Building Self-Adaptive Systems by Adaptation Patterns Integrated into Agent Methodologies

Mariachiara Puviani*, Giacomo Cabri, Nicola Capodieci, and Letizia Leonardi

Università di Modena e Reggio Emilia,
Modena, Italy
{mariachiara.puviani,giacomo.cabri,nicola.capodieci,letizia.leonardi}@unimore.it
http://www.unimore.it

Abstract. Adopting patterns, i.e. reusable solutions to generic problems, turns out to be useful to rely on tested solutions and to avoid reinventing the wheel. To this aim, we proposed to use adaptation patterns to build systems that exhibit self-adaptive features. However, these patterns would be more usable if integrated in a methodology exploited to develop a system. In this paper we show how our Catalogue of adaptation patterns can be integrated into methodologies for adaptive systems; more in detail, we consider methodologies which support the development of multi-agent systems that can be considered good examples of adaptive systems. The paper, in particular, shows the integration of our Catalogue of adaptive patterns into the PASSI methodology, together with the graphical tool that we developed to support it.

Keywords: Multi-Agent System, adaptation pattern, methodology.

1 Introduction

Intelligent software systems are playing an important role in many fields, but they are becoming more and more complex, requiring appropriate approaches for their development and maintenance. In particular, their complexity and the fact that they often cannot be stopped, introduce the need for some form of adaptation to the changes in the surrounding environment or in general in the execution conditions. So, self-adaptation is more and more a required feature of the complex intelligent systems, and must be carefully taken into consideration during the development, from a software engineering point of view, becoming one of the challenges for the discipline [14].

We define *Self-adaptation* as the ability of a software system or an application to automatically modify its structure and behaviour at runtime in order to ensure, maintain or recover some functional or non-functional properties, even in the case of unexpected changes to operating conditions or user requirements.

* The work is supported by the "Linea strategica SMART ICT FOR SMART SOCIAL WORLDS" of the Università di Modena e Reggio Emilia.

In the literature, we can find two approaches to develop self-adaptive systems: *parameter* adaptation and *compositional* adaptation [21]. Parameter adaptation means adapting the system's behaviour through changing parameters, while compositional adaptation is meant as a change of components (in terms of behaviour or whole structure). In our work, we focus on *compositional* adaptation, but in a more specific way: we do not aim to simply change the components in a system, but we aim to modify their behaviour (defined as the *pattern* that describes it) inside the system. This leads to conceive the adaptation of a system as the capability of changing its internal structure in order to make it behave differently, not only as the possibility of changing the system's components.

Anyway, there is a *lack* of support for designing and implementing self-adaptive systems, in terms of reusable and well-defined *components* and how composing them, so designers often start from scratch the development of a self-adaptive system. From this point of view, the availability of adaptation patterns is considered as useful means to introduce adaptation into a system. Further, the integration of adaptation patterns into a more general framework not only will help suggesting developers how to include adaptivity in their systems, but also this would take advantage of the methodologies and the tools exploited in the framework, leading to a fast and less error-prone development.

The general aim of our work is to propose a comprehensive approach that will guide developers during the development phases, from the system's specification to the system's implementation, in a complete framework for developing self-adaptive systems. To this purpose, we have defined four general steps for our work:

1. analysis of existing methodologies, choice of few of them and integration of our Catalogue of adaptation patterns into the chosen ones;
2. modification of the tools that support the chosen methodologies, for the creation of adaptive systems;
3. creation of a middleware that will merge the concepts coming from methodologies' tools, by means of Java classes;
4. evaluation of the framework, experimenting the creation of self-adaptive systems.

In a previous work [27], we have presented a preliminary result of the first step. In this paper, we extend the presentation of the results of the first step and add some results of the second and of the third steps, in particular related to a specific methodology.

With regard to the first step, we will show how our Catalogue of adaptation patterns can be integrated, in particult, into one existing methodology, PASSI; we will exploit the SPEM notation [34] to specify when and how our Catalogue can be considered in the development process proposed by a specific methodology.

With regard to the second step, we will present the graphical tool we have developed in order to support the previously mentioned integration.

With regard to the third step, we will show how the graphical tool produces a set of Java classes.

The reminder of this paper is as follows: in Section 2, we present the Catalogue of adaptation patterns, explaining its importance in connection with adaptive systems and methodologies. Further on, in Section 3 we introduce agent-oriented methodologies, along with the criteria we used to select a few of them, and we show how and where our Catalogue of adaptation patterns can be included into one of the chosen methodology, PASSI (Section 4); moreover, we present the developed graphical tool to exploit adaptation patterns in the PASSI methodology and to produce the needed Java classes. In Section 5, we present some work related to our approach and at the end, in Section 6 we conclude the paper and present some future works.

2 The Catalogue of Adaptation Patterns

Closed to self-adaptation are *Service-based systems* and *Agent-based systems*.

In literature, design patterns (or simply *patterns*) are defined as reusable solutions to recurring design problems and are a mainstream of software reuse practice [15, 22]. They crystallize a general solution to a common problem, so software developers can benefit from their reuse to develop systems. An *adaptation pattern* is a conceptual scheme that describes a specific adaptation mechanism. It specifies how the component/system architecture can express adaptivity.

An important task to develop a well performing self-adaptive system, is to understand which pattern to choose. In order to define how a pattern works in a self-adaptive system and which kind of systems is covered by a specific pattern, we wrote a Catalogue of adaptation patterns [28]. In this Catalogue, the different patterns are presented, and each of them describes the features of a specific adaptive system.

The adaptive behaviour inside a component or an ensemble is described in terms of feedback loops. In the Catalogue, the patterns are proposed with a specific description by means of a template, and with examples of use of the patterns in real systems, in order to simplify the selection of the right pattern to use. The use of a pattern permits the developer to be guided to make the system exhibit the required behaviour, even when unexpected situations occur.

The use of adaptation patterns to create self-adaptive systems has been tested in different fields and in many applications [20, 29], and guarantees correct results in systems that are frequently changing, not only in their internal conditions, but also in the environment where they are operating.

The use of a methodology could be very useful to develop self-adaptive systems. However, the current methodologies consider adaptation only at level of single components, instead of at the system's level. That is the reason why we consider necessary to introduce our Catalogue of adaptation patterns into methodologies: in fact, this will enact adaptivity at the level both of single component and of the entire system. By this integration, our Catalogue of adaptation patterns will support the methodologies in the creation of an adaptive system where the structural adaptation of the whole system is considered very relevant.

3 Agent-Oriented Methodologies for Adaptive Systems

In order to support application developers during the creation of an adaptive system, it is necessary to provide a methodology that support adaptation mechanisms starting from the system requirements. The initial idea is not to propose a methodology from scratch, but to have as a base a stable and well known methodology.

Moreover, we consider that "agents" are one of the most useful paradigms to build intelligence distributed systems, so we would like to use that paradigm to create adaptive systems. To do that, we started from the study of agent-oriented methodologies as a starting point to introduce adaptation features while building a system.

Considering MASs (Multi Agent Systems), it is generally accepted that analysis and design of agent-based systems require an Agent-Oriented Software Engineering (AOSE) methodology. There are now many mature AOSE methodologies [19], [4], including MaSE [13], Tropos [6], Gaia [36], Prometheus [24], INGENIAS [25], ADEM[1], ADELFE [5], SODA [1] and PASSI [11].

After different studies [32], we found out that to create a unified methodology that may have the most powerful features of every of the starting ones, is very difficult. For example, we are not able to prove if a new unified methodology covers all the possible scenarios, as happened for MAR&A [7], that is a composed methodology, but is not applicable to adaptive systems. Moreover, not all the composing methodologies use the same language or the same concepts, and translating them into unified terms will not be always easy. Furthermore, creating a new methodology for adaptive systems from scratch will not be easy as well. It may be yet another methodology, and there is no guarantee that it will be able to build all the adaptive systems.

All these reasons suggested us to not create a methodology dedicated to self-adaptive system, or to compose a methodology, but to start from well known and well defined methodologies, and to insert our Catalogue of adaptation patterns into them in order to have self-adaptive features.

Starting from our previous work [30], we found out that some AOSE methodologies, even if they are well known, are not up to date, or no more utilised to develop intelligent complex agents systems. So we selected only few methodologies that we consider suitable to build a self-adaptive system. The methodologies that we selected have some common features:

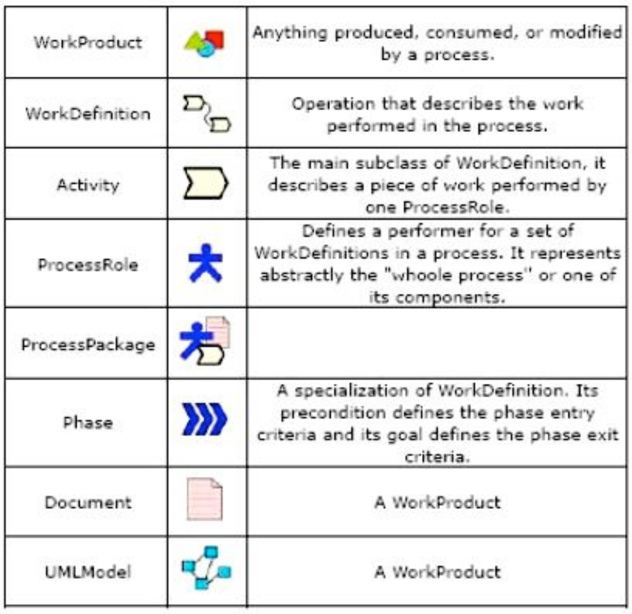
- they are updated (e.g. a new version of the methodology has been released in the last years);
- they have been tested in different distributed systems;
- they use well known paradigms like UML and the SPEM approach [34, 31] that will be very useful in order to introduce adaptation patterns;
- they use the concept of "role" to define adaptation patterns in a system;
- they have a supporting tool, or specific indication for the development of a system.

[1] http://www.whitestein.com/adem

The methodologies we selected for our work are: ADELFE, PASSI2 and SODA. For space reasons, in this paper we describe only PASSI2, along with explaining where introducing our Catalogue of adaptation patterns.

As said before, an common point of these methodologies is that all of them have been described using the SPEM (Software Process Engineering Meta-model) approach [23]. This will be useful in order to insert our Catalogue of adaptation patterns in terms of SPEM fragments. In this way, it will be possible to better define the concepts presented in patterns and to insert them into the different methodologies.

To improve reading of the paper, in Fig. 1 we report the definitions of some common notations used by SPEM.



WorkProduct		Anything produced, consumed, or modified by a process.
WorkDefinition		Operation that describes the work performed in the process.
Activity		The main subclass of WorkDefinition, it describes a piece of work performed by one ProcessRole.
ProcessRole		Defines a performer for a set of WorkDefinitions in a process. It represents abstractly the "whoole process" or one of its components.
ProcessPackage		
Phase		A specialization of WorkDefinition. Its precondition defines the phase entry criteria and its goal defines the phase exit criteria.
Document		A WorkProduct
UMLModel		A WorkProduct

Fig. 1. SPEM notations

4 Integrating Catalogue of adaptation patterns into Methodologies

In this section we present how to integrate our Catalogue of the adaptation patterns, in particular, in the PASSI methodology. Moreover, the last subsection sketches the interfaces of the graphical tool we developed to support the exploitation of the patterns in that methodology.

4.1 Integration in PASSI2

PASSI2 (Process for Agent Society Specification and Implementation) [12] is the evolution of PASSI [11], a methodology that aims at covering all the phases of a system development from the requirements' analysis to the deployment configuration, coding, and testing.

It is based on a meta-model describing the elements that constitute the system to be designed (agents, tasks, communications, roles) and what are the relationships among them. The importance of this description is in the lack of a universally accepted meta-model of MASs (differently from object-oriented systems) that makes unclear any agent design process that does not precisely define the structure of the system it aims to produce. PASSI2 has been designed keeping in mind the possibility of designing systems with the following peculiarities: (i) highly distributed, (ii) subject to a (relatively) low rate of requirements changes, (iii) openness (at runtime external systems and agents that are unknown at design time will interact with the system to be built). Robotics, workflow management, and information systems are the specific application areas where it has been wildly applied.

PASSI2 is composed of three models that address different design concerns and several phases, as we can see in Fig. 2. An important aspect of PASSI2 is that it uses standards as UML and adapts it to the need of representing agent systems through its extension mechanisms (constraints, tagged values and stereotypes).

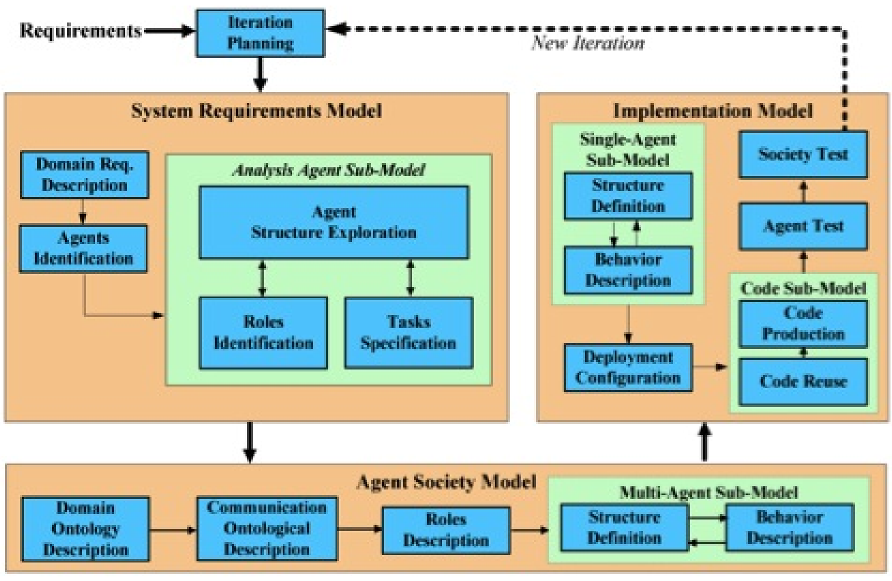


Fig. 2. PASSI2 models and phases

Synthetically, the models and phases of PASSI2 are:

1. System Requirements Model. A model of the system requirements in terms of agency and purpose. Developing this model involves:
   - Domain req. Description (DD). A functional description of the system using conventional use-case diagrams.
   - Agents Identification (AId). Separation of responsibility concerns into agents, represented as UML packages.
   - Roles Identification (RId). Use of sequence diagrams to represent each agent's responsibilities through role-specific scenarios.
   - Agent Structure Exploration (ASE). An analysis-level description of the agent structure in terms of tasks required for accomplishing the agent's functionalities.
   - Tasks Specification (TSp). Specification through state/activity diagrams of the capabilities of each agent.
2. Agent Society Model. A model of the social interactions and dependencies among the agents involved in the solution. Developing this model involves five phases:
   - Domain Ontology Description (DOD). Use of class diagrams to describe domain categories (concepts), actions that could affect their state and propositions about values of categories.
   - Communication Ontology Description (COD). Use of class diagrams to describe agents' communications in terms of referred ontology, interaction protocol and message content language.
   - Roles Description (RD). Use of class diagrams to show distinct roles played by agents, the tasks involved what the roles involve, communication capabilities and inter-agent dependencies in terms of services.
   - Multi-Agent Structure Definition (MASD). Use of conventional class diagrams to describe the structure of solution agent classes at the social level of abstraction.
   - Multi-Agent Behavior Description (MABD). Use of activity diagrams or state-charts to describe the behaviour of individual agents at the social level of abstraction.
3. Implementation Model. A model of the solution architecture in terms of classes, methods, deployment configuration, code and testing directives; it is composed of seven phases, the first two are performed at both the multi-agent (whole agent society) and single-agent abstraction level:
   - Single-Agent Structure Definition (SASD). Use of conventional class diagrams to describe the structure of solution agent classes at the implementation level of abstraction.
   - Single-Agent Behavior Description (SABD). Use of activity diagrams or state-charts to describe the behaviour of individual agents at the implementation level of abstraction.
   - Deployment Configuration (DC). Use of deployment diagrams to describe the allocation of agents to the available processing units and any constraints on migration, mobility and configuration of hosts and agent-running platforms.

- Code Reuse (CR). A library of patterns with associated reusable code to allow the automatic generation of significant portions of code.
- Code Production (CP). Source code of the target system that is manually completed.
- Agent Test. Verification of the single behaviour with regards to the original requirements of the system solved by the specific agent.
- Society Test. Validation of the correct interaction of the agents, performed in order to verify that they actually concur in solving problems that need cooperation. This test is done in the most real situation that can be simulated in the development environment.

The Iteration Planning phase is positioned at a higher level of abstraction, above the logical sequence of models and phases. It is at the base of every iterative incremental process and in our case consists of the analysis of the Problem Statement and all the other available documents (for instance outputs of previous iterations) in order to identify the requirements (and related risks) that should be faced in the next iteration (that is considered as the nineteen phase).

An important concept in PASSI2 is that of "role". A *role* is defined by the set of responsibilities defining the subjective behaviour of an agent in an interaction (conversation) with another one or in providing some service in one or more scenarios; an agent may play one or more roles at the same time. Roles are very important because they are considered a useful paradigm that can used to define the different patterns in a system [26].

Two are the main phases that involved roles: the *Role Identification phase*, into the System Requirements Model (Fig. 3), and the *Role Description phase* into the Agent Society Model (Fig. 4).

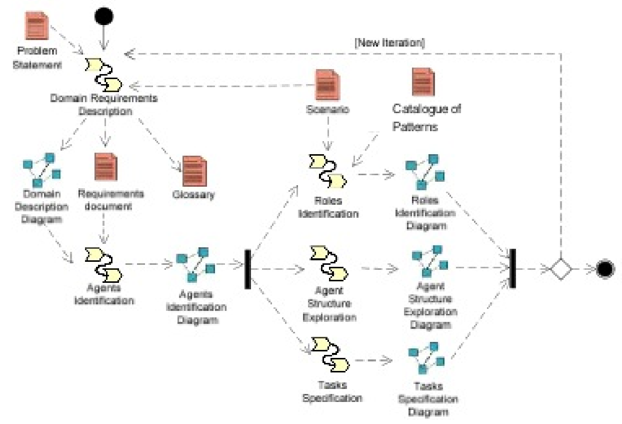

Problem Statement
[New Iteration]
Domain Requirements Description
Scenario
Catalogue of Patterns
Domain Description Diagram
Requirements document
Glossary
Roles Identification
Roles Identification Diagram
Agents Identification
Agents Identification Diagram
Agent Structure Exploration
Agent Structure Exploration Diagram
Tasks Specification
Tasks Specification Diagram

Fig. 3. PASSI2: System Requirements Model activities and resulting work products

The *Roles Identification* phase produces a set of sequence diagrams that specify scenarios from the agents' identification use case diagram. In this phase,

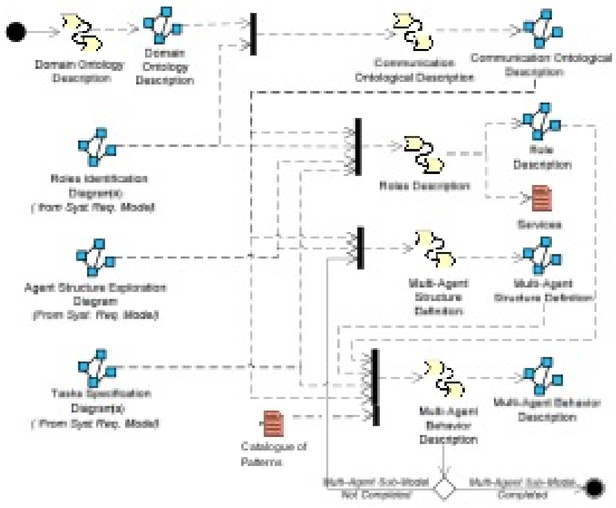

Domain Ontology Description
Domain Ontology Description
Communication Ontological Description
Communication Ontological Description
Roles Identification Diagram(s) (from Syst. Req. Model)
Roles Description
Role Description
Services
Agent Structure Exploration Diagram (From Syst. Req. Model)
Multi-Agent Structure Definition
Multi-Agent Structure Definition
Tasks Specification Diagram(s) (From Syst. Req. Model)
Catalogue of Patterns
Multi-Agent Behavior Description
Multi-Agent Behavior Description
Multi-Agent Sub-Model Not Completed
Multi-Agent Sub-Model Completed

Fig. 4. PASSI2: Agent Society Model activities and resulting work products

our Catalogue of adaptation patterns is added as input, in order to create specific roles able to describe an adaptive system. In that phase, roles are identified in the sense that agents' external manifestations are captured in sequence diagrams where agents participate playing one or more roles concurring to the evolution of the system dynamic.

Our Catalogue of adaptation patterns is also introduced in the *Roles Description* phase. This phase consists in modelling the lifecycle of each agent, looking at the roles it can play, the collaboration it needs, the communications in which it participates and, with the inclusion of the Catalogue of adaptation patterns, the adaptive system to develop. In the RD diagram all the rules of the society, laws of the society and the domain in which the agent operates are introduced. They could be expressed in plain text or OCL (Object Constraint Language) in order to have a more precise, formal description.

In PASSI2, the defined RD diagram is a class diagram where roles are classes grouped in packages representing agents. Roles can be connected by relationships representing changes of role, by dependencies for a service or the availability of a resource and by communications.

Specifically, in the *Agent Society Model*, the Catalogue of adaptation patterns is introduced for the Multi-Agent Behaviour Description (MABD), where agents are described in terms of their behaviour both from the social-exterior point of view and the internal flow of control, as we can see in Fig. 4. Here the Catalogue is necessary to identify which role to choose to obtain the system adaptation in the considered environment.

PASSI2 does not have a real Code Production Phase, but each programmer has to complete the code of the application starting from the design of the skeletons produced by the methodology.

The PASSI2 design methodology is supported by a specific design tool, granting a large number of automatisms during the design, and a pattern repository for the reuse practice; these are determinant in cutting down the time and cost for developing systems [10]. The toolkit is PTK (PASSI ToolKit). The PTK add-in can generate the code for all the skeletons of the agents, tasks and other classes included in the project. The pattern repository consists of a serie of reusable portions of agents and tasks. The repository also includes a list of tasks that can be applied to existing agents.

4.2 The Graphical Tool

We developed a graphical tool to be used as a complementary extension for PTK, with the purpose of allowing the designer of adaptive system to rapidly prototype different patterns to be applied to the same sets of tasks within the considered adaptive system. Our contribution aims to extend the concept of *Agent-Structure Exploration* as defined by the PASSI methodology (see Fig. 2), hence to focus the necessary design aspects needed to foster the dynamic creation of hierarchical structures of *autonomic components*. The word *component* is from now on used a substitute of agent, so that we are now able to operate a first distinction on the concept of Agent-Structure by dividing the agents of the considered population into two mutually exclusive categories: *Autonomic Managers* and *Service Components* (resp. AMs and SCs).

From the role description, the following phase of the PASSI methodology deals with "Behaviour Description", hence implying an unspecified relation between roles and behaviours. In our implementation, we detailed the relation between these two concepts, by proposing a logic that is consistent with the taxonomy of adaptation patterns as proposed in [28]. We defined a role as a collection of sequential and/or parallel behaviours and the concept of behaviour is an (implementation-wise) extension of the concept of Behaviour as intended in the JADE agent platform [3]. We therefore extended the definition of PTK so to support the definition of roles, behaviours, components and adaptive patterns so to have a tool able to generate the necessary artifacts for describing more complete adaptive systems.

As presented in section 2, an *adaptation pattern* is a conceptual scheme that describes a specific adaptation mechanism. It specifies how the component/system architecture can express adaptivity.

In [28], a taxonomy of adaptation patterns is defined in such a level of detail that we are able to translate most of the relevant aspects in an Object Oriented Programming point of view, with Java as our language of choice. In particular, we are now able to design abstract classes, interfaces and basic implementations of them (called default implementations) so to have re-usable structures of classes that the designer of adaptive systems can rely on.

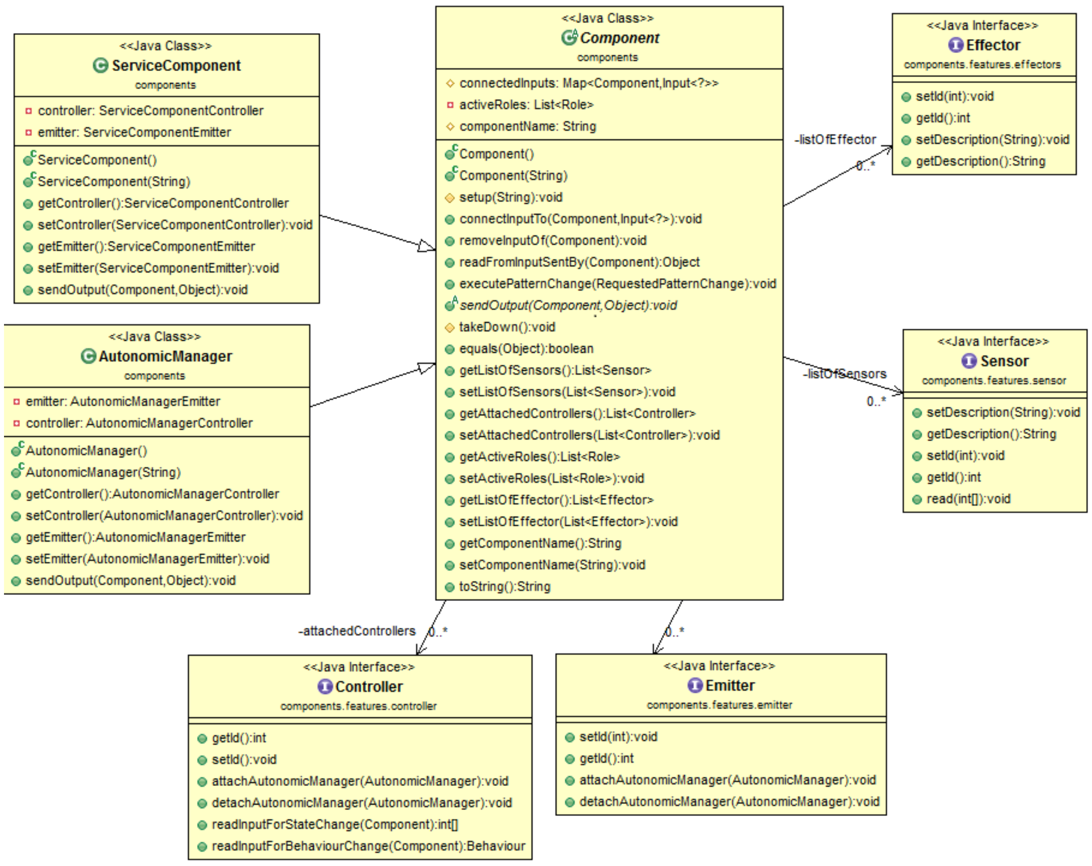
According to [28] a clear definition of the components interfaces helps us in understanding the mechanism of components' composition. The interfaces of a component can be described as a tuple of six elements:

– Input, used to receive information (e.g. service's request);
– Output, used to send information (e.g. service's reply);
– Sensor, that makes the component able to achieve information from the external (e.g. others components and/or the environment);
– Effector, that makes the component be able to manage the external (e.g. to act on the environment and manage other components);
– Emitter, used to emit status information to an external manager. This interface permits also to share information taken from the environment (using sensors) or other components;
– Controller, that makes it possible to an external adaptation manager to change and adapt the component's internal state.

According to our implementation, these elements refer to the generic concept of Component, hence the class Component is subsequently sub-classed into the classes *Service Component* and *Autonomic Manager*, with the latter one having the implicit role of adaptation manager. Service Components have a list of different sensors and actuators and those are mainly used for reading or applying modifications to the environment in which the ensemble is inserted. The list of sensors and actuators belonging to an Autonomic Manager is used to create connections with SCs or other AMs. Other important lists, associated with the component, are related to active roles, components attached to the emitter and components attached to the controller. Trivially, all the methods that allow us to modify these lists are accessible through the class hierarchy. The concept of component is an extension of the class *Agent* from the Jade agent platform, hence the component inherits methods such as *setup* and *takeDown* that refer to standard agent operations to be executed just after their creation or just before their removal from the considered context (i.e registering to or de-registering from a directory facilitator or other user-implemented yellow pages services).

A summarizing class diagram is in figure 5.

As shown in figure 5, components have the possibility to actively participate into a change of pattern. A pattern is a description of which component is connected to which other components inside the same adaptive system. More specifically, we can define a pattern by specifying for each component (AMs and SCs), which components are attached to the their emitters and controllers. By doing so we are able to re-create any combination of adaptation pattern that are referred as *taxonomies* in [28]. A pattern may or may not have an associated role, but is always bounded to a context. A context is an attribute of the environment and it is characterized by a list of *rules* and a list of *laws*. The difference between these latter ones is mostly case specific; however, as rule of thumb, we can specify that rules are used to regulate the single behaviour within a role, while laws deal with regulating the inter-component interactions (e.g: negotiations, elections etc.). As previously specified, roles are collections of



<<Java Class>>
ServiceComponent
components
controller: ServiceComponentController
emitter: ServiceComponentEmitter
ServiceComponent()
ServiceComponent(String)
getController():ServiceComponentController
setController(ServiceComponentController):void
getEmitter():ServiceComponentEmitter
setEmitter(ServiceComponentEmitter):void
sendOutput(Component,Object):void
<<Java Class>>
AutonomicManager
components
emitter: AutonomicManagerEmitter
controller: AutonomicManagerController
AutonomicManager()
AutonomicManager(String)
getController():AutonomicManagerController
setController(AutonomicManagerController):void
getEmitter():AutonomicManagerEmitter
setEmitter(AutonomicManagerEmitter):void
sendOutput(Component,Object):void
<<Java Class>>
Component
components
connectedInputs: Map<Component,Input<?>>
activeRoles: List<Role>
componentName: String
Component()
Component(String)
setup(String):void
connectInputTo(Component,Input<?>):void
removeInputOf(Component):void
readFromInputSentBy(Component):Object
executePatternChange(RequestedPatternChange):void
sendOutput(Component,Object):void
takeDown():void
equals(Object):boolean
getListOfSensors():List<Sensor>
setListOfSensors(List<Sensor>):void
getAttachedControllers():List<Controller>
setAttachedControllers(List<Controller>):void
getActiveRoles():List<Role>
setActiveRoles(List<Role>):void
getListOfEffector():List<Effector>
setListOfEffector(List<Effector>):void
getComponentName():String
setComponentName(String):void
toString():String
-listOfEffector
0..*
<<Java Interface>>
Effector
components.features.effectors
setId(int):void
getId():int
setDescription(String):void
getDescription():String
-listOfSensors
0..*
<<Java Interface>>
Sensor
components.features.sensor
setDescription(String):void
getDescription():String
setId(int):void
getId():int
read(int[]):void
-attachedControllers
0..*
<<Java Interface>>
Controller
components.features.controller
getId():int
setId():void
attachAutonomicManager(AutonomicManager):void
detachAutonomicManager(AutonomicManager):void
readInputForStateChange(Component):int[]
readInputForBehaviourChange(Component):Behaviour
0..*
<<Java Interface>>
Emitter
components.features.emitter
setId(int):void
getId():int
attachAutonomicManager(AutonomicManager):void
detachAutonomicManager(AutonomicManager):void

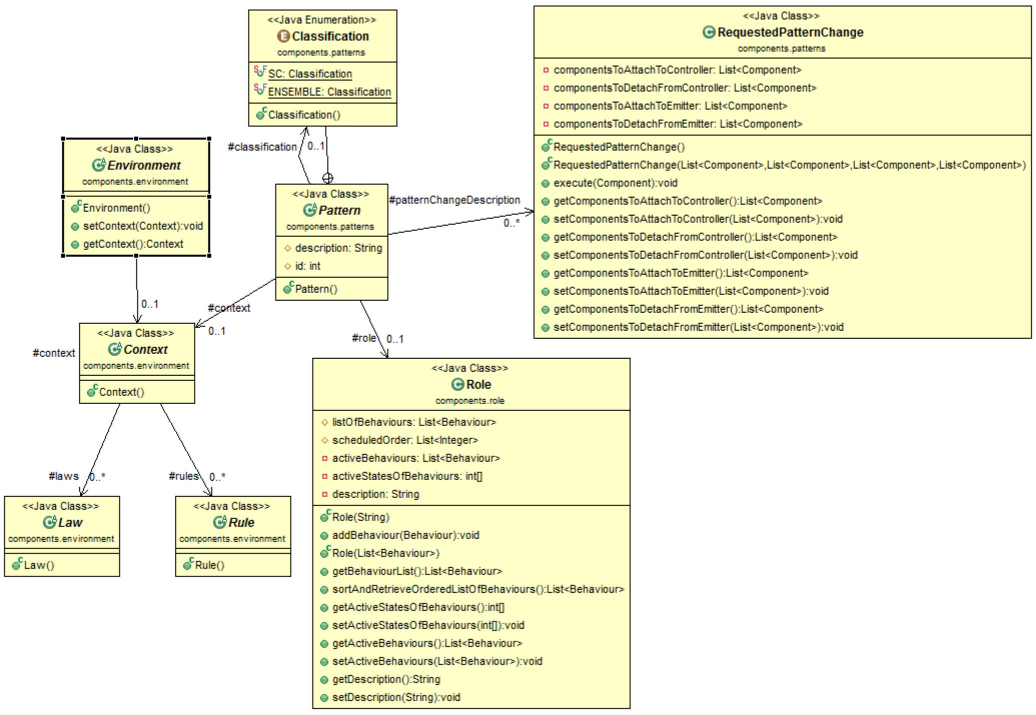
Fig. 5. UML Class diagram of the concept of Component

behaviours: this implies being able to use existing classes from the JADE package for defining behaviours as dynamically activated local computation within the agents/components: these computations can be sequential, parallel, cyclic or (using a JADE terminology) *one-shot*.

These considerations have been summarized in figure 6.

Now that we have a more detailed view on the concept of pattern and component, we were able to proceed in creating the extension for the PTK. We called this extension *Component Hierarchy Builder (CHB)*. Both these toolkits use tree structures as a mean of representing the considered system: a screen shot of both of them is in figure 7.

The PASSI toolkit (PTK) allows the designer to describe agent structures in terms of agents, tasks (as behaviours) and their relations. Agents can be characterized in low level implementation details: therefore for each agent we can specify which interfaces implements as well as defining attributes and methods' prototypes. Our extension of this PTK feature does not capture this level of



<<Java Enumeration>>
Classification
components.patterns
SC: Classification
ENSEMBLE: Classification
Classification()
<<Java Class>>
RequestedPatternChange
components.patterns
componentsToAttachToController: List<Component>
componentsToDetachFromController: List<Component>
componentsToAttachToEmitter: List<Component>
componentsToDetachFromEmitter: List<Component>
RequestedPatternChange()
RequestedPatternChange(List<Component>,List<Component>,List<Component>,List<Component>)
execute(Component):void
getComponentsToAttachToController():List<Component>
setComponentsToAttachToController(List<Component>):void
getComponentsToDetachFromController():List<Component>
setComponentsToDetachFromController(List<Component>):void
getComponentsToAttachToEmitter():List<Component>
setComponentsToAttachToEmitter(List<Component>):void
getComponentsToDetachFromEmitter():List<Component>
setComponentsToDetachFromEmitter(List<Component>):void
<<Java Class>>
Environment
components.environment
Environment()
setContext(Context):void
getContext():Context
#classification
0..1
<<Java Class>>
Pattern
components.patterns
description: String
id: int
Pattern()
#patternChangeDescription
0..*
0..1
#context
0..1
#context
<<Java Class>>
Context
components.environment
Context()
#role
0..1
<<Java Class>>
Role
components.role
listOfBehaviours: List<Behaviour>
scheduledOrder: List<Integer>
activeBehaviours: List<Behaviour>
activeStatesOfBehaviours: int[]
description: String
Role(String)
addBehaviour(Behaviour):void
Role(List<Behaviour>)
getBehaviourList():List<Behaviour>
sortAndRetrieveOrderedListOfBehaviours():List<Behaviour>
getActiveStatesOfBehaviours():int[]
setActiveStatesOfBehaviours(int[]):void
getActiveBehaviours():List<Behaviour>
setActiveBehaviours(List<Behaviour>):void
getDescription():String
setDescription(String):void
#laws
0..*
#rules
0..*
<<Java Class>>
Law
components.environment
Law()
<<Java Class>>
Rule
components.environment
Rule()

Fig. 6. UML Class diagram of the concept of Context, Pattern and Role

details, instead it complements it in order to have generic and reusable agents in the form of components. During the creation of a component in CHB, the user can decide to insert a new Autonomic Manager or a new Service Component and, for each, can specify which sensor and effectors are going to be used.

In PTK, tasks are behaviour and viceversa. In our implementation, tasks are not directly defined: instead roles are collection of behaviours and components can add scheduled roles by picking from all the roles that the user inserted in the CHB tree (figure 8).

In the example scenario depicted in figure 8, we can see a generic Service Component that is representing a mobile robot: it has proximity sensors and an effector constituted by a differential wheeled motor. As far as its scheduled roles are concerned, it has the task to explore an area and to operate the role of *Listener*. In order to know more about the listener role we have to open the corresponding CHB panel regarding roles and behaviours (see figure 9). The listener role is composed of two parallel behaviours that basically describe how the Service Component listens to its connected autonomic managers and (if necessary) enacts the requested state changes.

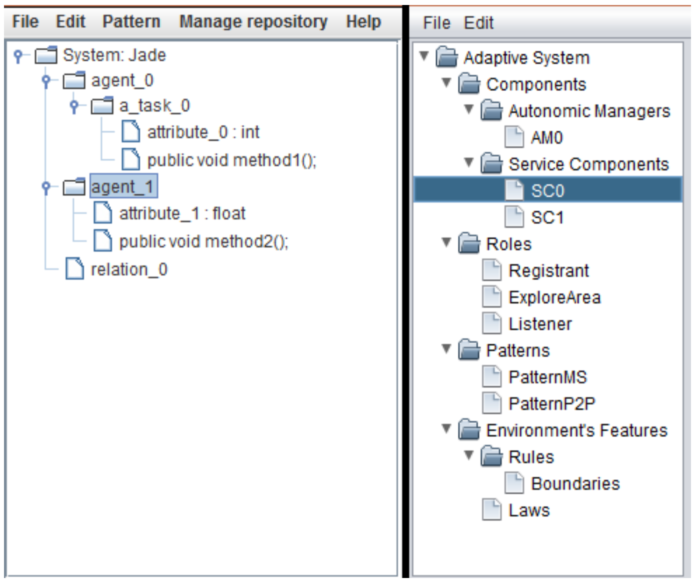
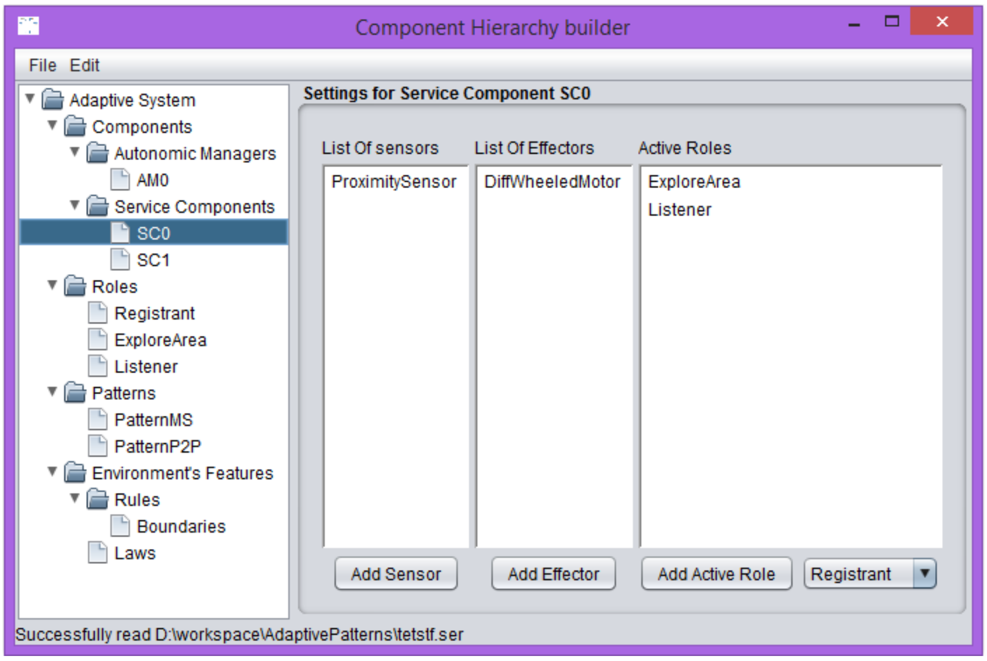


Fig. 7. Tree views of both PTK (left) and our proposed CHB (right)

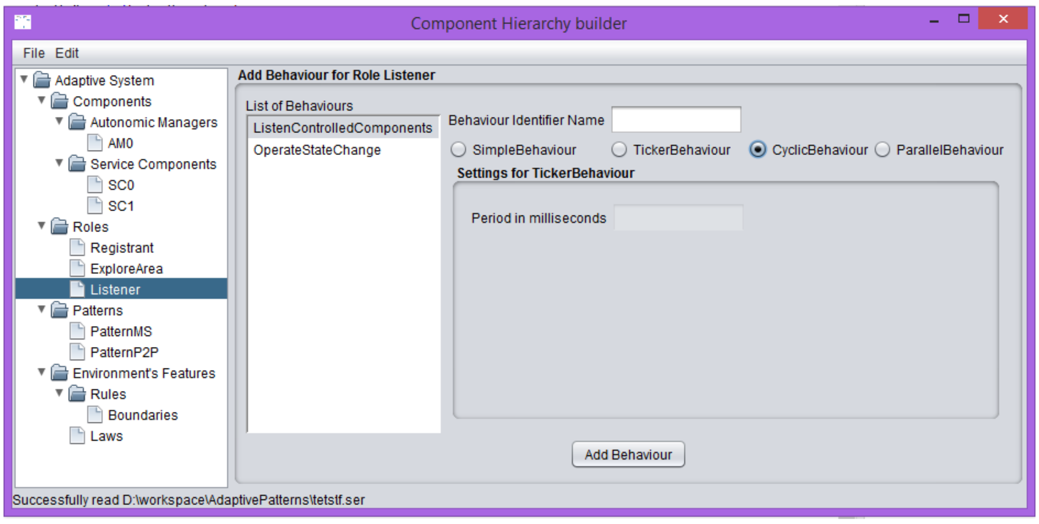


Settings for Service Component SC0

List Of sensors	List Of Effectors	Active Roles
ProximitySensor	DiffWheeledMotor	ExploreArea
		Listener

Successfully read D:\workspace\AdaptivePatterns\tetstf.ser

Fig. 8. A screen shot for the creation of a Service Component



SimpleBehaviour TickerBehaviour [✓ CyclicBehaviour] ParallelBehaviour

Successfully read D:\workspace\AdaptivePatterns\tetstf.ser

Fig. 9. A screen shot for the creation of a roles as collection of behaviours

In figure 9, we can see how for each behaviour composing the role, we can specify the sub-class of JADE's Behaviour class that better suits the component's needs.

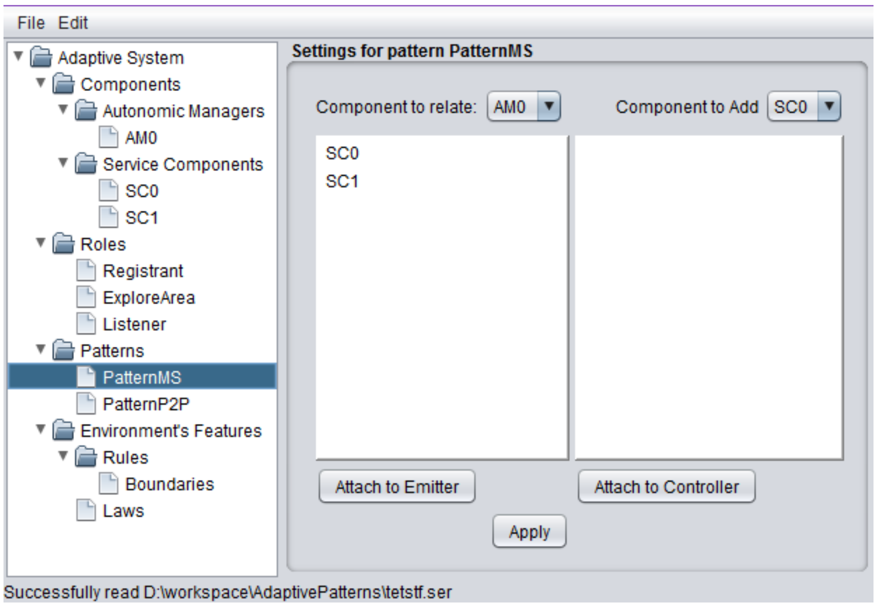
Using CHB, the user can specify adaptation pattern by indicating the connection topology among components, as specified as list of attached components in both controllers and emitters. This can be seen in figure 10, in which an example master-slave pattern is implemented and from the screen shot we can see how a generic Autonomic Manager is connected to both the Service Components that were previously inserted in the designed component structure.

Instead of generating code, CHB is able to serialize all the inserted information (components, patterns, context, roles etc...) inside a macro object that is called *adaptive system*. Such macro object can be serialized and deserialized so to allow the operations of modifications, export and import of previously saved component structures.

5 Related work

In literature, many approaches on patterns for self-adaptation exist, like the one of [33], [8] and [35]. However, in this paper we do not focus on the use of adaptation patterns to create self-adaptive systems, but on the definition of a useful methodology to create this kind of systems, with the aid of the patterns' approach.

In the last years, engineering research has tackled a well-defined problem and has carefully selected and combined existing solutions into a comprehensive development framework for self-adaptive systems.



Successfully read D:\workspace\AdaptivePatterns\tetstf.ser

Fig. 10. A screen shot for the creation of a pattern: point of view of the Autonomic Manager

A lot of projects like MADAM project[2] and the MUSIC project[3] tried to address adaptation in different scenarios, from both the theoretical and the practical perspective. For example, the MUSIC project [18] would like to introduce a model-driven development methodology [17] for self-adaptive context-aware applications. Different from us, this approach was to write a new methodology instead of exploiting the power of existing ones.

Other approaches as CARISMA [9] and RAINBOW [16] propose self-adaptation middleware or architectural styles to develop self-adaptive software, but they do not propose any methodology that will guide developers from the collection of requirements to implementation. Moreover, MOCAS (Model of Components for Adaptive Systems) propose a generic state-based component model which enables the self-adaptation of software components along with their coordination [2]; but like the other approaches, there are not concrete guidelines, considered as a methodology.

[2] Mobility and Adaptation-enabling Middleware, supported by the European Union under research grant 004159 lasting from September 2004 to March 2007.

[3] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments, supported by the European Union under research grant IST-035166 lasting from October 2006 to March 2010.

6 Conclusions

In this paper, we have proposed an approach to enrich methodologies for addressing adaptation in building self-adaptive systems. We considered, in specific, agent-oriented methodologies and we introduced in some of them our Catalogue of adaptation patterns that help in defining self-adaptive systems. In this paper we have shown how the patterns can be integrated in PASSI2, but our approach can be exploited in any methodologies for building adaptive systems. The possibility of easily inserting our Catalogue of adaptation patterns inside a methodology is based on the SPEM approach and permits harnessing the power of the chosen methodologies. Moreover, we have completed the methodologies process introducing our Catalogue of adaptation patterns also in the the supporting tools, in order to have all the steps completed. Then we have created a framework that permits matching the methodologies' concepts into agents' infrastructures.

As an ongoing work we are testing these modified methodologies in different scenarios, to have quantitative and qualitative results of the effectiveness of the methodologies.

References

1. aliCE Research Group et al. SODA home page, 2009.
2. Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. Mocas: A state-based component model for self-adaptation. In *Self-Adaptive and Self-Organizing Systems, 2009. SASO'09. Third IEEE International Conference on*, pages 206–215. IEEE, 2009.
3. Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE-a FIPA-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London, 1999.
4. Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli. *Methodologies and software engineering for agent systems: the agent-oriented software engineering handbook*, volume 11. Springer, 2004.
5. Carole Bernon, Marie-Pierre Gleizes, Sylvain Peyruqueou, and Gauthier Picard. Adelfe: A methodology for adaptive multi-agent systems engineering. In *Engineering Societies in the Agents World III*, pages 156–169. Springer, 2003.
6. Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.
7. Giacomo Cabri, Mariachiara Puviani, and Letizia Leonardi. The MAR&A methodology to develop agent systems. In *ICAART*, pages 501–506, 2009.
8. Giacomo Cabri, Mariachiara Puviani, and Franco Zambonelli. Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In *CTS*, pages 508–515. IEEE, 2011.
9. Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *Software Engineering, IEEE Transactions on*, 29(10):929–945, 2003.

10. ANTONIO Chella, Massimo Cossentino, and Luca Sabatucci. Tools and patterns in designing multi-agent systems with PASSI. *WSEAS Transactions on Communications*, 3(1):352–358, 2004.
11. Massimo Cossentino. From requirements to code with the PASSI methodology. *Agent-oriented methodologies*, 3690:79–106, 2005.
12. Massimo Cossentino and Valeria Seidita. Passi2–going towards maturity of the passi process. 2009.
13. Scott A DeLoach, Mark F Wood, and Clint H Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(03):231–258, 2001.
14. Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Paul L Snyder, and Giuseppe Valetto. A pattern-based architectural style for self-organizing software systems. *Drexel University, Department of Computer Science, Tech. Rep*, 6, 2012.
15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, Reading (MA), 1995.
16. David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.
17. Kurt Geihs, Roland Reichle, Michael Wagner, and Mohammad Ullah Khan. Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments. In *Software engineering for self-adaptive systems*, pages 146–163. Springer, 2009.
18. Svein Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli, and George Angelos Papadopoulos. A development framework and methodology for self-adapting applications in ubiquitous computing environments. *Journal of Systems and Software*, 85(12):2840–2859, 2012.
19. Brian Henderson-Sellers and Paolo Giorgini. *Agent-oriented methodologies.* IGI Global, 2005.
20. Philip Mayer, Annabelle Klarl, Rolf Hennicker, Mariachiara Puviani, Francesco Tiezzi, Rosario Pugliese, Jaroslav Keznikl, and Toma Bure. The autonomic cloud: a vision of voluntary, peer-2-peer cloud computing. In *Self-Adaptation and Self-Organizing Systems Workshops (SASOW), 2013 IEEE 7th International Conference on*, pages 89–94. IEEE, 2013.
21. Philip K McKinley, Seyed Masoud Sadjadi, Eric P Kasten, and Betty HC Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.
22. M. Morandini et al. On the use of the Goal-Oriented Paradigm for System Design and Law Compliance Reasoning. In *iStar*, volume 586, pages 71–75. CEUR-WS.org, 2010.
23. Object Management Group. SPEM. http://www.omg.org/technology/documents/formal/spem.htm, 1997.
24. Lin Padgham and Michael Winikoff. *Developing intelligent agent systems: A practical guide*, volume 13. John Wiley & Sons, 2005.
25. Juan Pavón, Jorge J Gómez-Sanz, and Rubén Fuentes. The INGENIAS methodology and tools. *Agent-oriented methodologies*, 9:236–276, 2005.
26. Mariachiara Puviani, Giacomo Cabri, and Letizia Leonardi. Enabling self-expression: the use of roles to dynamically change adaptation patterns. In *FOCAS 2014*. IEEE Computer Society, 2014.
27. Mariachiara Puviani, Giacomo Cabri, and Letizia Leonardi. Integrating adaptation patterns into agent methodologies to build self-adaptive systems. In *7th Inter-*

national conference on Agents and Artificial Intelligence (ICAART 2015), pages 99–106. SciTePress–Science and Technology Publications, 2015.

28. Mariachiara Puviani, Giacomo Cabri, and Franco Zambonelli. A taxonomy of architectural patterns for self-adaptive systems. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, pages 77–85. ACM, 2013.
29. Mariachiara Puviani, Giacomo Cabri, and Franco Zambonelli. Agent-based simulations of patterns for self-adaptive systems. In *ICAART 2014 - Proceedings of the 6th International Conference on Agents and Artificial Intelligence, Volume 1, ESEO, Angers, Loire Valley, France, 6-8 March, 2014*, pages 190–200. IEEE Computer Society, 2014.
30. Mariachiara Puviani, Massimo Cossentino, Giacomo Cabri, and Ambra Molesini. Building an agent methodology from fragments: the MEnSA experience. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 920–927. ACM, 2010.
31. Mariachiara Puviani, Giovanna Di Marzo Serugendo, Regina Frei, and Giacomo Cabri. Methodologies for self-organising systems: a SPEM approach. In *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology-Volume 02*, pages 66–69. IEEE Computer Society, 2009.
32. Mariachiara Puviani, Giovanna Di Marzo Serugendo, Regina Frei, and Giacomo Cabri. A method fragments approach to methodologies for engineering self-organizing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(3):33, 2012.
33. A.J. Ramirez and B.H.C. Cheng. 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. ACM, May 2010.
34. Valeria Seidita, Massimo Cossentino, and Salvatore Gaglio. Using and extending the SPEM specifications to represent agent oriented methodologies. In *Agent-Oriented Software Engineering IX*, pages 46–59. Springer, 2009.
35. Danny Weyns, Sam Malek, Jesper Andersson, and Bradley Schmerl. Introduction to the special issue on state of the art in engineering self-adaptive systems. *Journal of Systems and Software*, 85(12):2675–2677, 2012.
36. Franco Zambonelli, Nicholas R Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(3):317–370, 2003.