The rest of this paper is structured as follows. In Section II we provide background on MAPE. Section III introduces the mobile learning application, describes the problem, and outlines the architecture of the self-adaptive solution. In Section IV, we describe in detail the behavioral models of the self-adaptive system. Section V describes the required properties and discusses verification results. Section VI briefly explains the mapping of behavioral models to implementation. We draw conclusions and outline plans for future work in Section VII.

II. Background and Related Work

MAPE was introduced as the conceptual core of an autonomic manager, which is central to IBM's framework for Autonomic Computing [2]. The MAPE components realize the primary functions of a feedback loop. The *Monitor* component gathers relevant information from the underlying managed system and the environment. The *Analyze* component assesses the collected data to determine the system's need to satisfy the adaptation objectives. The *Plan* component constructs the actions necessary to achieve the system's objectives. Finally, *Execute* component carries out changes on the managed system. The additional *Knowledge* component maintains representations of the managed system and environment, adaptation objectives, and other relevant state that is shared by the MAPE components. MAPE is therefore also referred as MAPE-K.

Rainbow [7] offers a reusable architectural framework for building self-adaptive systems. The architectural layer that deals with self-adaptation, resembles similarities with a MAPE loop. Rainbow supports monitoring and adaptation of software systems that are distributed in a network. However, the control of adaptation is centralized. Another interesting example of a centralized feedback loop is described in [8]. The authors propose an approach to achieve QoS for service-based systems through an external MAPE loop. Formally specified requirements are automatically analyzed to identify and enforce optimal system configurations. The approach uses Markov models and probabilistic computation tree logic, and focuses on improving response time and dealing with failures.

A number of authors have studied interactions between feedback loops, which are more or less explicitly modeled as MAPE loops. [9] expresses structural constraints over an architectural specification that are used by component managers to automatically configure the system. [10] introduces a gossip protocol to make this approach scalable. [11] makes control loops explicit and present a UML profile for control loops that extends UML modeling concepts. [12] extends MAPE with support for inter-loop and intra-loop coordination. [13] introduces the concept of adaptive goal in service-based systems. Adaptive goals are responsible for adapting the goal model at runtime when needed. [14] presents a reference model for adaptive software that supports separation of concerns among feedback loops required to address control objectives over time. Finally, [6] describes patterns of interacting MAPE loops derived from implemented self-adaptive systems.

The work presented in this paper contributes to the presented background with a rigorous specification and verification of the behavior of the distinct components of MAPE loops and their interactions, for a concrete application.

III. Towards a Robust M-Learning Application

In this section we give a brief summary of the mobile learning application we developed, we pinpoint the robustness problem we faced with insufficient GPS accuracy, and we outline how we tackled this problem by extending the design of the legacy system with a self-adaptation layer.

A. Mobile-Learning Application

The mobile learning application supports outdoor learning activities, where students use GPS-enabled mobile devices. A learning activity takes place in the context of a lecture (of 1 or 1.5 hour) and is composed of a set of tasks (typically 4 to 8 tasks). An example of a learning activity is to measure and calculate properties of triangles, and one concrete task is to use triangulation techniques to find locations on the field given the three side measurements of a triangle, and having two of the triangle locations already marked on the field. Fig. 1 shows a use case scenario, where three groups of students (represented by MVDs) perform tasks of a learning activity.

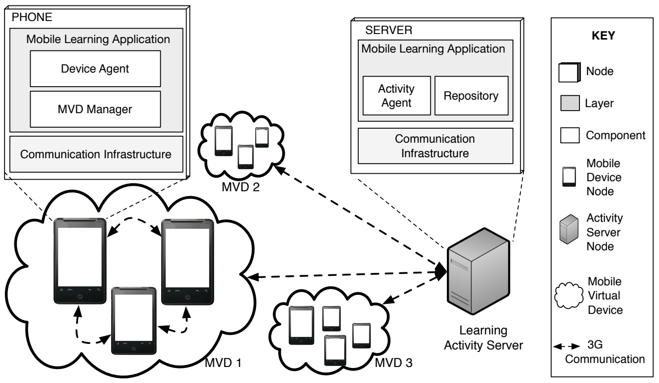

Fig. 1. Use case scenario of a learning activity

The application is conceived as a distributed agent-based system. A *Device Agent* deployed on each mobile *phone* provides the learning services to the student (gathering locations, calculating distances, etc.). The device agents of a group that work on the same tasks form an MVD. Within an MVD, one of the agents is elected as master, while the others serve as slaves. The *MVD Manager* is responsible for the management of the MVD. E.g., a new master is elected when the master phone runs out of energy. The master communicates via 3G with the *Server* using the *Communication Infrastructure*. Management of the tasks at the server is the responsibility of the *Activity Agent*. The master of each MVD receives new tasks from the activity agent at the server and reports the results back when a task is finished.

B. Problem Description

Due to changing environmental conditions, the GPS sensitivity can vary over time, which affects the accuracy of the measurements and may undermine the use of the application when conditions get worse. We were aware of the fact that GPS is not always accurate. However, it proved to be a bigger problem than our initial assumptions, up to the point where inaccurate measurements mislead students conclusions.

There are two main variables that determine the required quality of the GPS measurements during learning activities: the current GPS accuracy and the required level of accuracy for the task at hand. Fig. 2 illustrates how the GPS accuracy error typically evolves over time for a mobile device.

Depending on the given task, the allowed level of GPS accuracy errors can be different. As an example, an 8 meters accuracy error in the GPS acquisition has a higher impact when used in a 20 meter distance calculation than when used in a 60 meter distance. Therefore, there is a need for dynamically updating the required level of GPS accuracy for the measurements of each task. Fig. 2 shows two horizontal lines representing different accuracy requirements for Task-1 and Task-2, where the first task requires a lower level of accuracy (error lower than 11 meters) than the second (error lower than 7 meters).

The combination of the GPS quality and the requirements for a given task determine whether a mobile device is suitable to perform the distance measurements for the task. We marked two time instances in Fig. 2 representing points where the mobile device is not providing the required quality of measurements. At time stamp 20, while running Task-1, a 14 meter accuracy error is reported, failing the 11 meters accuracy that is required for the task. Notice that the available GPS quality may fulfill the requirements for one task, but fail for another task (see e.g., time stamp 40). From our experience, we learned that the required GPS quality of the mobile devices varies substantially. However, we noticed that during a learning activity, at most 20% of GPS modules failed to provide the required quality level, and this for a duration lower than 20% of the time of the learning activity. As a rule, we can state that (worst case in practice) less than 10% of the mobile devices are in an undesired state at the same time.

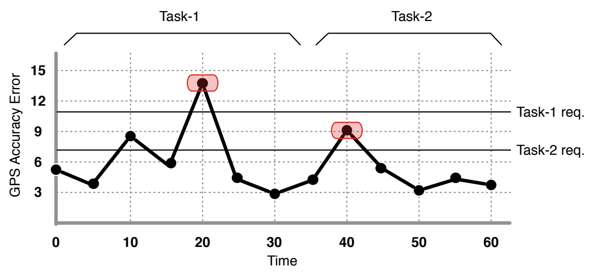

Fig. 2. Exemplification of GPS accuracy

In order to deal with failing devices, we need to take into account the requirements with respect to the required number of devices for the tasks. As explained before, a learning activity consists of a set of tasks. However, different tasks may require a different number of devices. For example, two mobile devices are sufficient to measure the diameter of a circle, while triangulation requires three mobile devices. Therefore, we need to take into account the number of required GPS devices per group (MVD) when handling failing devices. Typically, between 10 and 20% redundant phones are available.

Summarizing, the GPS accuracy of phones can degrade making them invalid for distance measurements. As a result, the number of mobile devices in a group may be insufficient to complete tasks successfully. Currently, the application does not support students with identifying the lack of sufficient quality of the GPS module and solving the problem by dynamically integrating available phones. To deal with this problem, we aim to enhance the current system with self-adaptation mechanisms that guarantee the robustness of the system with respect to decreasing GPS quality of mobile devices.

C. Adding a Self-Adaptive Layer

To realize the required robustness, we added a self-adaptive layer on top of the exiting system. We realized the self-adaptive layer using MAPE [2] loops, as shown in Fig. 3. Concretely, to provide robustness to the system we added two MAPE loops that deal with two concerns of robustness: the first loop deals with managing the availability of the GPS service based on the actual GPS service quality (left-side MAPE in Fig. 3); the second loop deals with managing the required number of GPS services of the current task for the MVD (right-side MAPE in Fig. 3). Probes and effectors enable the MAPE loops gathering the relevant information of the underlying managed system and applying the planned adaptations actions.

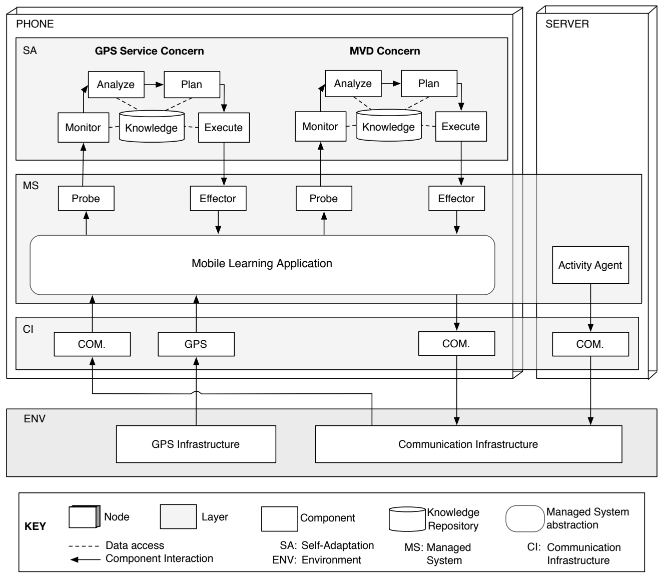

Fig. 3. Structural view of the self-adaptive system

The first MAPE loop (GPS Service Concern) is local to each mobile device. This loop monitors the quality of the GPS module, compares it with the required quality, and based on that, activates or deactivates the GPS service. When a GPS service is deactivated, it can trigger the second MAPE loop to start a self-healing process, that is, find a new device and add this to the MVD. We say *can* trigger, because there may be redundant phones in the MVD, so that no replacement is required.

The second MAPE loop (MVD Concern) is distributed over the devices of the MVD. This MAPE loop uses a master-slave pattern [6]. Fig. 4 shows the distribution of MAPE components for three phones. The master-slave pattern enables coordination of self-adaptation among nodes in a distributed system. Devices have similar roles (master and slave) both with respect to adaptation in the second MAPE loop and the functionality provided by mobile learning application (i.e., the managed system). All devices of an MVD (master and slaves) monitor the mobile learning application and execute adaptation actions on it, but only the master is responsible for analysis and planning adaptations.

If the master detects that the number of GPS services in the MVD is not sufficient for the current task, it looks for an additional service. If there is a free GPS service available, the device that provides that service is dynamically added to the MVD, if not, the master periodically re-checks.

The master role can be performed by any of the phones in an MVD, making the organization robust in case of a master failure. In this paper, we abstract from the mechanisms to elect a new master. We refer the interested reader to [15] for self-healing mechanisms to deal with failures of a master in a master-slave organization deployed in a distributed application.

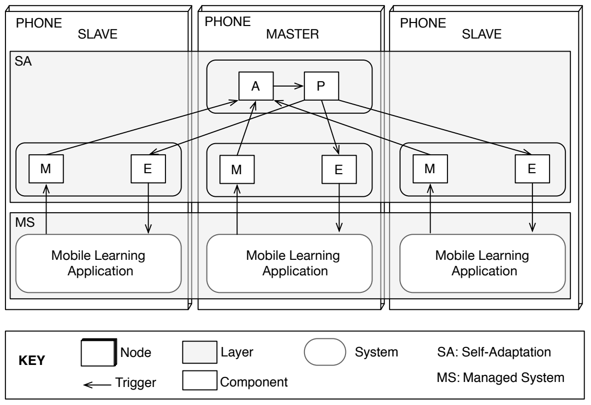

Fig. 4. Master/Slave MVD self-healing pattern (for 3 phones)

IV. Behavioral Design

The structural models of the self-adaptive layer described in the previous section show the primary building blocks of the MAPE loops and there interactions. These models are useful for explaining the adaptation mechanisms at a high-level of abstraction, and defining course-grained modules to implement the self-adaptive layer. However, to guarantee the robustness requirements, we need a rigorous specification of the self-adaptive *behaviors*, together with the properties that express the robustness requirements. This specification allows then to verify whether the self-adaptive behaviors comply to the properties. To that end, we formally specify the behavioral design of the self-adaptive layer.

In this research, we use Uppaal [16], a model checking tool that supports modeling of behaviors (also called processes) using timed automata and verification of the robustness properties expressed in timed computation tree logic (TCTL). Timed automata and TCTL provide an accessible formalism. Concretely, a timed automaton is a finite-state machine extended with clock variables, which are used to synchronize behaviors. The automata represent states in which a behavior can be found and define actions to be performed on the transition between states. Behaviors can communicate through channels by signal passing, where the sender process *x!* synchronizes with the receiver process *x?*. The automata can be complemented with expressions specified in a C-like language to define data structures (*struct* concept) and functions. Expressions in TCTL describe state and path formulae allowing the verification of properties of interest, such as *reachability* (a system should/can/cannot etc. reach a particular state or states), *liveness* (something eventually will hold), etc.

In the rest of this section, we describe the behaviors of the self-adaptive layer in three parts. We start by presenting the processes of the external world. Then, we present the behaviors of first MAPE loop (GPS Service Concern) and conclude with the behaviors of the second MAPE loop (MVD Concern). For the managed system, we only model the essential aspects that are required with respect to self-adaptation.

A. External World Processes

The need for self-adaption is triggered by changes in the external world. To that end, it is necessary to formally specify an abstraction of the external world. In our case, the external world consists of three behaviors: the Activity Agent, the Context, and the GPS Module. Fig. 5 shows the behaviors in relation to the MAPE loop for GPS service self-adaptation (which we discuss below).

An **Activity Agent**, located at the activity server, is in charge of setting the requirements for the GPS accuracy to perform the tasks, and the number of mobile devices that are required per group. Fig. 6 shows the automaton of the Activity Agent[1]. A first step initializes the distributed application, defining an initial deployment of phones to MVDs. Next, the Activity Agent is in charge to control the activity flow. On a periodic basis[2] (*Time_Activity*), the activity agent sends new tasks in the activity with new requirements (*SubmitTask* state), until the tasks in the activity (*TotalLoops*) are completed (*Final* state). Task requirements define the desired minimal accuracy necessary for the GPS modules and the number of GPS modules in each MVD (represented by *MASactivity.min_accuracy* and *MASactivity.number_GPS*) (see Fig. 5).

The environment influences the GPS module quality, potentially bringing a GPS service to an undesired state. The

[1]Transitions between states fire based on conditions and/or received signals (we place these above transition arrows) and can perform actions or send signals to other processes (we place these below transition arrows).

[2]The model abstracts the Activity Agent behavior by sending new requirements on a period basis. In practice, there is an activity flow between server and device agents based on the assignment and completion of tasks.

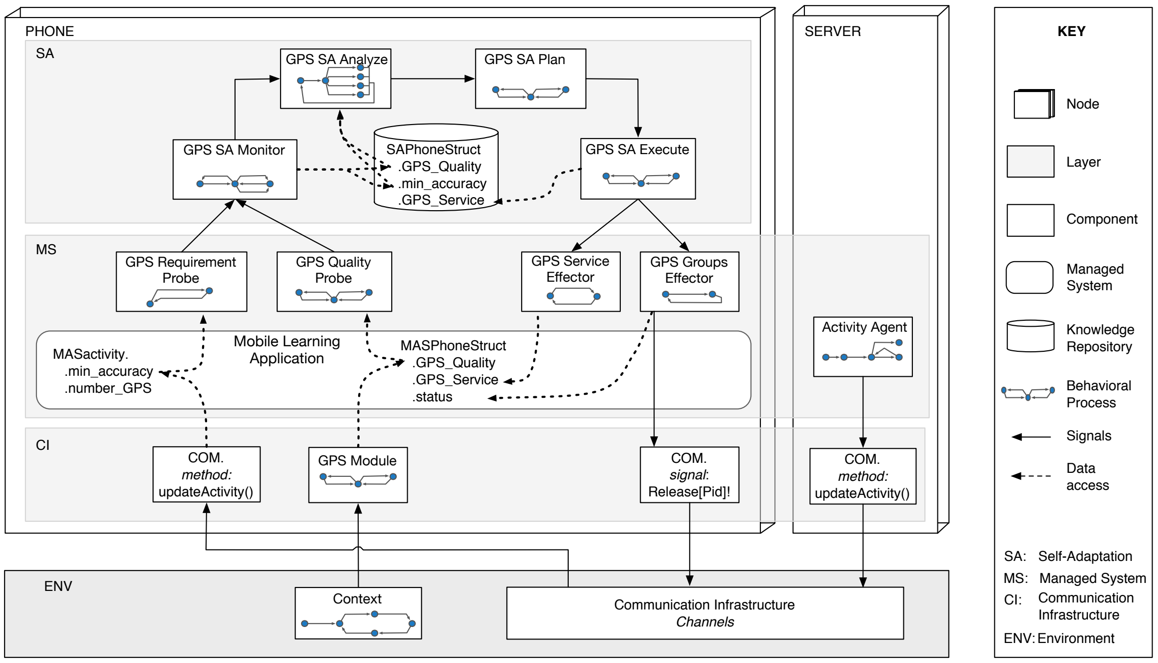

Fig. 5. Process mapping for GPS service self-adaptation

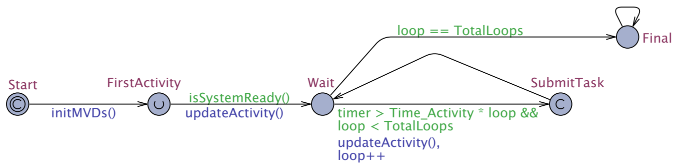

Fig. 6. Activity Agent

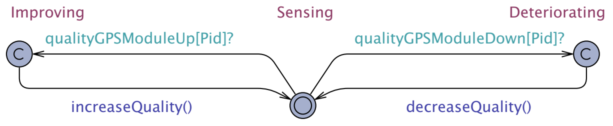

Fig. 8. GPS Module

environment is abstractly modeled by the **Context** automaton (Fig. 7), which describes states of the environment with respect to GPS interference: *Clean* or *Noisy*, and transitions at defined time instances (provided by the *getNextNoise* and *getNextRecover* functions). *Pid* refers to the phone ID. One context automaton is instantiated for each mobile device, allowing us to model the influence of the environment on each GPS module.

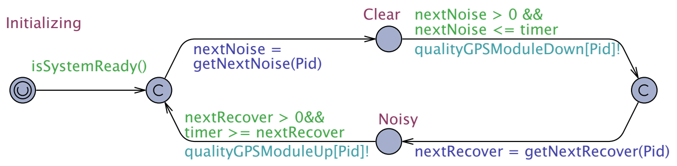

Fig. 7. Context

The **GPS Module** behavior, which is part of the Communication Infrastructure layer, is modeled by the automaton shown in Fig. 8. This behavior gets quality signals from the context (via *qualityGPSModuleUp* and *qualityGPSModuleDown*) that are used to update the representation in the system, represented by *MASPhoneStruct.GPS_Quality* (see Fig. 5).

As mentioned above, we abstracted the managed system to its essentials required to deal with self-adaptation. Concretely, the managed system is represented using structures in Uppaal. Snippet 1 illustrates how different elements of the mobile learning application are represented: the current information w.r.t. the GPS quality (*GPS_Quality*) and the service (*GPS_Service*) state, the participation in organizations (*status*), and temporary information used to determine changes on the GPS quality (*change_NotTreated, prev_quality*).

Snippet 1. *MASPhoneStruct*

```
struct{
  int GPS_Quality;  // Undesired or OK
  int GPS_Service;  // Deactivated or Active
  int status; // inMVD or Free
  bool change_NotTreated;  // internal (MAPE loop sync)
  int prev_quality; // internal (GPS quality)
}
```

Once we have a formal model of the external world and an abstraction of the managed system, we can model the self-adaptation processes.

B. GPS Service Self-Adaptation Processes

The GPS service self-adaptation processes model the first MAPE loop that deals with activating and deactivating GPS services based on the quality of the GPS signals. Fig. 5

shows the mapping of the behaviors of the MAPE loop to the components of the MAPE loop, shown in Fig. 3. There are two variables that can affect the suitability of a GPS module: the current task requirements and the GPS quality. Therefore, we model two probe processes that gather system information. Fig. 9 shows the automaton that represents the behavior of the **GPS Quality Probe**. The automaton contains a state in which the GPS quality is being sensed (*Probing*), and two additional states where the quality is *Increasing* and *Decreasing*[3]. In case the GPS quality is modified, the GPS Service Monitor is notified by sending a signal (*SAqualityGPSIncreased* and *SAqualityGPSDecreased*). Similarly, the **GPS Requirement Probe** captures changes in the activity requirements (*MAS-activity.min_accuracy*) to notify the monitor component (automaton not shown).

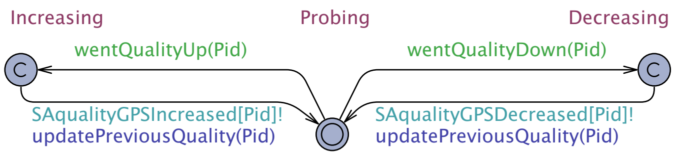
Increasing
Probing
Decreasing
wentQualityUp(Pid)
wentQualityDown(Pid)
SAqualityGPSIncreased[Pid]!
updatePreviousQuality(Pid)
SAqualityGPSDecreased[Pid]!
updatePreviousQuality(Pid)

Fig. 9. GPS Quality Probe

The **GPS SA Monitor** process (Fig. 10) is in charge of monitoring the underlying managed system and updating the knowledge repository (*SAPhoneStruct* in Fig. 5), supporting analysis and planning of self-adaptive actions. The automaton monitors two separated variables. On the left hand side, changes on the GPS requirements (initiated by the Activity Agent) are processed (*UpdateGPSReq*). On the right hand side, changes on the GPS quality are processed (*increase/decreaseQuality*). The automaton notifies the Analyze process when changes in the knowledge are detected (through the *SAGPSParametersChanged[Pid]* channel).

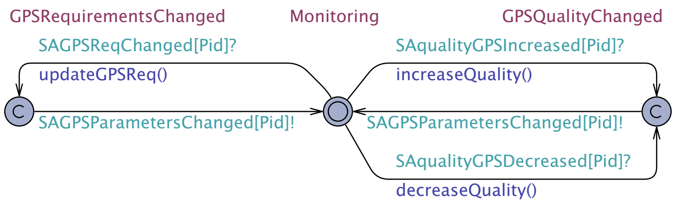

Fig. 10. GPS SA Monitor

The **GPS SA Analyze** process (Fig. 11) waits in the *Waiting* state for a trigger from the Monitor process to make a transition to the *Analyzing* state. One of four possible states can be reached (*KeepGood, KeepBad, ChangedGood, ChangedBad*), depending on the current GPS quality and the requirements to accomplish the current task. In case changes are identified (*ChangedGood, ChangedBad*), the Plan process is notified via a *SA_GPS_degraded/recovered* signal. Snippet 2 illustrates how the analyze functions to determine transitions to potential undesired states are specified in Uppaal.

[3]The C represent committed states where a behavior cannot delay.

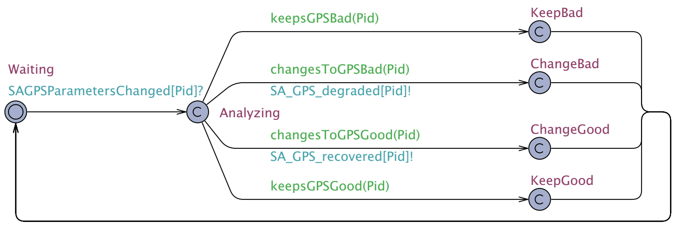

Fig. 11. GPS SA Analyze

The **GPS SA Plan** process (Fig. 12) is responsible for planning adaptation actions with respect to the GPS service. That is, deciding whether to turn on/off the GPS service provided by the phone (*ChangeToGood, ChangeToBad*).

Snippet 2. *changesToGPSBad()* function

```
bool changesToGPSBad(phone_id Pid){
    if( SAphoneStruct[Pid].GPS_Quality <
        SAphoneStruct[Pid].activity1.min_accuracy &&
        SAphoneStruct[Pid].GPS_Service == 1){
        return true;
    }else{
        return false;
    }
}
```

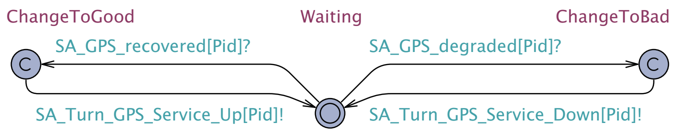

Fig. 12. GPS SA Plan

The **GPS SA Execute** process (Fig. 13) is in charge to apply the planned actions to the managed system. From a waiting state, it is triggered by the GPS SA Plan to make a transition and modify the GPS service via one of the states *SetGPSServiceUp* or *SetGPSServiceDown*.

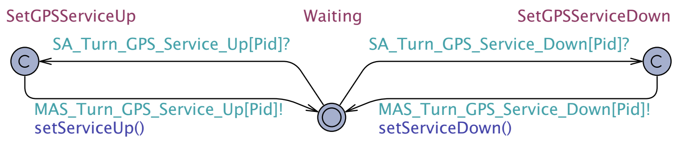
SetGPSServiceUp
Waiting
SetGPSServiceDown
SA_Turn_GPS_Service_Up[Pid]?
SA_Turn_GPS_Service_Down[Pid]?
MAS_Turn_GPS_Service_Up[Pid]!
setServiceUp()
MAS_Turn_GPS_Service_Down[Pid]!
setServiceDown()

Fig. 13. GPS SA Execute

To support the Execute process, two effectors are provided that perform the actual adaptations to the managed system. The **GPS Service Effector** process (Fig. 14) is in charge of activating/deactivating the GPS service on the managed system. This is represented by the MASPhoneStruct.GPS_Service variable (see Fig. 5). Additionally, a GPS Group Effector process is designed to remove a phone from an MVD in case the GPS Service is deactivated (automaton not shown).

C. MVD Self-Healing Processes

The MVD self-healing processes model the second MAPE loop that deals with recovery of undesired MVD states. Fig. 15

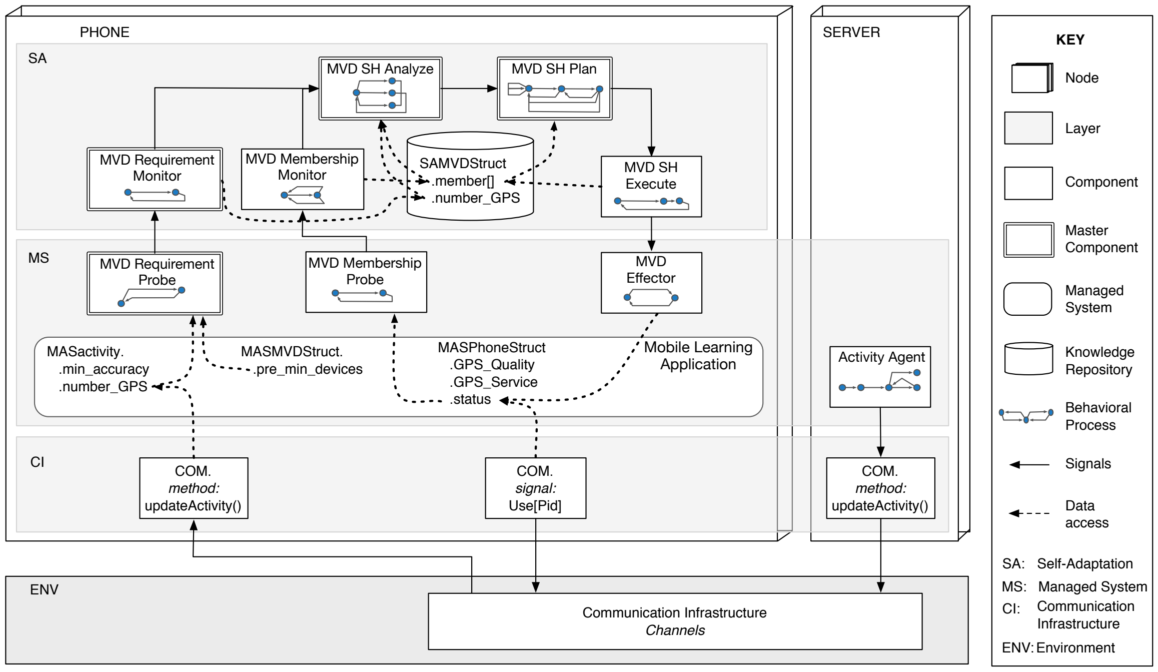

Fig. 15. Process mapping for MVD self-healing

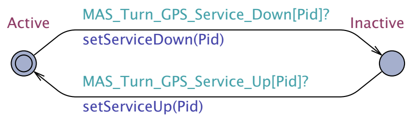
Active
MAS_Turn_GPS_Service_Down[Pid]?
setServiceDown(Pid)
Inactive
MAS_Turn_GPS_Service_Up[Pid]?
setServiceUp(Pid)

Fig. 14. GPS Service Effector

shows the mapping of the behaviors of the MAPE loop to the components of the MAPE loop, shown in Fig. 3

The **MVD Requirement Probe** process (Fig. 16) shows a behavior that gathers information about the group requirements for the current task (*MASactivity.number_GPS*, see Fig. 15). Changes that are detected are communicated to the corresponding Monitor process. *Mid* refers to the MVD ID.

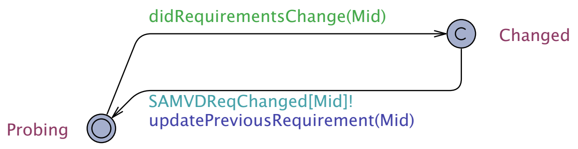

Fig. 16. MVD Requirement Probe

The **MVD Requirement Monitor** process (Fig. 17) is in charge of monitoring changes in the task requirements and, when signaled by the Requirement Probe, it updates the knowledge repository for the self-healing process (modeled as the *SAMVDStruct*, see Fig. 15). As shown in the Fig. 15, the probe and monitor processes that deal with the MVD requirements are only instantiated at the master device.

If a mobile device deactivates the GPS service resulting from the GPS service self-adaptation, a **MVD Membership**

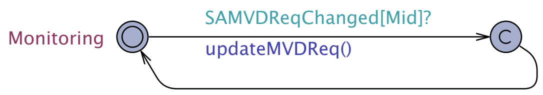

Fig. 17. MVD Requirement Monitor

Probe communicates the change to the **MVD Membership Monitor**, represented by the process in the Fig. 18. Unlike the MVD Requirement Probe and MVD Requirement Monitor processes, the MVD Membership Probe and the MVD Membership Monitor processes are instantiated at all mobile devices to gather the distributed information w.r.t. the MVD composition. The MVD Membership Monitor identifies whether an MVD is affected or not by a GPS service that is turned off (*myMVD != NOGROUP*).

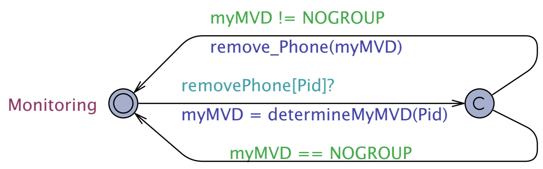

Fig. 18. MVD Membership Monitor

The **MVD SH Analyze** process (Fig. 19) is triggered by the **MVD SH Monitor**[4]. The Analysis process identifies three possible scenarios, that is, the MVD is *Complete* (the number of GPS services covers the requirements), *Incomplete* (GPS services are missing) or *Redundant* (there is redundancy of

[4]The Analyze process is logically triggered by Monitor processes; actually the processes interact indirectly via the knowledge repository of the SA layer.

GPS devices). The Analyze process communicates with the Plan process to start recovering the MVD that is missing GPS services (via *SH_MVD_Incomplete*) or to stop a possible search otherwise (via *SH_MVD_Redundant/Complete*).

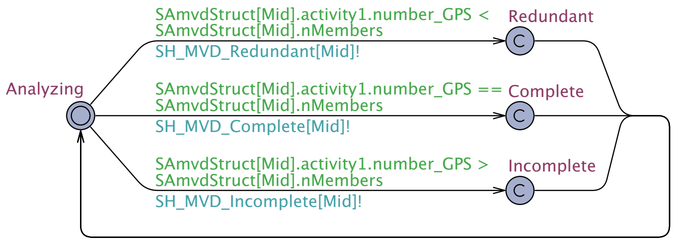
Analyzing
Redundant
Complete
Incomplete
SAmvdStruct[Mid].activity1.number_GPS < SAmvdStruct[Mid].nMembers
SH_MVD_Redundant[Mid]!
SAmvdStruct[Mid].activity1.number_GPS == SAmvdStruct[Mid].nMembers
SH_MVD_Complete[Mid]!
SAmvdStruct[Mid].activity1.number_GPS > SAmvdStruct[Mid].nMembers
SH_MVD_Incomplete[Mid]!

Fig. 19. MVD SH Analyze

When triggered by the Analyze process, the **MVD SH Plan** process (Fig. 20) initiates a search to locate a free phone (*found_Phone*) that offers a GPS service (*LookForFreeGPS*). In case the resource cannot be found, the process stays in the *NoFreeGPS* state and repeats the search until it finds a service and its goal is achieved (*AllFine*). The phone that provides the service (*found_Phone*) is notified to get integrated in the MVD. Only the master phone executes the Plan process.

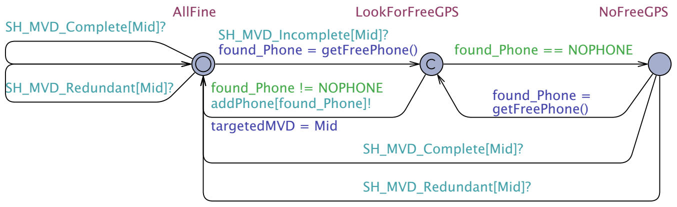

Fig. 20. MVD SH Plan

The **MVD SH Execute** process (Fig. 21) is in charge of applying the planned decisions for MVD self-healing. One Execute process is instantiated in each phone, in order to allow changing the phone state and integrating it into the correspondent MVD. The integration is performed through a **MVD Effector**, and results in changing the *MASPhoneStruct.status* (see Fig. 15).

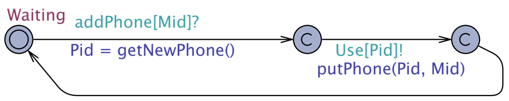

Fig. 21. MVD SH Execute

V. Verification of Self-Adaptation

Once we have modeled processes, we can formulate the self-adaptation requirements as logical expressions over the models. Uppaal uses a subset of TCTL (timed computation tree logic) to specify state and path formulae that can be verified. We discuss four groups of properties: functional correctness, GPS service adaptation, MVD self-healing, and MAPE loop interference.

A. Functional Correctness

To verify functional correctness, we check the absence of deadlock in the system (F1), and we check that for all tasks the required number of GPS services are available in each group. Deadlock is directly supported in Uppaal. F2 presents a concrete scenario that checks the required GPS services for group 1.

```
F1: A[] not deadlock
F2: A[] ServerAgent1.SubmitTask imply
     MASmvdStruct[1].nMembers >=
     MASactivity1.number_GPS
```

B. GPS Service Self-Adaptation

We define three robustness properties that allow verification of self-adaptation of the GPS service. R1 specifies that a GPS service, which provides an insufficient quality, will eventually be recognized by the Analyze process. R2 specifies that the GPS service of a GPS module with insufficient quality will actually be deactivated. Finally, R3 specifies that such GPS service will eventually be removed from any MVD. R1 and R2 are exemplified by defining an instance that analyzes cases on the phone number 1. R3 studies the scenario in which 3 groups (1, 2 and 3) do not contain the deteriorated phone 1.

```
R1: GPSModule(1).Deteriorating -->
     GPSSAAnalyze(1).ChangeBad
R2: GPSModule(1).Deteriorating -->
     MASphoneStruct[1].GPS_Service == DEACTIVATED
R3: GPSModule(1).Deteriorating -->
     MASmvdStruct[1].member[1] == NOT_USED &&
     MASmvdStruct[2].member[1] == NOT_USED &&
     MASmvdStruct[3].member[1] == NOT_USED
```

C. MVD Self-Healing

We define three properties that allow verification of self-healing of MVDs. R4 specifies that when an MVD is incomplete, eventually a search of a replacing GPS service will be initiated. R5 verifies that a search will eventually be completed successfully, and R6 that this will lead to the MVD being in a Complete (or Redundant) state.[5] R4 to R6 illustrate the rules applied to group 1.

```
R4: MVDSHAnalyze(1).Incomplete -->
     MVDSHPlan(1).LookForFreeGPS
R5: MVDSHPlan(1).LookForFreeGPS -->
     MVDSHPlan(1).AllFine
R6: MVDSHAnalyze(1).Incomplete -->
     MVDSHAnalyze(1).Complete ||
     MVDSHAnalyze(1).Redundant
```

D. MAPE Loop Interference

Finally, we verify that there is no interference between the MAPE loops. R7 specifies that the deactivation of a GPS service (as a result from adaptations in the first MAPE loop), is correctly handled by not including the undesired GPS service in any MVD (by the MVD self-healing process in the second MAPE loop). R8 specifies another required interference property that concerns the integration of a phone in a group

[5]The correctness of R5 and R6 relies on the assumption that only a fraction of the available GPS services can go down at the same time, and redundant services are available, as described in Section III.

(second MAPE loop) while the GPS service of this phone is deactivated (first MAPE loop) because it can no longer provide the required quality. In this case, the service of the failing phone should be replaced by another available service. R8 shows a concrete scenario where group 1 (*MVDSHPlan(1)* and *MASmvdStruct[1]*) has initially selected phone 2. In case the GPS service of phone 2 becomes *DEACTIVATED* during the integration process, phone 2 will be *NOT_USED* and a replacing service (*MVDSHPlan(1).found_Phone != 2*) will be selected. Finally, R9 specifies that, as a result of self-healing processes, a GPS service will not belong to two different MVDs at a time.

```
R7: GPSInternalEffector(1).Inactive -->
     MASmvdStruct[1].member[1] == NOT_USED &&
     MASmvdStruct[2].member[1] == NOT_USED &&
     MASmvdStruct[3].member[1] == NOT_USED
R8: MASphoneStruct[2].GPS_Service == DEACTIVATED &&
    MVDSHPlan(1).found_Phone == 2 -->
     MASmvdStruct[1].member[2] == NOT_USED &&
     MVDSHPlan(1).found_Phone != 2
R9:A[] forall(Pid:phone_id) forall(Mid1:MVD_id)
       forall(Mid2:MVD_id)
    MASmvdStruct[Mid1].member[Pid] == USED &&
    Mid1 != Mid2
     imply MASmvdStruct[Mid2].member[Pid]==NOT_USED
```

E. Verification and Results

We instantiated different scenarios with increasing number of phones and MVDs. Learning activities consisted of 6 tasks. Measurements confirm that the processing cost for verification grow exponentially with the complexity of the scenarios (number of phones and number or MVDs). In this particular domain, a scenario with 3 phones and 1 MVD required 393 ms to verify the deadlock property (F1), while 6 phones and 2 MVDs required around 21 minutes, and 12 phones and 3 MVDs required several hours to analyze. The analysis results show only small differences in terms of cost for verifying the other properties for the same scenario. For example, for a scenario with 1 MVD and 3 phones, the verification cost varied between 167 ms to 178 ms for properties F2 and the robustness properties R1 to R9.

VI. From Design to Implementation

The original mobile learning application was implemented using JADE [17]. JADE is a platform that provides facilities to develop distributed multi-agent systems, including services for message communication, service registration and discovery, etc. We briefly explain how we implemented the self-adaptive layer on top of the legacy system.

We mapped one-to-one the behavioral processes presented in Section IV to Java classes. Fig. 22 illustrates the classes that implement the MAPE loop of the GPS service concern. The probe processes are implemented as Jade behaviors, which are executed by the agent on each mobile device. Snippet 3 shows the implementation of the GPS Quality Probe. The code that is executed on a periodic basis (*TickerBehaviour*) has been granted access to the current GPS accuracy data via the *getAccuracy* interface offered by the *LocationManager*. Effector classes implement effectors that have access to the underlying system to adapt it when needed. For example, the *GPSServiceEffector* class implements an effector that consumes the *setGPSService()* method of the *PhoneManager* to activate (or deactivate) the GPS service when demanded.

Snippet 3. *GPSQualityProbleBehaviour* class

```
public class GPSQualityProbeBehaviour extends
     TickerBehaviour{
[...]
  public GPSQualityProbeBehaviour(Agent agent, long period,
        float threshold) {
    super(agent, period);
  }

  @Override
  protected void onTick() {
    float accuracy = LocationManager.getInstance().
         getMyLocation().getAccuracy();
    myLogger.log(Logger.INFO, "Tick! Check GPS accuracy");
    if(accuracy - prev_accuracy > Threshold){
      myLogger.log(Logger.WARNING, "GPS accuracy changed");
      GPSMonitor.getInstance().update("accuracy",accuracy);
    }
    pre_accuracy = accuracy;
  }
}
```

Analogously, we mapped the processes of the MVD self-healing loop to Java classes. For the communication between the masters and the activity agent on the one hand, and between the agents of MVDs on the other hand, we used *ACLMessages* (Agent Communication Language Messages) provided by JADE. These messages offer high-level communication primitives and supporting protocols, such as request-confirm, inform, etc.

VII. Conclusions and Future Work

In this paper, we explained how we have extended a legacy mobile learning application with a self-adaptation layer, making the system robust to degrading GPS accuracy. We designed the self- adaptive layer as a set of interacting MAPE loops distributed over the mobile devices. To guarantee the required robustness requirements, we used timed automata to specify the behaviors of the MAPE loops, and expressed the robustness requirements as formal properties in TCTL. The Uppaal tool allowed us to verify the properties. The behavioral design was then mapped to Java implementation.

While MAPE is widely recognized in the community, it is often only used as a conceptual guidance for the design of self-adaptive systems. In this paper, we used explicit modules for each of the adaptation functions of MAPE loops in the design of a self-adaptive application and mapped these modules one-to-one to an implementation. We report a number of lessons learned from this effort. Using explicit modules for each of the adaptation functions of MAPE loops makes it easier to:

- model the self-adaptive behavior, as the designer can focus on one activity at a time;
- model the interaction between the managed and managing system, as the interaction points are well-defined;
- reason about the behavior within and between MAPE loops, as a result of the clear separation of concerns;
- reason about the interactions between the managed and managing system;

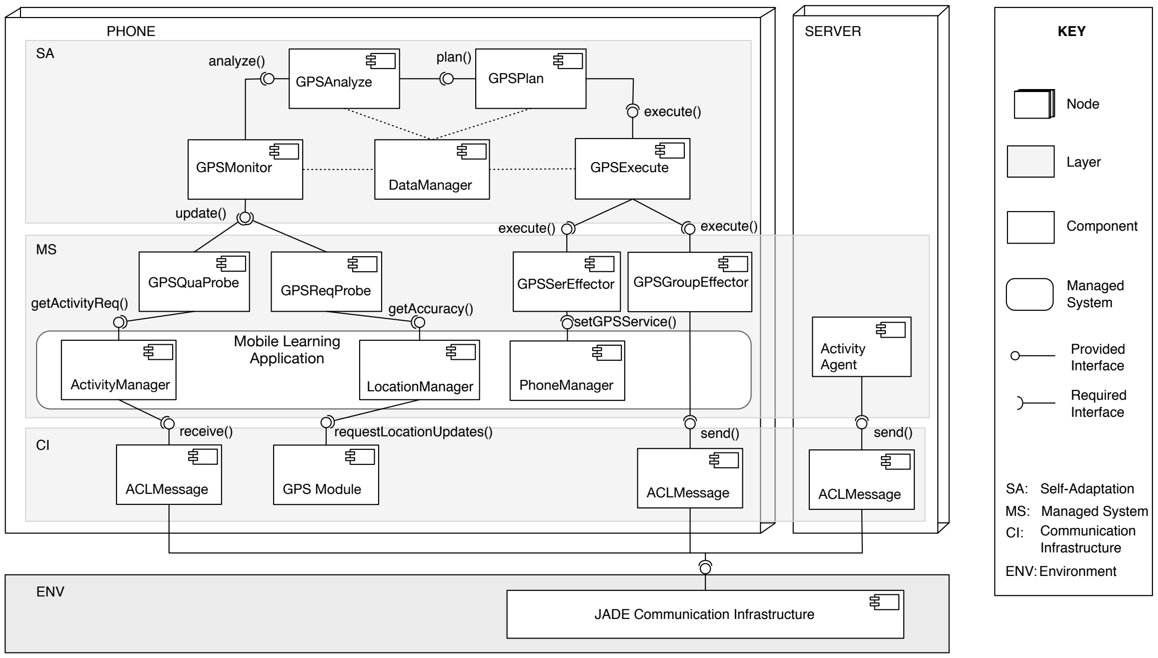

Fig. 22. UML component diagram of the GPS Service self-adaptation

- specify required properties, and these properties can be specified at a more fine-grained level;
- identify problems in the design;
- map design to implementation.

However, there are also some tradeoffs:

- some MAPE activities have a straightforward behavior, which may raise questions about the usefulness of a separate specification;
- the size of the design increases;
- the cost for verification grows.

Regarding the generalization of the approach, we remark that in the presented application, we could easily separate the adaptation behavior from the business logic. From other work in our team [15], it is clear that this separation is not always so easy to realize. This calls for more attention to the separation of managed and managing system and the study of their interactions, including formal verification.

As the next step, we first aim to further verify that the required properties hold in the implemented mobile learning application. To that end, we plan to check the compliance of traces derived from the verification of the design with traces obtained from the running implementation. Second, we plan to study the overhead implied by adding MAPE loops to the system at runtime, including cost in terms of resources, and communication due to interactions among MAPE loops.

References

[1] D. Gil de la Iglesia, *Uncertainties in Mobile Learning applications: Software Architecture Challenges*. Linnaeus University, 2012.

[2] J. Kephart and D. Chess, "The vision of autonomic computing," *IEEE Computer Society*, vol. 36, no. 1, 2003.

[3] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," *FOSE '07 2007 Future of Software Engineering*, 2007.

[4] R. de Lemos *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," ser. Lecture Notes in Computer Science vol. 7475. Springer, 2013.

[5] D. Weyns, S. Malek, and J. Andersson, "FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems," *ACM Transactions on Autonomous and Adaptive Systems*, vol. V, 2011.

[6] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, and K. G. H. Giese, *On Patterns for Decentralized Control in Self-Adaptive Systems*, ser. LNCS 7475, 2012.

[7] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *Computer*, vol. 37, no. 10, 2004.

[8] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic QoS Management and Optimization in Service-Based Systems," *Software Engineering, IEEE Trans.*, vol. 37, no. 3, 2011.

[9] I. Georgiadis, J. Magee, and J. Kramer, "Self-Organising Software Architectures for Distributed Systems," in *WOSS*, 2002.

[10] D. Sykes, J. Magee, and J. Kramer, "FlashMob: Distributed Adaptive Self-Assembly," in *Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011.

[11] R. Hebig, H. Giese, and B. Becker, "Making control loops explicit when architecting self-adaptive systems," in *International workshop on Self-organizing architectures*, ser. SOAR '10. ACM, 2010.

[12] P. Vromant, D. Weyns, S. Malek, and J. Andersson, "On Interacting Control Loops in Self-Adaptive Systems," in *Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '11. ACM, 2011.

[13] L. Baresi and L. Pasquale, "Live goals for adaptive service compositions," in *Software Eng. for Adaptive and Self-Managing Systems*, 2010.

[14] N. Villegas, G. Tamura, H. Müller, L. Duchien, and R. Casallas, "DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS. Springer, 2013.

[15] M. U. Iftikhar and D. Weyns, "A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System," in *Foundations of Coordination Languages and Self Adaptation*, 2012.

[16] G. Behrmann, A. David, P. Pettersson, W. Yi, and M. Hendriks, "UPPAAL 4.0," *In Quantitative Evaluation of Systems*, 2006.

[17] F. Bellifemine, "JADE: What it is and what it is next." in *Models and Methods of Analysis for Agent Based Systems*. Springer, 2001.