DOI 10.12694/scpe.v18i1.1235
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 91–103. http://www.scpe.org © 2017 SCPE

ENABLING AUTONOMIC COMPUTING SUPPORT FOR THE JADE AGENT PLATFORM

ALI FARAHANI[*], ESLAM NAZEMI[†], GIACOMO CABRI[‡], AND NICOLA CAPODIECI[§]

Abstract. Engineering complex distributed system is a real challenge documented in recent literature. Novel paradigms such as Autonomic Computing (AC) approaches appear to be the fittest engineering model in order to face performance instabilities of systems inserted in open and non-deterministic environments. To this purpose, the availability of appropriate development environments will facilitate the design of such systems. Standard agent development platforms represent a good starting point, but they generally lack of rigorous ways to define central AC-related concepts such as the prominent role of feedback loops and knowledge integration in decision making processes: we therefore believe that Agent Oriented Software Engineering (AOSE) can be substantially enriched by taking into account such concepts.

In this paper, a novel extension of the well-known JADE agent development environment is discussed. This extension enhances JADE in order to address the engineering process with Autonomic Computing support. It is called "Autonomic Computing Enabled JADE" or shortly ACE-JADE. The behavioral model of ACE-JADE will be thoroughly described in the context of a case study (NASA ANTS project).

Key words: JADE, Autonomic Computing, Multi-agent Systems, Development Environment

AMS subject classifications. 68T42

1. Introduction.

The complexity of information systems has grown dramatically in recent years. This complexity, which is the result of several factors like advances in hardware and infrastructure technology, the growth of Internet networks, etc., has inspired Autonomic Computing (AC) approaches, which enable software components to go through runtime adaptation processes in order to react to changes of their execution contexts [24, 22]. It is important to design software able to promptly react to these changes as unpredictable dynamic environments pose threats to the stability and performance of the designed system. In order to fully exploit the benefits of the mechanisms behind these adaptation processes, different architectures were introduced [20, 36] and they all derive from the first work that introduced the concept of Autonomic Computing, as described by IBM [22].

Autonomic Computing builds upon the concept of autonomous software entity and focuses on the interactions between the decisional process within the software entity and its surrounding environment. The general idea is to establish a feedback loop able to provide additional elements to be used in the decisional process of each software entity and a well-known architectural design for these feedback controlled mechanisms is the *MAPE loop*. This loop consists of four phases: Monitoring, Analyzing, Planning and Executing [24, 23]. During the *Monitor* phase, the system component collects and correlates information from the environment; this data collection phase is enabled through specific sensorial capabilities featured by the component. In the *Analyze* phase, the component will analyze the variables observed during the Monitor phase, so to determine the type and the magnitude of the needed reactions: this is instrumental to correctly respond to threatening environmental changes. Passing the result of analyze phase into the *Plan* phase, the component will select and conclude into more specific sub-set of actions to be performed in the environment to achieve a desired state. And at last, in the *Execute* phase, the component will enact the selected actions through effectors. [12].

As mentioned in [33], feedback loops foster self-* properties within the software systems. These kinds of capabilities are related to the awareness of a system about itself and its environment, hence the ability of a system to autonomously react to environmental changes. There is no general agreement over these concepts and the related terminology but there are main four properties (self-healing, self-protecting, self-optimizing and self-configuring) usually categorized as self-* properties or simply summarized under the term self-managing [24].

[*]Computer Science and Engineering Department, Shahid Beheshti University Tehran, Iran (a_farahani@sbu.ac.ir)
[†]Computer Science and Engineering Department, Shahid Beheshti University Tehran, Iran (nazemi@sbu.ac.ir)
[‡]Department of Physics, Informatics and Mathematics, Università di Modena e Reggio Emilia, Modena, Italy (giacomo.cabri@unimore.it)
[§]Department of Physics, Informatics and Mathematics, Università di Modena e Reggio Emilia, Modena, Italy (nicola.capodieci@unimore.it)

In this paper, we present how AC topics can seamlessly integrate within distributed system engineering methodologies such as Agent Oriented Software Engineering (AOSE). Distributed system is considered as another solution to large scale and complex problems, thanks to their high degree of scalability. Bringing more resources together to satisfy needs for more coverage and computation is a solution, even if managing them leads into another kind of complexity [37]. Having autonomous agents which can deal with changes by their own is a way which can heal the complexity. Appropriate design and simulation frameworks for distributed autonomous system are therefore valuable instruments for the system designer.

Multi-Agent Systems (MAS) is one of the main branches of distributed systems, enabling autonomous components to carry out tasks in a decentralized way, supporting flexibility and scalability. MAS architectures described in literature vary from abstract design methodologies to detailed engineering frameworks or platforms. FIPA (Foundation of Intelligent Physical Agents) proposes a complete reference architecture for MAS [28]. JADE (Java Agent Development Environment) is a FIPA compliance agent development environments [5, 4], which supports the development of multi-agent systems. Bringing AC into the JADE can enable the self-* properties in MAS architectures.

The work presented in this paper aims at extending JADE to bring support for Autonomic Computing in the JADE development environment. This idea has been preliminary discussed in [18] and in this paper we provide much deeper analyses and discussions regarding our novel extension. Section 2 presents the background and related work. Main elements of proposed architecture are discussed in Section 3. Section 4 will be about a case study and implementing the presented extension in a scenario. Conclusion and future work are discussed in Section 5.

2. Background and Related Work.

In this section, background information about self-adaptation (and autonomic computing in general) and related concepts are briefly presented, along with JADE as a development environment. The previous researches about providing JADE extensions are also discussed.

2.1. Autonomic Computing in software development.

In [17, 25] researchers tried to use component-based approach to help the development of autonomic software. In [17] an infrastructure named AUTONOMIA is presented, which foster engineering and deployment of autonomous applications. In [25] a framework for supporting development of a component-based application (self-managed application) is presented. These solutions represent proof-of-concepts in which we start to develop our extension.

In [7] a toolkit is presented (*ABLE*) for building multi-agent autonomic systems. This toolkit provides a lightweight Java agent framework with a set of beans able to support self-* properties development. This toolkit focuses more on the development of agents' self-* tasks rather than on characterizing the capabilities of multi-agent system. Also, in [14], a framework based on component-based viewpoint is introduced. A conceptual framework for designing autonomic components starting from the artificial immune system paradigm have been proposed in [11]. It covers the self-expression aspect of system based on inspiration from the natural immune system. Researches for bringing autonomic computing in system design and development are not bounded to only high level concepts like framework. For instance a research provides a language (SCEL, Software Component Ensemble Language) to be used in any framework for formally and rigorously describe coordination patterns for autonomic agents and their interactions [10].

The novelty of our work consists in providing an actual implementation in the form of a JADE extension of the topics explored as theoretical subjects by the cited articles described in this section. During the development of this extension, we realized that many important engineering artifacts were missing, thus we integrated this design effort with our own ideas.

2.2. JADE and its extensions.

JADE (Java Agent Development Environment) is a development environment which facilitates the process of multi-agent system development, providing a platform that supports the agent design and execution. Multi-agent systems and especially agents themselves have many reference architectures. JADE is compliant with FIPA agent architecture [5].

Adding **AC** (Autonomic Computing) ability to the agent development environment has been subject to previous research. For instance, in [15] an architecture-based extension for supporting feedback loops in agents has been discussed. In particular, authors [15] discussed the possibility of adding architectural patterns so to have feedback loops in MAS.

Most of the JADE extensions that are related to AC concept are built upon the *behaviour* concept, which can be considered as one of the JADE primary elements. In [8] JADE agent behaviour is extended in order to support *behaviour tree* and *flexible behaviour* in JADE. Researches in [1] and [21] bring *organization* and *role* concepts into JADE, but there is still no mention regarding control loops. Control loops have been exploited in the form of coordination patterns into the software development process to have an autonomic computing enabled development kit [30]. The idea of having a role-based design in multi-agent systems alongside coordination mechanisms is able to provide a methodology which its output can be used to generate JADE agent classes.

Other examples of JADE extension in order to provide more formal ways to model adaptivity can be found in [27], [9] and [26]. Also The idea about using JADE and MATLAB (Simulink) have been addressed in [32] as a JADE extension called MACSimJX. This extension is about creating multi-agent control systems. Using Simulink as a way to describe agents, MACSimJX enables in JADE to implement a multi-agent control system. Using this idea in micro-grid environment is discussed in [31].

The common idea behind these extensions is to bring something new into JADE in order to add new features or to exploit it in new application fields. Enhancing JADE with ACE-JADE will provide the ability to facilitate the process of implementing AC enabled software in MAS environment. In [19] a preliminary idea about bringing Autonomic Computing into JADE has been discussed and future challenges addressed. In this paper, the idea of JADE extension will be discussed with more detail.

2.3. Agent Development Platforms. JADE is not the only FIPA-compliant agent platform. There are some researches about self-* system development tools and frameworks (like [16] as a framework for multi-agent systems development in IoT environment) which mostly are compliant with FIPA standards. *Grasshopper* is presented in [2] as an agent development platform based on OMG MASIF and FIPA. A work-flow management system alongside an intelligent system business is supported in [2]. *Grasshopper*, however, appears to be not currently maintained, hence, in order to develop ACE-JADE, we decided to stick with JADE.

There are several development aspects related to ACE-JADE. In [6] researchers mention that knowledge management is one of the critical parts of the proposed development environment. One of the targeted ways for addressing this challenge is rule-based knowledge management. In distributed environment conflicts between each agent's knowledge and reasoning could be a problem. In [29] this challenge has been addressed. Also, some other researches are taking place without tying themselves to FIPA standard. Jackal [13] is one of these solutions. It presents a Java-based tool for development of agents. These kinds of solutions have a downside of not getting the same level of attention and support as the other frameworks that comply with known standards.

The idea of having *autonomic computing* support in the JADE emerged after realizing that all of the previously discussed existing research does not involve the implementation of tools and platform that can be used by the system engineers. The popularity of JADE, due to its open source Java implementation on the top of the FIPA standard, is the ideal starting point for such effort, that will build upon the findings of all the previously cited papers.

3. AC Extension of JADE (ACE-JADE). A framework for MAS can be addressed from two aspects; namely, (1) a *structural* aspect and (2) a *behavioral* aspect. Our MAS extension, therefore, will be discussed considering both these aspects, i.e. extending the JADE java classes related to structural part and also extending the behavior related classes.

More specifically, our AC enabled JADE extension is articulated in five points:

1. *Providing adaptive agents.* In MAS the central component is the *agent.* In order to have an autonomic system, we exploit the concept of agent in terms of generic autonomous software component.
2. *Implementing different feedback loops by extending the class `Behaviour`.* Like the structural concept of (`Agent`), there must be an extension of behavioural concepts in JADE (`Behaviour`) to enable the response to environmental changes.
3. *Adding Knowledge support in the adaptive agent.* Data and knowledge is a vital concept in AC, therefore it will be present in our JADE extension.
4. *Support for `Sensors` and `Effectors`.* Sensors and effectors are AC related concepts that deal with the ability of an agent/software component to obtain information regarding the surrounding environment and act on the same environment. Our extension will therefore model environment interactions through Sensors and Effectors classes.

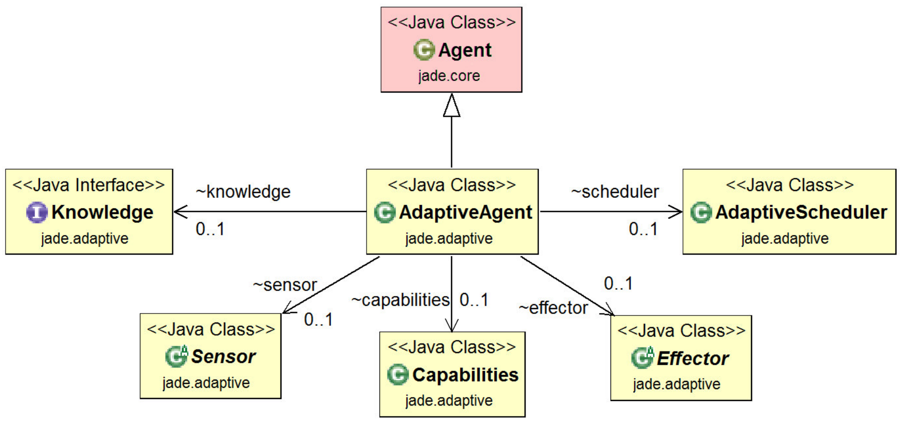


FIG. 3.1. *AdaptiveAgent class diagram.*

5. *Extending `Message` to support internal messages among different adaptive agents.* JADE original `Message` class cannot fulfill the need for information sharing in the AC, hence we had to provide an added layer of complexity to the original Message related classes. More specifically, we later detail how Sensors and Effectors will be implemented with a message passing interface.

The mentioned points are vital for making ACE-JADE. In the following sections, these five different aspects will be discussed. Each aspect will be discussed by a *class diagram* and if the behavioural aspect is involved, it will be analyzed though a *sequence diagram.* The class diagrams will explain parts that have been extended in the JADE original structure and the sequence diagram will show the mechanisms that our novel extension will provide in order to obtain the autonomic computing related features on top of JADE.

Starting from Fig. 3.1, all the subsequent class diagrams will use the following notation: in red, we indicate the original classes already present in the JADE standard package, whereas yellow artifacts are actually related to classes and interfaces provided by our extension.

3.1. Providing `AdaptiveAgent`. The `AdaptiveAgent` class (our implementation of autonomic software component) keeps references to `Sensor` and `Effector` instances that characterize the components (see Fig. 3.1). These agents are also composed by instances of `Capabilities` class, which define the list of acceptable *parameters* and list of feasible *actions* for such agents.

In addition to what can be seen in Fig. 3.1, we specify that for each `AdaptiveAgent` component, a set of *behaviours* for adaptive tasks is provided (by means of the class `AdaptiveBehaviourSet`). It consists of a set of `AdaptiveBehaviours` with a list of task priorities (`BehaviourOrder`). Also, there is another proposed class, `AdaptiveScheduler`, in which resides the implementation of how different behaviours are arbitrated in terms of ordering. This `AdaptiveScheduler` extends the `Scheduler` class of JADE. While there is not much to explain with regards to the scheduler, the behaviours are the most important aspect of our work and they are detailed in the next section.

3.2. Extending Behaviour. The most important aspect of AC is responding to changes in the environment in an autonomous and adaptive way. The reaction towards the dynamics of the environment is taken care by the behavioural aspect of ACE-JADE.

Autonomic computing enabled through MAPE feedback loop is composed of four sequential actions (Monitor, Analyze, Plan and Execute) and such actions involve the use of sensors and effectors. Each of these phases can be composed by a plurality of steps. As mentioned in [23], we identify different levels of adaptation according to whether all of these four phases are present: a distributed system with limited adaptation capabilities, for instance, might feature a simplified control loop, in which just the Monitor and Execute phases are present. By defining a control loop as a subset of combinations of the MAPE phases, ACE-JADE is therefore able to model different adaptation capabilities.

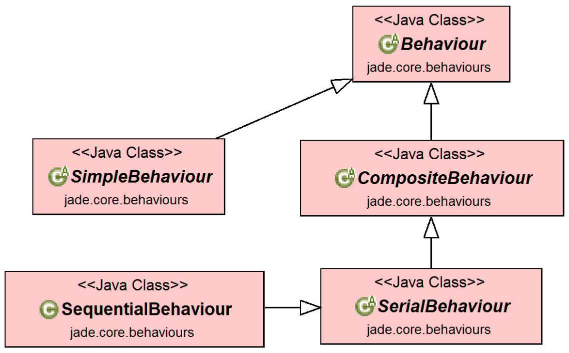


FIG. 3.2. *SerialBehaviour, SequentialBehaviour and CompositeBehaviour relations*

In ACE-JADE, MAPE control loops as a whole, their phases and each step for each phase is a JADE behaviour. From the JADE original `behaviour` package, we infer the class diagram in Fig. 3.2. These are the classes from which our custom control loops will be implemented, but also phases and single steps for each constituent part of the feedback loop.

In order to implement different control loops with each having a different level of adaptation, we have five different classes and they all extends the JADE original `CompositeBehaviour` class:

- class `AutonomicBehaviour`: which allows the implementation of all the MAPE four phases without focusing on a complex Knowledge representation logic.
- class `AdaptiveLoopBehaviour`: which also allows the implementation of all the MAPE phases and forces the user to implement specific Knowledge representation logic.
- class `PredictiveLoopBehaviour`: which allows the definition of simpler control loops, that just consider Monitoring, Analyze and Execution phase.
- class `ManagedLoopBehaviour`: same as above, but focusing on Monitor, Execute and Knowledge representation.
- class `BasicLoopBehaviour`: represents the simplest kind of feedback loop, as just Monitor and Execute phase can be exploited from this class.

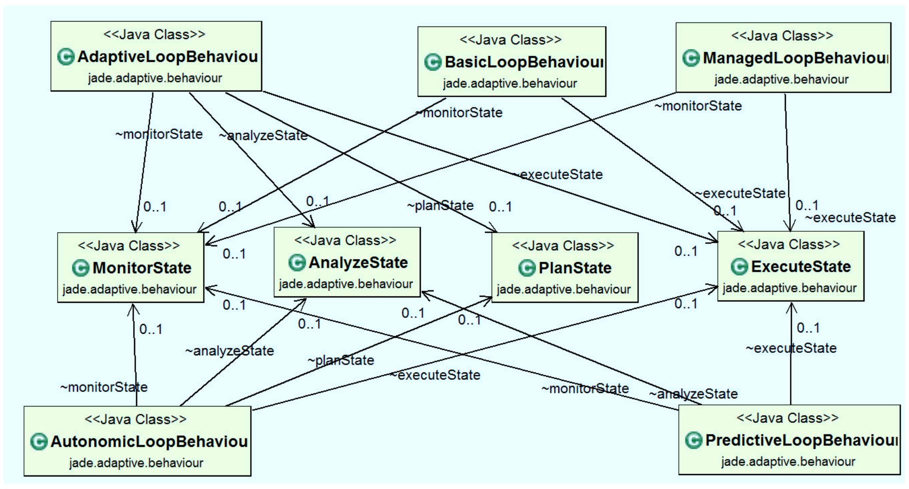
In other words, according to the level of desired self-adaptation, the user will select the appropriate class. The selected class relates to specific admissible MAPE phases (detailed in the previous class list). Each of these different classes are composed of one instance of each admissible phase state. A phase state is one of `MonitorState`, `AnalyzeState`, `PlanState` and `ExecuteState` (see Fig. 3.3). Phase state classes are derived from the JADE original class `SequentialBehaviour`.

Each phase state class is therefore composed of 1 to N corresponding steps (`Monitor`, `Analyze`, `Plan` and `Execute`). A step class is derived from the `SimpleBehaviour` class out of the JADE original `behaviour` package.

This particular class hierarchy allows us not only to tune the self-adaptive capability of our system, but also allows for implementing logic for each of the MAPE phase within an arbitrary complexity given by the composition of several steps. This is visible in Fig. 3.4, in which, for brevity, just one of the adaptive level is shown (`AutonomicLoopBehaviour`).

3.3. Introducing Knowledge. Another important requirement to introduce AC in JADE is to enable agents to manage knowledge and information, so an appropriate implementation of these concepts has been introduced in our extension of JADE. An interface named `Knowledge` is introduced, which mainly provides two methods: `toRule` and `fromRule`. The first method defines a behaviour in response to a predicate; such resulting behaviour will be inserted in a knowledge repository. Such knowledge repository can be queried with `fromRule` method.

The `Knowledge` interface is implemented by a specific class for each message type (between each MAPE step). `SensorKnowledge` and `EffectorKnowledge` are implemented to support the tasks of `Sensor` and `Effector`. These classes, with the help of `Capabilities`, will define which *parameters* should be sensed and which *actions* can be performed.



<<Java Class>> AdaptiveLoopBehaviou jade.adaptive.behaviour
<<Java Class>> BasicLoopBehaviou jade.adaptive.behaviour
<<Java Class>> ManagedLoopBehaviou jade.adaptive.behaviour
<<Java Class>> MonitorState jade.adaptive.behaviour
<<Java Class>> AnalyzeState jade.adaptive.behaviour
<<Java Class>> PlanState jade.adaptive.behaviour
<<Java Class>> ExecuteState jade.adaptive.behaviour
<<Java Class>> AutonomicLoopBehaviou jade.adaptive.behaviour
<<Java Class>> PredictiveLoopBehaviou jade.adaptive.behaviour
~monitorState
~analyzeState
~planState
~executeState
0..1

FIG. 3.3. *Behaviour class diagram.*

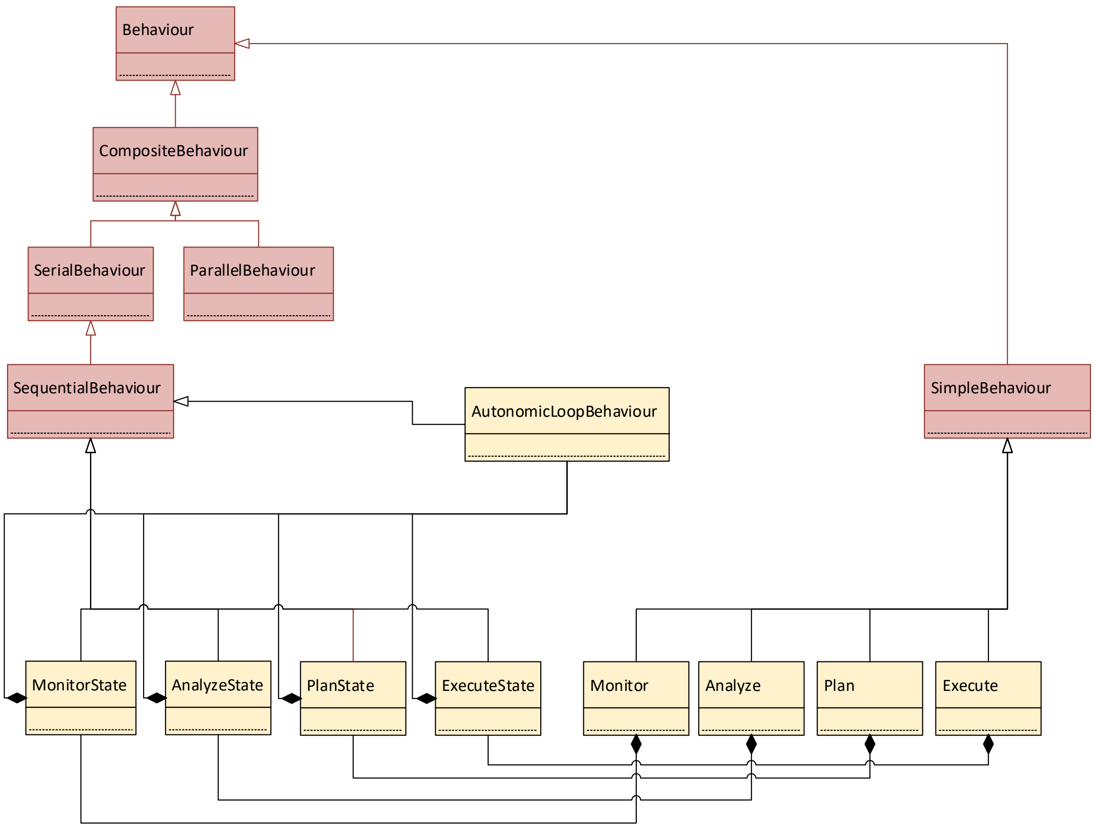


FIG. 3.4. *Holistic view of Class diagrams for Autonomic Loop Behaviour*

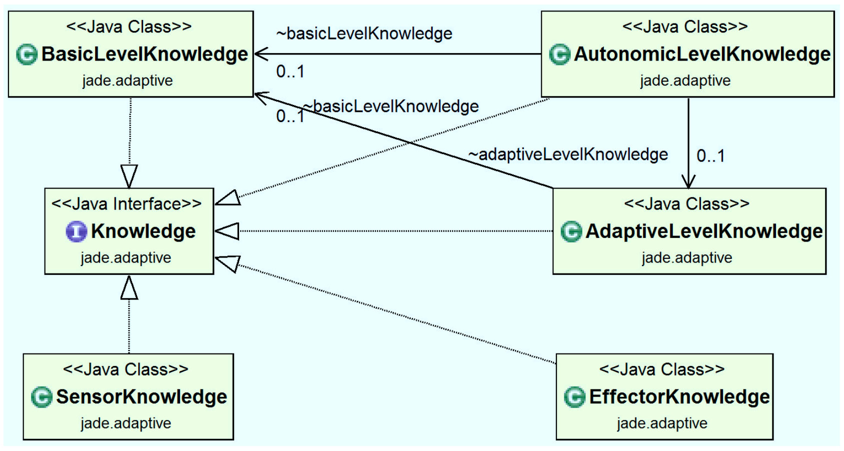


Fig. 3.5. *Knowledge-related class diagram.*

In [23] three levels of knowledge about the environment and reaction to the environment are presented. Each level has its own abstraction and each of the five different kinds of control loop: `BasicLevelKnowledge`, `AdaptiveLevelKnowledge` and `AutonomicLevelKnowledge`. The class diagram of the Knowledge-related class can be seen in Fig. 3.5. According to the complexity of the knowledge repository, the user can select which class to extend.

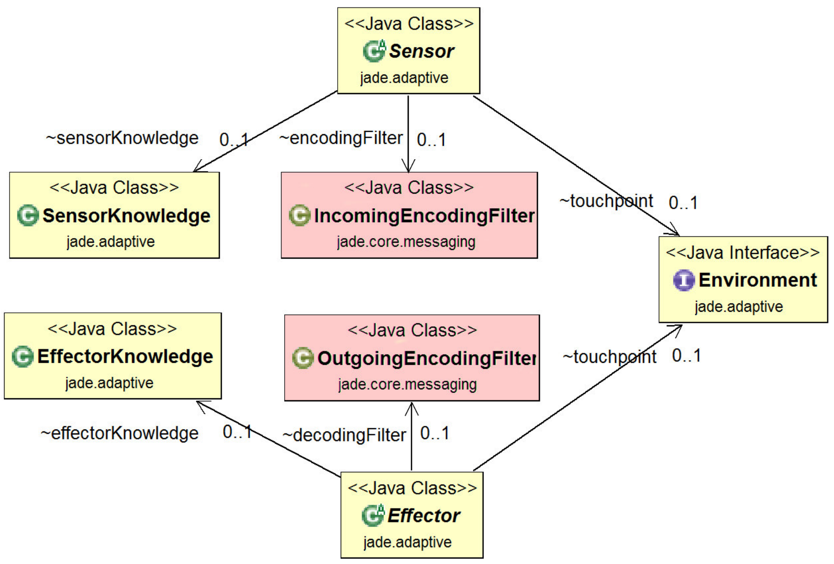
3.4. Support for Sensor and Effector. `AdaptiveAgent`s will have to deal with `Sensor`s and `Effector`s. These are classes for implementing the logic in which the environment can be sensed and modified according to the behavioural logic expressed by the implemented control loop. Such sensor might be further specified with capabilities. For addressing the capabilities, a class named `Capabilities` that shows the incoming known parameters and also the outgoing accessible parameters and their transformation function is introduced. The transformation operation takes place whenever sensed data needs to be translated in a different format, as different component of our system might implement different data representation schemes. This transformation is performed by two JADE original classes: `IncomingEncodingFilter` and `OutgoingEncodingFilter`, which will be discussed in next subsection. `Effector` is an extension of `OutgoingEncodingFilter` and translates in a different representation scheme the modification to be enacted to the environment.

The agent knows about the environment in which it is situated. The environment is represented by `Environment`, which is an interface that enables sensor and effector to know where they should get the data from and what they should effect. `AdaptiveAgent.environment` can be accessed through the `environment` variable in `Sensor` and `Effector`.

The class diagram of the `Sensor` and `Effector` class can be seen in Fig. 3.6.

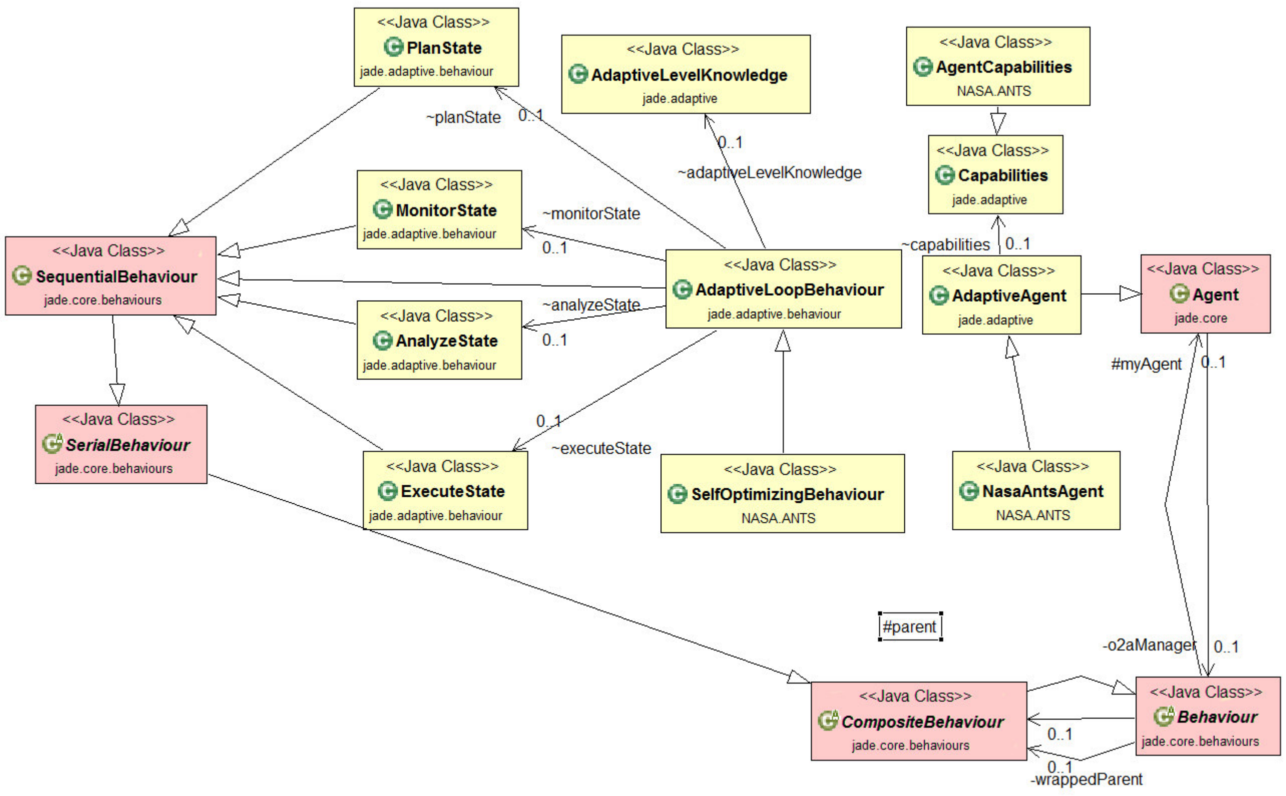
3.5. Support for Internal Messaging. Passing information and data within the system needs an appropriate *messaging platform*. JADE has its own messaging platform which is extended by ACE-JADE in order to help in order to deal with sensors and actuators of each component of the system.

- The `Sensor` ACE-JADE class will use the original `IncomingEncodingFilter` class for translating the data based on the data model which is presented by our added concept of `SensorKnowledge`. An instance of `IncomingEncodingFilter` should be created by the developer based on his/her needs and passed to the `SensorKnowledge`'s instance. The developer could translate the monitored parameters from environment into a set of control signals. Filters are used to process only the relevant subsets of information sensed from the environment.
- As far as effectors are concerned, filters are implemented through the original class `OutgoingEncoding-Filter`. This class is therefore able to filter outgoing commands related to the encoding of ACL messages [3]. Filtering rules can be further tuned with the ACE-JADE specific class `EffectorKnowledge`.



<<Java Class>> Sensor jade.adaptive
<<Java Class>> SensorKnowledge jade.adaptive
<<Java Class>> IncomingEncodingFilter jade.core.messaging
<<Java Interface>> Environment jade.adaptive
<<Java Class>> EffectorKnowledge jade.adaptive
<<Java Class>> OutgoingEncodingFilter jade.core.messaging
<<Java Class>> Effector jade.adaptive
~sensorKnowledge 0..1
~encodingFilter 0..1
~touchpoint 0..1
~effectorKnowledge 0..1
~decodingFilter 0..1
~touchpoint 0..1

Fig. 3.6. *Sensor and Effector class diagram.*



<<Java Class>> PlanState jade.adaptive.behaviour
<<Java Class>> AdaptiveLevelKnowledge jade.adaptive
<<Java Class>> AgentCapabilities NASA.ANTS
<<Java Class>> Capabilities jade.adaptive
<<Java Class>> MonitorState jade.adaptive.behaviour
<<Java Class>> SequentialBehaviour jade.core.behaviours
<<Java Class>> AdaptiveLoopBehaviour jade.adaptive.behaviour
<<Java Class>> AdaptiveAgent jade.adaptive
<<Java Class>> Agent jade.core
<<Java Class>> AnalyzeState jade.adaptive.behaviour
<<Java Class>> SerialBehaviour jade.core.behaviours
<<Java Class>> ExecuteState jade.adaptive.behaviour
<<Java Class>> SelfOptimizingBehaviour NASA.ANTS
<<Java Class>> NasaAntsAgent NASA.ANTS
<<Java Class>> CompositeBehaviour jade.core.behaviours
<<Java Class>> Behaviour jade.core.behaviours
~planState 0..1
~adaptiveLevelKnowledge 0..1
~monitorState 0..1
~capabilities 0..1
~analyzeState 0..1
~executeState 0..1
#myAgent 0..1
#parent
-o2aManager 0..1
0..1
-wrappedParent 0..1

Fig. 3.7. *Self-optimizing scenario implementation class diagram*

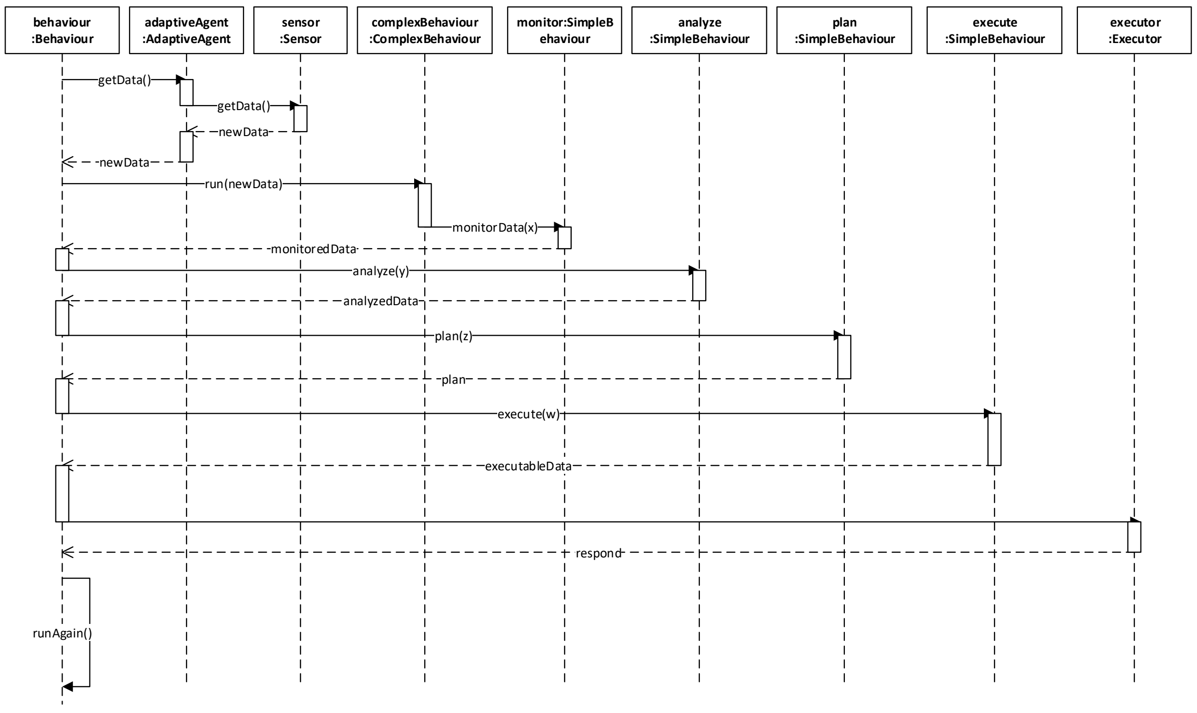


FIG. 3.8. *Sequence Diagram of a simple MAPE loop*

3.6. Behavioural Analysis. While in the previous sections we addressed the *structural* aspect of our extension, in this section we use sequence diagrams in order to better explain the *behavioural* aspects of ACE-JADE.

In Fig. 3.8, a sequence diagram of a control flow for a simple MAPE loop is reported. The `behaviour` is an active thread which, on a defined interval, starts to see if there is any new data available for the process as obtained by a `sensor`. For this purpose, the component-agent sends a message to `adaptiveAgent` and it passes the control to `sensor` in order to analyze the new data. Afterward, new data goes back to the `adaptiveAgent` and after that to `complexBehaviour`. For analyzing the data with MAPE loop, `complexBehaviour` passes the control through all the four steps of MAPE loop on after another. At last in this case, *executablePlan* as a result of `plan()` is executed with the help of `execute()`.

4. Implementing a Case Study. For better understanding the proposed AC extension to JADE, a simple scenario about managing failure (self-healing property) in *ruler* components of the **NASA ANTS** project is implemented. NASA ANTS is a new class mission from NASA [35]. NASA Autonomous Nano Technology Swarm is a milestone for autonomic computing concept and because of its specification, it represents a useful case study for autonomic computing distributed environment to be modeled with ACE-JADE. In the NASA ANTS project thousands of tiny space-craft weighting less than 1.5 kg will work cooperatively to explore the asteroid belt [35, 34]. In the project, three different kinds of space-crafts are working together to carry out the project mission:

- *Worker:* the largest number of space-crafts are of worker kind (80%). They carry different instruments for gathering data.
- *Ruler:* this kind of space-craft defines and update the rule of each space-craft in the team formation.
- *Messenger:* this kind of space-craft enables communication between workers, rules and the earth station.

All of these space-crafts have a degree of autonomy and adaptation to perform necessary tasks.

4.1. Scenario. We chose a scenario that can be addressed by designing an internal MAPE loop for each Messenger space-craft; each Messenger is therefore implmented by an instance of our `AdaptiveAgent` class, as the concrete example of an autonomic component. In this scenario, both *messengers* and *rulers* can perform a change in their states: the former by changing their location, the latter by updating their information about new space-crafts. Rulers therefore are in charge of specifying to the messenger, which messages have to be passed to the workers. The goal in this scenario is to enable *self-optimization* and *self-protection* for all these space-crafts.

In order to foster self-optimization and self-protection, a *messenger* is enhanced with two MAPE control loop. A first loop is defined to improve its positioning over time (as it can communicate within fixed ranges), and a second loop is in charge of detecting the status of local established connections with the other space-crafts, and react by changing the messenger direction of movement.

As far as the first MAPE loop is concerned, a messenger space-craft finds out that by changing to a given location it can reach more space-crafts. It can do this by measuring the density of the space-crafts within its range. This will represent the ability to *self-optimize*.

As far as the second loop is concerned, this loop which monitors the messenger special connections (such as the one with the ruler agents) and when it detects the probability of losing these connections it reacts by changing its movement direction in order not to get too far away from a ruler. This will represent the ability to *self-protect*.

Self-optimization and self-protection are tied together in this scenario: the two MAPE loops are designed in order to balance the goal of varying the position of the messenger so to have reach as many workers as possible, but at the same time its average movement direction shall not bring the messenger itself too far from a ruler, as this situation will threaten the ability of the system of propagating updated messages.

4.2. Implementation with ACE-JADE. We have implemented some new classes based on scenario issues. The list of the newly extended classes for this scenario is:

- `NasaAntsAgent:` specialization of the `Agent` which is implemented and used in the NASA ANTS project. Also, the `Capabilities` class is used to make differences between the three different kinds of the agents in NASA ANTS mission. The texttttNasaAntsAgent is therefore the messenger agent as introduced in the previous section.
- `SelfOptimizingBehaviour:` the extra structure which is needed for an algorithm to deal with environment and performing tasks for achieving *self-optimizing* is implemented in this class (first MAPE loop).
- `SelfProtectingBehaviour:` another structure which contains the *self-protecting* algorithm to maintain important connections of the *messenger* (second MAPE loop).

Also, besides the previous classes which have defined and instantiated in the system, the following classes are used in the case study:

- `AdaptiveLevelKnowledge`
- `Capabilities`
- `MonitorState`
- `AnalyzeState`
- `PlanState`
- `ExecuteState`
- `Sensor`
- `Effector`

Based on the location where the `NasaAntsAgent`'s instance is, it counts its nearby neighbors (within its range) in its monitoring phase and if the measured density is less than the defined density in the agent's *knowledge* (within the *analyze knowledge*, based on the planning *knowledge*, a random path will be picked and applied within the *execute* phase. It will cause a change in the location of the agent. This cycle will continue until the value of monitored neighbors exceeds the predefined number in analyze knowledge.

Simultaneously with the previous loop, another instance of a composite behaviour is running. This new object is instantiated from an implementation which supports agent's connectivity to important nodes. It will

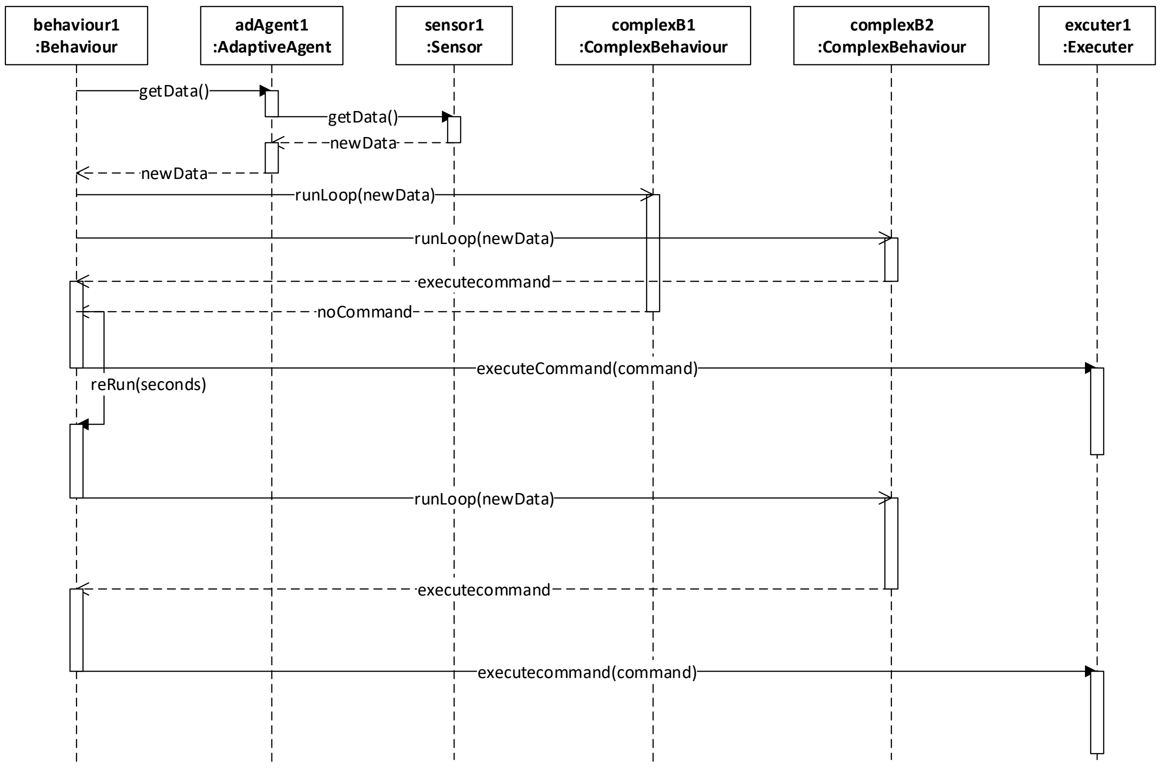


FIG. 4.1. *Sequence diagram of the implemented case study*

cover the self-protecting property and will go after being sure about continue connections which have been flagged as important connections. After the creation of the loop (by means of the appropriate *behaviour*) every defined period the behavior starts and measure the distance of the agent with flagged agents to be sure that the distance is not getting near to the agent's transmission rate. If so, it will start changing the speed of the agent in order to be sure that the other agents are not going to be farther away.

For better understanding the implementation, the sequence diagram of a system run is provided (see Fig. 4.1). The descriptions of objects are reported in the figure. In this sequence diagram two feedback loops are described (*self-optimizing* and *self-protecting*, as mentioned previously). In the first round, there is a run of the loop of `complexB1`, which is about the self-optimization. In this run, the result will be a set of commands (decisions about changes in the location) to improve the number of covered nodes. While this loop is running, another loop (`complexB2`) is called for running. The `complexB2` is about the self-protection. Because there was no change in the location, there is no result for that loop. But as it is reported in the diagram, another run is scheduled for this loop for some time later. After implementing the changes of the `complexB1` with the help of `executor1`, another round of `complexB2` running is started. In this round, because of changes in the location, the result is a correction of the location in order to be sure about being connected to important nodes. Afterward, this decision is applied with the help of `executor1`.

Meanwhile there are not similar researches which can be examined to compare our presented extension, we can compare the time and effort (number of classes) needed for implementing a system by standalone JADE with the proposed extension. This can give a overall proof of applicability and usefulness about the presented extension.

A NASA ANTS case study was implemented with ACE-JADE and also with standalone JADE. The time needed to implement the case-study with JADE was around 90 hours with 1 developer familiar with JADE. The same case study was prepared in around 75 hours with 1 developer which was familiar with ACE-JADE. The number of classes for JADE implementation was 16 classes and for ACE-JADE was 8. A point which should be considered is that ACE-JADE has more than 20 pre-implemented (or at least semi-implemented) classes with respect to JADE. Researchers think that with expanding the problems this improvement and time saving will have a bigger gap with developing a problem which needs autonomic computing as a necessary aspect with bare JADE.

5. Conclusion and Future Work. Having a multi-agent development environment that provides the support for autonomic computing features makes ease the process of creating adaptive multi-agent system. This support can shorten the development process and also can minimize the development expenses. Also, like any other development environment, using autonomic enabled development environment will increase the quality of the developed system.

An extension of JADE for having adaptive agents has been proposed in this paper. For this extension, changes concern the introduction of `AdaptiveAgent`, `Behaviour`, `Message` classes; also the `Sensor` and `Effector` classes are introduced as auxiliary classes to interact with the environment.

For a better explanation, a case study was introduced and implemented. NASA ANTS is a case study that is suitable for examining concepts in adaptive systems in distributed environment. A scenario for self-optimizing and self-protecting was implemented and explained in order to show the applicability of the JADE extension. For better understanding the proposed extension, two sequence diagrams have been discussed: one for describing the ordinary MAPE loop with details and another one is for describing the implemented case study.

In this extension, some aspects are still to be defined in a precise way. One of these aspects is about the usage of `Knowledge`. Besides the implementation of `Knowledge` class, further classes are needed to use it, for instance a *rule engine*; moreover, a *knowledge management* and a *knowledge reasoning* should be implemented by *rules* and *rule engines* in future work.

With regard to future work, we can sketch three directions. First, the introduction of more complex classes for behaviours to support complex adaptive properties (like self-healing) more easily. Also, the explicit implementation of the *knowledge* with well-known knowledge management framework for knowledge part in MAPE-K loop. Finally, the addition of more configurable details in order to support *domain specific* needs (for example, adapting this extension to Wireless Sensor Networks issues).

These kinds of extensions are important towards the definition of standards for AC in multi-agent systems.

REFERENCES

[1] M. Baldoni, G. Boella, M. Dorni, R. Grenna, and A. Mugnaini, *Adding Organizations and Roles as Primitives to the JADE Framework*, Proc. of WOA 2008: Dagli oggetti agli agenti, Evoluzione dell'agent development: metodologie, tool, piattaforme e linguaggi, (2008), pp. 84–92.

[2] C. Baumer, M. Breugst, S. Choy, and T. Magedanz, *Grasshoppera universal agent platform based on omg masif and fipa standards*, in First International Workshop on Mobile Agents for Telecommunication Applications (MATA99), Citeseer, 1999, pp. 1–18.

[3] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa, *Jade programmer's guide*, vol. 3, CSELT S.p.A., 2000.

[4] F. Bellifemine, A. Poggi, and G. Rimasa, *JADE: a FIPA2000 compliant agent development environment*, International Conference on Autonomous Agents and Multiagent Systems, (2001), pp. 216–217.

[5] F. Bellifemine, A. Poggi, and G. Rimassa, *JADE-A FIPA-compliant agent framework*, Proceedings of PAAM, (1999), pp. 97–108.

[6] F. Bellifemine, A. Poggi, and G. Rimassa, *Developing multi-agent systems with a fipa-compliant agent framework*, Software-Practice and Experience, 31 (2001), pp. 103–128.

[7] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao, *Able: A toolkit for building multiagent autonomic systems*, IBM Systems Journal, 41 (2002), pp. 350–371.

[8] I. Bojic, T. Lipic, M. Kusek, and G. Jezic, *Extending the JADE agent behaviour model with JBehaviourTrees Framework*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6682 LNAI (2011), pp. 159–168.

[9] S. Bouchenak, F. Boyer, N. De Palma, D. Hagimont, S. Sicard, and C. Taton, *JADE: A Framework for Autonomic Management of Legacy Systems*, Wiki.Jasmine.Ow2.Org, (2006), p. 20.

[10] G. Cabri, N. Capodieci, L. Cesari, R. De Nicola, R. Pugliese, F. Tiezzi, and F. Zambonelli, *Self-expression and dynamic attribute-based ensembles in scel*, in International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Springer, 2014, pp. 147–163.

[11] N. Capodieci, E. Hart, and G. Cabri, *Artificial immunology for collective adaptive systems design and implementation*, ACM Transactions on Autonomous and Adaptive Systems (TAAS), 11 (2016), p. 6.

[12] A. Computing, *An architectural blueprint for autonomic computing*, IBM White Paper, (2006).

[13] R. S. Cost, T. Finin, Y. Labrou, X. Luan, Y. Peng, I. Soboroff, J. Mayfield, and A. Boughannam, *Jackal: a java-based tool for agent development*, Working Papers of the AAAI-98 Workshop on Software Tools for Developing Agents, 1998.

[14] P.-C. David and T. Ledoux, *Towards a framework for self-adaptive component-based applications*, in IFIP International Conference on Distributed Applications and Interoperable Systems, Springer, 2003, pp. 1–14.

[15] N. H. Dhaminda B. Abeywickrama and F. Zambonelli, *Engineering and implementing software architectural patterns based on feedback loops*, Scalable Computing: Practice and Experience, 15 (2014), pp. 84–92.

[16] N. M. do Nascimento and C. J. P. de Lucena, *Fiot: An agent-based framework for self-adaptive and self-organizing applications based on the internet of things*, Information Sciences, 378 (2017), pp. 161–176.

[17] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao, *Autonomia: an autonomic computing environment*, in Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International, IEEE, 2003, pp. 61–68.

[18] A. Farahani, E. Nazemi, and G. Cabri, *Ace-jade, autonomic computing enabled jade*, in IEEE SASO 2016, IEEE, 2016.

[19] ———, *A self-healing architecture based on rainbow for industrial usage*, Scalable Computing: Practice and Experience, 17 (2016), pp. 351–368.

[20] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, *Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure*, IEEE Computer, 37 (2004), pp. 46–54.

[21] M. L. Griss, S. Fonseca, D. Cowan, and R. Kessler, *SmartAgent: Extending the JADE agent behavior model*, Proceedings of SEMAS, (2002), pp. 1–10.

[22] P. Horn, *autonomic computing : IBM's Perspective on the State of Information Technology*, IBM White Paper, (2001), pp. 1–10.

[23] B. Jacob, R. Lanyon-Hogg, D. K. Nadgir, and A. F. Yassin, *A Practical Guide to the IBM Autonomic Computing Toolkit*, IBM Redbooks, 2004.

[24] j. O. Kephart and D. M. Chess, *The vision of Autonomic Computing*, IEEE computer society, (2003), pp. 41–50.

[25] H. Liu, M. Parashar, and S. Hariri, *A componentbased progarmming framework for autonomic applications*, in Proceedings of the 1st IEEE International Conference on Autonomic Computing, 2004.

[26] Y. S. Lopes, E. J. T. Gonçalves, M. I. Cortés, and S. S. Emmanuel, *Extending JADE Framework to Support Different Internal Architectures of Agents*, in 9th European Workshop on Multi-agent Systems (EUMAS 2011), 2011, pp. 1–15.

[27] V. Markova, *Autonomous Agent Design Based On Jade Framework*, in Proceedings of the International Conference on Information Technologies (InfoTech-2013), no. September, 2013, pp. 19–20.

[28] P. O'Brien and R. Nicol, *FIPA - towards a standard for software agents*, BT Technology Journal, 16 (1998), pp. 51–59.

[29] D. Petcu, *A parallel rule-based system and its experimental usage in membrane computing*, Scalable Computing: Practice and Experience, 7 (2001).

[30] M. Puviani, G. Cabri, N. Capodieci, and L. Leonardi, *Building self-adaptive systems by adaptation patterns integrated into agent methodologies*, in International Conference on Agents and Artificial Intelligence, Springer, 2015, pp. 58–75.

[31] L. Raju, R. Milton, and S. Mahadevan, *Multiagent systems based modeling and implementation of dynamic energy management of smart microgrid using macsimjx*, The Scientific World Journal, 2016 (2016).

[32] C. R. Robinson, P. Mendham, and T. Clarke, *Macsimjx: A tool for enabling agent modelling with simulink using jade*, (2010).

[33] M. Salehie and L. Tahvildari, *Self-adaptive software: Landscape and research challenges*, ACM Transactions on Autonomous and Adaptive Systems, 4 (2009), pp. 14–56.

[34] W. Truszkowski, L. Hallock, J. Karlin, J. Rash, and G. Michael, *Autonomous and Autonomic Systems with Applications to NASA Intelligent Spacecraft Operations and Exploration Systems*, Springer, 2010.

[35] W. Truszkowski, M. Hinchey, J. Rash, and C. Rouff, *NASA's Swarm Missions: The challenge of Building Autonomous Software*, IT Pro, (2004), pp. 47–52.

[36] D. Weyns and R. Haesevoets, *The MACODO middleware for context-driven dynamic agent organizations*, ACM Transactions on Autonomous and Adaptive Systems (TAAS), 5 (2010).

[37] F. Zambonelli, *Self-management and the many facets of nonself*, IEEE Intelligent systems, 21 (2006), pp. 53–55.

Edited by: Dana Petcu
Received: December 24, 2016
Accepted: March 22, 2017