Enhancing Software Qualities in Multi-Agent Systems using Self-Adaptation

Didac Gil de la Iglesia, and Danny Weyns

DFM, Linnaeus University, Växjö, Sweden
{didac.gil-de-la-iglesia, danny.weyns}@lnu.se

Abstract. Engineering multi-agent systems (MAS) is known to be a complex task. One of the reasons lays in the complexity to combine multiple concerns that a MAS is expected to address, such as system functionality, coordination, robustness, etc. A well-recognized approach to manage system complexity is the use of self-adaptive (SA) mechanisms. Self-adaptation allows to adjust the system behavior in order to achieve certain software qualities (optimization, fault-tolerance, etc.). The key idea behind self-adaptation is complexity management through *separation of concerns*. In this paper we introduce SA-MAS, an architectural approach that integrates the functionalities provided by a MAS with software qualities offered by a SA solution. The paper presents a reference model for SA-MAS and applies it to a Mobile learning case, in which we deal with robustness properties. In addition, we apply formal verification techniques as an approach to guarantee the requirements of the SA-MAS application.

1 Introduction and Motivation

Multi-agent systems (MAS) have been applied in a broad number of fields, such as e-commerce, traffic and transportation, robotics and education applications. Among the benefits of these systems are autonomy of interacting entities, flexibility of entities that can come and go at will, efficiency as a result of local decision making, etc. Despite the benefits of MAS, engineering such systems pose huge engineering challenges because of their distributed behavior [1], and the need to deal with multiple concerns.

One recognized approach to manage complexity in MAS is by means of employing domain-specific middleware, where (parts of) the coordination is separated from agents that realize the system functionalities [2]. Recognized approaches are e-institutions, organization middlewares, and stigmergic approaches. Self-adaptation (SA) is a well recognized approach for dealing with the system complexity by separating logic that deals with particular runtime qualities [3, 4]. SA is used for adding so called self-* properties (self-healing, self-protection, self-optimization) [3] to address changing operating conditions in the system or its environment. SA is based on the design principle of *separation of concerns*. SA is a promising approach to tackle the complexity of engineering MAS by separating the logic that deals with quality concerns of interest from the domain functionality provided by agents and coordination, handled by agents supported by appropriate middleware.

SA has not deeply been studied in combination with MAS architectures. One reason resides on the interference of the adaptation logic with the functional agent logic, which becomes a conceptual conflict of breaking the autonomy of the agents. Nevertheless, intertwining MAS and SA should be avoided when possible to manage the system complexity. The separation of SA and MAS enables the independent study of the distributed system functionalities from the behaviors of self-adaptation mechanisms for robustness, performance, availability, etc. In addition, the separation would facilitate adding SA mechanism on top of an already existing multi-agent legacy platform, which would improve reusability of solutions.

This paper studies the use of self-adaptive multi-agent systems (SA-MAS) to handle the complexity of engineering MAS with particular quality concerns. What differentiates SA-MAS from previous approaches that aimed at combining MAS with a self-adaptive approach is a disciplined separation of logic for functionality and coordination from logic to deal with quality concerns. Our approach advocates for the construction of SA-MAS in a layered design. In this design, the MAS becomes a *managed* entity from the self-adaptation perspective, and components in the SA layer are *managing* entities of the system. In addition, we apply formal methods to guarantee the quality properties of interest during the engineering of SA-MAS.

The rest of this paper is structured as follows. In Section 2 we introduce a reference model for SA-MAS, and describe its benefits and tradeoffs. Section 3 presents a case scenario to show the application of SA-MAS for robustness concerns. We explain the application of the reference model to define a high-level architecture for the case, presented in Section 4. Section 5 provides a formal behavioral description of the SA-MAS software elements, demonstrating how we separated system functionalities and coordination from self-healing behavior. In Section 6, we formally specify the properties of interest of different concerns and explain verification. In Section 7, we briefly explain the implementation of the case. Section 8 discusses related efforts and Section 9 concludes the paper, providing an outline for future work.

2 SA-MAS

The principled idea of SA-MAS is to add SA mechanisms to MAS to deal with particular quality concerns. In particular, SA adds a supervising level to MAS that can adjust the MAS to deal with non-functional properties such as protecting the MAS from external security threats, or mitigating errors due to failures in the the operating environment.

2.1 Reference Model

Fig. 1 presents a reference model for a SA-MAS, which is conceived as a layered architecture where concerns and responsibilities from the MAS and self-adaptation subsystems are clearly separated. The part of Fig. 1 on the left hand side shows the primary layers. The bottom layer of the SA-MAS architecture provides the *communication infrastructure*, which encapsulates the means for communication between agents of the distributed system. The middle layer provides the *multi-agent system*, which deals with requirements of the domain at hand. The top layer provides *self-adaptation*, which can modifying the MAS layer, conditioned by the environment and system state, in order to cover system quality concerns. Fig. 1 (left) illustrates a distributed system with multiple nodes that share the three layers of a SA-MAS. In principle, the interactions should be restricted between connected layers, and a layer should not depend on layers upwards. In practice, there might be deviations from this principle.

The part of Fig. 1 on the right hand side refines the MAS and SA layers. The MAS layer is composed by autonomous agents that provide the required domain functionalities for the system at hand, and the (optional) coordination middleware that deals with coordination concerns.

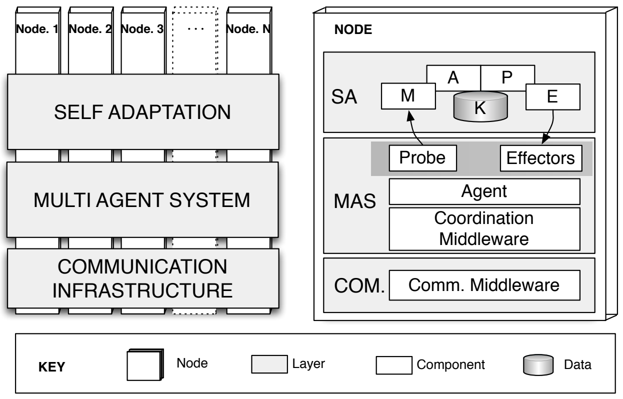


Fig. 1: Reference model for SA-MAS

The coordination middleware is optional as MAS may be designed with purely agents that communicate via exchanging messages. In this approach, agents are in charge of providing functionalities and handling coordination. An established approach of coordination middleware for MAS are electronic institutions, where agents interact through scenes in virtual markets [5]. Yet another approach is stigmergic coordination, where agents coordinate their behavior through the manipulation of virtual marks in the coordination infrastructure. Classic examples are digital pheromones [6] and gradient fields [7]. Recently, there has been an increasing interest in middleware support for organization management, some of this recent work on this is [8, 9].

Using a coordination middleware offers a separation of concerns, by separating domain functionalities (provided by agents) from coordination concerns (provided by the middleware). However, this separation does not consider quality properties as first class concerns. Support for example for fault tolerance is typically scattered over agents and coordination middleware, which results in complex designs that are difficult to understand and maintain.

To overcome this limitation, SA provides the flexibility to alter the behavior of certain modules in a (possibly legacy) system to realize quality properties of interest as first-class concerns. A common approach to realize SA is by a MAPE-K control loop [3, 10], as shown in the right part of Fig. 1. The knowledge component (K) maintains a runtime representation of the managed MAS, which typically consists of local knowledge models at the nodes of the system, and a representation of the adaptation goals. A monitor component (M) gathers information from the underlying MAS and possibly the system's environment in order to update the knowledge models, providing the subsequent computations of the control loop with the necessary data. An analyze component (A) examines the knowledge previously gathered by the monitor, and based on the adaptation goals draws conclusions on which further actions should be undertaken by the SA system. A plan component (P) puts together a series of adaptation actions to resolve the problem identified by the analyzer. This set of actions to the managed MAS is then carried out by an execution component (E).

A SA can support the behavior modification of a single agent, such as publishing new services in the MAS, and modifying the internal service implementation. A SA can also support the adaptation of the organization, by requesting modifications on the structure of agents in the organization (peer-to-peer, hierarchies, federations, etc. [11]) or modifying the list of members in the organization. In order to allow interaction between the MAS and the SA layer, probes and effectors should be included. Probes are components included in the MAS that allow a monitoring component to collect relevant information of the managed system. Similarly, effectors in the MAS are required to apply the effects of the execute component in the SA. Adding these two component in a MAS implies an intrusive task. This is a tradeoff that breaks the agent autonomy and requires additional components in the MAS, but has a noticeable impact reducing the system complexity to enhance system quality properties.

Sometimes, several quality concerns need to be considered when modeling a MAS. This may require that several self-adaptation components are placed in the SA layer, to address the multiple quality concerns. Examples could be system performance and robustness. Often, these quality concerns become cross-cutting, therefore it is important to keep the self-adaptation components separated when possible, to reduce their individual complexity and encourage their scalability and reusability. [12] presents inter-loop and intra-loop coordination in a SA-MAS application. An alternative approach to deal with multiple concerns is pre-emption to switch adaption concerns dynamically [13].

2.2 Formal Analysis

While SA mechanisms are placed to provide quality properties to the multi-agent system, it is critical to guarantee that SA behaviors perform the desired outcomes. This is, verify that the adaptation processes performed by a SA component achieve the desired quality concerns. Formal methods provide us the means to rigorously describe and verify the behavior of an entity. Examples such as deadlocks and livelocks [1] need to be analyzed, but also properties that are domain and concern dependent. In order to verify the SA-MAS designs, the behaviors of layers on the architecture need to be formally modeled, including probes and effectors that interact with agents and organization and self-adaptation processes. In addition, a formal representation of the external world is necessary. The environment in which the system is placed provokes (through disturbances, failures, etc.) changes in the system behavior, on which the SA should react. Other components of the system that may trigger self-adaptation should be modeled as well. There is a range number of variables in the external world that can be considered when modeling our system. However, we may create a model that presents an abstraction of the environment, which describes the behavior with the basic variables that can influence the SA-MAS.

3 M-Learning Scenario

To illustrate the design of a SA-MAS, in this section we present a case study on the field of m-Learning. In [14], a MAS solution based on mobile devices was developed. The solution covered the functional requirements for a set of pedagogical outdoor activities to train geometry concepts. Distance calculations were required to perform the activities and mobile devices with GPS were used as tools to support this requirement. Distances could be measured based on the locations of two or more mobile devices. For collaboration purposes, the students were arranged in groups of 3, which means that each group could acquire three locations at a time and the distances between them. In this study, devices grouped in an organization are named Mobile Virtual Device (MVD). The activities took place during the winter season, wherefore the weather conditions could affect the GPS accuracy and reliability measurements. From our experience, no more than two over six devices would permanently degradate the GPS signal to the extent that it becomes undesirable for the distance measurements. The Fig. 2 illustrates an example of the accuracy error of a GPS module over a period of time. The two dotted lines indicate the max. accuracy error allowed for two different tasks.

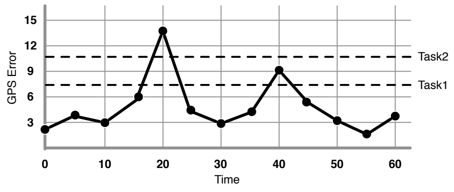
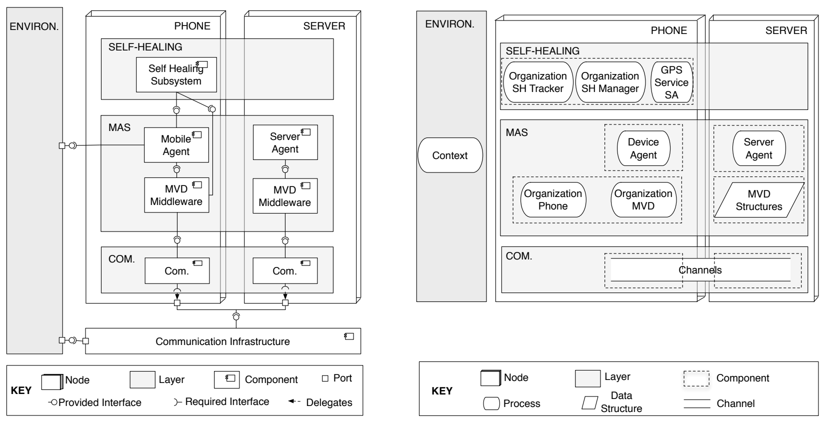


Fig. 2: Example of GPS quality

Quality requirements: The characteristics provided by this legacy system could not guarantee the robustness of the system under uncertain environmental conditions, which could provoke misleading measurements (for example in t=20). Previous studies have analyzed organization self-healing mechanisms for individual device failures [15]. In this scenario we focus on the quality of the GPS modules. Previous studies [16] show that the accuracy of a GPS can vary depending on the environment conditions.

4 Overview of the architectural design

To address a GPS service failure, a SA layer complements the previously implemented MAS architecture. Fig. 3a illustrates the architectural composition of the SA-MAS after including an additional SA layer to the legacy system. Fig. 3b presents the behavioral composition of the SA-MAS. Separation of concerns are also present in the behavioral structure of the SA-MAS, splitting self-adaptation processes from functional processes provided in the MAS. The reader should notice that, in order to provide guarantees on the SA behaviors, an abstraction of the environment needs to be described by an additional process.



(a) Component diagram deployed in server and phones

(b) Design model of server, phones and environment (component mapping)

Fig. 3: Collaborative m-Learning system

5 Behavioral Models

We provide a formal verification of the system behavior to guarantee its correctness w.r.t. main properties. Uppaal allows to describe an abstraction of a system's behavior to rigorously design and verify desired system properties. Processes are defined as timed automata, which allow use of guards for the edges, execute basic program functions and communicate with other processes through signal passing. The SA-MAS described in section 3 is divided in three behavioral areas. Probes and effectors processes describe the additional components required in the MAS. These connect with the Self-Adaptation processes and the external world. Notice that probes and effector models describe behaviors to acquire information from the MAS and effect agent's and organization's behaviors as represented in Fig. 1, but do not describe internal functionalities in the MAS.

5.1 Probes and Effector processes

To be able to gather information w.r.t. the MAS and modify its behavior, probes and effectors components are created. In this subsection we describe the processes to interact with *Agents* and the *MVD Middleware*.

Device Agent. The mobile agents have the autonomy to offer locations services to other nodes in the system. However, one influencing variable on the service quality is the GPS quality. A probe can allow gathering the service quality to prepare the required SA mechanisms. The process that describes the service quality state is represented by a timed automaton, containing a *Desired* and an *Undesired* state. For each device agent, there is an instance of the process running. Signals, providing from the environment, referring to the GPS state are used to decide transitions between the two basic states.

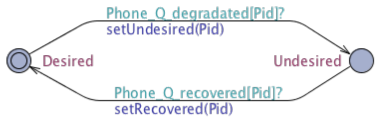
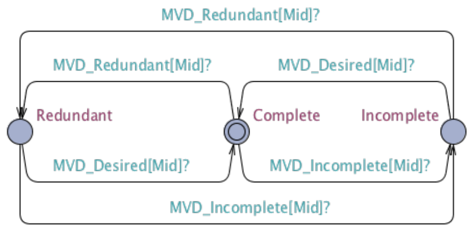


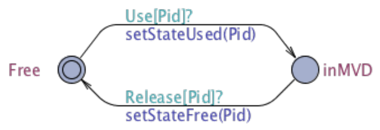
Fig. 4: Device agent process

Organization MVD. Parallelly, probes and effectors are necessary to allow SA of the organizations. The first organization process (Fig. 5a) is described by an automaton that contains three basic states of a MVD. Based on the number of GPS resources in the MVD and the requirements of the task in process, the MVD can be in a *Complete* (there is an equality between the two variables), *Incomplete* (there is a lack of GPS resources) or *Redundant* state (there is an excess of GPS resources). There exist a unique instance of the MVD process for each MVD in the activity.

Organization Phone. There are two basic states in which a phone can be found (Fig. 5b), depending on if it is a member of a MVD (*inMVD*) or not (*Free*). The state of a phone device offers basic information to determine potential resources to be used in the self-healing processes. Therefore, transitions between the phone states are subjected to *Release* and *Use* signals originated during the self-healing process and described in the following subsection.



(a) Organization MVD



(b) Organization Phone

Fig. 5: Organization processes

5.2 Self-Adaptation processes.

The SA processes describe the four basic roles of the managing system: Monitor, Analyze, Plan and Execute. Two different self-adaptation ambients are identified. The first, *GPS Service Self-Adaptation*, deals with the quality of the GPS service and determines when a service can be offered (or not) in the MAS. The second, *Organization Self-Healing*, deals with the health of existing MVD w.r.t. the activity requirements.

GPS Service Self-Adaptation. This process is responsible for monitoring the accuracy of the GPS module in the mobile device and to request the device agent to enable or disable the GPS location service. It has two basic states (*Normal* and *Degradated*). A GPS Service Self-adaptation process (Fig. 6) runs on each mobile device. This process *monitors* the GPS quality by receiving increments or decrements of its quality (*qualityUp* and *qualityDown* signals) from the environment. An *analysis* determines the reliability of the GPS service based on system *knowledge*: the current service accuracy and the activity requirements. The *plan* and *execution* stages are performed communicating with the device agent, by sending *Phone_Q_** signals. In case a used GPS becomes degradated, a signal *removePhone* is fired. This allows notifying the organization self-healing subsystem w.r.t. the unsuitability of maintaining the phone inside the organization.

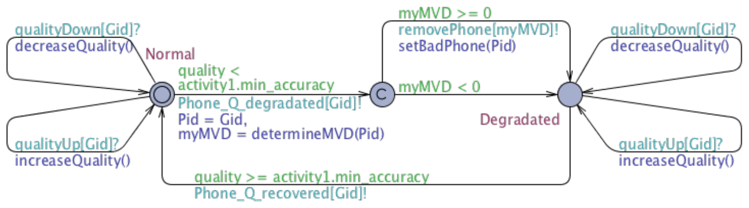


Fig. 6: GPS Service Self-adaptation process

Organization Self-Healing. The organization self-healing process is represented by two interconnected automata. These processes are in charge of providing robustness to the system, assuring that the related MVD is directed to the Complete state in case of failures. Two automata are instantiated for each MVD. The four MAPE roles are distributed among the two automata, being the *monitor*, *analysis* and *execute* in the tracker process (Fig. 7) and the *plan* role in the manager process (Fig. 8).

The tracker process contains three main states, *Complete*, *Incomplete* and *Redundant*, and is in charge of maintaining the internal state of the MVD. The tracker monitors arriving and leaving phones into the MVD. Phones can leave an organization due to the adaptation outcomes presented in the GPS service self-adaptation process. The tracker analyzes the completeness of the group w.r.t. the current activity (*knowledge*). If the automaton is found in Incomplete state, a signal is submitted to the Self-Healing manager to determine the adaptation

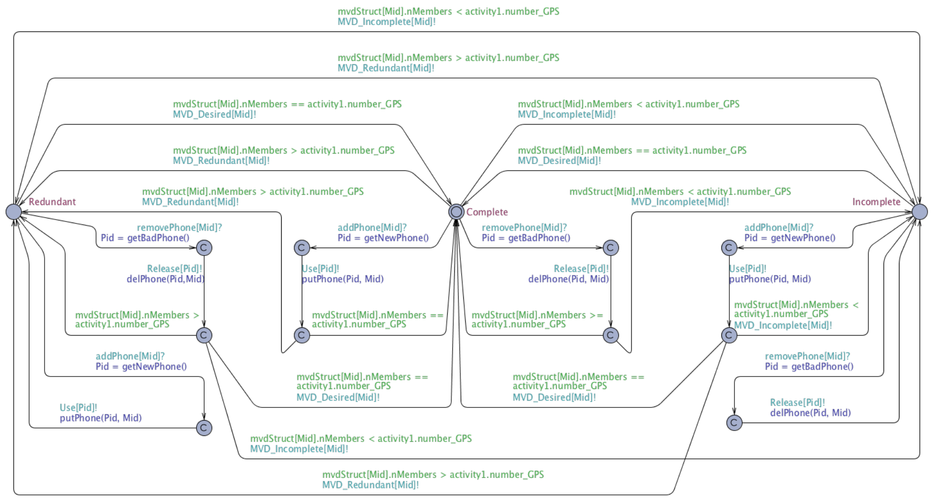

mvdStruct[Mid].nMembers < activity1.number_GPS
MVD_Incomplete[Mid]!
mvdStruct[Mid].nMembers > activity1.number_GPS
MVD_Redundant[Mid]!
mvdStruct[Mid].nMembers == activity1.number_GPS
MVD_Desired[Mid]!
mvdStruct[Mid].nMembers < activity1.number_GPS
MVD_Incomplete[Mid]!
mvdStruct[Mid].nMembers > activity1.number_GPS
MVD_Redundant[Mid]!
mvdStruct[Mid].nMembers == activity1.number_GPS
MVD_Desired[Mid]!
mvdStruct[Mid].nMembers > activity1.number_GPS
MVD_Redundant[Mid]!
mvdStruct[Mid].nMembers < activity1.number_GPS
MVD_Incomplete[Mid]!
Redundant
Complete
Incomplete
removePhone[Mid]?
Pid = getBadPhone()
addPhone[Mid]?
Pid = getNewPhone()
removePhone[Mid]?
Pid = getBadPhone()
addPhone[Mid]?
Pid = getNewPhone()
Release[Pid]!
delPhone(Pid,Mid)
Use[Pid]!
putPhone(Pid, Mid)
Release[Pid]!
delPhone(Pid, Mid)
Use[Pid]!
putPhone(Pid, Mid)
mvdStruct[Mid].nMembers > activity1.number_GPS
mvdStruct[Mid].nMembers == activity1.number_GPS
mvdStruct[Mid].nMembers >= activity1.number_GPS
mvdStruct[Mid].nMembers < activity1.number_GPS
MVD_Incomplete[Mid]!
addPhone[Mid]?
Pid = getNewPhone()
removePhone[Mid]?
Pid = getBadPhone()
mvdStruct[Mid].nMembers == activity1.number_GPS
MVD_Desired[Mid]!
mvdStruct[Mid].nMembers == activity1.number_GPS
MVD_Desired[Mid]!
Use[Pid]!
putPhone(Pid, Mid)
Release[Pid]!
delPhone(Pid, Mid)
mvdStruct[Mid].nMembers < activity1.number_GPS
MVD_Incomplete[Mid]!
mvdStruct[Mid].nMembers > activity1.number_GPS
MVD_Redundant[Mid]!

Fig. 7: MVD tracker process

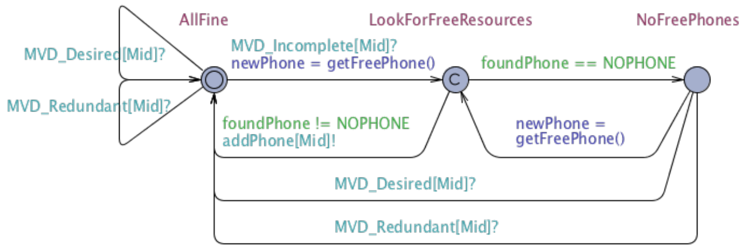


Fig. 8: Self-healing Manager process

actions required to recover a desired state on the managed MVD. The plan process is described by the second automaton (Fig. 8). When the MVD is found in the Incomplete state, the *MVD_Incomplete* signal is fired. Then, a search for a Free and Desired phone to be included in the MVD is started (*getFreePhone*). A search can become unsatisfactory (*NoFreePhones*) when no phones are available. In such scenario, the organization self-healing plan process repeats until the search becomes satisfactory. The plan resolution determines a *newPhone* to be included in the Incomplete MVD and fires the *addPhone* signal to communicate with the tracker process. The *execution* of the adaptation actions include the update of the phone (*Use*) and organization (*Desired* and *Redundant*) states.

5.3 External World.

The external world can affect the behavior of our system, and it is therefore necessary to provide a formal representation of it. The external world contains any component not included in our system that can affect a SA-MAS. In this subsection we model an abstraction of the external world that has an impact on our system robustness. The server agent behavior modifies the GPS accuracy requirements for the tasks. Context where the activity takes places can affect the GPS reception.

Server Agent. The SA process does not modify the server behavior. However, this component modifies the required GPS accuracy for the activity, which has an impact on a quality property. Therefore, it is necessary to model the server agent behavior as part of the external world. The behavior of the server agent has a first stage in which the platform is initialized for the activity, defining groups (MVDs) in the activity and an initial deployment of nodes in each group. In a second stage, the agent controls the activity flow updating the task requirements and GPS quality requirements.

Context. An abstraction of the environment with the concerned aspects are presented in the following automaton (Fig. 9). The context influences the GPS quality, which can force its degradation or recovery. This process simulates an environment that sets the GPS *qualityUp* or *qualityDown*, triggering modifications on the GPS service self-adaptation process, to self-adapt the state of the location service provided by the device agents. One instance of the context runs for each GPS device, allowing to trace independent GPS-based location service behaviors.

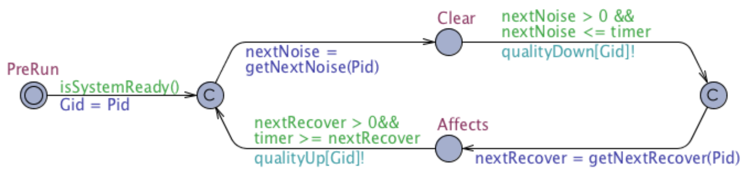


Fig. 9: Context (GPS signal)

5.4 System

One instance of the system, defining the processes that are involved, can be created to study the desired quality properties. The following presents a system with a Server updating the activity requirements every 100 time units, and instances of the MAS-related components, SA and External world models. For analysis purposes, the system is composed by 6 phones to be used in 2 MVDs. The activity is composed by two tasks (Task1 and Task2), requiring 1 and 2 phones per group respectively. The following are the system declarations defined in Uppaal.

```
ServerAgent1 = ServerAgent(100);
system DeviceAgent, PhoneState, MVD,       //MAS (Probes and Effectors)
       GPS_Agent, SHTracker, SHManager,    //SA
       Context, ServerAgent1,              //External World
       ActivityServerInit;
```

```
const int Phones=6;   //Number of phones
const int MVD=2;      //Number of groups
int Task1=1, Task2=2; //Phones required per task
```

We stimulate undesired GPS conditions and MVD states provoking errors on the GPS module. The following arrays define time instances from when the *GPS module* quality can be changed.

```
int GPSSetNoise[Phones][Nerrors] =    {{2,0,5, 70},{2,0,30, 120}, ...};
int GPSRemoveNoise[Phones][Nerrors] = {{2,0,25,100},{2,0,80,160}, ...};
```

6 Properties

After the processes have been modeled, Uppaal assists us to verify that concern properties are satisfied. Properties are verified by querying the system using TCTL (timed computation tree logic) statements. The main focus of this study is the robustness of the system, provided by the SA. The verification of the properties is evaluated based on the scenario presented in 5.4.

6.1 MAS (Functional correctness of Probes and Effectors)

Due to the dynamism of the server agent, and changes in the environment, the organizations require frequent modifications. Therefore, it is critical that no *deadlock* is found in the system (F1). Deadlock is directly supported in Uppaal. The second property (F2) guarantees that an organization will eventually reach a *Complete* state if new activity requirements had led it to an *Incomplete* state.

```
F1:  A[] not deadlock

F2:  A<> forall(Mid:MVD_id)
      ServerAgent1.SubmitTask && MVD(Mid).Incomplete
      imply MVD(Mid).Complete
```

6.2 SA (Robustness)

The verification of the self-adaptation behaviors for robustness is divided into four properties. The first property defines that if a MVD organization reaches an Incomplete state, eventually the self-healing system will detect the failure and look for a resource to recover the organization (R1). This can be expressed as a safety property, stating that the self-healing manager will not consider the MVD as being in a *AllFine* state if the structure is *Incomplete*.

```
R1: A[] forall(Mid:MVD_id)
    MVD(Mid).Incomplete imply !SHManager(Mid).AllFine

R2: A<> forall(Mid:MVD_id)
      SHManager(Mid).LookForFreeResources
      imply SHManager(Mid).AllFine
```

```
R3: A<> forall(Mid:MVD_id)
      MVD(Mid).Incomplete imply MVD(Mid).Complete || MVD(Mid).Redundant

R4: A<> forall(Mid:MVD_id) forall(Pid:phone_id)
      PhoneState(Pid).Free
      imply mvdStruct[Mid].member[Pid]==0
```

R2 verifies that when the self-healing manager searches for a free resource, will eventually find one, under the assumption that no more than two phones will be Undesired after all the environment changes have been processed. A consequence from the previous two properties states that if a MVD organization gets into an Incomplete state it will eventually recover from it (R3). Finally we verify that in case a phone detects a GPS degradation, it will eventually be excluded from any involved organization (R4). The *mvsStruct* represents a global data structure containing all the MVDs in the activity (see Fig. 3b).

These properties allow us to determine the correctness of the SA layer behavior on top the legacy MAS, analyzing that self-adaptive actions are correctly designed and transmitted to the MAS through probes and effectors.

6.3 Results

The separation of concerns in a SA-MAS allowed us to study and verify functional properties of the system, while relieving us from focusing on self-adaptation processes. Similarly, the layered design of the SA-MAS allows us to concentrate on the verification of robustness properties.

Describing a SA-MAS using formal methods offers rigor but also is an additional effort for the development process. Moreover, quality properties verification is a demanding task that scales up exponentially due to the multiple options that derive from each state in the SA-MAS. To provide some measurements, verification of deadlock in a system with one MVD and 3 phones required 6.72 seconds, 14.85 seconds for 2 MVDs and 6 phones and up to more than 4 hours for 3 MVDs and 12 phones.

7 Implementation and Lessons Learnt

Extending a legacy system with a SA layer involves intrusive actions to be performed on the adapted system. The SA processes require internal information from the legacy system, therefore it is necessary to implement a set of *probes* components in the MAS that allow the data gathering that is necessary for the monitoring and analysis process. Moreover, behavior and organization modifications in the MAS require the inclusion of *effectors* that realize the plans defined in the SA. Our scenario, considers a MAS implemented in JADE [17], which uses software agents to cover functional requirements. The system is extended by new agent behaviors that allow data acquisition and agent behavior modifications.

The additional SA layer in the SA-MAS needs to be deployed on the mobile devices in a distributed manner. This is implemented by additional distributed agents in the system that perform the Monitor, Analyze, Plan and Execute roles [10] and interact with the MAS through the probes and effectors mentioned above. The formal models that describe processes for probes and effectors and SA components presented in 5 communicate through a basic signal passing mechanism in Uppaal. However, this process becomes more complex in the development phase. The modifications on the legacy system include additional communication protocols to interact with the newly added SA layer.

Moreover, the new SA layer in the SA-MAS includes processing efforts for the device. In addition to the domain functionalities provided by the agents in the MAS, processes related to self-healing need to be executed, which derives into a processing overhead.

The rigorous description and verification of properties through formal methods have also provided advantages for the implementation design. Detecting non-verified properties becomes valuable help to identify errors in the SA-MAS behavior. As an example, during the formal modeling process, we detected an incorrect definition of an organization in Redundant state. This would occur when an incoming task would reduce the number of required GPS and the MVD had been recovered from an Incomplete state.

8 Related work

Few researchers have performed explorative studies on self-adaptation in decentralized systems and MAS. [18] uses an auction-based mechanism to coordinate resource usage in self-adaptive multi-agent systems and defines areas in which an agent can bid for the resources. This approach is aligned with the MVD concept used in our SA-MAS. The authors' approach can be applied as a Planning mechanism inside a MAPE-K in the self-adaptive layer. [19] describes the key attributes of decentralized self-adaptive systems derived from a number of case studies. This study can be considered as an extension of the FORMS work, providing an architectural view to separate the functional requirements provided by a MAS from the quality properties offered by a SA layer. [20] expresses structural constraints over an architectural specification that are used by component managers to automatically configure the system. Managers exchange information using reliable broadcast channels, restricting scalability. This approach contrast with the use of local organizations (MVD) where self-healing approaches are applied, which would improve scalability issues. [21] introduces a gossip protocol to overcome the scalability limitation in distributed systems. This mechanism could be implemented in our SA-MAS reference model to disseminate necessary information for the SA of organizations in the MAS. The authors in [20] observe an evolution limitation in their approach in case of new requirements. Our layered SA-MAS approach can be an answer to this question, by reducing the level of intrusive modifications in the MAS limiting to probes and effectors. [22] presents the K-Components Framework to dynamically modify and evaluate a graph representations of a decentralized system, to apply the planned changes to the running system. The framework allows separating adaptive and functional system behaviors.

A related approach is presented in [23] that uses meta-agents to overcome the inflexibility of some MAS platforms. The author states that *"each appli-*

cation agent rests on a small set of interconnected and interacting meta-agents which provide the application agent with its abilities to encapsulate behavior and to communicate" [23]. The offered interfaces of the meta-agents facilitates the modification of the agent behavior and implementation. However, the approach does not provide a systematic solution to deal with quality properties as first class citizens.

9 Discussion and Future work

In this paper we introduced a reference model for SA-MAS, an architecture design that combines the benefits of multi-agent systems with quality properties that a self-adaptive mechanism can offer. We argued for the benefits of an additional SA layer on top of a MAS system in order to offer separation of concerns. This approach becomes beneficial for legacy MAS, but demands an intrusive step to include probes and effectors in the MAS in order to communicate with the SA. In order to verify that quality properties are provided, we have applied formal methods, which offer the means to rigorously describe behavior processes and verify desired properties. We have illustrated the separation of concerns in SA-MAS with a Mobile learning case study, where SA has been applied both for individual agent and organization adaptations.

Coming efforts include an exhaustive analysis of the implemented SA-MAS in real settings. This task will include the evaluation of the system behavior under undesired states, to study the reaction of an implemented SA-MAS for guaranteeing the desired quality goals. Secondly, traces from the use of the SA-MAS in real settings need to be compared with simulations performed in Uppaal. This task needs to be performed using Model-based testing techniques [24]. This study will strength the guarantees of the SA-MAS, to evidence that properties formally verified are transferred to the implemented system.

The question *How well does the system scale up?* needs to be answered, both by simulation in formal models and through empirical studies. The presented scenario describes a mobile learning activity, where numbers of participants vary in the range of 15-40 students. The simulation in Uppaal has been studied for a limited amount of mobile phones and has evidenced initial scalability issues. Morever, in other SA-MAS scenarios these numbers could become critical, which would require of further studies to determine its viability.

References

1. Wooldridge, M.: An introduction to multiagent systems. John Wiley & Sons (2002)
2. Weyns, D., Parunak, H.V.D.: Environments for multiagent systems state-of-the-art and research challenges. Springer (2005)
3. Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Computer Society **36**(1) (2003) 41–50
4. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. FOSE '07 2007 Future of Software Engineering (2007) 259–268

5. Esteva, M., Rosell, B., Rodriguez-Aguilar, J.A., Arcos, J.L.: AMELI: An Agent-Based Middleware for Electronic Institutions. In: Third International Joint Conference on Autonomous Agents and Multiagent Systems. Volume 1 of AAMAS '04., Washington, DC, USA, IEEE Computer Society (2004) 236–243
6. Parunak, H.V.D.: Digital pheromone mechanisms for coordination of unmanned vehicles. In: First international joint conference on Autonomous agents and multiagent systems. AAMAS '02, New York, New York, USA, ACM (2002) 449–450
7. Mamei, M., Zambonelli, F., Leonardi, L.: Cofields: a physically inspired approach to motion coordination. Pervasive Computing, IEEE **3**(2) (2004) 52–61
8. Weyns, D., Haesevoets, R., Helleboogh, A., Holvoet, T., Joosen, W.: The MACODO middleware for context-driven dynamic agent organizations. ACM Trans. Auton. Adapt. Syst. **5**(1) (2010) 3:1–3:28
9. Hübner, J.F., Boissier, O., Kitio, R., Ricci, A.: Instrumenting multi-agent organisations with organisational artifacts and agents. In: Autonomous Agents and Multi-Agent Systems. Volume 20. (April 2009) 369–400
10. Weyns, D., Malek, S., Andersson, J.: FORMS: a formal reference model for self-adaptation. In: 7th International Conference on Autonomic Computing. (2010) 205–214
11. Isern, D., Sánchez, D., Moreno, A.: Organizational structures supported by agent-oriented methodologies. Journal of Systems and Software **84**(2) (2011) 169–184
12. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On Interacting Control Loops in Self-Adaptive Systems. In: Software Engineering for Adaptive and Self-Managing Systems. (2011)
13. Raheja, R., Cheng, S.W., Garlan, D., Schmerl, B.: Improving architecture-based self-adaptation using preemption. In: First international conference on Self-organizing architectures, Berlin, Heidelberg, Springer-Verlag (2010) 21–37
14. Gil de la Iglesia, D.: Uncertainties in Mobile Learning applications: Software Architecture Challenges. Licentiate thesis, Linnaeus University (2012)
15. Iftikhar, M.U., Weyns, D.: A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System. Electronic Proceedings in Theoretical Computer Science **91** (August 2012) 45–62
16. Wing, M., Eklund, A., Kellogg, L.: Consumer-grade global positioning system (GPS) accuracy and reliability. Journal of Forestry (2005)
17. Bellifemine, F., Caire, G., Greenwood, D.: Developing Muti-Agent Systems with Jade. John Wiley & Sons, Liverpool, UK (2007)
18. Malek, S., Mikic-rakic, M., Medvidovic, N.: A decentralized redeployment algorithm for improving the availability of distributed systems. In: 3rd International Conference on Component Deployment. (2005)
19. Weyns, D., Malek, S., Andersson, J.: On Decentralized Self-Adaptation: Lessons from the Trenches & Challenges for the Future. In: Software Engineering for Adaptive and Self-Managing Systems, IEEE (2010)
20. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: Workshop on Self-Healing Systems, ACM (2002)
21. Sykes, D., Magee, J., Kramer, J.: FlashMob: Distributed Adaptive Self-Assembly. In: Software Engineering for Adaptive and Self-Managing Systems, ACM (2011)
22. Dowling, J.: Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems. PhD thesis, Univ. of Dublin (2004)
23. Lynch, S.: Using Meta-Agents to Build MAS Platforms and Middleware. In: 3rd International Conference on Agents and Artificial Intelligence. (2011) 28–30
24. Apfelbaum, L., Doyle, J.: Model based testing. Software Quality Week Conference (1997) 1–14