initiated

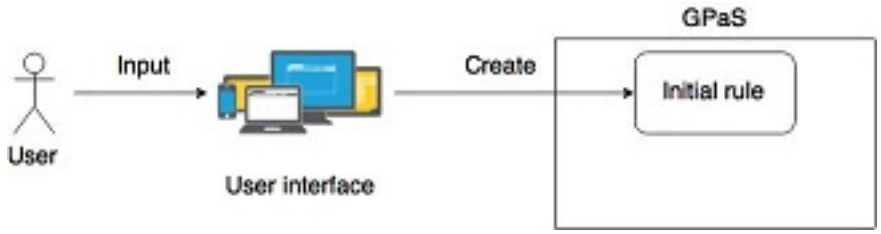

Figure 4.10: User initiating the GPaS

In a User-User setup the GPaS is started by the action of a user. This is done trough a user interface where the user can enter the data required by the initial rule of the GPaS and create this rule by submitting the data entered (Figure 4.10). The execution of the GPaS will then proceed normally in the background.

4.2.3.2 Monitored

For the monitored property of a GPaS, we propose two different approaches of interacting with the user.

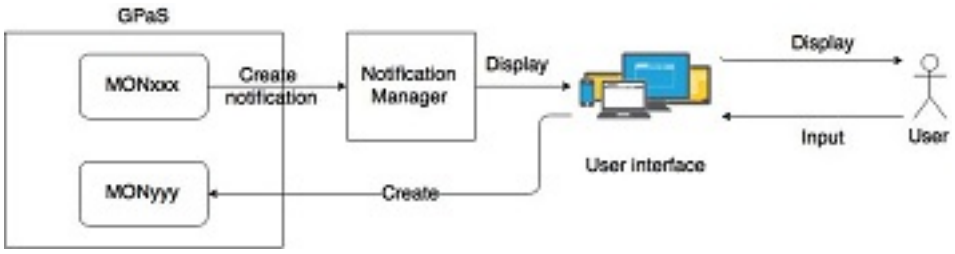

Figure 4.11: Monitoring through user interface

The first approach, depicted in Figure 4.11, sends a notification to the user through the notification manager. The notification provides the necessary user interface to the user to enter the data needed by the GPaS. After the data was entered the GPaS continues its normal execution with the information received.

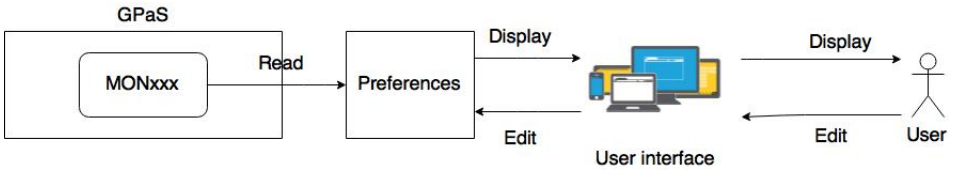

Figure 4.12: Monitoring through preferences

In the second approach (Figure 4.12) the user is not directly asked for input. The user can edit his preferences through a user interface at any time. When needed, the GPaS can then fetch the required information from the preferences and continue its execution without interruption.

In the context of proactivity the second approach is the most appropriate one as the user can edit his preferences whenever he wants and the GPaS can carry out its task in the background without having to wait for user input. However, in some situations there is no way around the first approach, like when the user needs to make a precise choice. Of course the preferences can allow the GPaS to greatly reduce the set of possible choices but the final choice between the remaining options still needs to be done by the user.

5 Examples of model applications

Contents

In this chapter we will present three example applications to show how the previously presented templates can be used to easily create GPaS for specific applications. Each application is described in four parts: General description, static diagram, dynamic diagram and a detailed description of each rule in the GPaSs. The static diagram shows the parameters of the different rules along with their types. It also represents the creation of rules by arrows. Solid arrows mean that the rule at which the arrow is pointing is created on the same engine where the rule creating it, is located. Dashed arrows mean that the rule at which the arrow is pointing is created on a remote engine (Figure 5.1). The numbers on the arrows indicate how many rules are created by one single rule of a given type.

The dynamic diagrams represent single executions of the GPaS where an arrow simply indicates the creation of a rule. Moreover the dynamic diagrams contain color codes in order to show to which template a rule belongs. White rules are rules that do not belong to any template. Most of the time these rules continue the execution of the GPaS after a successful collaboration. The three examples are chosen to match the three different types of architectures a GPaS can have, namely System-System, System-User and User-User. In this chapter we will concentrate on the first two applications by presenting the rules needed for the GPaSs. In the next chapter we will then present the third application as a proof of concept with the rules, database and user interface needed to ensure the correct functioning of the application.

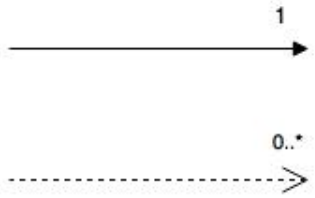

Figure 5.1: Static diagram arrows

5.1 Airplane collision avoidance (System-System)

5.1.1 General description and assumptions

The idea of this application is to have a system for airplanes to avoid collisions for other airplanes. Every airplane will be equipped with a proactive engine, have an ID and plane model and will regularly send its position as well as relevant data of its movement like speed, flight angles and fuel level to the nearest airport, which also has a proactive engine running. In order to simplify the application for this thesis we will assume that there exists only one airport and that every plane will send the data to this airport. We also assume that the auto-pilot of the airplanes can take a command of the form (duration,angles,speed), meaning that the auto-pilot will adapt the heading angles and speed of the airplane to those of the command for the duration given and then go back to its normal route.

5.1.2 Template usage

In this section we describe which templates were used in order to design the GPaS of this application and give a reason why they were used.
First of all, the application for which we want to create a GPaS is an application between systems, as there is no human user involved in the execution of this application. Therefore we are using the 'continuous' template, which will allow the GPaS to keep running forever.
As we are designing a GPaS for the application, the collaborative and fault-tolerant properties also need to be satisfied. In regards to the collaboration, the best template to use was the reactive collaboration one. The reasoning behind this is that the PEs running on the airplanes know that they need to send updates regularly to the closest airport. It is not the case that the airports first need to analyze local data and then decide that they need data from the airplanes and then ask them for it. This means that using the reactive collaboration template will allow us to cut the communication needed by half (only updates are sent as opposed to asking for data and then receiving an answer). As we are using the reactive collaboration template, we thus also need to use the reactive fault-tolerant template as it is specif-

ically designed to be used together with the reactive collaboration one and handle possible problems.

5.1.3 Static diagram

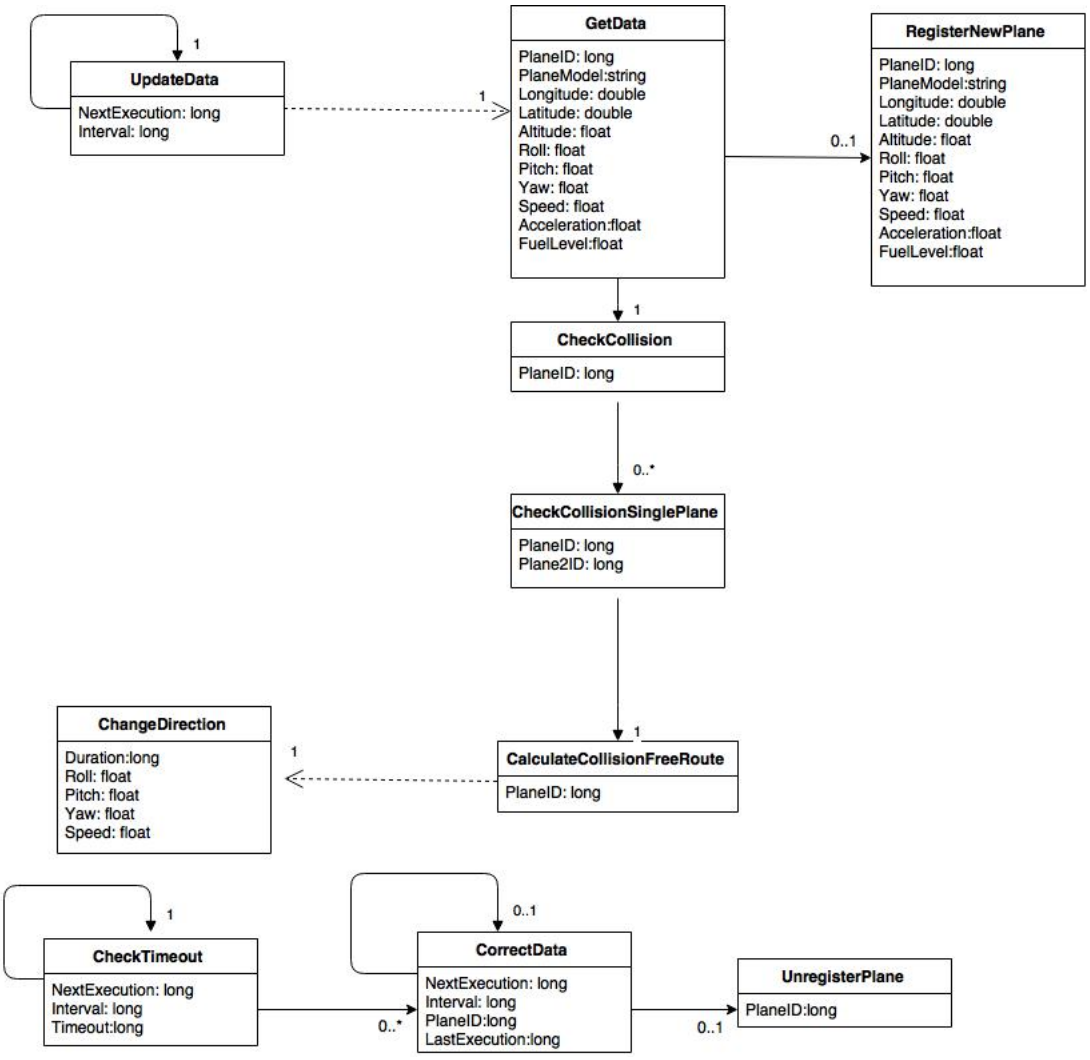

Figure 5.2: Airplane collision avoidance static diagram

5.1.4 Sequence diagrams

The three sequence diagrams represent the following story line. A brand new airplane takes off for its first flight. The airport will receive the first data, recognize that it is a new plane and register it (Figure 5.3). After a

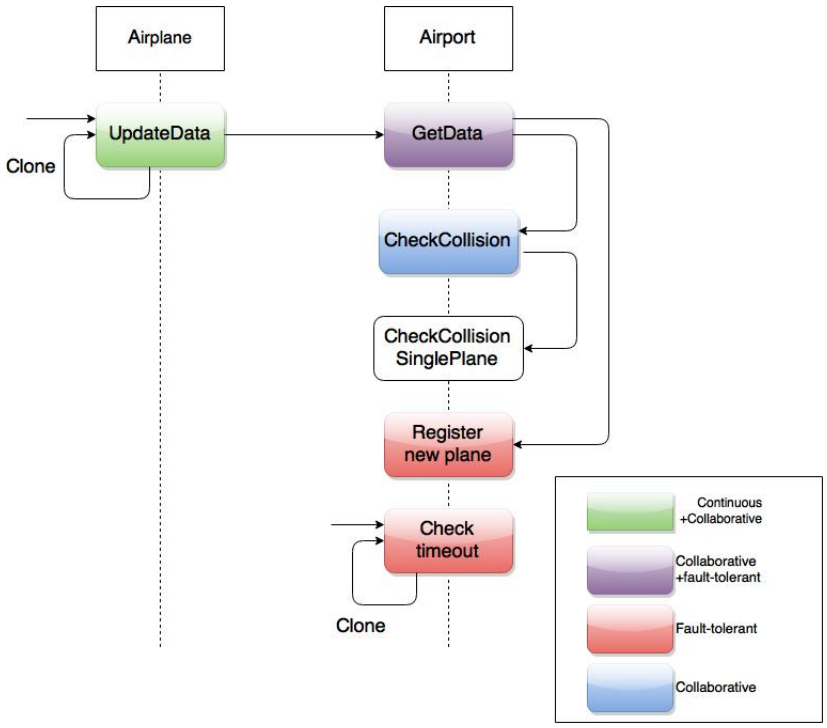

Figure 5.3: Registration of new air plane and collision check

while and several information updates by the plane, the proactive engine of the airport recognizes that the plane will collide with another plane in the next minutes if it continues with its current route and thus the airport will send commands to the plane to correct the course of the plane in order to avoid the collision (Figure 5.4).

A few hours later, the plane does not send any more data to the airport. The engine on the airport recognizes this thanks to the timeout rule and proceeds to calculate the new position of the plane based on the old data. After some time, during which the plane still did not send any data, the engine on the airport calculated (estimation) that the plane is out of fuel and considers it as lost and thus removes the data of that plane from the database so that it will not be used for any future collision checks for other planes (Figure 5.5).

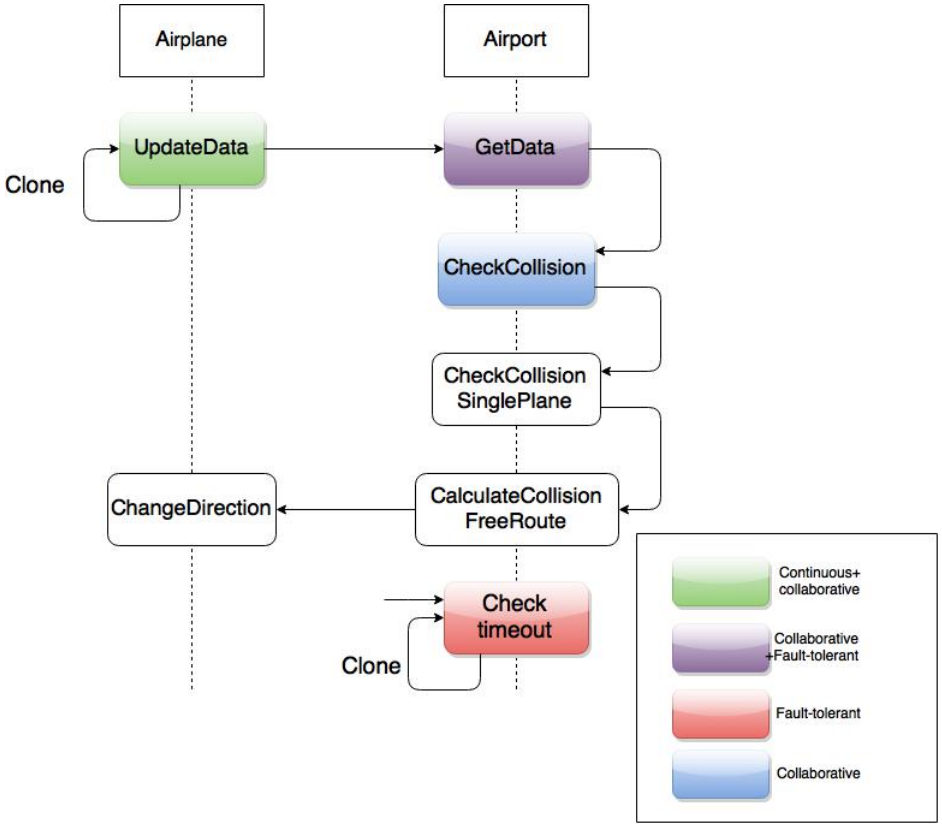

Figure 5.4: Collision check with subsequent route correction

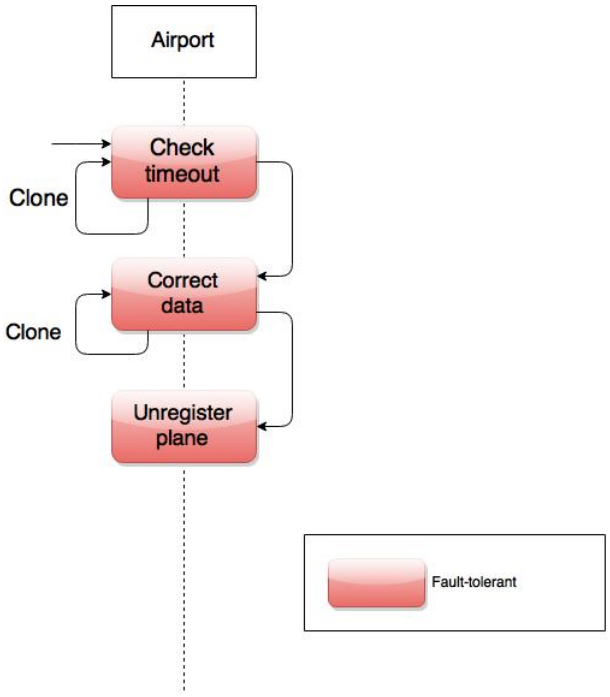

Figure 5.5: Error correction and plane removal

5.1.5 Airplane rules

5.1.5.1 Update Data

Parameters:

nextExecution	Time when the data should be sent again
interval	Number of seconds between executions

Description:

This rule gathers important information from the airplane's system such as current speed, position, flight angles, acceleration and fuel level (in l) every interval seconds and sends a GetData rule with this information to the nearest airport. It will then clone itself with nextExecution=currentime+interval.

Pseudo code:

```
dataAcquisition:
  long planeID=getPlaneID();
  String planeModel=getPlaneModel();
  float longitude=getCurrentLatitude();
  float latitude=getCurrentLongitude();
  float altitude=getCurrentAltitude();
  float speed=getCurrentSpeed();
  float acceleration=getCurrentAcceleration();
  float roll= getCurrentRollAngle();
  float pitch= getCurrentPitchAngle();
  float yaw= getCurrentYawAngle();
  float fuel= getCurrentFuelLevel();
activationGuards:
  if(getCurrenttime()>=nextExecution){
    return true;
  }else{
    return false;
  }
conditions:
  return true;
actions:
  sendGetDataRuleToAirport(planeID,planeModel,longitude,latitude
      ,altitude,speed,acceleration,roll,pitch,yaw,fuel);
rule generation:
  if(activationGuards()){
    createUpdateDataRule(nextExecution+interval,interval);
  }else{
    createUpdateDataRule(nextExecution,interval);
  }
```

5.1.5.2 Change Direction

Parameters:

duration	How long the auto-pilot will follow this new route before going back to its original one
roll	Future roll angle of the plane (0ï¿½ if wings both have the same altitude)
pitch	Future pitch angle of the plane (positive when gaining altitude)
yaw	Future yaw angle of the plane (0ï¿½ when heading north)
speed	Future speed of the plane

Description:
This rule receives route correction data from the airport and passes them as a command to the auto-pilot of the plane.
Pseudo code:

```
dataAcquisition:
  None
activationGuards:
  return true;
conditions:
  return true;
actions:
  commandAutoPilot(duration,roll,pitch,yaw,speed);
rule generation:
  None
```

5.1.6 Airport rules

5.1.6.1 Get Data

Parameters:

planeID	Id of the plane that sent this information
planeModel	Model of the plane that sent this information
longitude	Latest longitude position of the plane
latitude	Latest latitude position of the plane
altitude	Latest altitude of the plane
speed	Speed of the plane
acceleration	Acceleration of the plane
roll	Roll angle of the plane (0ï¿½ if wings both have the same altitude)
pitch	Pitch angle of the plane (positive when gaining altitude)
yaw	Yaw angle of the plane (0ï¿½ when heading north)
fuelLevel	Fuel level of the plane (in l)

Description:
If the plane is already registered, this rule will save the data into the database and update the time a message was last received for the plane with the id planeID. Then it will create a CheckCollision rule for this plane. If the plane is not registered, this rule will first create a RegisterNewPlane rule with all the data and the create a CheckCollision rule for this plane.
Pseudo code:

```
dataAcquisition:
  boolean registered=isRegistered(planeID);
activationGuards:
  return registered;
conditions:
  return true;
actions:
    saveToDatabase(planeID,longitude,latitude,altitude,speed,
        acceleration,roll,pitch,yaw,fuelLevel);
    updateLastReceivedMessageTime(planeID,currentTime);

rule generation:
  if(!activationGuards()){
    createRegisterNewPlaneRule(planeID,planeModel,longitude,
        latitude,altitude,speed,acceleration,roll,pitch,yaw,
        fuelLevel);
  }
  createCheckCollisionRule(planeID);
```

5.1.6.2 Register new plane

Parameters:

planeID	Id of the new plane
planeModel	Model of the new plane
longitude	Longitude position of the plane
latitude	Latitude position of the plane
altitude	Altitude of the plane
speed	Speed of the plane
acceleration	Acceleration of the plane
roll	Roll angle of the plane (0ï¿½ if wings both have the same altitude)
pitch	Pitch angle of the plane (positive when gaining altitude)
yaw	Yaw angle of the plane (0ï¿½ when heading north)
fuelLevel	Fuel level of the plane (in l)

Description:
This rule will add the new plane to the list of registered planes along with the current time, which represents the time a message was last received from this plane. It will also save the initial data received by this plane.
Pseudo code:

```
dataAcquisition:
  None
activationGuards:
  return true;
conditions:
  return true;
actions:
  registerPlane(planeID,planeModel,getCurrentTime());
  saveToDatabase(planeID,longitude,latitude,altitude,speed,
      acceleration,roll,pitch,yaw,fuelLevel);
rule generation:
  None
```

5.1.6.3 Check Collision

Parameters:

planeID	Id of the airplane for which collisions should be checked

Description:
This rule creates a CheckCollisionSinglePlane for every airplane that is in a radius of 500 km of the airplane for which possible collisions should be checked.

Pseudo code:

```
dataAcquisition:
  float longitude=getLatestLongitude(planeID);
  float latitude=getLatestLatitude(planeID);
  float altitude=getLatestAltitude(planeID);
activationGuards:
  return true;
conditions:
  return true;
actions:
  List<long> nearAirplaneIDs=calculateNearAirplanes(longitude,
      latitude,altitude,500);
rule generation:
  foreach(id in nearAirplaneIDs){
    createCheckCollisionSinglePlaneRule(planeID,id);
  }
```

5.1.6.4 Unregister Plane

Parameters:

planeID Id of the airplane which should be unregistered

Description:

This rule removes all received data from a plane and unregisters the plane so that calculations for other planes will not be falsified by the data of a lost plane.

Pseudo code:

```
dataAcquisition:
  None
activationGuards:
  return true;
conditions:
  return true;
actions:
  removeAllData(planeID);
  unregisterPlane(planeID);
rule generation:
  None
```

5.1.6.5 Calculate Collision Free Route

Parameters:

planeID Id of the airplane for which collisions should be checked

Description:

This rule calculates new angles for the given airplane based on all airplanes

in a radius of 500km and sends a ChangeDirection rule with the calculated data to the airplane with the id planeID.
Pseudo code:

```
dataAcquisition:
  None
activationGuards:
  return true;
conditions:
  return true;
actions:
  Route newRoute=calculateNewRoute(planeID,500);
  long duration=newRoute.getDuration();
  long roll=newRoute.getRoll();
  long pitch=newRoute.getPitch();
  long yaw=newRoute.getYaw();
  long speed=newRoute.getSpeed();
rule generation:
  sendChangeDirectionRuleToPlane(planeID,duration,roll,pitch,yaw
      ,speed);
```

5.1.6.6 Check Timeout

Parameters:

nextExecution	Time this rule will be executed for the next time
interval	Time between two executions of this rule
timeout	Time after which a plane is considered as timed out

Description:
This rule periodically checks for all unflagged planes (planes that are not considered yet as timed out) if they are timed out, meaning if the last message received from a plane was too long ago. For every timed out plane this rule then creates a CorrectDataRule.
Pseudo code:

```
dataAcquisition:
  List<Long> registeredPlaneIDs=
      getRegisteredAndUnFlaggedPlaneIDs();
activationGuards:
  if(getCurrenttime()>=nextExecution ){
    return true;
  }else{
    return false;
  }
conditions:
  return true;
actions:
```

```
    foreach(planeID in registeredPlaneIDs){
      long lastReceived=getTimeOfLastReceivedMessage();
      if(lastReceived-getCurrentTime()>timeout){
        createCorrectDataRule(getCurrentTime(),interval,planeID);
        flag(planeID);
      }
    }
  rule generation:
    if(activationGuards()){
      createCheckTimeoutRule(nextExecution+interval,interval,
          timeout);
    }else{
      createCheckTimeoutRule(nextExecution,interval,timeout);
    }
```

5.1.6.7 Correct Data

Parameters:

nextExecution	Time this rule will be executed for the next time
interval	Time between two executions of this rule
planeID	ID of the plane whose data needs to be corrected
lastExecution	The time this rule was last executed

Description:
This rule periodically calculates the new position,speed and fuel Level of a given plane based on the last position, speed, acceleration and angles. If the altitude or the fuel Level of the calculated data reaches 0, the plane is considered as lost and a UnregisterPlane rule is created. If the time of the last received message from the plane is more recent then this rule, this rule stops cloning itself and unflags the plane.
Pseudo code:

```
  dataAcquisition:
    long lastReceived=getTimeOfLastReceivedMessage();
    boolean endExecution=(lastReceived>lastExecution);
  activationGuards:
    if(getCurrenttime()>=nextExecution ){
      return true;
    }else{
      return false;
    }
  conditions:
    return !endExecution;
  actions:
    calculateAndSaveNewData(planeID);
  rule generation:
    if(!endExecution){
      if(activationGuards()){
```

```
        createCorrectDataRule(nextExecution+interval,interval,
            planeID,getCurrentTime());
      }else{
        createCorrectDataRule(nextExecution,interval,planeID,
            lastExecution);
      }
    }else{
      createUnregisterPlaneRule(planeID);
      unFlag(planeID);
    }
```

5.1.6.8 Check Collision Single Plane

Parameters:

planeID	Id of the airplane for which collisions should be checked
planeID2	Id of an airplane that is in a radius of 500km of the first airplane

Description:
This rule checks whether two airplanes will collide in the next 15 minutes if they both continue to fly in the same way like they did when the data was updated last. If there will be a collision a CalculateCollisionFreeRoute rule is created.
Pseudo code:

```
dataAcquisition:
  None
activationGuards:
  return true;
conditions:
  return true;
actions:
  boolean collision=checkCollision(planeID,planeID2,15);
rule generation:
  if(collision){
    createCalculateCollisionFreeRouteRule(planeID);
  }
```

5.2 House temperature negotiation (System-User)

5.2.1 General description and assumptions

In this application a house possesses a proactive engine that is connected to the heating system. It will check which smartphones are currently in the house and ask for a preference for the temperature set by the user of the smartphone. It will then try to find a common agreement on the ideal

temperature for the house with respect to the preferences and the priorities of each user (devices that visit the house more often get a higher priority) and adapt the temperature through the heating system. For this application we suppose that every user visiting the house possesses a smartphone with Bluetooth enabled and has a minimum and maximum preferred temperature set in the preferences.

5.2.2 Template usage

In this application we are creating a GPaS that involves a system that has no direct interaction with a user (PE of the house) and systems that are running on the smartphones of the users and receive indirect input from the user by reading the preferences set by the user. As the application to regulate the temperature should react to the presence of users in the house it needs to satisfy the ambient property as well as the continuous property as there needs to be a rule that keeps the GPaS running. This is achieved by one rule which clones itself at every execution (continuous) and at every activation calls the proximity manager of the PE, which uses Bluetooth in this case to determine which users are currently in the house (ambient).
For the mandatory collaborative and fault-tolerant properties we chose to use one of the active collaboration templates in this case, more precisely the all-at-once one. In fact, in this application, the PEs on the smartphones do not know when they need to send data. Thus using the reactive collaboration template makes no sense. Furthermore, for this case, the all-at-once collaboration template has a slight advantage over the one-by-one template in combination with the fault-tolerance pattern. If a user shuts down its phone after the detection of his device and before his device could communicate the preferences of the user to the system of the house, using the all-at-once collaboration template will allow the fault-tolerant template to still take into account the preferences of all the other available users while using the one-by-one template could possibly lead to every user being ignored (if the first device disconnected), at least until the next execution of the GPaS.

5.2.3 Static diagram

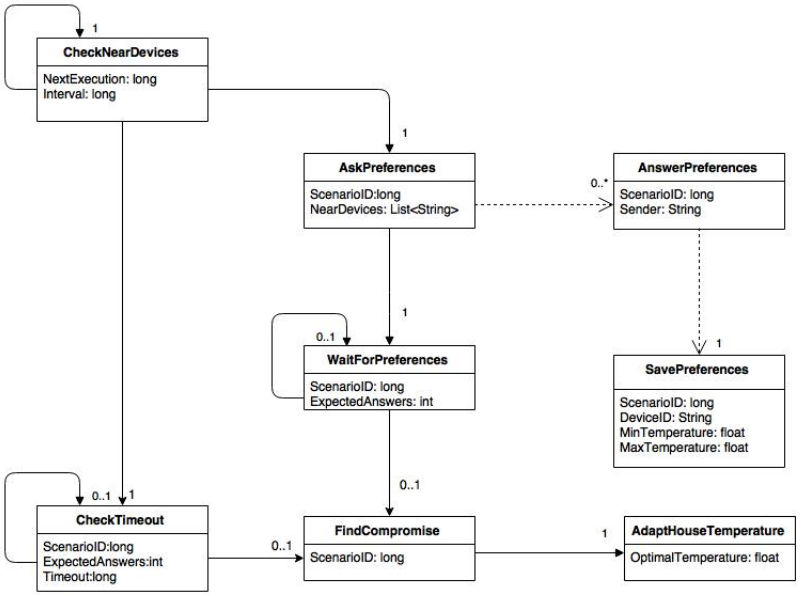

Figure 5.6: Normal execution for the temperature negotiation

5.2.4 Sequence diagrams

For this application we consider 2 different executions. In the first one (Figure 5.7) everything is happening like expected. The proactive engine of the house detects the devices of the guests that are currently in the house. It then proceeds to ask all of them for their preferences and waits for their answers. After that every answer arrived it then tries to find a compromise between the present guests and adapts the house temperature accordingly.

In the second execution (Figure 5.8) the proactive engine of the house again detects all the guests of the house and asks them for their preferences. But this time something went wrong and at least one of the devices did not respond by sending back its preferences. After the some time the check timeout rule detects this and unregisters the scenario so that future answers (if they would arrive) will be ignored. The engine then just bases its decision for the optimal temperature on the preferences it received and adapts the house temperature accordingly.

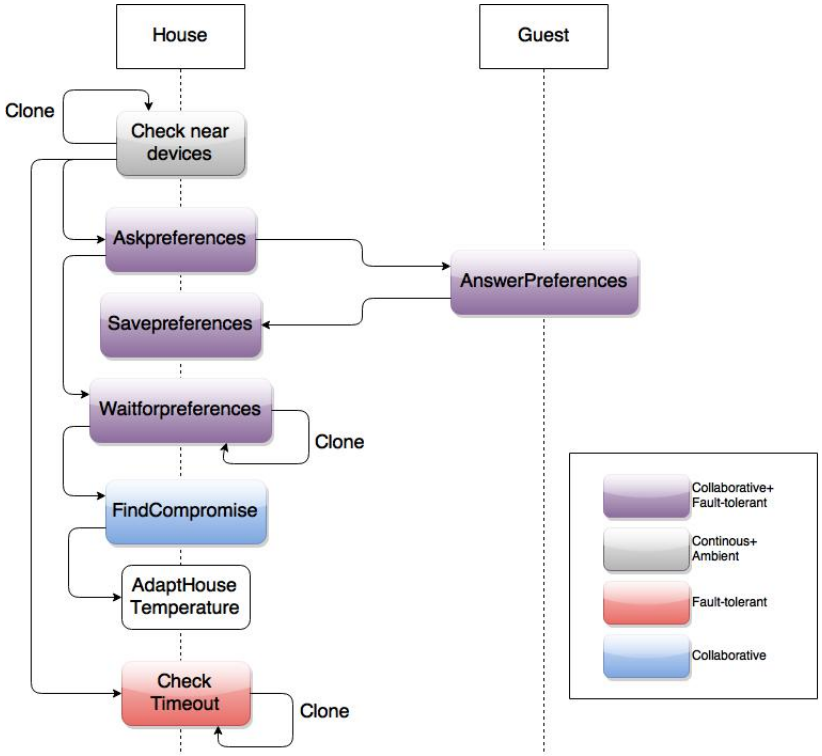

Figure 5.7: Normal execution for the temperature negotiation

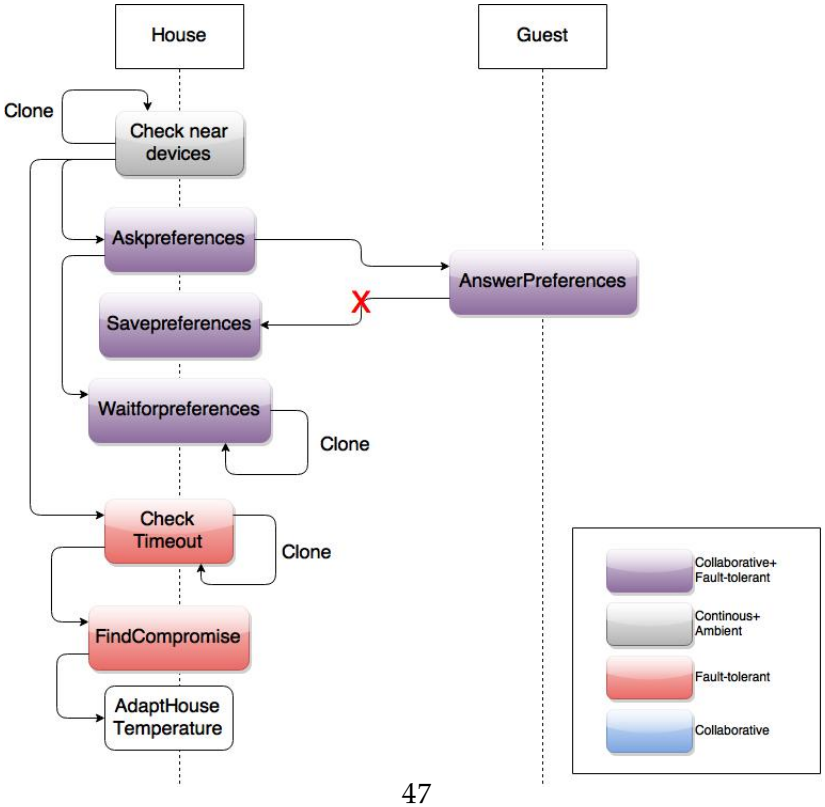

Figure 5.8: Execution for temperature negotiation with timed out guests

5.2.5 House rules

5.2.5.1 Check near devices

Parameters:

nextExecution	Time this rule should be executed for the next time
interval	Time interval between two executions of this rule

Description:

This rule checks which devices are currently in the house (through Bluetooth) and updates the priority level of each device. It then registers the current instance of the scenario in the engine and gets a ScenarioID, which it uses to create an AskPreferences rule with the list of near devices and a CheckTimeout rule with the number of devices. Finally this rule will clone itself.

Pseudo code:

```
dataAcquisition:
  None
activationGuards:
  if(getCurrentTime()>nextExecution){
    return true;
  }else{
    return false;
  }
conditions:
  return true;
actions:
  List<String> nearDevicesList= ProximityManager.getNearDevices
      ();
  updatePriorities(nearDevicesList);
  long scenarioID=registerNewScenario("TemperatureNegotiation");
rule generation:
  if(activationGuards()){
    createAskPreferencesRule(scenarioID,nearDevicesList);
    createCheckTimeoutRule(scenarioID,nearDevicesList.size()
        ,300);
    createCheckNearDevicesRule(nextExecution+interval,interval);
  }else{
    createCheckNearDevicesRule(nextExecution,interval);
  }
```

5.2.5.2 Ask preferences

Parameters:

scenarioID	ID of this scenario instance
nearDevices	List of devices that are currently in the house

Description:
This rule sends an AnswerPreferences rule to every device in the near devices list and then creates a WaitForPreferences rule with the ScenarioID and the number of near devices.
Pseudo code:

```
dataAcquisition:
  boolean registered=isRegistered(scenarioID);
activationGuards:
  return registered;
conditions:
  return true;
actions:
  foreach(device in nearDevices){
    sendAnswerPreferencesRuleTo(device,scenarioID,
        getLocalDeviceID());
  }
rule generation:
  if(activationGuards()){
    createWaitForPreferencesRule(scenarioID,nearDevices.size());
  }
```

5.2.5.3 Save preferences

Parameters:

scenarioID	ID of this scenario instance
deviceID	ID of the device that sent this preference
minTemperature	Minimum preferred temperature
maxTemperature	Maximum preferred temperature

Description:
This rule saves the received preferences into the database, but only if the scenario is still registered.
Pseudo code:

```
dataAcquisition:
  boolean registered=isRegistered(scenarioID);
activationGuards:
  return registered;
conditions:
  return true;
actions:
  saveIntoDatabase(scenarioID,deviceID,minTemperature,
      maxTemperature);
rule generation:
  None
```

5.2.5.4 Wait for preferences

Parameters:

scenarioID	ID of this scenario instance
expectedAnswers	The number of devices that are supposed to send back their preferences

Description:
This rule waits until all answers for the preferences arrived and then creates a FindCompromise rule. If the scenario was unregistered due to a timeout this rule stops.
Pseudo code:

```
dataAcquisition:
  boolean registered=isRegistered(scenarioID);
  int numberOfReceivedAnswers= getNumberOfReceivedAnswers(
      scenarioID);
activationGuards:
  return registered;
conditions:
  return true;
actions:
  None
rule generation:
  if(activationGuards()){
    createWaitForPreferencesRule(scenarioID,expectedAnswers);
    if(numberOfReceivedAnswers==expectedAnswers){
      createFindCompromiseRule(scenarioID);
    }
  }
```

5.2.5.5 Find Compromise

Parameters:

scenarioID	ID of this scenario instance

Description:
This rule retrieves the saved preferences by using the scenarioID and tries to find the best temperature based on the preferred temperature levels and the priorities of the different guests. It then unregisters the Scenario and creates an AdaptHouseTemperature rule with the calculated optimal temperature.
Pseudo code:

```
dataAcquisition:
  boolean registered=isRegistered(scenarioID);
activationGuards:
  return registered;
conditions:
```

```
    return true;
  actions:
    float optimalTemperature=calculateOptimalTemperature(
        scenarioID);
    unregister(scenarioID);
  rule generation:
    if(activationGuards()){
        createAdaptHouseTemperatureRule(optimalTemperature);
    }
```

5.2.5.6 Adapt House Temperature

Parameters:

optimalTemperature	The optimal temperature calculated based on the preferences of the guests

Description:
This rule will notify the heating system about the new temperature that should be reached in the house.
Pseudo code:

```
  dataAcquisition:
    None
  activationGuards:
    return true;
  conditions:
    return true;
  actions:
    HeatingSystem.setTemperature(optimalTemperature);
  rule generation:
    None
```

5.2.5.7 Check timeout

Parameters:

scenarioID	ID of this scenario instance
expectedAnswers	Number of expected answers
timeout	time after which the scenario will be unregistered

Description:
This rule checks whether the time limit for the collaboration was exceeded. If it was it checks if any devices answered and if this is the case it creates a FindCompromise rule. Otherwise it will just unregister the scenario and stop. If all answers arrived this rule will stop.
Pseudo code:

```
  dataAcquisition:
```

```
    int numberOfReceivedAnswers=getNumberOfReceivedAnswers(
        scenarioID);
    long scenarioCreationTime=getScenarioCreationTime(scenarioID);
  activationGuards:
    return (numberOfReceivedAnswers!=expectedAnswers);
  conditions:
    return true;
  actions:
    None
  rule generation:
    if(activationGuards()){
        if(scenarioCreationTime-getCurrentTime()>timeout){
          if(numberOfReceivedAnswers>0){
            createFindCompromiseRule(scenarioID);
          }else{
            unregister(scenarioID);
          }
        }else{
          createCheckTimeoutRule(optimalTemperature,
              expectedAnswers, timeout);
        }
    }
```

5.2.6 Guest rules

5.2.6.1 AnswerPreferences

Parameters:

scenarioID	ID of this scenario instance
sender	Id of the engine that asked for the preferences of this device

Description:
This rule simply sends the preferences of this device to the engine that asked for it.
Pseudo code:

```
  dataAcquisition:
    float minTemperature=getMinTempFromPreferences();
    float maxTemperature=getMaxTempFromPreferences();
  activationGuards:
    return true;
  conditions:
    return true;
  actions:
    sendSavePreferencesRuleTo(sender,scenarioID,getLocalDeviceID()
        ,minTemperature,maxTemperature);
  rule generation:
    None
```

5.3 FAME (User-User)

5.3.1 General description and assumptions

In short, this application helps the user find meeting dates and times for a group of people. The user has to indicate a time span for when the meeting should take place (for example next week), set a duration and choose the participants from a list of friends. The application will then calculate free time slots based on the calendars on the smartphones. The user then will receive a list of possible meeting dates from which he can choose one. Finally his choice is communicated to the other participants and added to everyones calendar.

For this application we assume that the user already has a few people in his friends list (with the ID of their devices needed for the communication). We also assume that everyone participating in the negotiation has synchronized his calendar on his smartphone.

5.3.2 Template usage

In this application the GPaS initially needs specific information of a user before it can be started. It thus needs to satisfy the 'initiated' property for which the user has to enter data through a user interface, and only after that the first rule of the GPaS is created with the information entered. The GPaS also needs input from the user at certain points of the scenario. Here we can differentiate between two different cases. The first case is the part of the scenario for which the data needed for taking a decision can be foreseen. For this application this would be the situation when the GPaS is fetching data from the calendar of the users (Monitored template with user preferences 4.12). The second case is when the GPaS is unable to make a clear decision with the preferences of the user at his disposal. In this case the GPaS proposes all possible solutions (here meeting dates) to the user (Monitored template with user interface 4.12), by sending him a notification. After the user made a decision the next rule of the GPaS is created and the execution can continue.

Concerning the mandatory collaborative and fault-tolerant properties, we chose to use the one-by-one collaborative template, which has slight advantages for this application in comparison of the all-at-once collaboration template. For this application it makes no sense to ask every participant of the meeting for their preferences if it is already sure that no meeting can be found after contacting the first one. Also if a device does not respond because it is offline, it is not useful to continue the rest of the execution as the user wants everyone in the participants list to attend the meeting.

Therefore the fault-tolerant template should make sure that the user is notified about potential problems so that he can decide whether he wants to restart the GPaS with different parameters or maybe try again.

5.3.3 Static diagram

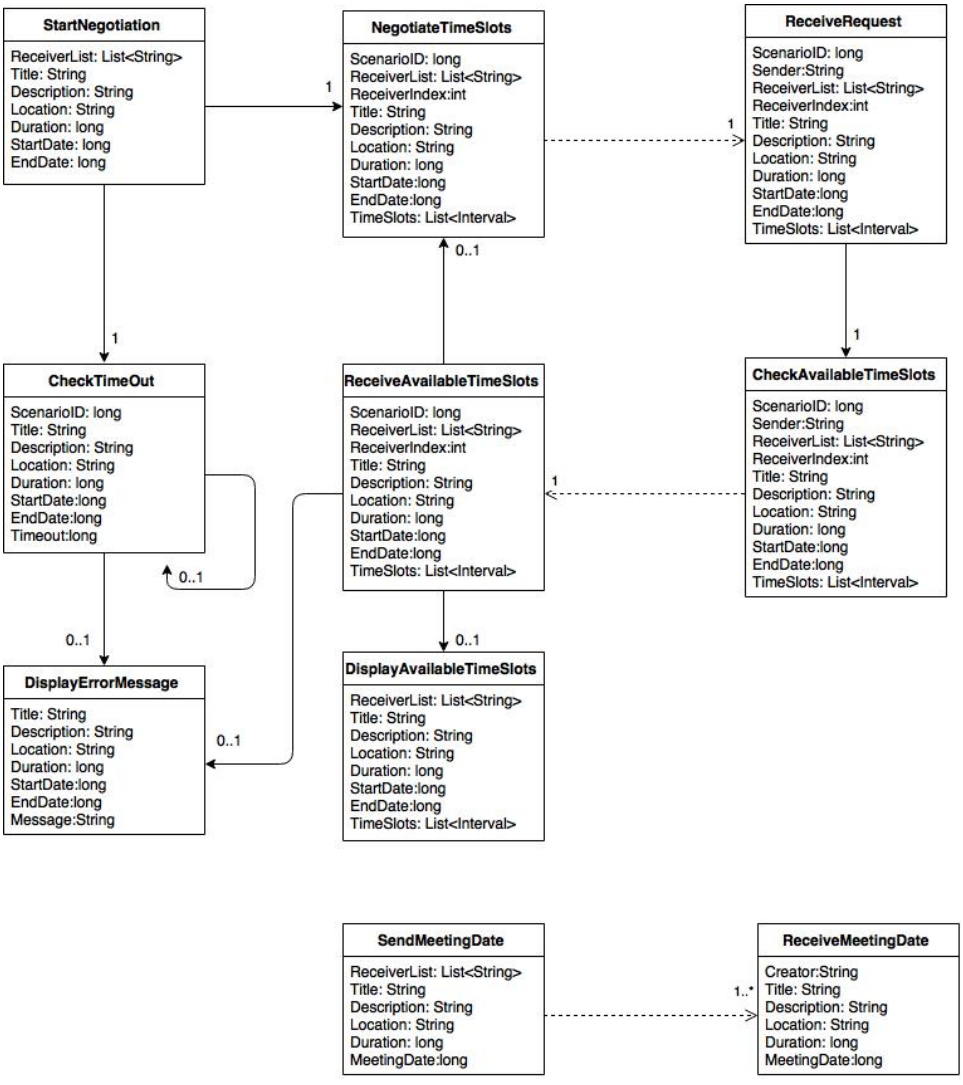

Figure 5.9: FAME GPaS : Static diagram

5.3.4 Sequence diagrams

For this application we present two sequence diagrams. In the first one (Figure 5.10), the GPaS does not encounter any issues. The user initiates

the GPaS by entering the required data in the user interface. Then the local calendar is fetched and unavailable time slots are eliminated from the remaining result set. Then this result set is send to the first participant of the meeting. Again, unavailable time slots are removed from the remaining result set and the data is send back to the initiator of the GPaS, which will then relay the data to the next participant. This continues without errors until every participant removed his unavailable time slots from the final result set. The result set is then presented to the user that initiated the GPaS and he chooses one of the possible dates. The final choice is communicated to the participants of the meeting.

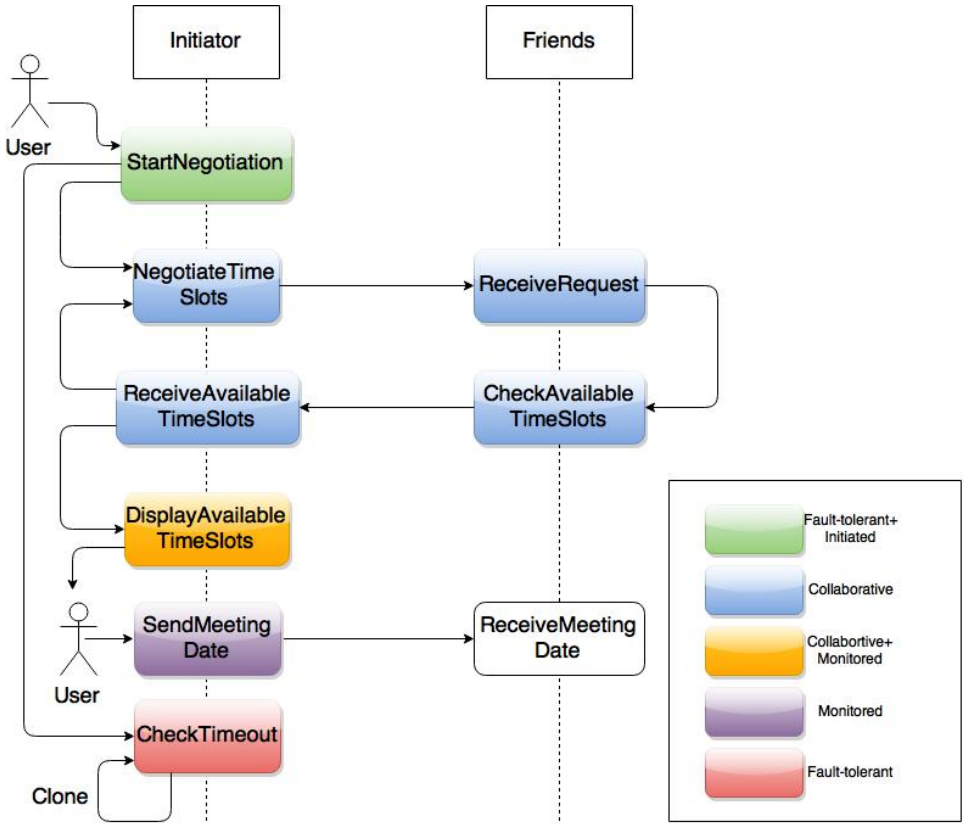

Figure 5.10: Normal execution of FAME GPaS

In the second diagram (Figure 5.11) things do not go as well. The GPaS executes normally until the time limit for the negotiation process is exceeded. An error message is displayed to the user.

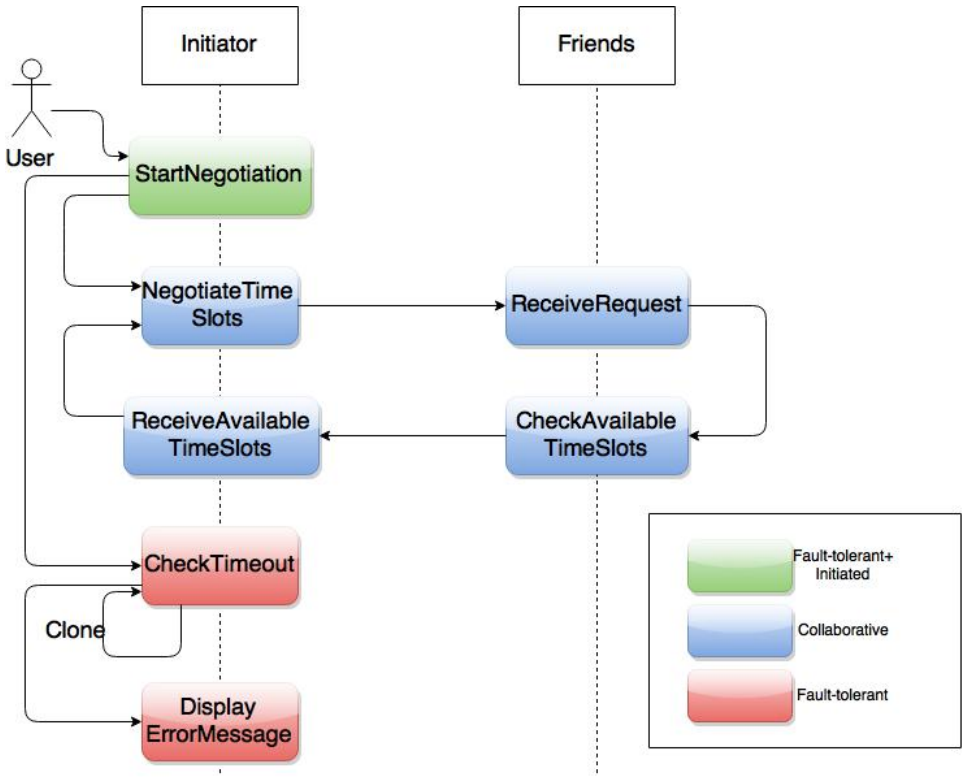

Figure 5.11: Failed execution of FAME GPaS

5.3.5 Initiator rules

5.3.5.1 Start negotiation

Parameters:

receiverList	List of participants
title	Title of the meeting
description	Description of the meeting
location	Location where the meeting will take place
duration	Duration of the meeting
startDate	Beginning of time period for which a meeting should be found
endDate	End of time period for which a meeting should be found

Description:

This rule is created after the user entered the necessary data. It will first register this instance of the scenario and get back a scenarioID. It will then calculate the available time slots based on the startDate, endDate, duration and on the meetings in the calendar of the user. Finally it will create a Ne-

gotiateTimeSlots rule with the calculated data as well as a CheckTimeout rule.
Pseudo code:

```
dataAcquisition:
  List<TimeSlot> meetingDates=
      getUnavailableTimeslotsFromCalendar(startDate,endDate);
activationGuards:
  return true;
conditions:
  return true;
actions:
  long scenarioID=registerNewScenario(ï¿½FAMEï¿½) ;
  List<TimeSlot> availableTimeSlots=calculateAvailableTimeslots(
      startDate,endDate,duration,meetingDates);
rule generation:
  createNegotiateTimeSlotsRule(scenarioID, receiverList,0, title
      , description, location, duration, startDate, endDate,
      availableTimeSlots);
  createCheckTimeoutRule(scenarioID,title,description,location,
      duration,startDate,endDate,1800);
```

5.3.5.2 Negotiate time slots

Parameters:

scenarioId	Id of the scenario in the initiators database
receiverList	List of participants
receiverIndex	Index of the next person in the list which will take part in the negotiation
title	Title of the meeting
description	Description of the meeting
location	Location where the meeting will take place
duration	Duration of the meeting
startDate	Beginning of time period for which a meeting should be found
endDate	End of time period for which a meeting should be found
timeSlots	List of available time slots for the meeting

Description:
This rule sends the data in form of a ReceiveRequest rule to the person in the receiverList at the index receiverIndex.
Pseudo code:

```
dataAcquisition:
  boolean registered=isRegistered(scenarioId);
activationGuards:
  return registered;
```

```
conditions:
  return true;
actions:
    String receiver=receiverList.get(receiverIndex);
    receiverIndex++;
    sendReceiveRequestRuleTo(receiver,scenarioId,
        getLocalDeviceId(),receiverList,receiverIndex,title,
        description,location,duration,startDate,endDate,
        timeSlots);
rule generation:
  None
```

5.3.5.3 Receive available time slots

Parameters:

scenarioId	Id of the scenario in the initiators database
receiverList	List of participants
receiverIndex	Index of the next person in the list which will take part in the negotiation
title	Title of the meeting
description	Description of the meeting
location	Location where the meeting will take place
duration	Duration of the meeting
startDate	Beginning of time period for which a meeting should be found
endDate	End of time period for which a meeting should be found
timeSlots	List of available time slots for the meeting

Description:

If the scenario was unregistered, this rule will not perform any actions. If the timeSlots list is empty, meaning that there is no available date for a meeting, this rule will unregister the scenario and create a DisplayErrorMessage rule. If the end of the receiverList was reached, the algorithm has finished and this rule will also unregister the scenario and create a DisplayAvailableTimeSlots rule with the list of available time slots. Otherwise it will create a NegotiateTimeSlots rule in order to continue the negotiation.

Pseudo code:

```
dataAcquisition:
  boolean registered=isRegistered(scenarioId);
activationGuards:
  return registered;
conditions:
  return empty(timeSlots) || (receiverIndex>=receiverList.size()
      );
```

```
actions:
  unregister(scenarioId);
rule generation:
  if(activationGuards()){
    if(!empty(timeSlots)){
      if(receiverIndex<receiverList.size()){
        createNegotiateTimeSlotsRule(scenarioId,receiverList,
            receiverIndex,title,description,location,duration,
            startDate,endDate,timeSlots);
      }else{
        createDisplayAvailableTimeSlotsRule(receiverList,title,
            description,location,duration,startDate,endDate,
            timeSlots);
      }
    }else{
      createDisplayErrorMessageRule(title,description,location,
          duration,startDate,endDate,"No possible meetings");
    }
  }
```

5.3.5.4 Display available time slots

Parameters:

receiverList	List of participants
title	Title of the meeting
description	Description of the meeting
location	Location where the meeting will take place
duration	Duration of the meeting
startDate	Beginning of time period for which a meeting should be found
endDate	End of time period for which a meeting should be found
timeSlots	List of available time slots for the meeting

Description:

This rule sends a notification to the user with the list of possible meeting dates and times from which the user can choose one.

Pseudo code:

```
dataAcquisition:
  None
activationGuards:
  return true;
conditions:
  return true;
actions:
  NotificationManager.displayChooseMeetingDateNotification(
      receiverList,title,description,location,duration,startDate
```

```
        ,endDate,timeSlots);
  rule generation:
    return true;
```

5.3.5.5 Send meeting date

Parameters:

receiverList	List of participants
title	Title of the meeting
description	Description of the meeting
location	Location where the meeting will take place
duration	Duration of the meeting
meetingDate	Final date for the meeting

Description:
This rule will add the chosen date for the meeting into the calendar of the user along with the participants (this way the participants will receive an email where they can choose to accept or decline the meeting) and notify all participants about it by sending a ReceiveMeetingDate rule to them.

Pseudo code:

```
  dataAcquisition:
    None
  activationGuards:
    return true;
  conditions:
    return true;
  actions:
    addToCalendar(title,description,location,duration,meetingDate,
        receiverList);
    foreach(receiver in receiverList){
      sendReceiveMeetingDateTo(receiver,getLocalusername(),title,
          description,location,duration,meetingDate);
    }
  rule generation:
    None
```

5.3.5.6 Check timeout

Parameters:

scenarioId	Id of the scenario in the initiators database
title	Title of the meeting
description	Description of the meeting
location	Location where the meeting will take place
duration	Duration of the meeting
startDate	Beginning of time period for which a meeting should be found
endDate	End of time period for which a meeting should be found
timeout	Time limit after which the scenario will be stopped

Description:
If the duration of the scenario exceeds the time limit, this rule will create a DisplayErrorMessage rule with the appropriate message.
Pseudo code:

```
dataAcquisition:
  boolean registered=isRegistered(scenarioId);
  long scenarioCreationTime=getScenarioCreationTime(scenarioId)
      ;
activationGuards:
  return registered;
conditions:
  return true;
actions:

rule generation:
  if(activationGuards()){
    if(scenarioCreationTime-getCurrentTime()>timeout){
      createDisplayErrorMessageRule(title,description,location,
          duration,startDate,endDate,"Negotiation timed out");
    }else{
      createCheckTimeouRule(scenarioId,title,description,
          location,duration,startDate,endDate,timeout);
    }
  }
```

5.3.5.7 Display error message

Parameters:

title	Title of the meeting
description	Description of the meeting
location	Location where the meeting will take place
duration	Duration of the meeting
startDate	Beginning of time period for which a meeting should be found
endDate	End of time period for which a meeting should be found
message	Error message to be displayed

Description:
This rule displays an error message to the user with the reason why no meeting dates where found.
Pseudo code:

```
dataAcquisition:
  None
activationGuards:
  return true;
conditions:
  return true;
actions:
  NotificationManager.displayErrorMessageNotification(title,
      description,location,duration,startDate,endDate,message);
rule generation:
  None
```

5.3.6 Friends rules

5.3.6.1 Receive meeting date

Parameters:

creator	Creator of the meeting
title	Title of the meeting
description	Description of the meeting
location	Location where the meeting will take place
duration	Duration of the meeting
meetingDate	Date of the meeting

Description:
This rule displays the chosen meeting date to the participants.
Pseudo code:

```
dataAcquisition:
  None
```

```
activationGuards:
  return true;
conditions:
  return true;
actions:
  NotificationManager.displayNewMeetingNotification(creator,
      title,description,location,duration,meetingDate);
rule generation:
  None
```

5.3.6.2 Receive request

Parameters:	scenarioId	Id of the scenario in the initiators database
	sender	Id of the phone from which the request was sent
	receiverList	List of participants
	receiverIndex	Index of the next person in the list which will take part in the negotiation
	title	Title of the meeting
	description	Description of the meeting
	location	Location where the meeting will take place
	duration	Duration of the meeting
	startDate	Beginning of time period for which a meeting should be found
	endDate	End of time period for which a meeting should be found
	timeSlots	List of available time slots for the meeting

Description:
This rule creates a CheckAvailableTimeSlots rule with the data it received.

Pseudo code:

```
dataAcquisition:
  None
activationGuards:
  return true;
conditions:
  return true;
actions:
  None
rule generation:
  createCheckAvailableTimeSlotsRule(scenarioId,sender,
      receiverList,receiverIndex,title,description,location,
      duration,startDate,endDate,timeSlots);
```

5.3.6.3 Check available time slots

Parameters:

scenarioId	Id of the scenario in the initiators database
sender	Id of the phone from which the request was sent
receiverList	List of participants
receiverIndex	Index of the next person in the list which will take part in the negotiation
title	Title of the meeting
description	Description of the meeting
location	Location where the meeting will take place
duration	Duration of the meeting
startDate	Beginning of time period for which a meeting should be found
endDate	End of time period for which a meeting should be found
timeSlots	List of available time slots for the meeting

Description:
This rule calculates a new list of available time slots based on the old list and the calendar of the current phone. When finished it sends the updated list back to the initiator of the meeting request.
Pseudo code:

```
dataAcquisition:
  List<TimeSlot> meetingDates=
      getUnavailableTimeslotsFromCalendar(startDate,endDate);
activationGuards:
  return true;
conditions:
  return true;
actions:
  List<TimeSlot> availableTimeSlots=calculateAvailableTimeslots(
      timeSlots ,duration,meetingDates);
  sendReceiveAvailableTimeSlotsRuleTo(sender,scenarioId,
      receiverList,receiverIndex,title,description,location,
      duration,startDate,endDate,availableTimeSlots);
rule generation:
  None
```

6 Proof of concept:FAME

Contents

In these section we are going to present the implemented FAME application. This includes database tables (outside of the ones from the rules), User interface and the most important part, the algorithm to calculate the time slots where the user is available based on the meetings in the calendar and the free time slots that were calculated previously.

6.1 Database

Outside of the database tables that are used to store the different rules, the FAME application only needs two additional tables (Figure 6.1), one to register Scenarios for the error handling and one to save the friend list of the user. The confirmed column of the FAME_User is used to not display a friend if the friend did not accept the friend request yet.

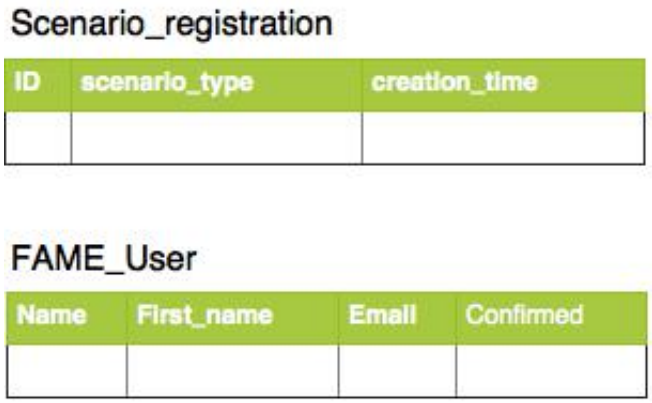

Figure 6.1: Extra database table

6.2 User interface

6.2.1 Start Negotiation screen

The Start Negotiation Screen, is the screen from which the GPaS is started (Figure 6.2). The user can enter the title of the meeting, a description of the meeting, select a period of time during which the meeting should be scheduled, choose a duration for the meeting, enter the location for the meeting and finally the User can add participants to the meeting from a list of friends (Figure 6.3).

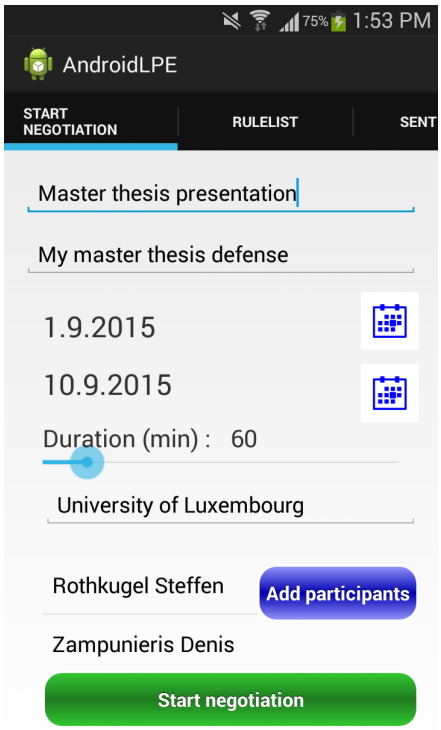

Figure 6.2: Start negotiation screen

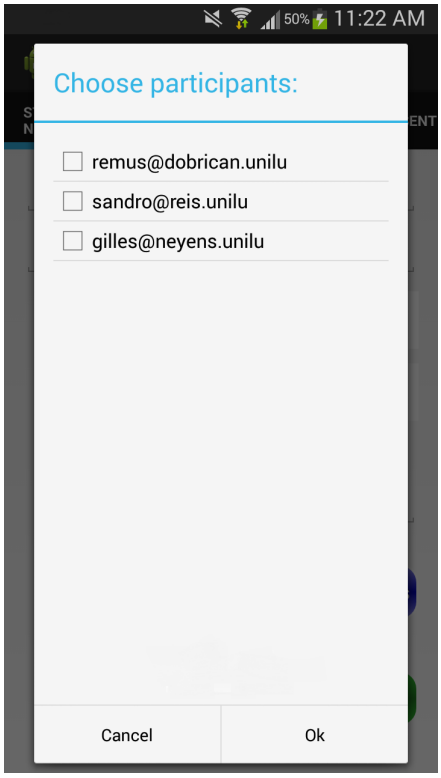

Figure 6.3: Choose participants screen

6.2.2 Meeting Suggestion Notification

The meeting suggestion notification (Figure 6.4) is the notification that displays the dates and times for which every participant of the meeting is available to the user who initiated the negotiation. The user can choose the date by checking the box in front of it and choose the available times for each date from a drop down menu. By pressing choose the user agrees on the date and time and the meeting is added to his and to the participants calendars. If none of the proposed dates is satisfactory, the user can cancel the negotiation.

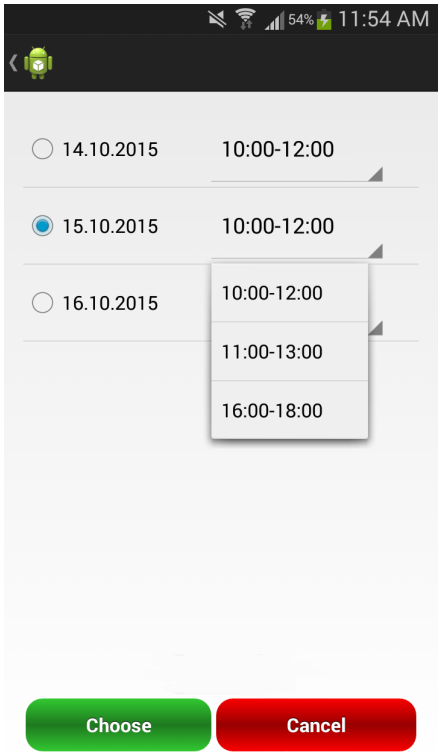

Figure 6.4: Meeting suggestion notification

6.3 Algorithm

The following algorithm calculates free time slots based on a list of time slots for which a user is not available, a list of time slots for which the previous users in the negotiation are available (Only one time slot with the start and end date chosen by the user if we are at the beginning of the negotiation) and the duration of the meeting we want to schedule. For every time slot where the user is not available, it is checked whether it intersects with any free time slot. If it is the case the list of free time slots is updated. At the end of every iteration the free time slots that are shorter than the duration of the meeting we want to schedule are removed.

```
private static ArrayList<TimeSlot> calculateFreeTimeSlots(
    ArrayList<TimeSlot> free,ArrayList<TimeSlot> occupied, long
    duration){
    for(int j=0;j<occupied.size();j++){
      ArrayList<Integer> listToremove=new ArrayList<Integer>();
      ArrayList<TimeSlot> listToAdd=new ArrayList<TimeSlot>();
      for(int i=0;i<free.size();i++){
          //if the meeting intersects with the free slot
          if((occupied.get(j).getEnd()>free.get(i).getStart()) &&
```

```
            (occupied.get(j).getStart()<free.get(i).getEnd())){
          if(occupied.get(j).getStart()<=free.get(i).getStart())
              {
            if(occupied.get(j).getEnd()>=free.get(i).getEnd()){
                //Meeting fully overlaps with the free time slot
              //remove the time slot
              listToremove.add(i);
            }else if(occupied.get(j).getEnd()<free.get(i).getEnd
                ()){ //Meeting only partially overlaps with the
                free time slot
              //remove time slot
              listToremove.add(i);
              //add a new time slot starting at the end of the
                  meeting
              listToAdd.add(new TimeSlot(occupied.get(j).getEnd
                  (), free.get(i).getEnd()));
            }
          }else if(occupied.get(j).getStart()>free.get(i).
              getStart()){
            if(occupied.get(j).getEnd()>=free.get(i).getEnd()){
                //Meeting partially overlaps with the free time
                slot
              //remove time slot
              listToremove.add(i);
              //add a new time slot ending at the beginning of
                  the meeting
              listToAdd.add(new TimeSlot(free.get(i).getStart(),
                  occupied.get(j).getStart()));
            }else if(occupied.get(j).getEnd()<free.get(i).getEnd
                ()){ //Meeting is haapening in the middle of a
                free time slot
              //remove time slot
              listToremove.add(i);
              //add a new time slot ending at the beginning of
                  the meeting
              listToAdd.add(new TimeSlot(free.get(i).getStart(),
                  occupied.get(j).getStart()));
              //add a new time slot starting at the end of the
                  meeting
              listToAdd.add(new TimeSlot(occupied.get(j).getEnd
                  (), free.get(i).getEnd()));
            }
          }
        }
      }
      for(int i=listToremove.size()-1;i>=0;i--){
        free.remove(listToremove.get(i).intValue());
      }
      for(int i=0;i<listToAdd.size();i++){
        free.add(listToAdd.get(i));
      }
      //remove time slots that are shorter than the duration for
          the meeting we want to schedule
      for(int i=free.size()-1;i>=0;i--){
```

```
            if(free.get(i).duration()<duration){
              free.remove(i);
            }
          }
        }
        return free;
}
```

6.4 Future features

User preferences To further enhance the users experience, it is possible to add several features to this application in the future. The most important one would be to allow the user to set preferences for when he wants and does not want to have meetings and which time of the day he prefers having a meeting when several options are available. Allowing the different users to further restrict the periods of time they are available will allow the algorithm to considerably reduce and refine the set of possible meeting dates and times, which will make the choice easier for the user initiating the meeting request. By having the user's preferences for some time of the day, the GPaS can rank the remaining possibilities for a meeting and help the initiator even more in making a choice.

Groups Especially in work environments meetings happen between the same group or core group of persons. Therefore it would be interesting to add an option for the users to create groups with people from their friend list, which will reduce the time needed to initiate the meeting request and make sure that one does not forget to invite all the most important persons for a meeting anymore.

Participants hierarchy Speaking of important persons for a meeting, it arrives quite often that not every person that is invited to a meeting absolutely needs to be there. It would therefore be interesting to have the possibility to choose importance levels for different participants and also have a minimum participants number. For the importance levels one could imagine different types of participants. There could be mandatory participants, which need to be available for the meeting, otherwise the meeting request fails. Optional participants would be nice to have for the meeting but they do not cause the meeting request to fail if not available. However, they can still influence the ranking of the meetings, which will then also take into account how many participants are available for a given period. One could also imagine a third type of participants called group participants (not necessary related to previously user defined groups). This type is useful if for example two work teams like the programming team and

the graphic design team need to find a meeting together. It is not mandatory for a specific person to attend the meeting, but there should be at least one person from each team (or at least two, etc.) that attends the meeting. The initiator of the request would then set the type for these participants to 'group participants' and assign them a group name and for each group a minimum number of participants. The algorithm will then fail if there are not enough available participants for each group.
When implementing this change it will probably be best to change the collaboration template from one-by-one to all-at-once and update the rules accordingly, as not every participant in the list absolutely needs to attend the meeting anymore.

Geolocation of meetings A current limitation of the FAME application is that it does not take into account the location of the different meetings in order to calculate available time slots for a meeting. This means that is possible to have a meeting that ends at 10 a.m. in Kirchberg, a meeting that starts at 10:05 a.m. and ends at 11 a.m. in Belval and a meeting again that starts at 11.05 a.m. in Kirchberg. To prevent this issue, the application needs to interpret the locations in the calendar as a GPS location and then for example use the Google Direction API in order to calculate the estimated time needed to get from one location to another and take this estimation into account when calculating available time slots.

Scalability and performance It is currently unknown how this application will perform when confronted with a large number of users. Therefore tests need to be made which check how the application handles meeting requests for a large group of people. These tests should include the time needed for finding meeting propositions as well as the impact of the negotiation on the performance of the phone, the battery usage and the amount of data transmitted. If the application performs poorly in these tests, possible solutions would be to limit the number of participants for a meeting and to limit the duration of the period in which the application will try to find a meeting.

7 Conclusion

This chapter summarizes the most important points of this thesis and presents several possible future research directions related to the work of this thesis.

Contents

7.1 Summary

The goal of this thesis was to find and define properties for GPaS and use these properties in order to create rule templates and design principles, which will allow us to create future GPaSs more easily. The main properties that we introduced were the collaborative and fault-tolerant properties, which both are important fro GPaS in the context of networks of proactive engines. Both of these properties lead to the creation of several rule templates, which have different advantages and disadvantages depending on the type of application one needs to create. However, what they all have in common is that they standardize and facilitate the future creation of GPaS and thus also the creation of applications based on the currently existing PE. We then defined a few properties depending on the actors involved in a GPaS.

The templates and design principles were then used to create the GPaSs for three example applications: Airplane collision avoidance, Temperature negotiation and FAME. It was made clear how the templates can be applied and used in different situations as well as how the different templates can interact and merge into each other in order to form a GPaS. Finally, we implemented the FAME application for Android-based smartphones, presented the user interface and algorithm needed for the realization of this application and gave a few suggestions for future improvements and additional features for this application at the end.

7.2 Future work

The work we presented in this thesis may still be enhanced in the future or even lead the project of proactive engines to other research directions. In this section we are going to present different possibilities for future work related to this thesis.

Extension and adaption of properties and templates The properties and templates presented in this thesis are not necessarily the only existing ones. It is possible that, while creating new applications, we discover and identify new properties of GPaS, which will then lead to the refinement or even the creation of new rule templates for GPaS. Each newly created rule template will then reduce the time needed for the creation of future GPaS and therefore also reduce the time for the creation of new applications using our PE.

Performance tests While we already performed some performance tests for mobile devices [5] it would be interesting to extend these tests for mobile devices and to benchmark the performance of the different rule templates presented in this thesis. These tests should include CPU and battery usage measurements as well as the total amount of data communicated during an execution of the same GPaS using different rule templates. The results will then allow us to optimize and refine the different templates for different situations.

Move in the direction of autonomous systems Proactive systems always have a human supervisor who makes sure that the system is running correctly and is responsible for the maintenance of the system. However, there are a many cases in which it is not optimal or even downright impossible to have a human taking care of these tasks. For example it is really dangerous for astronauts to do repairs in open space. Therefore the International Space Station has a robotic arm called Dextre, which does these repairs for them. However, one of the cameras on Dextre broke. If Dextre was a proactive system, this would require an astronaut to go and change it out, which would kind of defeat the purpose why Dextre was initially built. For such cases, autonomous systems were developed. Autonomous systems are systems that have different properties to help them to avoid as much as possible the intervention of a human administrator. Dextre is such an autonomous system and therefore he was able to exchange its broken camera with a working one. It would be interesting to see if, in the future, we could use our existing proactive system to create autonomous systems for different applications and how this solution compares to already existing autonomous systems in terms of performance, reliability and flexibility.

Bibliography

[1] R. Dobrican and D. Zampunieris, "A Proactive Approach for Information Sharing Strategies in an Environment of Multiple Connected Ubiquitous Devices," in *Proceedings of the 11th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC 2014)*. IEEE, 2014, pp. 763–771.

[2] A. Lella and A. Lipsman, "The US Mobile App report," http://www.comscore.com/Insights/Presentations-and-Whitepapers/2014/The-US-Mobile-App-Report, 2014, [Online; published 21-August-2014].

[3] M. Müller, "Connection of proactive engines," *University of Luxembourg*, 2013.

[4] G. Neyens, "Communication of proactive engines," *University of Luxembourg*, 2013.

[5] G. Neyens, R. A. Dobrican, and D. Zampunieris, "Enhancing Mobile Devices with Cooperative Proactive Computing," in *COLLA 2015, The Fifth International Conference on Advanced Collaborative Networks, Systems and Applications*, 2015.

[6] D. L. Tennenhouse, "Proactive Computing," in *Communications of the ACM*, 2000, pp. 43–50.

[7] R. Want, T. Pering, and D. L. Tennenhouse, "Comparing autonomic and proactive computing," in *IBM Systems Journal*, 2003, pp. 42(1):129–135.

[8] D. Shirnin, S. Reis, and D. Zampunieris, "Experimentation of proactive computing in context aware systems: Case study of human-computer interactions in e-learning environment," *Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2013 IEEE International Multi-Disciplinary Conference on*, pp. 272–279, 26-28 February 2013.

[9] S. M. Dias, S. Reis, and D. Zampunieris, "Personalized, Adaptive and Intelligent Support for Online Assignments Based on Proactive Computing," in *2012 IEEE 12th International Conference on Advanced Learning Technologies*. Rome, Italy: IEEE, Jul. 2012, pp. 668–669.

[10] R. A. Dobrican, S. Reis, and D. Zampunieris, "Empirical Investigations on Community Building and Collaborative Work inside a LMS

using Proactive Computing," in *Proceedings of E-Learn - World Conference on E-Learning 2013 Conference*. Theo Bastiaens and Gary Marks, 2013.

[11] D. Shirnin, "Formalising the twofold structure of a proactive system:proof of concept on deterministic and probabilistic levels," *University of Luxembourg*, 2014.

[12] S. Reis, D. Shirnin, and D. Zampunieris, "Design of proactive scenarios and rules for enhanced e-learning," in *CSEDU 2012 - Proceedings of the 4th International Conference on Computer Supported Education*. Porto, Portugal: SciTePress, 2012, pp. 253–258.

[13] D. Zampunieris, "Implementation of a Proactive Learning Management System," in *E-Learn World Conference on E-Learning in Corporate, Government, Healthcare and Higher Education*, Hawaii, 2006, pp. 3145–3151.

[14] D. J. Cook, J. C. Augusto, and V. R. Jakkula, "Ambient intelligence: Technologies, applications, and opportunities," *Pervasive and Mobile Computing*, vol. 5, no. 4, pp. 277–298, 2009.

[15] R. Boden, "Project Jacquard: Google and Leviâ€™s collaborate on interactive clothing," http://www.nfcworld.com/2015/06/01/335628/project-jacquard-google-and-levis-collaborate-on-interactive-clothing/, 2015, [Online; published 1-June-2015].

[16] S. Helal, W. Mann, J. King, Y. Kaddoura, E. Jansen *et al.*, "The gator tech smart house: A programmable pervasive space," *Computer*, vol. 38, no. 3, pp. 50–60, 2005.

[17] G. M. Youngblood, D. J. Cook, and L. B. Holder, "A learning architecture for automating the intelligent environment," in *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, vol. 20, no. 3. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005, p. 1576.

[18] K. Martinez, J. K. Hart, and R. Ong, "Environmental sensor networks," *Computer*, vol. 37, no. 8, pp. 50–56, 2004.

[19] A. Pentland, "Perceptual environments," *Smart Environments: Technologies, Protocols, and Applications*, pp. 345–359, 2005.

[20] D. Franklin, "Cooperating with people: The intelligent classroom," in *AAAI/IAAI*. Citeseer, 1998, pp. 555–560.

[21] M. Stottinger, "Context-awareness in industrial environments," *Software Engineering. Hagenberg, FH Hagenber*, vol. 68, 2004.

[22] J. C. Tang and S. L. Minneman, "Videodraw: A video interface for collaborative drawing," *ACM Trans. Inf. Syst.*, vol. 9, no. 2, pp. 170–184, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/123078.128729

[23] M. Handley and J. Crowcroft, "Network text editor (nte): A scalable shared text editor for the mbone," in *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '97. New York, NY, USA: ACM, 1997, pp. 197–208. [Online]. Available: http://doi.acm.org/10.1145/263105.263167

[24] M. Roseman and S. Greenberg, *ACM Transactions on Computer-Human Interaction (TOCHI)*, pp. 66–106.

Acronyms

FIFO First in First out. 3

GCM Google Cloud Messaging. 5, 6

GPaS Global Proactive Scenario. i, 1, 7, 9, 66, 72

PaS Proactive Scenario. i, 7, 9

PE Proactive Engine. 1, 3, 5, 7, 13, 33, 45, 72, 73

PEMD Proactive Engine for Mobile Devices. 2

PEMDs Proactive Engines for Mobile Devices. 5

PS Proactive System. 3

RRS Rule-running system. 3–5