Agent-based Simulations of Patterns for Self-adaptive Systems

Mariachiara Puviani[1], Giacomo Cabri[2] and 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*

Keywords: Adaptation Pattern, Taxonomy, MAS.

Abstract: Self-adaptive systems are distributed computing systems composed of different components that can adapt their behavior 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, detailing how it was carried out, and to present a "table of applicability" that completes the initial taxonomy of patterns and provides a further support for the developers.

1 INTRODUCTION

A complex distributed self-adaptive system consists of multiple components that are deployed on multiple nodes connected via some network (Weyns et al., 2013), whose administration must be automated, to avoid human manual management that would introduce cost, duration, slowness, and error-proneness. Self-adaptive systems can also optimize their performance under changing operating conditions.

In order to build self-adaptive systems, developers can choose a specific adaptation pattern among available ones. In a previous work we have cataloged them (Puviani, 2012b), 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.

Starting from the catalogue of adaptation patterns, we wrote a taxonomy table (Puviani et al., 2013) 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 behaviour of different patterns. The simulations allow us to define a "table of applicability" that will complete the initial taxonomy of 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 (Ferber, 1999) in order to simulate complex self-adaptive systems. This is because, as said before, agents exhibit features very relevant for self-adaptation.

Moreover, the chosen mechanism used to implement adaptation patterns is the "role based approach" (Cabri and Capodieci, 2013). Roles are a set of behaviours 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. This 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. Section 2 discusses related work in the area. In Section 3 we present the taxonomy of adaptation patterns and show some examples; while in Section 4 we present the "role based approach" and how agents are used o develop patterns. Section 5 presents a case study and simulations implemented using RoleSystem. Section 6 shows how these simulations help us to enrich the initial taxonomy and validate the use of adaptation patterns in developing self-adaptive systems. Section 6 discusses the results of the work and concludes the paper.

2 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 (Cheng et al., 2009; Salehie and Tahvildari, 2009; Weyns et al., 2012a). However, a comprehensive and 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 systems. For example, Gomaa et al. (Gomaa and Hashimoto, 2012) introduce 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.

Grounded on earlier works on architectural self-adaptation approaches, the FORMS model (FOrmal Reference Model for Self-adaptation) (Weyns et al., 2012b) 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. Closest to our approach, Weyns et al. (Weyns et al., 2012c) 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 chose between them.

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 (Puviani et al., 2013), 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 is no work that uses agents to simulate self-adaptive systems, exception of the use of AMAS proposed for example by (Bonabeau et al., 1999) and (Georgé et al., 2009).

3 USE OF PATTERNS

Because patterns are able to describe generic solutions for a recurring design 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 is covered by a specific pattern, we created a preliminary taxonomy presented in Figure 1 (Puviani et al., 2013).

This table describes the different patterns arranged 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 – fourth row.

How these patterns are useful to build self-adaptive systems has been demonstrated in different works like (Puviani, 2012a), (Puviani et al., 2012a), (Puviani et al., 2012b) and (Mayer et al., 2013).

With the aid of this taxonomy table and the catalogue of pattern presented in (Puviani, 2012b), we are able to understand which pattern can describe the desired features of 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 analysing, with the aid of simulations, the use of adaptation patterns. So, we started analysing the "basic" pattern of each ensemble level (the single level is delegated to the internal of each agent)[1].

[1]For space reasons we report only the studies on the first column of patterns in the taxonomy table

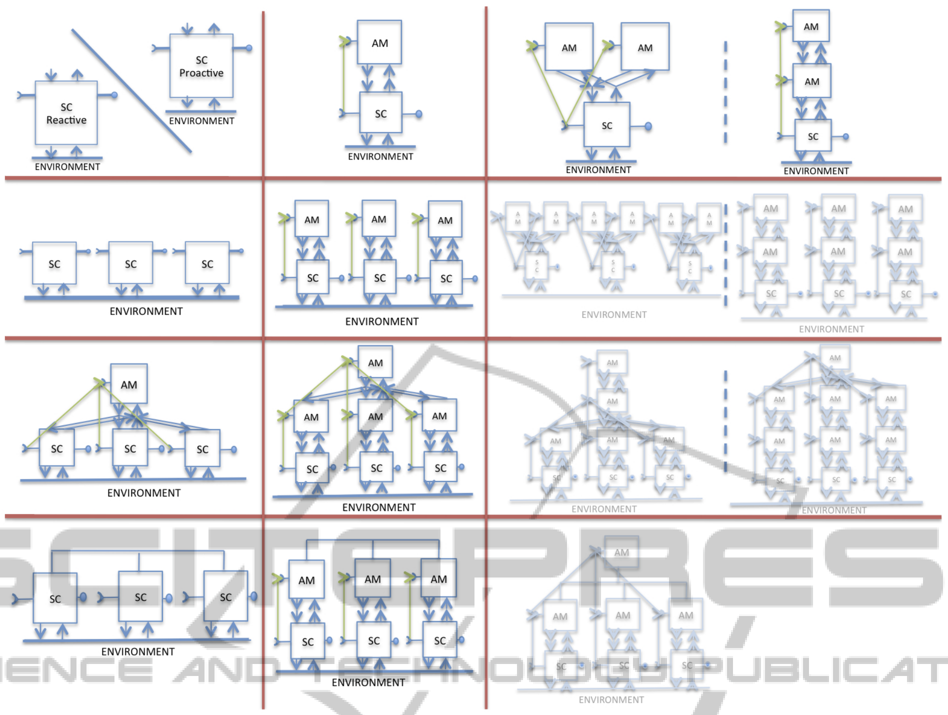


Figure 1: Taxonomy of self-adaptation patterns based on different composition mechanisms.

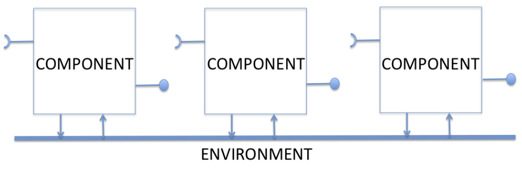


Figure 2: Reactive Stigmergy SCE Pattern.

First of all we present in the next subsections the studied patterns. For space limitations in the following we only report the "context" of the system that will apply that pattern, and its main "behaviour".

3.1 Reactive Stigmergy Service Components Ensemble Pattern

Context. This patterns has to be adopted when:

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

Behaviour. 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 behaviour due to these changes.

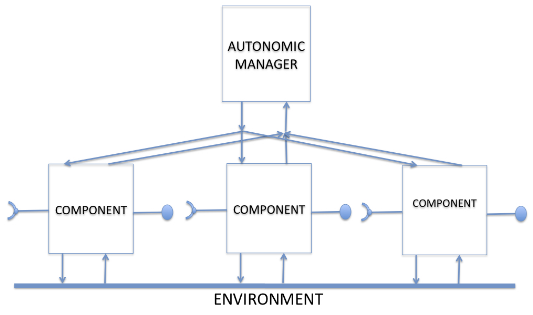


Figure 3: Centralised AM SCE Pattern.

3.2 Centralised 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 centralised feedback loop is more suitable because a single AM has a global vision on the system;
4. there are few components composing the ensemble.

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

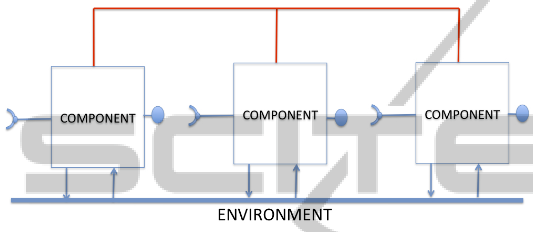


Figure 4: P2P Negotiation SCE Pattern.

3.3 P2P Negotiation Service Components Ensemble 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.

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

4 AGENTS AND THE ROLE BASED APPROACH

To evaluate patterns, we have implemented them using roles.

The "role theory" (Biddle, 1979) 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 (Fowler, 1997). As the author said, role is defined as "a set of behaviours common to different entities, with the possibility to apply them to an entity in order to change its capabilities and behaviour".

Roles are very important not only because can be applied to existing entities to change their behaviour, 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 (Bäumer et al., 1998), 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 role are deserving 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 components 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 modeled as a set of capabilities and an expected behavior, both related to the component (i.e. agent) that plays such role.

Based on the concept of role, we exploit RoleSystem (Cabri et al., 2003). 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 (Bellifemine et al., 2002), a FIPA compliant agent platform.

The RoleSystem infrastructure is divided into 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 layer of agents by exchanging ACL messages formatted in an appropriate way.

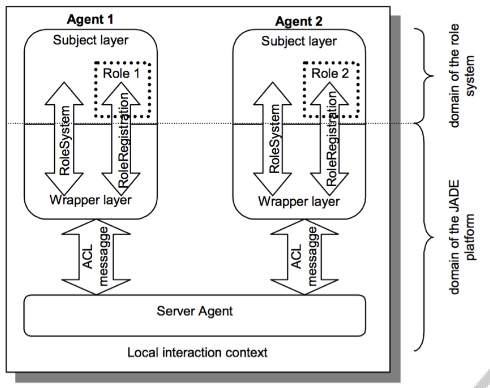


Figure 5: RoleSystem: separation of domain.

Every agent is a subject that perform some actions and on which some event can happened. 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.

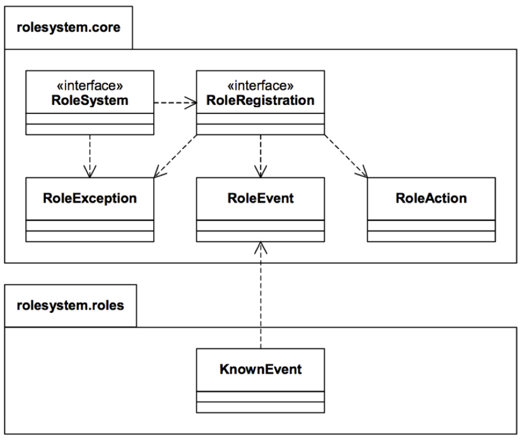


Figure 6: RoleSystem packages.

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 5).

5 CASE STUDY

Since this work has been carried on in the framework of the ASCENS project[2], we use one of the case study presented in the project to develop RoleSystem

[2]http://ascens-ist.eu

classes and to implement different simulations.

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

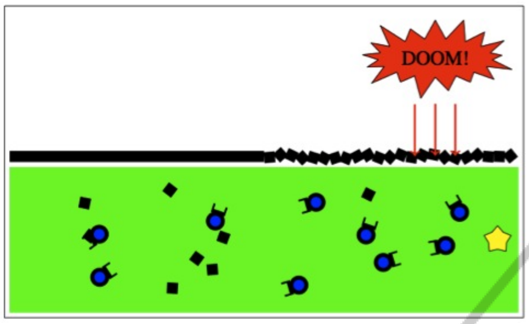


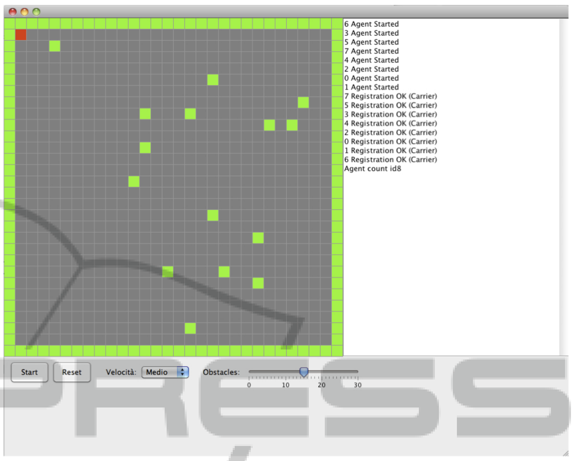
Figure 7: Disaster recovery scenario - ASCENS case study.

As reported in (Serbedzija et al., 2012), 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 analyse the *environment exploration* task, to understand how different patterns behave in order to build such a system.

5.1 Environment Exploration 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 a room (e.g. each position is randomly chosen once a robot is entered in the room). 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 size of the environment.

To implement the simulations of this scenario we have defined a room 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 cannot be removed by a robot. Due to the size of the room we initially implemented 16 robots moving in (they are a right amount compared to the size of the room, in order to be a swarm). In Figure 8 we can see a screenshot of the environment.



```
6 Agent Started
3 Agent Started
5 Agent Started
7 Agent Started
4 Agent Started
2 Agent Started
0 Agent Started
1 Agent Started
7 Registration OK (Carrier)
5 Registration OK (Carrier)
3 Registration OK (Carrier)
4 Registration OK (Carrier)
2 Registration OK (Carrier)
0 Registration OK (Carrier)
1 Registration OK (Carrier)
6 Registration OK (Carrier)
Agent count id8
```

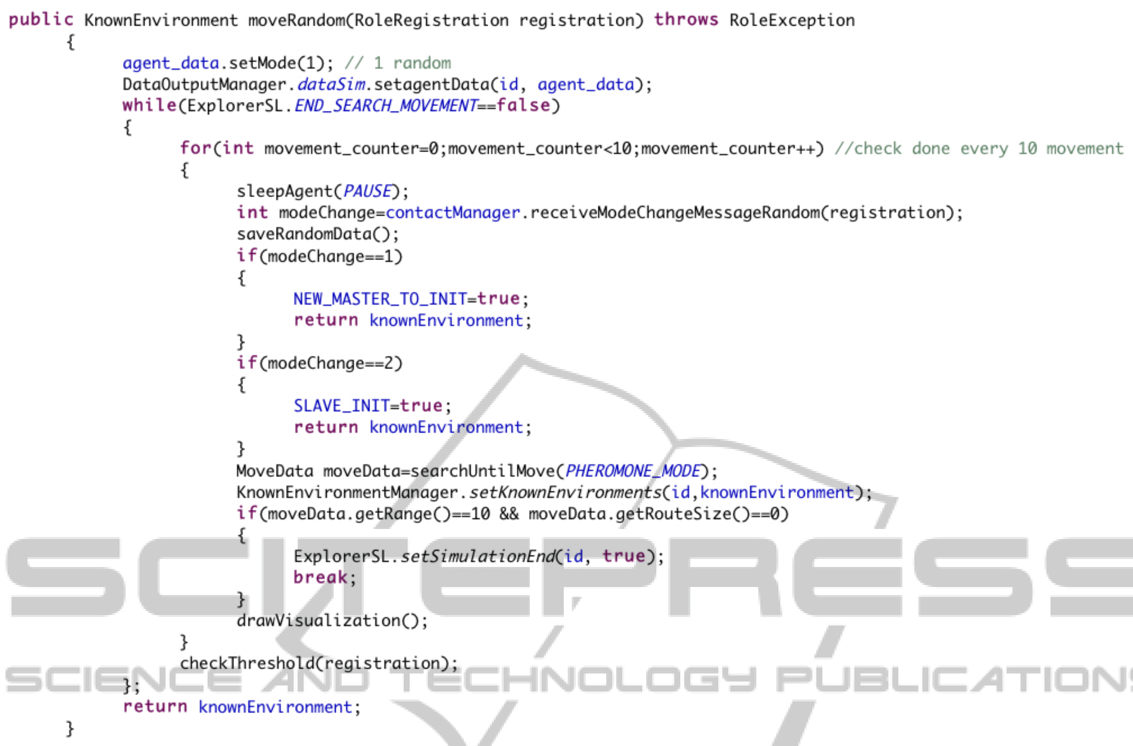
Figure 8: Environment of the “environment exploration scenario”.

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.

We performed hundreds of simulations using this pattern and in Figure 10 we show the average time of the goal’s satisfaction (A), using 16 robots, and the standard deviation. As we can see, the average time is about 418000 millisec. In these simulations every robot colors the pavement of the room (while it leaves the pheromone) of a different colour. However, at the end of these simulations, only few colors appear on the screen. This is because it happens that most of the time a robot explores again a cell just explored by



```
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: Code of the “Explorer” role.

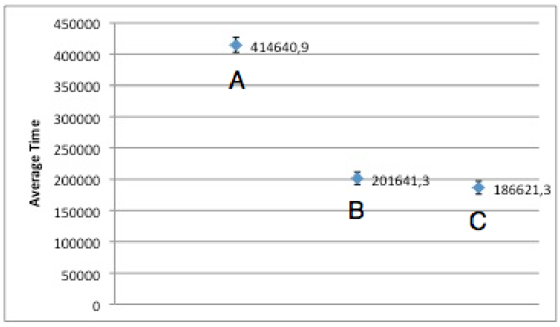


Figure 10: Cross-section of 10 simulations that uses the Reactive Stigmergy pattern.

another robot. This is a drawback of patterns based on stigmergy.

To overcome this, we implemented the same system using also the the “Centralised AM” pattern and the “P2P” pattern.

For the “Centralised AM” pattern we implemented the “Manager” and the “Slave” roles. An example of code of the “Manager” role is reported in Figure 11.

The manager, communicating with all the robots, receives the information about their positions and the explored area. Then it merges all the received information to create a map of the explored area and it shares this map with 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 12.

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 find 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, adopting this pattern the average time of the goal’s satisfaction (B) is better than the same system developed using the “Reactive Stigmergy” pattern; nevertheless, it remains high (it was compared with expected results studied in previous works). This is due to the need for 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

```
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 11: Example of the “Manager” role.

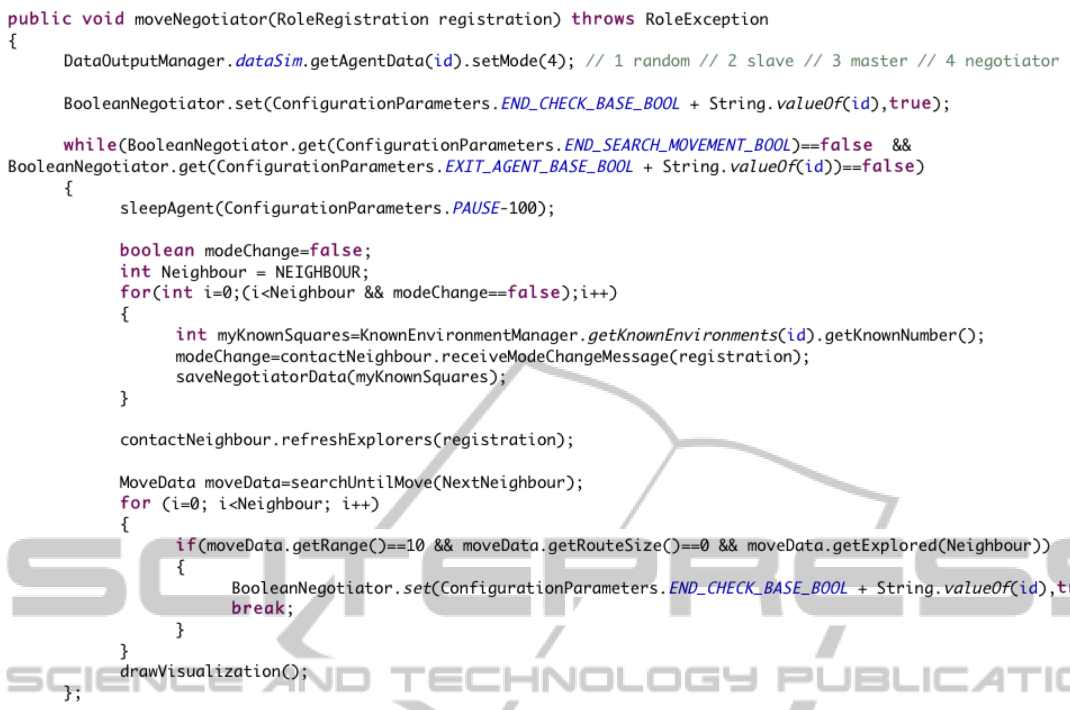
```
public void moveSlave(RoleRegistration registration) throws RoleException
    {
        agent_data.setMode(2); // 1 random // 2 slave // 3 master
        DataOutputManager.dataSim.setagentData(id, agent_data);
        while(ExplorerSL.END_SEARCH_MOVEMENT==false)
        {
            for(int movement_counter=0;movement_counter<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 12: Example of the “Slave” role.

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 one robot may become AM. Other solutions uses negotiation algorithms.

Instead, the “P2P” pattern does not suffer from the limitation of having a single point of failure. Furthermore, more than 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



```
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 13: Example of the “Negotiator” role.

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 13.

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

In Figure 14 we represent the comparison of the use of the different patterns in term of time of goal satisfaction. We can see that the “P2P” pattern has on average the best performances related to the use of the “Centralised AM” pattern without and with failure of the AM, and related to the “Reactive Stigmergy” pattern where the time is 120% more that the “P2P” one.

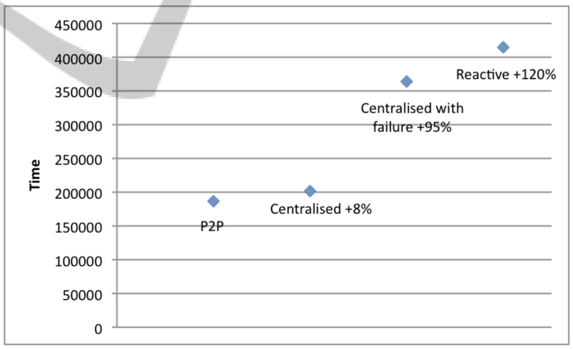
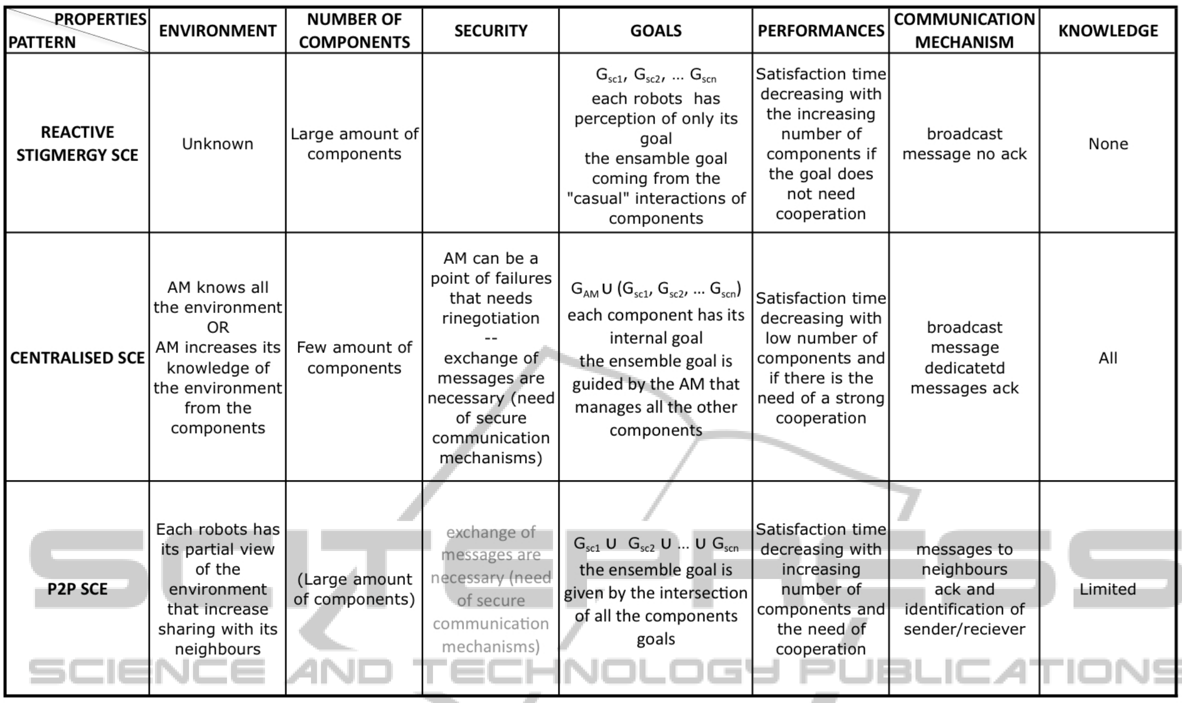


Figure 14: Comparison of the average time of goal satisfaction.

6 CONCLUSIONS

As emerged from the simulations, where all patterns end satisfying the goal, 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 of the system) simulations show that some perform better than others.

This implementation of different patterns on the same case study, gives us the possibility to add important specification to the initial taxonomy. The analysis of the different simulations reported above, permits us to write a “table of applicability” that describes patterns and their performances (see Figure 15). As said before, due to space limitations, 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 (Puviani, 2012b)). 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 way adaptation is propagated inside



PROPERTIES / PATTERN	ENVIRONMENT	NUMBER OF COMPONENTS	SECURITY	GOALS	PERFORMANCES	COMMUNICATION MECHANISM	KNOWLEDGE
REACTIVE STIGMERGY SCE	Unknown	Large amount of components		G_{sc1}, G_{sc2}, … G_{scn} each robots has perception of only its goal the ensamble goal coming from the "casual" interactions of components	Satisfaction time decreasing with the increasing number of components if the goal does not need cooperation	broadcast message no ack	None
CENTRALISED SCE	AM knows all the environment OR AM increases its knowledge of the environment from the components	Few amount of components	AM can be a point of failures that needs rinegotiation -- exchange of messages are necessary (need of secure communication mechanisms)	$G_{AM} \cup (G_{sc1}, G_{sc2}, … G_{scn})$ each component has its internal goal the ensemble goal is guided by the AM that manages all the other components	Satisfaction time decreasing with low number of components and if there is the need of a strong cooperation	broadcast message dedicatetd messages ack	All
P2P SCE	Each robots has its partial view of the environment that increase sharing with its neighbours	(Large amount of components)	exchange of messages are necessary (need of secure communication mechanisms)	$G_{sc1} \cup G_{sc2} \cup … \cup G_{scn}$ the ensemble goal is given by the intersection of all the components goals	Satisfaction time decreasing with increasing number of components and the need of cooperation	messages to neighbours ack and identification of sender/reciever	Limited

Figure 15: "Table of applicability" of patterns.

the pattern remains the same for all the patterns of a specific level (same row). We aim at extending the simulations to other patterns in order to complete the "table of applicability".

In the table we can see which pattern better creates a self-adaptive system under certain conditions, and which features a pattern shows. For example, if we would like to develop a system where the environment is perfectly known from at least one component, the "Centralised AM" pattern will perform better than the "Reactive Stigmergy" SCE pattern. Otherwise, if in our system we have no mechanisms for a direct communication for adaptation between components, the "Reactive Stigmergy" SCE pattern has to be preferred.

Agents enabled us to simulate patterns for complex self-adaptive systems, and with the aid of adaptation patterns, users are able to develop the most performative self-adaptive system, based on different system conditions.

On this way future work can concern implementing simulations in different scenarios that will confirm and complete our "table of applicability".

ACKNOWLEDGEMENTS

Work supported by the ASCENS project EU FP7-FET, Contract No. 257414.

REFERENCES

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

Bellifemine, F., Caire, G., Trucco, T., and Rimassa, G. (2002). Jade programmers guide. *Jade version*, 3.

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

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

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*.

Cabri, G., Leonardi, L., and Zambonelli, F. (2003). 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*.

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.

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

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

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.

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, Zrich, Switzerland. IEEE Computer Society.

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.

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

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

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 361367, Shrivenham, UK.

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.

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.

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

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.

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.

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.

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.

Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., and Göschka, K. M. (2013). On patterns for decentralized control in self-adaptive systems. In *Software Engineering for Self-Adaptive Systems II*, pages 76–107. Springer.