EAI Endorsed Transactions
on Self-Adaptive Systems
Research Article ICST.ORG

Patterns for self-adaptive systems: agent-based simulations

Mariachiara Puviani[1,*], Giacomo Cabri [2], Franco Zambonelli[3]

[1]DIEF, Università di Modena e Reggio Emilia, Via Vignolese 905/b, Modena, Italy
[2]FIM, Università di Modena e Reggio Emilia, Via Campi 213/b, Modena, Italy
[3]DISMI, Università di Modena e Reggio Emilia, Via Amendola 2, Reggio Emilia, Italy

Abstract

Self-adaptive systems are distributed computing systems that can adapt their behavior and structure to different kinds of conditions. This adaptation does not concern the single components only, but the entire system. In a previous work we have identified several patterns for self-adaptation, classifying them by means of a taxonomy, which aims at being a support for developers of self-adaptive systems. Starting from that theoretical work, we have simulated the described self-adaptation patterns, in order to better understand the concrete and real features of each pattern. The contribution of this paper is to report about the simulation work of three patterns as examples, detailing how it was carried out, in order to provide a further support for the developers.

Received on 23 October 2014; accepted on 19 January 2015, published on 28 January 2015

Keywords: Adaptation pattern, taxonomy, MAS, role

doi:10.4108/sas.1.1.e4

1. Introduction

Software is today the main enabler of many of the appliances and devices omnipresent in our daily life [27]. Software systems are catching on in different domains and environments. These domains are characterized by mobility, rapid changes in operation conditions along with changes in the systems' environment, and so on. To face to these problems, self-adaptation has been proposed as a solution to manage the growing complexity of systems: with the continuous increase in runtime scale and complexity of software systems, self-adaptation has assumed a central role in the software engineering development, and has been often mentioned as one of the defining challenges for the discipline [11].

In software engineering, self-adaptation is defined as the ability of a system to autonomously adapt its behavior to dynamic operating conditions [28]. Software systems need to adapt to failures, changes in their computing and physical environments, and to the availability of new or upgraded services [9]. Adaptation can occur at two levels: at the level of a single component, and at the level of the entire system viewed as an ensemble of components.

In order to build self-adaptive systems, developers can choose a specific adaptation pattern among available ones. In a the ASCENS project[1] we have catalogued them [18], and such a catalogue turns out to be useful, because a pattern describes a generic solution for a recurring design problem and the application of these adaptation patterns helps to develop a system that exhibits specific adaptation features and that is able to self-adapt during all its life. Often the same problem can be solved with different approaches, which means that different adaptation patterns can be used. Because the system's performances are based on environmental/external conditions, if a different pattern describes the system, it makes the system behave and also perform in different way (related to the same environment/external conditions). This leads to

★Please ensure that you use the most up to date class file, available from EAI at http://doc.eai.eu/publications/transactions/latex/

*Corresponding author. Email: mariachiara.puviani@unimore.it

[1]http://www.ascens-ist.eu/

consider that for different conditions, different patterns are better than others.

Furthermore, in a system, features like the execution time, that are considered non-functional requirements, may become functional requirements during systems life (e.g. if batteries discharge in a robot). Using patterns (that is able to describe them), these non-functional requirements can be well implemented in order to make the system to adapt to these kind of new possible situations.

Starting from the catalogue of adaptation patterns, we wrote a taxonomy table [22] that will help developer to choose the most suitable pattern for their systems. Moreover, to better understand (and take advantages from) the specific features of each adaptation pattern and its applicability, we have simulated the behavior of different patterns. The simulations allow us to define a "table of applicability" that will complete the initial taxonomy of patterns. In this paper we report the simulation of three patterns, taken as examples; the same approach can be adopted for the other patterns. The choice of these three patterns is motivated by the fact that they are the "basic" patterns of each ensemble level of adaptation patterns.

In simulating self-adaptive systems we take advantages of *agents*. Software agents represent an interesting paradigm to develop intelligent and distributed systems, because of their autonomy, proactiveness and reactivity. In addition to that, their sociality enables the distribution of the application logic in different agents that can interact with each other and with the host environment. In our work we use Multi Agent Systems [10] in order to simulate complex self-adaptive systems. This is because, as said before, agents exhibit features very relevant for self-adaptation.

The chosen mechanism used to implement adaptation patterns is the "role based approach" [5]. Roles are sets of behaviors common to different entities that can be applied to the context in which a component is behaving. An adaptive pattern can be described in terms of the roles the different components play. That will allow us to apply different roles to agents, in order to simulate different adaptation patterns.

The remainder of this paper is organized as follows. In Section 2 we present the taxonomy of adaptation patterns and show some adaptation patterns; in Section 3 we present the "role based approach" and how agents are used to develop the considered patterns. Section 4 presents two different tasks relates to a case study, and simulations implemented using RoleSystem. Moreover this section shows how these simulations help us to improve the initial taxonomy and validate the use of adaptation patterns in developing self-adaptive systems. Section 5 discusses related work in the area, while Section 6 discuss the results of the work and conclude the paper.

2. Use of patterns

Because patterns are able to describe generic solutions for a recurring deign problem, their use to design self-adaptive systems is very relevant. Moreover, a very important task to develop a well performing self-adaptive system, is to understand which pattern to choose. Defining how a pattern works in a self-adaptive system and which kind of systems are covered by a specific pattern, we create a preliminary taxonomy presented in Figure 1 [22].

In order to understand the table we have to define the basic components that are used in. A Service Component (SC) is a context dependent component. The component lives and acts inside an environment and with other components that create an ensemble. Its interactions both with the environment and with the other components constitute its context. Moreover, an Autonomic Manager (AM) is an AC that externalizes the feedback loop used to adapt one or more components.

This table describes the different patterns listed in different levels:

- the single components level - first row;
- the ensemble level where the environment is considered as the means of adaptation (e.g. a bio-inspired system) - second row;
- the ensemble level where adaptation is delegated to an external agent called Autonomic Manager (AM) that manages all the other agents (e.g. centralised system with regard to the adaptation aspects) - third row;
- the ensemble level where adaptation is delegated to the agents themselves though their direct communication of adaptation mechanisms - forth row.

How these patterns are useful to build self-adaptive systems has been demonstrated in different works like [17], [19], [20] and [16].

With the aid of this taxonomy table and the catalogue of pattern presented in [18], we are able to understand which pattern better describe a system. For example the table tells us that to build a system where it is important that the mechanisms for adaptation are shared between all the components, one of the patterns of the fourth row can be used.

However, to better understand the functionality of adaptation patterns, we aim at deeper analyzing, with the aid of simulations, the use of adaptation patterns. So we started analyzing the "basic" pattern of each ensemble level (the single level is delegated to the internal of each agent). In this article we only report the studies on the first column of patterns - starting from the taxonomy table, while the other columns are

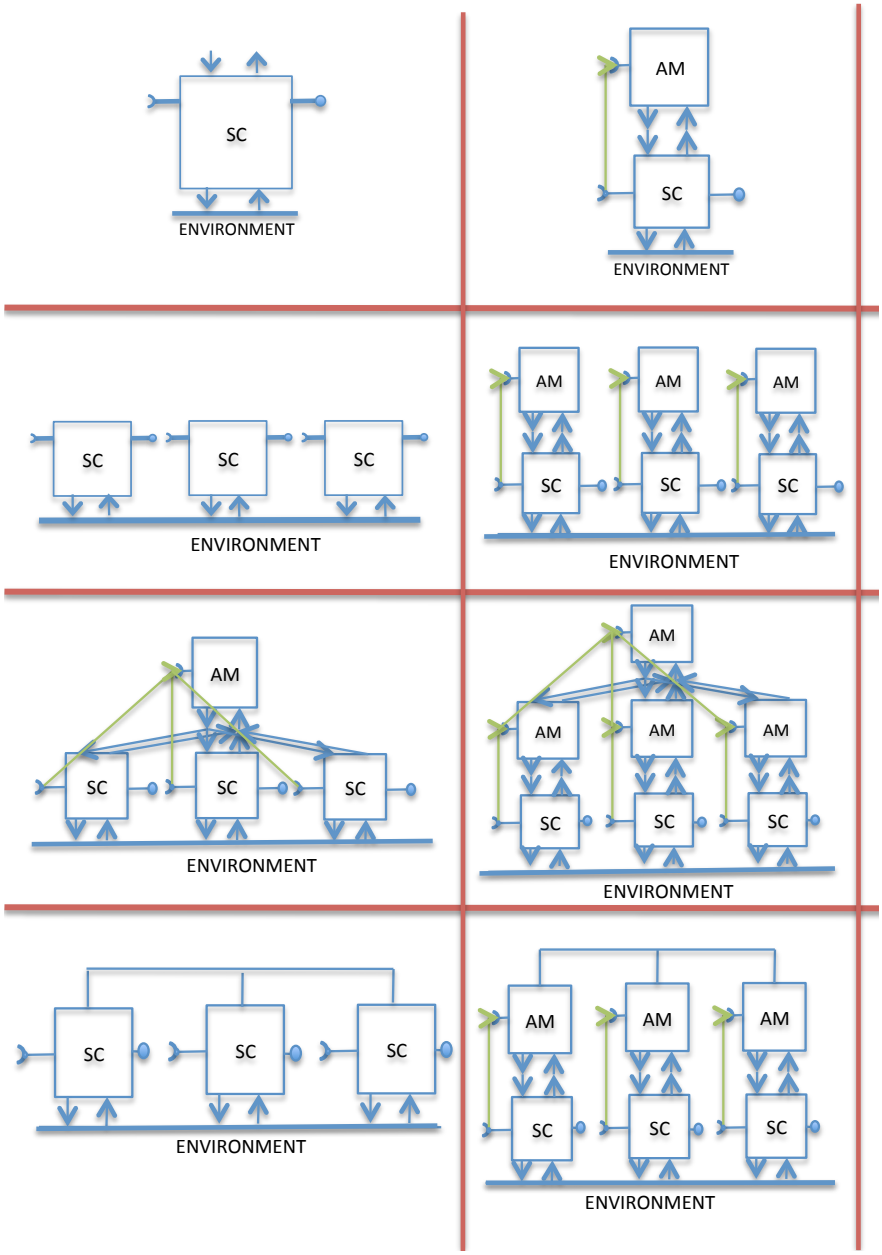

Figure 1. Part of the taxonomy of self-adaptation patterns based on different composition mechanisms

derived from this, adding more and more levels of AMs that are able to manage adaptation of different parts of the system (SCs or other AMs). These three patterns are the most specific that can be chosen in order to describe the different class of patterns. Moreover they include some of the most important class of adaptation: adaptation via stigmergy, adaptation using direct communication between peers, and adaptation propagated via a supervisor (AM).

First of all we present in the next subsections some of the studied patterns. In term of importance, we report a shorter description of each pattern. We report the "context" of the system that will apply that pattern, its main "behavior", its structure (see the different Figures), and its "consequences".

2.1. Reactive Stigmergy Service Components Ensemble Pattern

Context: This pattern has to be adopted when:

1. there are a large amount of components acting together;

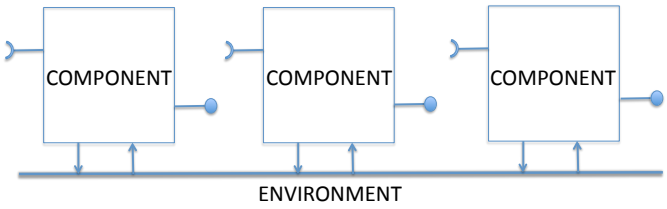

Figure 2. Reactive Stigmergy SCE Pattern

2. the components need to be simple component, without having a lot of knowledge in their internal;

3. the environment is frequently changing;

4. the components are not able to directly communicate one with the other.

Behavior: This pattern has not a direct feedback loop. Each single component acts like a bio-inspired component (e.g. an ant). To satisfy its simple goal, the component acts in the environment that senses with its "sensors" and reacts to the changes in it with its "effectors". The different components are not able to communicate one with the other, but are able to propagate information (their actions) in the environment. Than they are able to sense the environment changes (other components reactions) and adapt their behavior due to these changes.

Consequences: If the components composing the systems are proactive, their behavior is defined inside the components along with their internal goal. The behavior of the whole system cannot be *a priori* defined. It emerges from the collective behavior of the ensemble. The components do not require a large amount of knowledge. The reaction of each component is quick and does not need other managers because adaptation is propagated via environment. The interaction model is an entirely indirect one.

2.2. Centralized AM Service Components Ensemble Pattern

Context: This patterns has to be adopted when:

1. the components are simple and an AM is necessary to manage adaptation;

2. a direct communication between components is necessary;

3. a centralized feedback loop is more suitable because a single AM has a global vision on the system;

4. there are few components composing the ensemble.

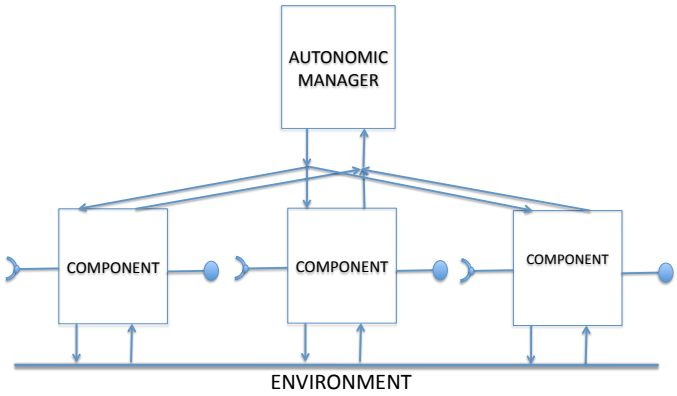

Figure 3. Centralized AM SCE Pattern

Behaviour: This pattern is designed around a unique feedback loop. All the components are managed by a unique AM that "controls" all the components behavior and, sharing knowledge about all the components, is able to propagate adaptation.

Consequences: A unique AM is more efficient to manage adaptation over the entire system, but it may become a point of failure. If the AM fails, mechanisms of negotiation have to be applied in order to create another AM, when it is possible.

2.3. P2P Negotiation Service Components Ensemble Pattern

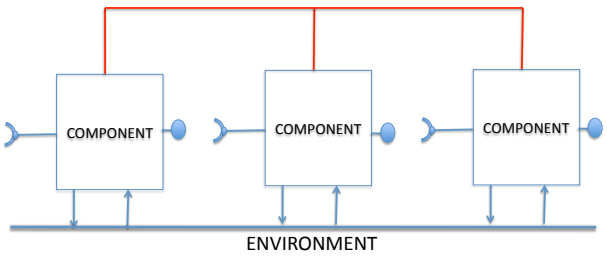

Figure 4. P2P Negotiation SCE Pattern

Context: This patterns has to be adopted when:

1. the components are proactive;

2. the components need to directly communicate one with the other to propagate adaptation.

Behavior: Each component is managed by an internal and implicit AM. The components directly communicate one with the other with a P2P communication protocol.

Consequences: The communicate between components makes it possible to share knowledge.

3. Agents and the Role based approach

To evaluate patterns, we implement them using roles. In the following we present the "role theory" and the concept of roles in subsection 3.1, while in subsection 3.2 we introduce the implementation of roles, using RoleSystem.

3.1. Roles

The "role theory" [3] has been applied to several computer science fields. For this reason there are several definitions of the concept of role, depending on the considered scenario. The interesting feature of roles is that they can be used as a paradigm to smartly model the view of a complex system [12]. As the author said, role is defined as "a set of behaviors common to different entities, with the possibility to apply them to an entity in order to change its capabilities and behavior".

Roles are very important not only because can be applied to existing entities to change their behavior, but also because they can be reused in different situations. For this reason they are considered as solutions common to different problems, as the same way patterns are considered as solutions common to different systems. Roles can also be used to manage interactions between components. These interactions are not *a priori* defined between components, but only occur when the roles involved in the interaction are associated to components. It is important to note that roles are tied to the local execution environment, thus they represent context-dependent views of entities running in that environment [1], granting adaptability. Moreover roles grant portability and generality: since they are tied to each interaction context, they hide context details to components, which are free to discard those "low-level" details. For all these reasons roles can be useful to build adaptation patterns.

In order to play a role, a component must assume it. In other words, a component must choose a specific role that means that the role assumption is considered an active process of the component's adaptation. In complex system, using the agent paradigm, we assume that a role is a software component (e.g. a Java class) that can be added to each service component of the ensemble. In our approach, a role is modelled as a set of capabilities and an expected behavior, both related to the component (i.e. agent) that plays such role.

There are some features that are proprietary of a role (independently to the component they are applied to):

- a role is temporary, a component may play it in a well-defined period of time or in a well-defined context;
- a role is generic, it is not tightly bound to a specific application, but it expresses general properties that can be used in different applications and then for different components;
- a role is related to a context, each environment can impose its own rules and can grant some local capabilities, forcing components to assume specific roles in order to adapt to such environment.

So roles represent the behavior that components are expected to show; who expects such behavior are entities external to components themselves, mainly organizations [32] and environments. This model leads to a twofold viewpoint of the role. From the application point of view, the role allows a set of capabilities, which can be exploited by components to carry out their tasks. From the environment point of view, the role imposes a defined behavior to the entities that assumes it.

We are especially focused on the latter point that is a component assuming a given role is expected to exhibit a specific behavior. In our specific case, since agents are at least reactive components, they are sensible to what happens in the environment where they live, they can analyze the perceived information and infer what behavior can apply.

In the Agent field, "roles" represent a cross-cut view of the agent space, and thus can be adopted to model dynamic and open environments. Roles can be applied to agents in order to both enhance their capabilities, granting a better adaptability, and to model interactions and coordination in MAS systems [6].

3.2. RoleSystem and roles implementation

Based on the concept of role, we exploit RoleSystem [7]. RoleSystem is an interaction infrastructure completely written in Java. This will grant high portability and the capability to be associated with the main agent platforms. The agent platform we chose to exploit RoleSystem is Jade [2], a FIPA compliant agent platform, and perhaps the most exploited one; but we have associated RoleSystem also to Aglets [15], a mobile agent platform originated from IBM and now open source.

The RoleSystem infrastructure is divided in two parts, as shown in Figure 5: the upper one is independent of the agent platform, while the lower part is bound to the chosen agent platform (i.e. Jade).

As said before, agents are a key point of RoleSystem. In applications exploiting the RoleSystem infrastructure, an agent is composed of two layers: the subject layer, representing the subject of the role - independent of the platform; and the wrapper layer, which is the Jade agent in charge of supporting the subject layer.

A specific agent, called Server Agent, is in charge of managing the roles and their interactions for each context/environment. It interacts with the wrapper

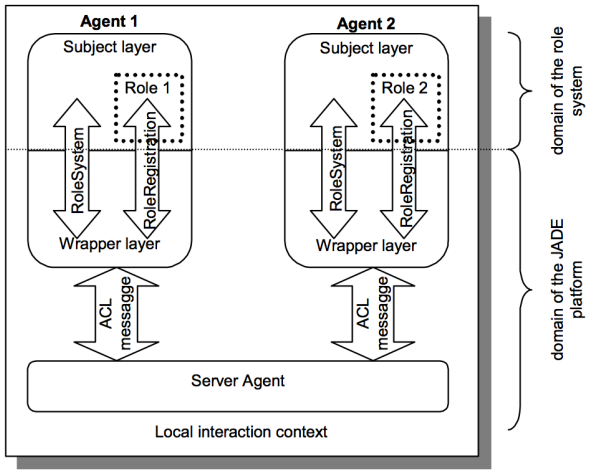

Figure 5. RoleSystem: separation of domain

layer of agents by exchanging ACL messages formatted in an appropriate way.

Every agent is a subject that performs some actions and on which some event can happen. So a "role" is defined as a set of actions that an agent assuming that role can play, and a set of events that an agent assuming that role can recognize. Based on this idea, every agent can choose the role on its temporary and immediate necessities.

We do not start from scratch, but starting from the existing RoleSystem, we developed several Java classes, both platform-independent and related to Jade. The main platform-independent classes, that are in the *rolesystem.core* and *rolesystem.roles* packages, are reported in the UML diagram of Figure 6.

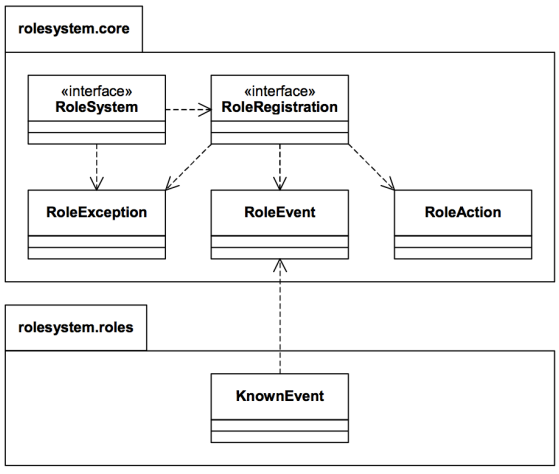

Figure 6. RoleSystem packages

The connection between the subject layer and the wrapper layer is granted by two Java objects, instances of classes implementing respectively the *RoleSystem* and *RoleRegistration* interfaces, which provide methods to register agents with roles, to search for agents playing a given role, to listen for events and to perform actions. The *RoleSystem* interface enables agents to perform preliminary operations needed to assume a specific role; while the *RoleRegistration* interface enables agents to perform operations on the system via a specific registration (i.e., after that the agent has assumed a role).

To play a role, an agent has to obtain an object that implements the *RoleRegistration* interface, invoking the *reqRegistration* method. The returned object represents the association between the agent and the specific role. As soon as an agent does not need to play the assumed role, it can release the role registration via the *dismiss* method. If the agent wants to assume a role again (one just assumed or another one), it has to require another registration via the *reqRegistration* method.

A role is implemented by an abstract class, where the features of the role are expressed by static fields and static methods. The class that implements a role has the same name of the role, and it is part of a package that represents the application scenario for such role. A static and final field called ROLE_ID identifies the role. Each action defined in a role is built by a static method, which is in charge of creating an appropriate instance of the class *RoleAction* and returning it to the caller. Such a static method has the same name of the corresponding action and one or two parameters: the former one is the agent addressee of the event corresponding to the action; the latter parameter (optional) is the information content to perform that action.

To perform an action, an agent playing a given role must obtain the appropriate *RoleAction* instance, invoking the corresponding static method of the role class. Then, it has to invoke the *doAction* method of *RoleRegistration* to actually perform the action, supplying the previously created instance of *RoleAction*. Then, when the server agent receives the request to perform the action via the wrapper layer, translates it into a known event, and sends it to the addressee agent. To find partners to interact with, an agent exploits the *searchForRoleAgent* methods made available by the *RoleSystem* interface, by which the agent can get a list of the registrations related to a specific role, each one specified by an identifier.

Starting from RoleSystem we have implemented a scenario of swarm robotics. There we develop dedicated Java classes representing the used roles (see Section 4).

4. Evaluated case studies

In our work, we consider the use of roles as an ideal starting point to implement different simulations that develop self-adaptive systems. With the aid of RoleSystem, we implement agents systems to develop

self-adaptive systems, starting from different case studies.

Since this work has been carried on in the context of the ASCENS project, we use two different tasks of one of the case study presented in the project to develop RoleSystem classes and to implement different simulations.

In particular, we take care of a swarm robotics scenario: the *disaster recovery* scenario (Figure 7).

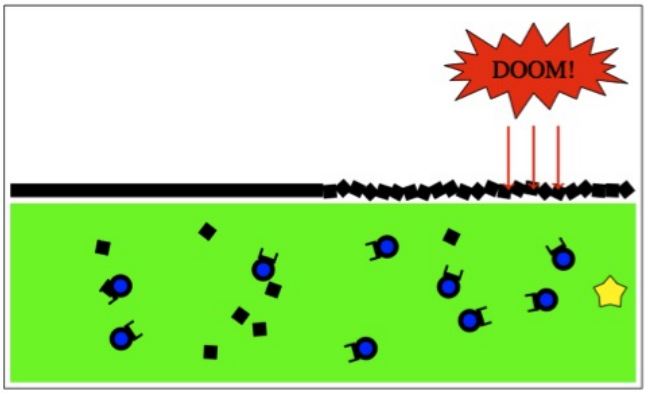

Figure 7. Disaster recovery scenario - ASCENS case study

As reported in [26], in the *disaster recovery* case study, we imagine that a disaster happened, such as the catastrophic failure of a nuclear plant, or a major fire in a large building. We also imagine that an activity of search and rescue must be carried out. For instance, people may be trapped inside a building and they must be found and brought to safety. Given the high danger of operating in such environment, it is realistic to think that an ensemble of robots could be used to perform the most dangerous activities. Among these activities, four are very relevant: exploring the environment, mapping dangerous areas and targets to rescue, performing the rescue, and seal the dangerous areas. In particular, we analyze and present here, two task to understand how different patterns behave in order to build such a system: the *environment exploration* task and the *perform the rescue* task. The two tasks and their implementations are presented in the following.

In this scenario, the necessity to develop an adaptive system is given by the presence of an unknown environment that is continuously changing: the dangerous areas can enlarge or a new one can be created, new obstacles can appear, the number of victims is not a priori known and can change. Moreover, the system will need not only to adapt to external conditions, but also in its composition because some of the components could broke or remain imprisoned in the disaster.

4.1. "Environment exploration" task implementation

In this scenario, an ensemble of robots (simulated by agents) has the task to explore the environment in order to map it. The ensemble is composed of a given number of autonomous robots that are initially randomly distributed in an area (e.g. each position is randomly chosen once a robot is entered in the area). The goal of the ensemble is strictly connected to the utility of minimizing the time of completion of the task. Another important utility of the system is to provide an equal work balancing for all the components that form the ensemble. A single robot does not know in advance the number of components of the ensemble. At the same way the robot does not know anything on the shape and the length of the environment.

To implement the simulations of this scenario we have defined an area of the size of 30x30 cells, with 15 obstacles in, randomly distributed. The obstacles can be walls or something else (e.g. ruins from an explosion) that are not possible to be removed from a robot. Due to the size of the area we initially implemented 16 robots moving in (they are a right amount compared to the size of the area, in order to be a swarm). The choice of using 16 robots was made after a lot of experiments in real world scenarios: using real robots in an arena of 3x3 meters and with the described obstacles. From these experiments we saw that using less robots was too time consuming (due to the large part of ground to explore), and using more robots, they spent a lot of time avoiding not only obstacles but also the other robots.

In Figure 8 we can see a screenshot of the environment.

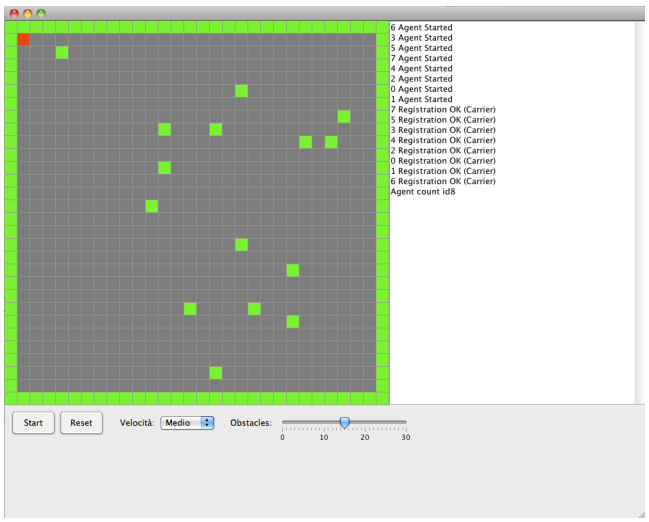

Figure 8. Environment of the "environment exploration scenario"

In order to implement each pattern, we wrote one or more specific role with RoleSystem, which robots must play during their life.

The first pattern we used to implement the system was the "Reactive Stigmergy" pattern. This is because the system's requirements are well described in the context of this pattern, and because usually, in swarm robotics, the stigmergic mechanism is the one that best fits for adaptation.

For this pattern, we implemented the "Explorer" role. With this role a robot starts to explore the cells in its surrounding, following an anti-clockwise random movement. Using the *moveRandom(RoleRegistration registration)* method described in the following, a robot leaves a pheromone every time it explores a new cell, and when it senses this pheromone, left by other robots in a cell, it is able to understand that this cell has been explored by others. The simulation ends when every robot does not find any free cell unexplored (by itself) in a range of 10 cells. An example of code of the "Explorer" role is reported in Figure 9.

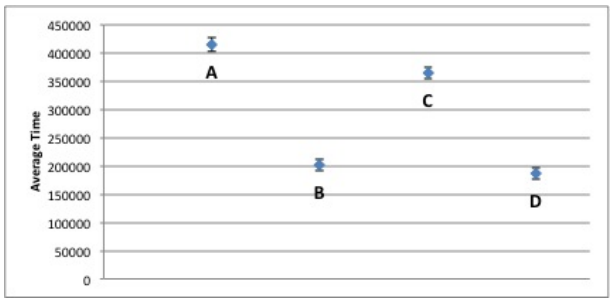

Figure 10. Average time of goal's satisfaction and standard deviation with different patterns: (A) Reactive Stigmergy pattern, (B) Centralized AM pattern, (C) Centralized AM Pattern with failure, (D) P2P Negotiation pattern

We performed hundreds of simulations using this pattern and in Figure 10 - (A) we show the average time of the goal's satisfaction, using 16 robots, and the standard deviation. As we can see, the average time is about 418000 millisec. In these simulations every robot colours the pavement of the area (while it lets the pheromone) of a different colour. However, at the end of these simulations, only few colours appear on the screen, as we can see from Figure 11. This is because it happens that most of the time a robot explores again a cell just explored by another robot. This is a drawback of patterns based on stigmergy.

To overcome this, we implemented the same system using also the "Centralized AM" pattern and the "P2P" pattern.

For the "Centralized AM" pattern we implemented the "Manager" and the "Slave" roles. An example of code of the "Manager" role is reported in Figure 12.

The manager, communicating with all the robots, receives the information about their positions and the explored area. Then it merges all the information received to create a map of the explored area and it

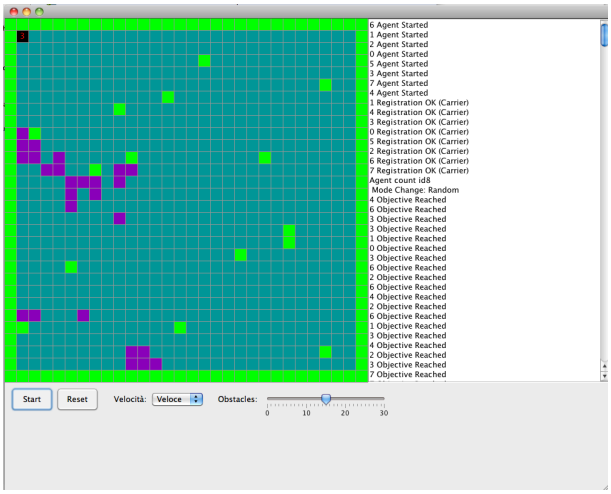

Figure 11. Screenshot of a simulation implementing the Reactive Stigmergy pattern

shares this map to the other robots that are able to understand the perimeter of the known area in order to explore the exterior of that area.

An example of the code of the "Slave" role is reported in Figure 13.

With this method the robot-slave is able to update its knowledge with the data of all the other robots, by updating its *myKnownSquares* variable with data given by the manager.

The exploration ends if the built map finds a closed perimeter explored (the perimeter is outlined by obstacles in closed cells) or if all the robots do not find unexplored areas in the surrounding of a radius of 10 cells.

As we can see from Figure 10 - (B), the average time of the goal's satisfaction is better than the same system developed using the "Reactive Stigmergy" pattern, but it remains high (it was compared with expected results studied in previous works). This is due to the necessity of coordinating a large number of robots in an unknown environment. The fact of not knowing the environment makes it very difficult to well coordinate the robots, because the manager knows the other robots' positions and the explored area, but it does not know which area still misses to be explored.

Moreover, if the AM fails (example not reported in this simulation), it has to be replaced by negotiation. Here, the simplest and less time consuming strategy for negotiation is that the first robot that senses the AM is out of work, become the new AM. This solution is also the most dangerous one because more than two robots may become AM. Other solutions use negotiation algorithms.

Instead, the "P2P" pattern does not suffer from the limitation of having a single point of failure.

```
public KnownEnvironment moveRandom(RoleRegistration registration) throws RoleException
      {
            agent_data.setMode(1); // 1 random
            DataOutputManager.dataSim.setagentData(id, agent_data);
            while(ExplorerSL.END_SEARCH_MOVEMENT==false)
            {
                  for(int movement_counter=0;movement_counter<10;movement_counter++) //check done every 10 movement
                  {
                        sleepAgent(PAUSE);
                        int modeChange=contactManager.receiveModeChangeMessageRandom(registration);
                        saveRandomData();
                        if(modeChange==1)
                        {
                              NEW_MASTER_TO_INIT=true;
                              return knownEnvironment;
                        }
                        if(modeChange==2)
                        {
                              SLAVE_INIT=true;
                              return knownEnvironment;
                        }
                        MoveData moveData=searchUntilMove(PHEROMONE_MODE);
                        KnownEnvironmentManager.setKnownEnvironments(id,knownEnvironment);
                        if(moveData.getRange()==10 && moveData.getRouteSize()==0)
                        {
                              ExplorerSL.setSimulationEnd(id, true);
                              break;
                        }
                        drawVisualization();
                  }
                  checkThreshold(registration);
            };
            return knownEnvironment;
      }
```

Figure 9. Fragment of code of the "Explorer" role

```
public void moveMaster(RoleRegistration registration) throws RoleException
{
      ExplorerSL.setSimulationEnd(id, true);
      agent_data.setMode(3); // 1 random // 2 slave // 3 master
      DataOutputManager.dataSim.setagentData(id, agent_data);
      while(ExplorerSL.END_SEARCH_MOVEMENT==false)
      {
            boolean modeChange=false;
            for(int i=0;(i<ExplorerSL.id_counter) && modeChange==false;i++)
            {
                  sleepAgent(800);
                  int myKnownSquares=KnownEnvironmentManager.getKnownEnvironments(id).getKnownNumber();
                  modeChange=contactManager.receiveModeChangeMessage(registration);
                  saveMasterSlaveData(myKnownSquares);
            }
contactManager.refreshExplorers(registration,KnownEnvironmentManager.getKnownEnvironments(id));
            if(modeChange==true)
            {
                  ExplorerSL.setSimulationEnd(id, false);
                  break;
            }
      };
}
```

Figure 12. Fragment of code of the "Manager" role

Furthermore, more that the "Reactive Stigmergy" pattern, in this pattern the adaptation mechanism is shared between components. In the "P2P" pattern we developed the "Negotiator" role that permits a component to share its knowledge with neighbours in a range of 10 cells and to negotiate with them which cell has to be explored next.

An example of the code of the "Negotiator" role is reported in Figure 14.

With this role the time necessary to exchange adaptation messages between neighbours does not invalidate the ultimate satisfaction of the goal: components are able to coordinate themselves in little groups and if some of them expire, the adaptation of the whole system goes on, as the Negotiator role allows

```
public void moveSlave(RoleRegistration registration) throws RoleException
      {
            agent_data.setMode(2); // 1 random // 2 slave // 3 master
            DataOutputManager.dataSim.setagentData(id, agent_data);
            while(ExplorerSL.END_SEARCH_MOVEMENT==false)
            {
                  for(int movement_counter=0;movement_counter<5;movement_counter++) //check every n movement
                  {
                        sleepAgent(PAUSE);
                        int myKnownSquares=KnownEnvironmentManager.getKnownEnvironments(id).getKnownNumber();
                        boolean modeChange=contactManager.receiveModeChangeMessage(registration);
                        saveMasterSlaveData(myKnownSquares);
                        if(modeChange==true)
                        {
                              return;
                        }
                        MoveData moveData=searchUntilMove(false);
                        KnownEnvironmentManager.setKnownEnvironments(id,knownEnvironment);
                        if(moveData.getRange()==10 && moveData.getRouteSize()==0)
                        {
                              ExplorerSL.setSimulationEnd(id, true);
                              break;
                        }
                        drawVisualization();
                  }
                  if(ExplorerSL.searchType!=2)
                  {
                       checkThresholdSlave(registration);
                  }
      contactManager.sendExplorationData(registration,KnownEnvironmentManager.getKnownEnvironments(id));
            };

            return;
      }
```

Figure 13. Fragment of code of the "Slave" role

```
public void moveNegotiator(RoleRegistration registration) throws RoleException
{
      DataOutputManager.dataSim.getAgentData(id).setMode(4); // 1 random // 2 slave // 3 master // 4 negotiator

      BooleanNegotiator.set(ConfigurationParameters.END_CHECK_BASE_BOOL + String.valueOf(id),true);

      while(BooleanNegotiator.get(ConfigurationParameters.END_SEARCH_MOVEMENT_BOOL)==false  &&
BooleanNegotiator.get(ConfigurationParameters.EXIT_AGENT_BASE_BOOL + String.valueOf(id))==false)
      {
            sleepAgent(ConfigurationParameters.PAUSE-100);

            boolean modeChange=false;
            int Neighbour = NEIGHBOUR;
            for(int i=0;(i<Neighbour && modeChange==false);i++)
            {
                  int myKnownSquares=KnownEnvironmentManager.getKnownEnvironments(id).getKnownNumber();
                  modeChange=contactNeighbour.receiveModeChangeMessage(registration);
                  saveNegotiatorData(myKnownSquares);
            }

            contactNeighbour.refreshExplorers(registration);

            MoveData moveData=searchUntilMove(NextNeighbour);
            for (i=0; i<Neighbour; i++)
            {
                  if(moveData.getRange()==10 && moveData.getRouteSize()==0 && moveData.getExplored(Neighbour))
                  {
                        BooleanNegotiator.set(ConfigurationParameters.END_CHECK_BASE_BOOL + String.valueOf(id),true);
                        break;
                  }
            }
            drawVisualization();
      };
```

Figure 14. Fragment of code of the "Negotiator" role

it. Figure 10 - (D) shows the average time in simulations that use this pattern.

4.2. "Performing the rescue" task implementation

In this scenario, an ensemble of robots (simulated by agents) has the task to find objects (i.e. people to assist and rescue) and to carry them to the way out of the area (e.g. a blazing building). As for the previous task, the ensemble is composed of a given number of autonomous robots that are initially randomly distributed in an area.

The goal of the ensemble is strictly connected to the utility of minimizing the time of completion of the task. A single robot does not know in advance the number of components of the ensemble. At the same way the robot does not know anything on the environment (number and position of obstacles) and on the objects to rescue (number and position).

For the implementation of the simulations, we use the same area of the previous scenario. Here we initially implement 8 robots moving in. In Figure 15 we can see a screenshot of the environment where yellow dots are objects to rescue, black dots are the agents moving around, the green dots are the obstacles to avoid and the red dot is the light that indicate the way out to the area.

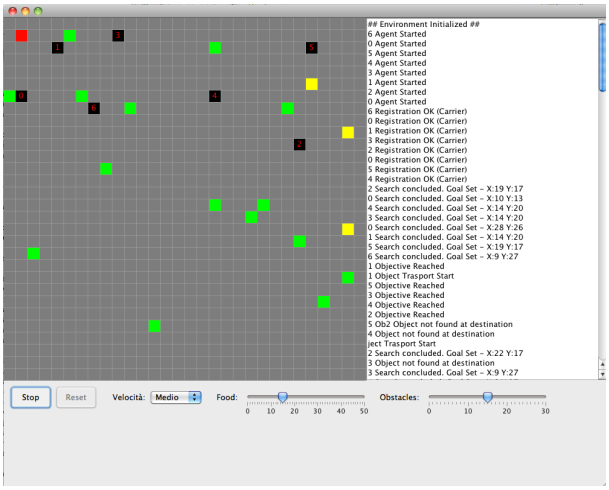

Figure 15. Environment of the "performing the rescue scenario"

As for the previous implementation, in order to implement each pattern, we wrote one or more specific role with RoleSystem, which robots must play during their life. The implementation of the different patterns is very similar to the previous example.

For the implementation of the "Reactive Stigmergy" pattern, we wrote the "Carrier" role. With this role a robot randomly move in order to find an object. First of all it explores the cells in its surrounding. If it does not find anything, it moves randomly to another position and starts again to search. When it finds an object, it carries it and goes back to the way out of the area (red light). The simulation ends when every robot does not find any object in a range of 10 random movements.

An example of code of the "Carrier" role is reported in Figure 16.

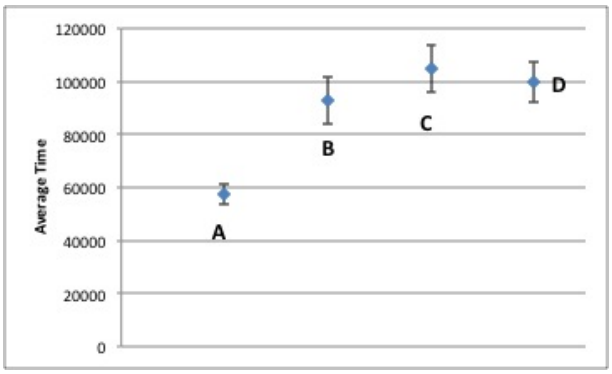

Figure 17. Average time of goal's satisfaction and standard deviation with different patterns: (A) Reactive Stigmergy pattern, (B) Centralized AM pattern, (C) Centralized AM pattern with failures, (D)) P2P Negotiation pattern

We performed hundreds of simulations using this pattern and in Figure 17 - (A) we show the average time of the goal's satisfaction, using 8 robots, and the standard deviation. As we can see, the average time is about 57000 millisec.

To verify the choice of the pattern, we implemented the same system using also the "Centralized AM" pattern and the "P2P" pattern.

For the "Centralized AM" pattern we implemented the "Manager" and the "Slave" roles. An example of code of the "Manager" role is reported in Figure 18.

The manager, communicating with all the robots, receives the information about their positions and what they find. Then it merges all the information received to and creates a map of the explored area to steer all the slaves in unexplored areas to find new objects. Here the most of the time spent, is used by the manager to merge the different information and to indicate new ways to slaves. The exploration ends if the manager does not receive new information about carried objects from all the slaves, in a time of 1000 millisec.

The code of the "Slave", not reported here, permits the slave to send information to the Manager about its position and when it finds an object to carry. Moreover, the "Slave" is able to receive from the Manager indication about where to go in exploration.

As we can see from Figure 17 - (B), the average time of the goal's satisfaction is worse than the same system developed using the "Reactive Stigmergy" pattern, because in an unknowing environment, where is also not know the number of objects to rescue, the operations to coordinate all the agents is time expansive.

From the study of this pattern, we can see that instead it will be very useful in systems where a robot is not able to carry an object alone and collaboration in term

```
private void Carrier_Logic(RoleRegistration registration, PositionSq goal) throws RoleException
    {
         Square test=EnvironmentManager.pickObject(new PositionSq(goal));
         if(test==null)
         {
              DataOutputManager.printString(id + " Object not found at destination");
              agent_data.inclostObjects();
              DataOutputManager.dataSim.setagentData(id, agent_data);
              return;
         }
         else
         {
              agent_data.incobjectCarried();
              DataOutputManager.dataSim.setagentData(id, agent_data);
         }
         movementManager.carryObjectSimple();
         return;
    }
```

Figure 16. Fragment of code of the "Carrier" role

```
private void ruleManager(RoleRegistration registration) throws RoleException
{
    EventData data=communicationManager.requestEvent(Carrier.KE_youHelpAtPosition);
    if(data!=null) // If i received a help request, communicate and reach the sender
    {
     DataOutputManager.printString(id +" Coordinator Request Received. Sending data");

     registration=roleSystem.reqRegistration(Slave.ROLE_ID);
     AgentMap.setAgentRoleID(id, registration.whoAmI());

     PositionSq positionToReach=data.getContent().getPosition();
     int contactID=data.getContent().getAgentID();
     int contactRoleID=AgentMap.getRoleIdByIndex(contactID);

     movementForSlave.reachObjectNoInt(new PositionSq(positionToReach));
     DataOutputManager.printString(id +" Other agent Reached");
     registration.doAction(Slave.agentReached(contactRoleID,new MessagePayload()));

     DataOutputManager.printString(id +" Collaborative Carry End");
    }
}
```

Figure 18. Fragment of code of the "Manager" role

of adaptive strategy is necessary. This pattern can be used when this specific task occurs, if only a specific group of robots (the ones necessary to carry the object) can adopt this pattern. Here, the known situation and environment (a robot has an heavy object), makes the application of the pattern the better solution.

However the exploitation of this approach, called "Self-expression" [21], is outside from the intent of this paper.

Using this pattern, the considerations made for the previous scenario are still valid. If the AM fails it has to be replaced by negotiation. This will further increases the time for goal's satisfaction.

The "P2P" pattern does not suffer from the limitation of having a single point of failure. But the mechanisms of information sharing about components, is useful only to make robots not to explore a cell twice, but not to find more objects. In the "P2P" pattern we developed the "Negotiator" role that permits a component to share its knowledge with neighbours in a range of 10 cells and to negotiate with them which cell has to be explored next.

The code of the "Negotiator" role, reported in Figure 19, is very similar to the code used in the previous scenario because the shared information are not about the goal's satisfaction, but about how to implement a utility (time of satisfaction).

Figure 17 - (D) shows the average time in simulations that use this pattern. The time is higher than the "Centralized AM" patter, because the exchange of information is higher.

5. Related Work

The interest in engineering complex distributed self-adaptive systems is growing more and more in the last years, as shown by the number of surveys and overviews on the topic [8, 24, 29]. However, a comprehensive and

```
public void moveNegotiator(RoleRegistration registration) throws RoleException
{
     BooleanNegotiator.set(ConfigurationParameters.END_CHECK_BASE_BOOL + String.valueOf(id),true);
     while(BooleanNegotiator.get(ConfigurationParameters.END_SEARCH_MOVEMENT_BOOL)==false &&
BooleanNegotiator.get(ConfigurationParameters.EXIT_AGENT_BASE_BOOL + String.valueOf(id))==false &&
(Negotiatior.carrying() == false))
          {
          sleepAgent(ConfigurationParameters.PAUSE-100);
          boolean modeChange=false;
          int Neighbour = NEIGHBOUR;
          for(int i=0;(i<Neighbour && modeChange==false);i++) {
               int myKnownSquares=KnownEnvironmentManager.getKnownEnvironments(id).getKnownNumber();
          modeChange=contactNeighbour.receiveModeChangeMessage(registration);
          saveNegotiatorData(myKnownSquares);
          }
          contactNeighbour.refreshExplorers(registration);
          MoveData moveData=searchUntilMove(NextNeighbour);
          for (i=0; i<Neighbour; i++)
          {
               if(moveData.getRange()==10 && moveData.getRouteSize()==0 && moveData.getExplored(Neighbour))
          {
                    BooleanNegotiator.set(ConfigurationParameters.END_CHECK_BASE_BOOL +
          String.valueOf(id),true);
                    break;
               }
          }
}
```

Figure 19. Fragment of code of the "Negotiator" role

rationally-organized analysis of architectural patterns for self-adaptation, and their use, is still missing, despite the potential advantages of such a contribution.

Going into details, some works are very relevant considering patterns for adaptive system. For example, Gomaa et al. [14] introduces the concept of "software adaptation patterns" and specifies how they can be used in software adaptation of service-oriented architectures. However, they do not give a complete overview of their use. Ramirez and Cheng [23] go somehow farther, by expanding upon the most common patterns for adaptation actions and representing them – the same as we do – in a more standard way. Again, though, their focus is different from that of a detailed and comprehensive analysis of architectural self-adaptation patterns and their use.

Grounded on earlier works on architectural self-adaptation approaches, the FORMS model (FOrmal Reference Model for Self-adaptation) [30] enables engineers to describe, study and evaluate alternative design choices for self-adaptive systems. FORMS defines a shared vocabulary of adaptive primitives that – while simple and concise – can be used to precisely define arbitrary complex self-adaptive systems, and can support engineers in expressing their design choices, there included those related to the architectural patterns for feedback loops. FORMS does not have the ambition to analyze and classify architectural self-adaptation patterns, and rather has to be considered as a potentially useful complement to our work. Closest to our approach, Weyns et al. [31] introduce the concept of patterns for self-adaptive systems based on control loops. The authors describe how control loops are able to enforce adaptivity in a system, and present a set of patterns, but it is not clear how to choose between them. Differently from our scope this paper does not aim to describe a comprehensive set of patterns for adaptive systems, but basically identifies in the patterns, how different MAPE loops interact with each other.

Regarding the use of roles, we have to underline that role models are used quite heavily in the RBAC (Role Based Access Control) approach (e.g. [25]), especially to extend MAS in agent oriented computing, but none of them is focused on the description of patterns using roles.

In summary, the analysis of the related work in the area shows that literature on adaptive systems is very rich: patterns are a good mechanism that can be adopted to promote and support self-adaptation, but are not too much investigated, especially there is not a clear idea on how to help developers to choose among patterns for their systems. After our taxonomy proposed in [22], experiments that support the use of patterns to build self-adaptive systems are still missing, and that is why we proceed with our work. From our knowledge, there are no works that use agents to simulate self-adaptive systems, exception of the use of AMAS proposed for example by [4] and [13].

6. Conclusions

As emerged from the simulations of the two different scenario of the robotics case study, all patterns end satisfying the goal and all the developed patterns make it possible to have a self-adaptive system. However, in

different conditions (due to the environment or to other components composing of the system) simulations show that some perform better than other. For example, in the first scenario the Reactive Stigmergy pattern has the worst performances because there are no coordination between components of the systems and they risk to spend a lot of time analysing part of the ground that have been already explored. The same pattern instead, has the best performances in the second scenario where it is not important to explore "all" the ground, but the time saved because of the absence of communication and coordination between components is important in find objects to rescue.

This implementation of different patterns on the same case study, gives us the possibility to add important specification to the initial taxonomy. As said before, in this paper we reported simulations and analysis only of the basic patterns, one for each level of adaptation - first column of the taxonomy table (see [18]). Moreover, we need to recall that, for each pattern, the addition of an external manager over a component (that can be a component or another AM) will add a level of autonomicity but the why adaptation is propagated inside the pattern remain the same for all the pattern of a specific level (same row).

As a future work, we aim at extending the simulations to other patterns with the goal of completing the taxonomy table and implementing guidelines to help developers to create self-adaptive systems.

7. Acknowledgements

Work supported by the ASCENS project EU FP7-FET, Contract No. 257414 and by the "Linea strategica SMART ICT FOR SMART SOCIAL WORLDS" of the Università di Modena e Reggio Emilia.

References

[1] Bäumer, D., Riehle, D., Siberski, W., and Wulf, M. (1998). The role object pattern. In *Washington University Dept. of Computer Science*. Citeseer.

[2] Bellifemine, F., Caire, G., Trucco, T., and Rimassa, G. (2002). Jade programmerï£¡s guide. *Jade version*, 3.

[3] Biddle, B. (1979). *Role theory: Concepts and research*. Krieger Pub Co.

[4] Bonabeau, E., Dorigo, M., and Theraulaz, G. (1999). *Swarm intelligence: from natural to artificial systems*. Oxford University Press, USA.

[5] Cabri, G. and Capodieci, N. (2013). Runtime Change of Collaboration Patterns in Autonomic Systems: Motivations and Perspectives. In *Proceedings of the Ninth International Symposium on Frontiers of Information Systems and Network Applications (FINA), Barcelona, Spain, March 2013*.

[6] Cabri, G., Leonardi, L., and Zambonelli, F. (2003a). Implementing role-based interactions for internet agents. In *Applications and the Internet, 2003. Proceedings. 2003 Symposium on*, pages 380–387. IEEE.

[7] Cabri, G., Leonardi, L., and Zambonelli, F. (2003b). Implementing Role-based Interactions for Internet Agents. In *The 2003 International Symposium on Applications and the Internet (SAINT), Best Paper Award Orlando,Florida,USA, January 2003*.

[8] Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., et al. (2009). Software engineering for self-adaptive systems: A research roadmap. In Cheng, B., Lemos, R. d., Inverardi, P., and Magee, J., editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer.

[9] Edwards, G. et al. (2009). Architecture-driven self-adaptation and self-management in robotics systems. In *Software Engineering for Adaptive and Self-Managing Systems, 2009*, pages 142–151, Vancouver, BC, Canada. IEEE.

[10] Ferber, J. (1999). *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading.

[11] Fernandez-Marquez, J. L., Serugendo, G. D. M., Snyder, P. L., and Valetto, G. (2012). A pattern-based architectural style for self-organizing software systems. *Drexel University, Department of Computer Science, Tech. Rep*, 6.

[12] Fowler, M. (1997). Dealing with roles. In *Proceedings of PLoP*, volume 97, Monticello, Illinois, USA.

[13] Georgé, J.-P., Peyruqueou, S., Régis, C., and Glize, P. (2009). Experiencing self-adaptive mas for real-time decision support systems. In Demazeau, Y., Pavón, J., Corchado, J., and Bajo, J., editors, *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009)*, volume 55 of *Advances in Intelligent and Soft Computing*, pages 302–309. Springer Berlin Heidelberg.

[14] Gomaa, H. and Hashimoto, K. (2012). Dynamic self-adaptation for distributed service-oriented transactions. In *Proceedings of the 2012 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 11–20, Zï£¡rich, Switzerland. IEEE Computer Society.

[15] Lange, D. B. and Mitsuru, O. (1998). *Programming and Deploying Java Mobile Agents Aglets*. Addison-Wesley Longman Publishing Co., Inc.

[16] Mayer, P., Klarl, A., Hennicker, R., Puviani, M., Tiezzi, F., Pugliese, R., Keznikl, J., and Bureš, T. (2013). The Autonomic Cloud: A Vision of Voluntary, Peer-2-Peer Cloud Computing. In *Proceedings of 3rd Workshop on Challenges for Achieving Self-Awareness in Autonomic Systems*, pages 1–6, Philadelphia, USA.

[17] Puviani, M. (2012a). Adaptive System's Configuration in a Swarm Robotics Scenario. *Awareness magazine*, page 3.

[18] Puviani, M. (2012b). Tr 4.2: Catalogue of architectural adaptation patterns. Technical report, ASCENS Project.

[19] Puviani, M., Cabri, G., and Frei, R. (2012a). Self-healing in Ensembles' Adaptive Collaborative Patterns. In *1st International Conference on Through-life Engineering Services (TESConf 2012)*, page 361ï£¡367, Shrivenham, UK.

[20] Puviani, M., Cabri, G., and Leonardi, L. (2012b). Adaptive Patterns for Intelligent Distributed Systems: a Swarm Robotics Case Study. In *Proceedings of the 6th International Symposium on Intelligent Distributed Computing - IDC 2012*, pages 241 – 246. Sringer.

[21] Puviani, M., Cabri, G., and Leonardi, L. (2014). Enabling self-expression: the use of roles to dynamically change adaptation patterns. In *Self-Adaptive and Self-Organizing Systems Workshops (FoCAS), Fifth IEEE Conference on*, Ann Arbor, Michigan, USA. IEEE Computer Society.

[22] Puviani, M., Cabri, G., and Zambonelli, F. (2013). A Taxonomy of Architectural Patterns for Self-adaptive Systems. In *Sixth International C* Conference on Computer Science & Software Engineering*, pages 77–85, Porto (P). ACM.

[23] Ramirez, A. and Cheng, B. (2010). Design patterns for developing dynamically adaptive systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 49–58, Cape Town, South Africa. ACM.

[24] Salehie, M. and Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14.

[25] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *Computer*, 29(2):38–47.

[26] Serbedzija, N., Massink, M., Brambilla, M., Latella, D., Dorigo, M., and Birattari, M. (2012). Ensemble model syntheses with robot, cloud computing and e-mobility. *ASCENS Deliverable D*, 7.

[27] Weiser, M. (1991). The computer for the 21st century. *Scientific american*, 265(3):94–104.

[28] Weyns, D. and Holvoet, T. (2007). An architectural strategy for self-adapting systems. In *Software Engineering for Adaptive and Self-Managing Systems, 2007*, page 3, Minnesota, USA. IEEE Computer Society.

[29] Weyns, D., Iftikhar, M., Malek, S., and Andersson, J. (2012a). Claims and supporting evidence for self-adaptive systems: A literature study. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 89 –98.

[30] Weyns, D., Malek, S., and Andersson, J. (2012b). Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems*, 7(1):8.

[31] Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., and Göschka, K. (2012c). On patterns for decentralized control in self-adaptive systems. *Software Engineering for Self-Adaptive Systems II*, pages 76–107.

[32] Zambonelli, F., Jennings, N., and Wooldridge, M. (2001). Organisational rules as an abstraction for the analysis and design of multi-agent systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(03):303–328.