# On Programming and Policing Autonomic Computing Systems ⋆

M. Loreti[1], A. Margheri[1,2], R. Pugliese[1], and F. Tiezzi[3]

[1] Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy
[2] Università di Pisa, Largo Bruno Pontecorvo, 3 - 56127 Pisa, Italy
[3] IMT Advanced Studies Lucca, Piazza S. Francesco, 19 - 55100, Lucca, Italy

**Abstract.** To tackle the complexity of autonomic computing systems it is crucial to provide methods supporting their systematic and principled development. Using the PSCEL language, autonomic systems can be described in terms of the constituent components and their reciprocal interactions. The computational behaviour of components is defined in a procedural style, by the programming constructs, while the adaptation logic is defined in a declarative style, by the policing constructs. In this paper we introduce a suite of practical software tools for programming and policing autonomic computing systems in PSCEL. Specifically, we integrate a Java-based runtime environment, supporting the execution of programming constructs, with the code corresponding to the policing ones. The integrated, semantic-driven framework also permits simulating and analysing PSCEL programs. Usability and potentialities of the approach are illustrated by means of a robot swarm case study.

**Keywords:** Autonomic systems, Semantic-driven development tools, Robot swarms

## 1 Introduction

*Autonomic computing systems* [1] are self-managing computing systems, capable of autonomously adapting to unpredictable changes in order to achieve desired behaviours, while hiding at the same time intrinsic complexity to users. Since their first appearance they are becoming more common and integrated with a variety of other heterogeneous and interactive systems. The resulting systems usually include massive numbers of components, featuring complex interactions in open and non-deterministic environments. To enable systematic and principled development of autonomic computing systems it is then crucial to provide high level, linguistic abstractions – capable of describing how the different components are brought together to form the overall system architecture – together with a clear identification of the adaptation logic and an unambiguous account of the semantics.

⋆ This work has been partially sponsored by the EU project ASCENS (257414) and by the Italian MIUR PRIN project CINA (2010LHT4KM).

In this paper we introduce some software tools for programming and policing autonomic computing systems in PSCEL (*Policed SCEL*) [2]. This is a language with a formally defined semantics which results from the integration of SCEL and FACPL. SCEL [3] is one of the many languages for programming autonomic computing systems that have been proposed in the literature (see e.g. [4,5,6,7,8]). In SCEL, autonomic systems are programmed in terms of the constituent components and their reciprocal interactions. Components result from the aggregation of knowledge and behaviours, according to some policies. Knowledge acquisition and behaviour manipulation allow components to self-adapt. *Ensembles* of components are dynamically formed and referred to in communication actions by means of predicates over component *attributes*. These latter ones describe components' public features such as identity, functionalities, spatial coordinates, trust level, etc. that may dynamically change. FACPL [9,10] is a simple, yet expressive, language for defining access control, resource usage and adaptation policies. Policy specifications are intuitive and easy to maintain because of their declarative nature, therefore policy languages (see e.g. [11,6,12]) are receiving much attention in many research fields. In FACPL, policies are sets of rules specifying strategies, requirements, constraints, guidelines, etc. about the behaviour of systems and their components.

PSCEL appropriately integrates the linguistic abstractions of the two languages on which it is based. It is thus possible to develop autonomic computing systems in terms of software components capable of adapting their behaviour for reacting to new requirements or environment changes. For example, it is possible to define policies implementing adaptation strategies by exploiting specific actions that are produced at runtime as an effect of policy evaluation and are used to modify the behaviour of components. Moreover, policies can depend on the values of components' attributes (reflecting the status of components and their environment) and can be dynamically replaced as a reaction to system changes. Dynamically changing policies are indeed a powerful means for controlling, in a natural and clear way, the evolution of autonomic systems having a very high degree of dynamism, which in principle would be quite difficult to manage.

According to the *separation of concerns* principle, PSCEL design decouples the functional aspects from the adaptation ones. In fact, the application logic generating the computational behaviour of components is defined in a procedural style, by the programming constructs, while the adaptation logic is defined in a declarative style, by the policing constructs. At run-time, as clarified by the language operational semantics [13] and by the description of the supporting Java runtime environment (see Section 4), the adaptation actions generated by policy evaluation will be executed as part of components' behaviour.

The two languages at the basis of PSCEL come equipped with specific software tools providing development and run-time support to SCEL systems and FACPL policies, separately. In particular, SCEL programs can be executed and simulated in the jRESP environment. This environment provides an API allowing Java programs to use the SCEL linguistic constructs for controlling the computation and interaction of autonomic components, and for defining the ar-

| | | | |
|---|---|---|---|
| Systems: | $S$ | $::=$ | $\mathcal{I}[\mathcal{K}, \Pi, P] \mid S_1 \parallel S_2 \mid (\nu n)S$ |
| Processes: | $P$ | $::=$ | $\mathbf{nil} \mid a.P \mid P_1 + P_2 \mid P_1 \mid P_2 \mid X \mid A(\bar{p})$ |
| Actions: | $a$ | $::=$ | $\mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c \mid \mathbf{fresh}(n)$ |
| | | $\mid$ | $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ |
| Destinations: | $c$ | $::=$ | $n \mid x \mid \mathsf{self} \mid \mathcal{P} \mid p$ |
| Knowledge: | $\mathcal{K}$ | $::=$ | $\emptyset \mid \langle t \rangle \mid \mathcal{K}_1 \parallel \mathcal{K}_2$ |
| Items: | $t$ | $::=$ | $e \mid c \mid P \mid t_1, t_2$ |
| Templates: | $T$ | $::=$ | $e \mid c \mid ?x \mid ?X \mid T_1, T_2$ |

**Table 1.**Programming constructs (Policies $\Pi$ are in Table 2)

chitecture of systems and ensembles. jRESP API serves as a guidance to assist programmers in the implementation of autonomic systems, which turns out to be simplified with respect to using 'pure' Java. Finally, jRESP provides specific components that can be used to simulate and analyze SCEL programs. The development and the enforcement of FACPL policies, instead, are supported by an Eclipse IDE and a Java library for the policy evaluation process. Once the desired policies have been written with the IDE, they can be automatically transformed in Java classes according to the rules defining the FACPL's semantics.

The main contribution of this work is the definition of a practical tool suite supporting the development, execution, simulation and analysis of PSCEL programs. This is based on the integration of the Java code resulting from FACPL policies with the jRESP code corresponding to a SCEL system. In the integrated code, FACPL classes are invoked for authorizing interactions among components, while jRESP code is able to modify its workflow for executing the adaptation actions returned by policies evaluation. Usability and potentialities of this approach are illustrated by means of a simple, yet illustrative, case study of autonomic computing borrowed from the robotics domain. We show a complete specification of the case study, together with its simulation and analysis through jRESP.

The rest of the paper is organized as follows. Section 2 briefly reports the syntax of PSCEL. Section 3 presents the PSCEL specification of two scenarios of the robotics case study. Section 4 presents the development tools; it also describes the main features of jRESP and shows how it can be used to execute, as Java code, the PSCEL specification of the scenarios. Section 5 reviews more strictly related work. Finally, Section 6 concludes the paper by touching upon directions for future work.

## 2 PSCEL Syntax

In this section we review the syntax of PSCEL in two steps, by introducing first the constructs for programming autonomic computing systems and then the constructs for policing their behaviour. We also informally present the semantics of the different constructs (the interested reader is referred to [13] for a formal account of the semantics).

The constructs for programming autonomic computing systems are presented in Table 1. The key notion is that of *component* $\mathcal{I}[\mathcal{K}, \Pi, P]$ that consists of:

– An *interface* $\mathcal{I}$ publishing and making available structural and behavioural information about the component itself in the form of *attributes*, i.e. names acting as references to information stored in the component's repository.
– A *knowledge repository* $\mathcal{K}$ managing component's data.
– A set of *policies* $\Pi$ regulating the interaction with other components.
– A *process* $P$, together with a set of process definitions.

It is worth noticing that there is a clear separation of concerns: the normal computational behaviour of a component is defined in the process $P$, while the adaptation logic is defined in the policies $\Pi$. At runtime, the adaptation actions generated by the policy evaluation will be executed, of course, as part of the component's process.

We describe below the syntactic categories of the language.

Systems aggregate components through the *composition* operator, as in $S_1 \parallel S_2$. It is also possible to restrict the scope of a name, say $n$, by using the *name restriction* operator $(\nu n)S$.

Processes are the active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* $(a.P)$, *nondeterministic choice* $(P_1 + P_2)$, (interleaved) *parallel composition* $(P_1 \mid P_2)$, *process variable* $(X)$, and *parametrized process invocation* $(A(\bar{p}))$. Process variables can support *higher-order* communication, namely the capability to exchange (the code of) a process, and possibly execute it, by first adding an item containing the process to a knowledge repository and then retrieving/withdrawing this item while binding the process to a process variable. We let $A$ to range over a set of parametrized *process identifiers* that are used in recursive process definitions. We also assume that each process identifier $A$ has a *single* definition of the form $A(\bar{f}) \triangleq P$, with $\bar{p}$ and $\bar{f}$ denoting lists of actual and formal parameters, respectively.

Processes can perform five different types of actions. Actions $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$ and $\mathbf{put}(t)@c$ are used to manage shared knowledge repositories by withdrawing/retrieving/adding information items from/to the knowledge repository identified by $c$. These actions exploit templates $T$ to select knowledge items $t$ in the repositories. Action $\mathbf{fresh}(n)$ introduces a scope restriction for the name $n$ thus this name is guaranteed to be *fresh*, i.e. different from any other name previously used. Action $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ creates a new component $\mathcal{I}[\mathcal{K}, \Pi, P]$. Actions **get** and **qry** may cause the process executing them to wait for the wanted item if it is not (yet) available in the knowledge repository. The two actions differ for the fact that **get** removes the found item from the target repository while **qry** leaves the repository unchanged. Actions **put**, **fresh** and **new** can be instead immediately executed.

Knowledge items are *tuples*, i.e. sequences of values, while templates are sequences of values and variables. Knowledge repositories are then *tuple spaces*, i.e. (possibly empty) multisets of tuples. Values within tuples can either be destinations $c$, or processes $P$ or, more generally, can result from the evaluation of some given expression $e$. We assume that expressions may contain attribute names, *boolean*, *integer*, *float* and *string* values and variables, together with the corresponding standard operators. To pick a tuple out from a tuple space by

means of a given template, the *pattern-matching* mechanism is used: a tuple matches a template if they have the same number of elements and corresponding elements have matching values or variables; variables match any value of the same type ($?\,x$ and $?\,X$ are used to bind variables to values and processes, respectively), and two values match only if they are identical. If more tuples match a given template, one of them is arbitrarily chosen.

Different entities may be used as the DESTINATION $c$ of an action. As a matter of notation, $n$ ranges over component names, while $x$ ranges over variables for names. The distinguished variable $\mathsf{self}$ can be used by processes to refer to the name of the component hosting them. The destination can also be a *predicate* $\mathcal{P}$ or the name $p$, exposed as an attribute in the interface of the component, of a predicate that may dynamically change. A predicate is a boolean-valued expression obtained by applying standard operators to relations between components attributes and expressions.

In actions using a predicate $\mathcal{P}$ to indicate the destination (directly or via a name $p$), predicates act as 'guards' specifying *all* components that may be affected by the execution of the action, i.e. a component must satisfy $\mathcal{P}$ to be the target of the action. Thus, actions $\mathbf{put}(t)@n$ and $\mathbf{put}(t)@\mathcal{P}$ give rise to two different primitive forms of communication: the former is a *point-to-point* communication, while the latter is a sort of *group-oriented* communication. The set of components satisfying a given predicate $\mathcal{P}$ used as the destination of a communication action can be considered as the *ensemble* with which the process performing the action intends to interact. For example, to dynamically characterize the members of an ensemble that have the same role, say *landmark*, by assuming that attribute *role* belongs to the interface of any component willing to be part of the ensemble, one can write *role*="*landmark*".

Each action is executed only if it is authorized by the policies in force at the component willing to perform the action. The policies define authorization predicates, to grant or forbid actions, and *obligations*, i.e. actions that should be performed in conjunction with the enforcement of an authorization decision. They correspond to, e.g., updating a log file, sending a message, generating an event, setting an attribute. For example, if an action is forbidden due to unavailable resources, it can be needed to execute some other actions to reconfigure system's resources.

The constructs for policing autonomic computing systems are presented in Table 2. Notationally, symbol ? stands for optional elements, $*$ for (possibly empty) sequences, and $+$ for non-empty sequences. For the sake of readability, whenever an element is missing, we also omit the possibly related keyword; thus, e.g., we simply write $(d\ \ \mathsf{target}:\tau\,)$ in place of rule $(d\ \ \mathsf{target}:\tau\ \ \mathsf{condition}:\ \mathsf{obl}:)$.

A POLICY AUTOMATON $\Pi$ explicitly represents the fact that the policies in force at any given component can dynamically change while the component evolves. It is a pair $\langle\, A,\, \pi\,\rangle$, where

– $A$ is an automaton of the form $\langle \mathit{Policies}, \mathit{Targets}, \mathcal{T}\rangle$ where the set of states *Policies* contains all the POLICIES that can be in force at different times, the set of labels *Targets* contains the security relevant events (expressed as

| | | | |
|---|---|---|---|
| POLICY AUTOMATA: | $\Pi$ | ::= | $\langle A, \pi \rangle$ |
| POLICIES: | $\pi$ | ::= | $\langle \alpha$ target : $\tau^?$ rules : $r^+$ obl : $o^* \rangle$ |
| | | \| | $\{\alpha$ target : $\tau^?$ policies : $\pi^+$ obl : $o^* \}$ |
| COMBINING ALGORITHMS: | $\alpha$ | ::= | deny-overrides \| permit-overrides |
| | | \| | deny-unless-permit \| permit-unless-deny |
| | | \| | first-applicable \| only-one-applicable |
| RULES: | $r$ | ::= | $(d$ target : $\tau^?$ condition : $be^?$ obl : $o^*)$ |
| DECISIONS: | $d$ | ::= | permit \| deny |
| TARGETS: | $\tau$ | ::= | $f(pv,sn)$ \| $\tau \wedge \tau$ \| $\tau \vee \tau$ |
| MATCHING FUNCTIONS: | $f$ | ::= | equal \| not-equal \| greater-than |
| | | \| | less-than \| greater-than-or-equal |
| | | \| | less-than-or-equal \| pattern-match |
| OBLIGATIONS: | $o$ | ::= | $[\, d \;\; s \,]$ |
| OBLIGATION ACTIONS: | $s$ | ::= | $\epsilon$ \| $a.s$ |

**Table 2.** Policing constructs

TARGETS) that can trigger policy modification and the set of transitions $\mathcal{T} \subseteq (Policies \times Targets \times Policies)$ represents policy replacement.
– $\pi \in Policies$ is the current state of $A$.

A POLICY is either an atomic policy $\langle \ldots \rangle$ or a set of policies $\{\ldots\}$. An *atomic policy* (resp. *policy set*) is made of a target, a set of rules (resp. policy/policy sets) combined through one of the combining algorithms, and a set of obligations.

A TARGET indicates the *authorization requests* to which a policy/rule applies. It is either an atomic target or a pair of simpler targets combined using the standard logic operators $\wedge$ and $\vee$. An *atomic target* $f(pv,sn)$ is a triple denoting the application of a matching function $f$ to *policy values* $pv$ from the policy and to policy values from the evaluation context identified by *attribute (structured) names*[4] $sn$. In fact, an attribute name refers to a specific attribute of the request or of the environment, which is available through the evaluation context. In this way, an authorization decision can be based on some characteristics of the request, e.g. subjects' or objects' identity, or of the environment, e.g. presence of charging stations. For example, the target less-than($10\%$,*subject/batteryLevel*) matches whenever the battery level of the subject component is less than $10\%$. Similarly, the structured name *action/action-id* refers to the identifier of the action to be performed (such as **get**, **qry**, **put**, etc.) and, thus, the target equal(**qry**,*action/action-id*) matches whenever such an action is the retrieving one. Instead, for checking the content of the exchanged data in a communication action, via a template $T$, we can use the target pattern-match($T$,*action/item*).

Rules $(\ldots)$ are the basic elements for request evaluation. A RULE defines the tests that must be successfully passed by attributes for returning a positive or negative DECISION — i.e. permit or deny — to the enclosing policy. This decision is returned only if the target is 'applicable', i.e. the request matches the target; otherwise the evaluation of the rule returns not-applicable. Rule applicability can

[4] A structured name has the form *name/name*, where the first name stands for a category name and the second for an attribute name.

be further refined by the CONDITION expression *be*, which permits more complex calculations than those permitted in target expressions. *be* is a boolean term of the expression language used for defining item or template fields in Table 1, extended with policy values and structured names.

A COMBINING ALGORITHM computes the authorization decision corresponding to a given request by combining a set of rules/policies' evaluation results. PSCEL provides six algorithms but, due to lack of space, here we only present permit-unless-deny, which is used in the case study in Section 3 (the descriptions of the other algorithms is reported in [13]): if any rule/policy in the considered set evaluates to deny, then the result of the combination returned by permit-unless-deny is deny; otherwise, the result of the combination is permit (i.e., not-applicable is never returned).

An OBLIGATION is a sequence ($\epsilon$ denotes the empty one) of actions that should be performed in conjunction with the enforcement of an authorization decision. It is returned when the authorization decision for the enclosing element, i.e. rule, policy or policy set, is the same as the one attached to the obligation. An OBLIGATION ACTION is a process action which (with abuse of notation) may also contain structured names that are fulfilled during request evaluation. Thus, fulfilled obligation actions coincide with the (process) actions defined in Table 1. For example, the obligation $[\,\mathsf{deny}\ \mathbf{put}(\text{“}\mathsf{direction}\text{”}, env/station.x, env/station.y)@\mathsf{self}]$ could be fulfilled, w.r.t. a given request, as follows $\mathbf{put}(\text{“}\mathsf{direction}\text{”}, 10, 13)@\mathsf{self}$. It is used to set the robot's direction towards the position $(10, 13)$ corresponding to the location of a charging station perceived in the robot's environment.

## 3 PSCEL at Work on a Robot Swarm Case Study

In this section, we show the effectiveness of the PSCEL approach by modelling a robot swarm case study [14] defined in the EU project ASCENS [15]. We consider a scenario where a swarm of robots spreads throughout a given area where some kind of disaster has happened. The goal of the robots is to locate and rescue possible victims. As common in swarm robotics, all robots playing the same role execute the same code. According to the separation of concerns principle fostered by PSCEL, this code consists of two parts: *(i)* a process, defining the functional behaviour; and *(ii)* a collection of policies, regulating the interactions among robots and with their environment and generating the (adaptation) actions to react to specific (internal or environmental) conditions. This combination permits conveniently designing and enacting a collaborative swarm behaviour aiming at achieving the goal of rescuing the victims. We propose two different scenarios of the disaster case study that differ for the capabilities of the robots, mainly due to the availability of the GPS tracking system.

### 3.1 Scenario 1: different types of robot for different roles

The first scenario includes two different kinds of robots: *landmarks* and *workers*. Landmarks randomly explore the area of the disaster looking for victims. When

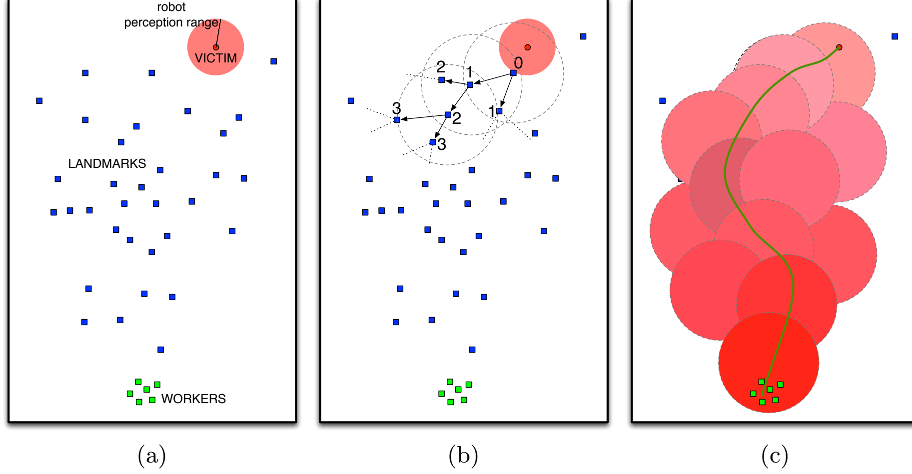


**Fig. 1.** Scenario 1: (a) scenario setting, (b) computational field creation, (c) computational field usage

a victim is found, its position is spread among landmarks, which stop to move. In this way, on the basis of the landmarks' positions and the information they receive, it is generated a sort of computational field [16] leading workers to the victim. Workers are the robots devoted to perform the actual rescuing task. They are initially motionless and are activated by informed landmarks. A graphical representation of the scenario, the creation of the computation filed and its use are depicted in Figure 1. For the sake of simplicity, we consider here just one victim and all workers go to rescue him when it is found. The scenario could be accommodated to deal with more victims by organizing landmarks and workers in different teams. We will deal with multiple victims in the next scenario.

This scenario can be modelled in PSCEL as

$$\textsc{Landmark}_1 \parallel \ldots \parallel \textsc{Landmark}_n \parallel \textsc{Worker}_1 \parallel \ldots \parallel \textsc{Worker}_m$$

where $\textsc{Landmark}_i$ and $\textsc{Worker}_j$ are PSCEL components of the form $\mathcal{I}_{L_i}[\mathcal{K}_{L_i}, \Pi_L, P_L]$ and $\mathcal{I}_{W_j}[\mathcal{K}_{W_j}, \Pi_W, P_W]$ modelling the two kinds of robots, respectively. Notably, all landmarks (resp. workers) enforce the same policy $\Pi_L$ (resp. $\Pi_W$) and execute the same process $P_L$ (resp. $P_W$).

In particular, the process run by a landmark is as follows

$$\begin{aligned} P_L \triangleq (\, & \mathbf{qry}(\text{“victimPerceived”}, \mathsf{true})@\mathsf{self}. \\ & \quad \mathbf{put}(\text{“victim”}, \mathsf{self}, 0)@\mathsf{self} \\ & + \mathbf{qry}(\text{“victim”}, ?id, ?d)@(role = \text{“}landmark\text{”}). \\ & \quad \mathbf{put}(\text{“victim”}, \mathsf{self}, d+1\text{”})@\mathsf{self}\,) \\ & \mid \mathit{RandomWalk} \mid \mathit{IsMoving} \end{aligned}$$

The landmark follows a random walk to explore the disaster area. To this aim, the process *RandomWalk* randomly selects a direction that is followed until either a wall is hit or a stop signal is sent to the wheels actuator. A landmark

stops when one of the following cases holds: *(i)* a victim is found (i.e., a tuple victimPerceived with value true is retrieved via a **qry** action from the local repository), or *(ii)* a message with the victim's position is published by a robot of the landmark ensemble. In the former case, the landmark starts the generation of the computational field, i.e. it publishes in its repository a victim tuple indicating that it is at distance 0 (measured in terms of 'number of hops') from the victim. In the latter case, instead, the robot non-deterministically retrieves a victim tuple from one robot of the landmark ensemble (i.e., the group of robots satisfying the predicate *role*="*landmark*") and locally publishes a victim tuple with the distance increased by one. It is worth noticing that the robots' range of communication is limited and, hence, the accessed ensemble may not contain all landmarks, but just the reachable ones. However, the range of communication is not explicitly specified in the PSCEL code, as well as in the jRESP one. Indeed, this is a physical constraint that will be only defined in the model of the physical scenario used by the jRESP simulation environment (see Section 4.2).

To stop a landmark immediately after the execution of one of the two **qry** actions, we define the following policy

$$
\begin{array}{l}
\langle\,\textsf{permit-unless-deny}\\
\quad \textsf{rules}:(\textsf{permit}\;\;\textsf{target}:\textsf{equal}(\mathbf{qry},action/action\text{-}id)\\
\qquad\qquad\qquad\qquad\qquad \wedge\;(\textsf{pattern-match}((\text{“victim”},_,_),action/item)\\
\qquad\qquad\qquad\qquad\qquad\quad \vee\;\textsf{pattern-match}((\text{“victimPerceived”},\textsf{true}),action/item))\\
\qquad\qquad\qquad \textsf{obl}:[\,\textsf{permit}\;\;\mathbf{put}(\text{“stop”})@\textsf{self}]\,)\,\rangle
\end{array}
$$

The policy contains a positive rule, whose only purpose is to return the obligation $\mathbf{put}(\text{“stop”})@\textsf{self}$ when one of the **qry** actions is executed. This action requests the wheels actuator to stop the movement.

The *RandomWalk* process calculates the random direction followed by the landmark for exploring the arena. When the proximity sensor signals a possible collision, by means of the tuple $\langle\text{“collision”},\textsf{true}\rangle$, a new random direction is calculated. This behaviour corresponds to the following PSCEL process

$$
\begin{array}{rl}
RandomWalk \triangleq & \mathbf{put}(\text{“direction”},2\pi\textsf{rand}())@\textsf{self}.\\
& \mathbf{qry}(\text{“collision”},\textsf{true})@\textsf{self}.RandomWalk
\end{array}
$$

The process defines only the direction of the motion not the will of moving.

During the movement, in order to check the level of charge of the battery and possibly halting the robot when the battery is low, we need to capture the movement status. This information is represented by the tuple $\langle\text{“isMoving”}\rangle$, produced by the wheels sensor, and monitored by the following process

$$
IsMoving \triangleq \mathbf{qry}(\text{“isMoving”})@\textsf{self}.IsMoving
$$

The reading of this datum is exploited by the following authorization rule (which must be added to the landmark's policy above)

$$
\begin{array}{l}
(\textsf{deny}\;\;\textsf{target}:\textsf{equal}(\mathbf{qry},action/action\text{-}id)\\
\qquad\qquad\quad \wedge\;\textsf{pattern-match}((\text{“isMoving”}),action/item)\\
\qquad\qquad\quad \wedge\;\textsf{less-than}(10\%,subject/batteryLevel)\\
\quad \textsf{obl}:[\,\textsf{deny}\;\;\mathbf{put}(\text{“stop”})@\textsf{self}]\,)
\end{array}
$$

to generate a stop action when the battery level is lower than 10%. In such a case, the robot will wait for new batteries and, eventually, restart the exploration.

Finally, the process for the worker is as follows

$$
\begin{aligned}
P_W \triangleq\ & \mathbf{qry}(\text{“victim”}, ?id, ?d)@(role{=}\text{“}landmark\text{”}).\\
& \mathbf{put}(\text{“start”})@\mathsf{self}.\\
& \mathbf{put}(\text{“direction”}, towards(id))@\mathsf{self}.\\
& \mathbf{while}(d > 0)\{d := d - 1.\\
& \qquad \mathbf{qry}(\text{“victim”}, ?id, d)@(role{=}\text{“}landmark\text{”}).\\
& \qquad \mathbf{put}(\text{“direction”}, towards(id))@\mathsf{self})\}.\\
& \mathbf{qry}(\text{“victimPerceived”}, \mathsf{true})@\mathsf{self}.\\
& \mathbf{put}(\text{“rescue”})@\mathsf{self}
\end{aligned}
$$

When the information about the discovery of a victim is retrieved by a worker (i.e., a victim tuple is read), the robot starts moving by following the direction indicated by the computational field defined by the landmarks. When the victim is reached, i.e. the tuple with distance 0 is read, the sensor perceives the victim and the worker starts the rescuing procedure.

For the worker process we do not report here any policy. Such policies could add additional actions when the worker is activated under specific conditions, e.g. a camera could be turned on in case there is enough daylight.

### 3.2 Scenario 2: the same type of robot for two different roles

In the second scenario of the case study, we consider robots with the same characteristics (in particular, all of them are equipped with a GPS tracking system) and capable of playing both the *explorer* and *rescuer* role. Thus, using its GPS, each robot can directly reach a given position (specified by coordinates $(x, y)$) and avoid the use of the computational field as in the previous scenario. A robot plays the *explorer* role during the exploration of the environment to locate the victim position, and the *rescuer* role when it is moving to reach a victim. Notably, the role changes according to the sensors and data values, e.g. this happens when the robot is close to a victim that needs help.

Therefore, this second scenario is modelled as a set of components ( $\text{ROBOT}_1 \parallel \ldots \parallel \text{ROBOT}_n$ ), where $\text{ROBOT}_i$ has the form $\mathcal{I}_{R_i}[\mathcal{K}_{R_i}, \Pi_R, P_R]$. Each robot initially plays the explorer role and possibly change it when victims are found. The behaviour of a single robot corresponds to the following PSCEL process

$$
\begin{aligned}
P_R \triangleq\ & (\ \mathbf{qry}(\text{“victimPerceived”}, \mathsf{true})@\mathsf{self}.\\
& \qquad \mathbf{put}(\text{“victim”}, x, y, 3)@\mathsf{self}.\ \mathbf{put}(\text{“rescue”})@\mathsf{self}\\
& \quad + \mathbf{get}(\text{“victim”}, ?x_v, ?y_v, ?count)@(role{=}\text{“}rescuer\text{”}).\ HelpRescuer\ )\\
& \mid RandomWalk \ \mid\ IsMoving
\end{aligned}
$$

Besides the processes *RandomWalk* and *IsMoving* still present for managing the movement, the robot recognises the presence of a victim by means of the **qry** action, while it helps other robots for rescuing a victim by means of the **get** action and according to the *HelpRescuer* process definition. When a victim is

found, an information about his position (retrieved by the attributes $x$ and $y$ of the robot's interface) and the number of other robots needed for rescuing him (3 robots in our case, but a solution with a varying number can be easily accommodated) is locally published.

The *HelpRescuer* process is defined as follows

$$\begin{aligned} \mathit{HelpRescuer} \triangleq\ & \mathbf{if}\ (\mathit{count} > 1)\ \mathbf{then}\ \{\ \mathbf{put}(\text{“victim”}, x_v, y_v, \mathit{count}\text{-}1)@\mathsf{self}\ \}. \\ & \mathbf{put}(\text{“direction”}, x_v, y_v)@\mathsf{self}. \\ & \mathbf{qry}(\text{“position”}, x_v, y_v)@\mathsf{self}.\ \mathbf{put}(\text{“rescue”})@\mathsf{self} \end{aligned}$$

This process is triggered by a victim tuple retrieved from the rescuers ensemble (see $P_R$). The tuple indicates that additional robots (whose number is stored in *count*) are needed at position $(x_v, y_v)$ to rescue a victim. If more than one robot is needed, a new victim tuple is published (with decremented counter). Then, the robot, which became a rescuer, goes towards the victim position and, once reaches him (i.e., the current position coincides with the victim's one), it starts the rescuing procedure. It is worth noticing that, if more victims are in the scenario, different groups of rescuers will be spontaneously organised to rescue them. To avoid that more than one group is formed for the same victim, we assume that the sensor of an explorer used to perceive the victim is configured so that a victim that is already receiving assistance by some rescuers is not detected as a victim.

An explorer changes its role to rescuer when it finds a victim or helps other rescuers. Each role corresponds to a different enforced policy, and the transition triggering the policy change is defined as follows

$$\textsc{Explorer} \xrightarrow[\qquad]{\begin{array}{l}(\ \mathsf{equal}(\mathbf{qry}, \mathit{action/action\text{-}id}) \\ \quad \wedge\ \mathsf{pattern\text{-}match}((\text{“victimPerceived”}, \mathsf{true}), \mathit{action/item})\ ) \\ \vee\ (\ \mathsf{equal}(\mathbf{get}, \mathit{action/action\text{-}id}) \\ \quad \wedge\ \mathsf{pattern\text{-}match}((\text{“victim”}, _, _, _), \mathit{action/item})\ )\end{array}} \textsc{Rescuer}$$

Thus, the explorer policy change to the rescuer one either when a victimPerceived tuple is read or when a victim tuple is consumed.

The policy enforced in the EXPLORER state is as follows

$$\begin{array}{l}\langle\ \mathsf{permit\text{-}unless\text{-}deny} \\ \quad \mathsf{rules}: (\mathsf{permit}\ \ \mathsf{target}: \mathsf{equal}(\mathbf{qry}, \mathit{action/action\text{-}id}) \\ \qquad\qquad\qquad\qquad \wedge\ \mathsf{pattern\text{-}match}((\text{“victimPerceived”}, \mathsf{true}), \mathit{action/item})) \\ \qquad\qquad\quad \mathsf{obl}: [\ \mathsf{permit}\ \ \mathbf{put}(\text{“stop”})@\mathsf{self}.\mathbf{put}(\mathit{role}, \text{“}\mathit{rescuer}\text{”})@\mathsf{self}]\ ) \\ \qquad\quad (\mathsf{deny}\ \ \mathsf{target}: \mathsf{equal}(\mathbf{qry}, \mathit{action/action\text{-}id}) \\ \qquad\qquad\qquad\quad \wedge\ \mathsf{pattern\text{-}match}((\text{“isMoving”}), \mathit{action/item}) \\ \qquad\qquad\qquad\quad \wedge\ \mathsf{less\text{-}than}(20\%, \mathit{subject/batteryLevel}) \\ \qquad\qquad\quad \mathsf{obl}: [\ \mathsf{deny}\ \ \mathbf{put}(\text{“direction”}, \mathit{env/station.x}, \mathit{env/station.y})@\mathsf{self}]\ )\rangle\end{array}$$

The first rule stops the robot when a victim is found and changes the interface attribute *role* to *rescuer* by means of the action $\mathbf{put}(\mathit{role}, \text{“}\mathit{rescuer}\text{”})@\mathsf{self}$. The second rule monitors the battery level and redirects the robot to the recharging station when the level is low. Notably, with respect to the previous scenario, the

battery level is considered low when it is less than 20%, which should ensure enough battery power to allow the explorer to reach the recharging station or to rescue the victim, if he would be found in the meanwhile. We assume that each robot can obtain the position of the charging station, which is retrieved here by means of the interface attributes $env/station.x$ and $env/station.y$.

The policy enforced in the RESCUER status is instead as follows

$$\begin{array}{l}\langle\, \mathsf{permit\text{-}unless\text{-}deny} \\ \quad \mathsf{rules} : (\mathsf{permit}\ \ \mathsf{target} : \mathsf{equal}(\mathbf{qry}, action/action\text{-}id) \\ \qquad\qquad\qquad\qquad \wedge\ \mathsf{pattern\text{-}match}((\text{“position”}, _, _), action/item)) \\ \qquad\qquad\quad \mathsf{obl} : [\,\mathsf{permit}\ \ \mathbf{put}(\text{“stop”})@\mathsf{self}]\,)\,\rangle \end{array}$$

The policy, as previously, stops the robot when the victim is reached.

## 4 Deployment and simulation of PSCEL programs

jRESP[5] is a Java runtime environment providing a framework for developing autonomic and adaptive systems according to the SCEL paradigm. Specifically, jRESP provides an API that permits using in Java programs the SCEL's linguistic constructs for controlling the computation and interaction of autonomic components, and for defining the architecture of systems and ensembles.

Like SCEL, jRESP has been designed to accommodate alternative instantiations of specific knowledge and policy managers that may change for tailoring to different application domains.

A detailed description of the jRESP architecture and its basic features can be found in [3]. In this section we will briefly present jRESP and its basic elements and the specific classes we have implemented to integrate FACPL in jRESP. The new classes, which specialize the jRESP architecture, have been included in a specific package enabling the execution of PSCEL programs.

**Components.** PSCEL components are implemented via the class PscelNode. Nodes are executed over virtual machines or physical devices providing access to input/output devices and network connections. A node aggregates a tuple space, a set of running processes, and a set of FACPL policies. Structural and behavioural information about a node are collected into an *interface* via *attribute collectors*. Nodes interact via *ports* supporting both *point-to-point* and *group-oriented* communications.

**Knowledge repository.** Since PSCEL specializes the knowledge repositories of SCEL's components as *tuple spaces*, the version of jRESP considered here provides an implementation of the interface Knowledge of PscelNodes. This is the class TupleSpace that defines the methods for withdrawing/retrieving/adding pieces of knowledge from/to repositories. Knowledge items are defined as *tuples*, i.e. sequences of Objects, that can be collected into a knowledge repository. They can be retrieved/withdrawn via pattern-matching through Templates, consisting of a sequence of actual and formal TemplateFields.

[5] jRESP website: `http://jresp.sourceforge.net/`.

External data can be collected into a knowledge repository via *sensors*. Each sensor can be associated to a logical or physical device providing data that can be retrieved by processes and that can be the subject of adaptation. Similarly, *actuators* can be used to send data to an external device or service attached to a node. This approach allows processes to control exogenous devices that identify logical/physical actuators.

The interface associated to a node is computed by exploiting *attribute collectors*. Each such collector is able to inspect the local knowledge and to compute the value of the attributes. This mechanism equips a node with *reflective capabilities* allowing a component to self-project the image of its state on the interface. Indeed, when the local knowledge is updated the involved collectors are *automatically* activated and the node interface is modified accordingly.

**Network Infrastructure.** Each PscelNode is equipped with a set of ports for interacting with other components. A port is identified by an *address* that can be used to refer to other jRESP components. Indeed, each jRESP node can be addressed via a pair composed of the node name and the address of one of its ports. The abstract class AbstractPort implements the generic behaviour of a port. It implements the communication protocol used by jRESP components to interact with each other. The class AbstractPort also provides the instruments to dispatch messages to components. However, in AbstractPort the methods used for sending messages via a specific communication network/media are abstract. Also the method used to retrieve the address associated to a port is abstract in AbstractPort. The concrete classes defining specific kinds of ports extend AbstractPort to provide concrete implementations of the above outlined abstract methods, so to use different underlying network infrastructures (e.g., Internet, Ad-hoc networks, ...). An additional instance, named VitualPort, is used to *simulate* nodes interaction within a single application without using a specific network infrastructure. Indeed, VirtualPort implements a port where interactions take place through a memory buffer.

**Behaviours.** Processes are implemented as threads via the abstract class Agent, which provides the methods implementing the PSCEL actions. In fact, they can be used for generating fresh names, for instantiating new components and for withdrawing/retrieving/adding items from/to shared knowledge repositories. The latter methods extend those considered in Knowledge with another parameter identifying either the (possibly remote) node where the target repository is located or the group of nodes whose repositories have to be accessed. As previously mentioned, group-oriented interactions are supported by the communication protocols defined in the node ports and by attribute collectors.

### 4.1 Integration of FACPL in jRESP

In jRESP policies are used to regulate the interaction between the different internal parts of components and their mutual interactions. Indeed, when a method of an instance of the class Agent is invoked, its execution is delegated to the policy associated to the node where the agent is running. The policy can then control

the execution of the action (for instance, by suspending a behaviour when some access rights are missing) and, possibly, define additional *behaviours*. Different kinds of policies can be easily integrated in jRESP by implementing the interface IPolicy. Currently, two implementations of this latter interface are included in jRESP: NodePolicy and PolicyAutomaton. NodePolicy is the policy enforced by default in each node. It always allows any operations, thus directly delegating the execution of each action to the associated node. PolicyAutomaton implements instead a generic POLICY AUTOMATON $\Pi$ (like those presented in Section 2). In this way, transitions caused by the execution of agent actions can trigger changes of the policies. In particular, a PolicyAutomaton consists of a set of PolicyStates, each of which identifies the possible policies enforced in the node, and of a reference to the current state, which is used to *evaluate* agent actions with respect to the current policies. This automaton can be easily integrated with various policy languages, although here we focus on its integration with FACPL policies.

The full integration of FACPL in jRESP can be now achieved by considering the class FacplPolicyState that, by extending PolicyState, relies on the Java-translated FACPL policies. This Java code is automatically obtained by using the FACPL IDE available for the Eclipse platform from the FACPL website [17].

When a PolicyAutomaton receives a request for the execution of a given action, first of all an AutorisationRequest is created. This is the object identifying the PSCEL action the node wants to perform, thus it provides information about the kind of action performed, its argument, its target and the list of attributes currently published in the node interface. The created AuthorizationRequest is then evaluated with respect to the current policy state via the (abstract) method evaluate(AutorisationRequest r) defined in the class PolicyState. In the class FacplPolicyState this method delegates the authorization to the referred FACPL policy. The method returns an instance of the class AuthorisationResponse, which contains a *decision*, i.e. permit or deny, and a set of *obligations*. The latter ones are rendered as a sequence of Actions that must be performed just after the completion of the requested action. Hence, if the decision is permit, the requested action is completed as soon as the obligations are executed. Instead, if the decision is deny, the requested action cannot be performed. In this case, first the obligations possibly returned along with the decision must be executed, then a new AutorisationRequest is created and evaluated in order to establish executability of the requested action.

Finally, the evaluation of a request by a PolicyAutomaton can trigger an update of its current state. Indeed, for each state, a sequence of *transitions* are stored in the automaton. These are instances of the class PolicyAutomatonTransition that provides two methods: apply(AutorisationRequest r): boolean and nextState(): PolicyState. A transition is *enabled* if the first method returns true. The next state is then obtained by invoking nextState() on the first enabled transition. If no transition is enabled, the current state is not changed.

In the first scenario of Section 3 the PolicyAutomaton associated to each PscelNode contains only a single state; this is the FacplPolicyState that interacts with the considered Java-translated FACPL policies. Instead, in the second

scenario, the PolicyAutomaton consists of two states that enforce *explorer* and *rescuer* behaviour, respectively. The PolicyAutomatonTransition associated in the automaton to the *explorer* state is the following:

```
public class ExplorerToRescuer implements PolicyAutomatonTransition {
	public boolean apply( AutorisationRequest req ) {
		return ( (req.getActionId() == ActionID.QUERY)
			&& (new Template(
				new ActualTemplateField( "VICTIM_PERCEIVED" ) ,
				new ActualTemplateField( true ) ).match( req.getItem() )))
			||((req.getActionId() == ActionID.GET)
			&& (new Template(
				new ActualTemplateField( "VICTIM" ) ,
				new FormalTemplateField( Object.class ) ,
				new FormalTemplateField( Object.class ) ,
				new FormalTemplateField( Object.class ) ).match( req.getItem() )));
	}
	public PolicyState nextState() { return new FacplPolicyState( new Policy_Rescuer() ); }
}
```

In the code above, Policy_Rescuer is the Java-translated FACPL policy associated to the policy presented at the end of Section 3.

### 4.2 Simulating robots in jRESP

To support analysis of adaptive systems specified in PSCEL, the jRESP environment provides a set of classes that permits simulating jRESP *programs*. These classes enable the execution of *virtual components* over a simulation environment that can control component interactions and collect relevant simulation data. In fact, although in principle jRESP code could be directly executed in real robots (provided that a Java Virtual Machine is running on them and that jRESP's sensors and actuators invoke the API of the corresponding robots' devices), this may not be always possible. Therefore, jRESP also provides simulation facilities.

To set-up the simulation environment in jRESP one has first of all to define a class that provides the machinery to manage the physical data of the scenario. These data include, e.g., robots position, direction and speed. In our case, we consider the class ScenarioArena that, in addition to the above mentioned data, also provides the methods for updating robots position and computing collisions. These methods are periodically executed by the jRESP simulation environment. For the sake of simplicity, in the simulation, only collisions with the borders of the arena are considered, while collisions among robots are ignored.

For both scenarios, in jRESP we consider a network of PscelNodes each of which identifies a single robot. We assume that each robot/node is equipped with sensors, like *collision* and *victim* detection sensors, which are used to retrieve information about the state of the robot and of its working environment. All the above mentioned sensors are built via the class ScenarioArena and permit to directly access the data associated to the state of the simulated physical environment. Similarly, each instance of PscelNode modelling a robot is equipped with *actuators* used to control robots movement, like *direction* and *stop* actuators. Also these actuators are built via the class ScenarioArena and permit to update the parameters of the simulated physical environment when the corresponding data are received. For instance, when the *RandomWalk* process running at the

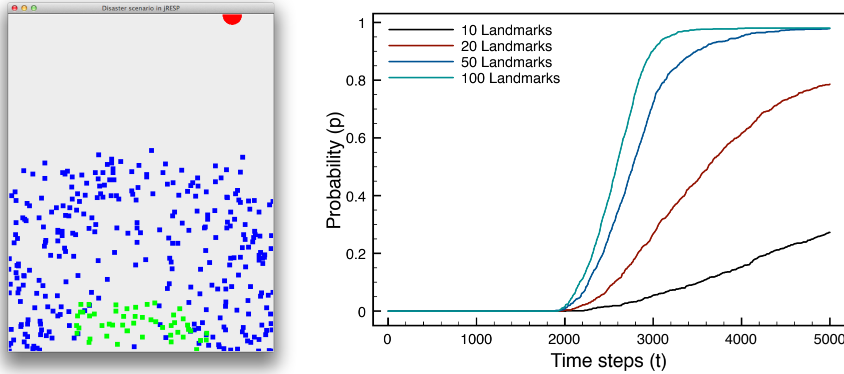


**Fig. 2.** Simulation and analysis of the first robot swarm scenario in jRESP

node corresponding to robot $i$ produces a tuple of the form $\langle$"*direction*"$, dir\rangle$, the local direction actuator sets to *dir* the direction of the robot $i$ in the ScenarioArena. This behaviour mimics the fact that in a *real* robot the actuator directly interacts with the wheels controller.

Each PscelNode also executes the agents presented in the previous sections. For instance, the *RandomWalk* process is rendered in jRESP as reported below:

```
public class RandomWalk extends Agent {
    Random r = new Random();
    public RandomWalk() { super("RandomWalk"); }
    @Override
    protected void doRun() throws IOException, InterruptedException{
        while (true) {
            double dir = r.nextDouble()∗2∗Math.PI;
            put( new Tuple( "direction" , dir) , Self.SELF );
            query( new Template( new ActualTemplateField( "COLLISION" ) ,
                    new ActualTemplateField( true ) ), Self.SELF );
        }
    }
}
```

By relying on the jRESP simulation environment, a prototype framework for *statistical model-checking* has been also developed. A randomized algorithm is used to verify whether the implementation of a system satisfies a specific property with a certain degree of confidence. Indeed, the statistical model-checker is parameterized with respect to a given *tolerance* $\varepsilon$ and *error probability* $p$. The used algorithm guarantees that the difference between the computed value and the exact one is greater than $\varepsilon$ with a probability that is less than $p$.

The model-checker included in jRESP can be used to verify *reachability properties*. These permit evaluating the probability to reach, within a given deadline, a configuration where a given predicate on collected data is satisfied. In our first scenario, this analysis technique is used to study how the number of landmark robots affects the probability to reach the victim within a given deadline.

In Figure 2, we report a screenshot of the robots simulation (left-hand side) and the results of the analysis (right-hand side). In the screenshot, a red semicircle represents the locations of the victim, while blue (dark grey in b/w print) and green (light grey in b/w print) squares represent landmark and worker

robots, respectively. The analysis results are represented as a chart showing the probability of rescuing the victim within a given time according to different numbers of landmark robots (i.e., 10, 20, 50 and 100). Notably, the victim can be rescued only after 2000 time steps and, beyond a certain threshold, increasing the number of robots is not worthy (in fact, the difference in terms of rescuing time between 100 and 50 robots is marginal with respect to the cost of deploying a double number of robots).

## 5 Related work

Autonomic computing systems are currently studied within many research communities. To deal with such systems different approaches have been advocated both for programming them, like multi-agent systems, component-based design and context-oriented programming, and for regulating their behavior, mainly through policy languages. Below, we mention the most closely related works.

Multi-agent systems (as e.g. [18,19,4]) pursue the importance of the knowledge representation and how it is handled for choosing adaptive actions. PSCEL, instead, bases the knowledge repository implementation on tuple-spaces, which is a more flexible and lightweight mechanism to, e.g., support adaptive context-aware activities in pervasive computing scenarios.

Component-based design has been indicated as a key approach for adaptive software design [20]. A relevant example in this field is Fractal [21], a hierarchical component model that, in addition to standard component-based systems, permits defining systems with a less rigid structure by means of components without completely fixed boundaries. However, communication among components is still defined via connectors and system adaptation is obtained by adding, removing or modifying components and/or connectors. Communication and adaptation in PSCEL, instead, are more flexible, and, hence, more adequate to deal with highly dynamic ensembles.

Another paradigm advocated to program autonomic systems [22] is Context-Oriented Programming (COP) [23]. It exploits ad-hoc linguistic constructs to define context-dependent behavioral variations and their run-time activation. The most of the literature on COP is devoted to the design and implementation of concrete programming languages (a comparison can be found in [24]). Only few works provide a foundational account, like e.g. [25], focussing on an object-oriented language extended with COP facilities. All these approaches are however quite different from ours, that instead focusses on distribution and attribute-based aggregations and supports a highly dynamic notion of adaptation regulated by policies.

As concerns policy languages, many such languages have been recently developed for managing different aspects of programs' behaviour as, e.g., adaptation and autonomic computing. For example, a policy-based approach to autonomic computing issues has also been proposed by IBM through a simplified policy language [12], which, however, comes without a precise syntax and semantics. [6] introduces PobSAM, a policy-based formalism that combines an actor-based

model, for specifying the computational aspects of system elements, and a configuration algebra, for defining autonomic managers that, in response to changes, lead the adaptation of the system configuration according to given adaptation policies. This formalism relies on a predefined notion of policies expressed as Event-Condition-Action (ECA) rules. Adaptation policies are specific ECA rules that change the manager configurations. PSCEL constructs for defining policies, being strictly integrated with a powerful autonomic programming language, is more flexible and expressive permitting not only to produce adaptation actions, but also authorisation controls and resource assignments. Moreover, the full integration of obligation actions with the programming constructs permits a runtime code generation and, hence, enables more flexible adaptation strategies. A policy language for which a number of toolkits have been developed and applied to various autonomous and pervasive systems is Ponder [11]. The language uses two separate types of policies for authorisation and obligation. Policies of the former type have the aim of establishing if an operation can be performed, while those of the latter type basically are ECA rules. Differently from Ponder, and similarly to more recent languages (e.g. XACML), in PSCEL obligations are expressed as part of authorisation policies, thus providing a more uniform specification approach.

Finally, the international standard XACML, which FACPL is inspired to, defines policy specifications in XML format without a formal description of the evaluation process. FACPL instead has a compact and intuitive syntax and is endowed with a formal semantics based on solid mathematical foundations. These features, as well as its supporting software tools, make FACPL easy to learn and use. This motivates our choice of FACPL as policy language to be integrated with the programming constructs provided by SCEL.

## 6 Conclusion

In this paper we tackled the issue of practically programming and policing autonomic computing systems. To this aim, we propose the use of the formal language PSCEL, which fosters an approach based on the 'separation of concerns' principle. Indeed, on the one hand, the behaviour of autonomic components and their ensembles are programmed through the SCEL constructs. On the other hand, the interactions between components and the adaptation actions to be performed in reaction to changes in their working environment are regulated by means of FACPL policies. From a practical perspective, SCEL specifications can be implemented in Java by relying on the jRESP runtime environment, while FACPL policies can be developed and automatically translated in Java by using a specific tool suite. The main contribution of this paper is the integration of these Java-based tools in order to provide a uniform software framework for the development and execution of PSCEL programs. In order to illustrate how jRESP supports simulation and analysis of autonomic systems specified in PSCEL, we have exploited a simple case study from the robotics domain.

As a future work, we plan to improve the practical applicability of the PSCEL approach by extending the Eclipse-based IDE for FACPL policies with the possibility of defining SCEL specifications. In this way, an autonomic system will be completely specified at high-level of abstraction using PSCEL's constructs and then automatically transformed in a Java application integrating the code corresponding to SCEL behaviours and FACPL policies. Moreover, to assess the potentialities of PSCEL tools, we also plan to consider other application domains and case studies among those developed within the ASCENS project, concerning cooperative e-vehicles and cloud systems.

## References


1. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer **36** (2003) 41–50
2. Margheri, A., Pugliese, R., Tiezzi, F.: Linguistic Abstractions for Programming and Policing Autonomic Computing Systems. In: UIC/ATC, IEEE (2013) 404–409
3. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. ACM TAAS (2014) To appear.
4. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems **16**(3) (2008) 214–248
5. Ashley-Rollman, M.P., Goldstein, S.C., Lee, P., Mowry, T.C., Pillai, P.: Meld: A declarative approach to programming ensembles. In: IROS, IEEE (2007) 2794–2800
6. Khakpour, N., Jalili, S., Talcott, C.L., Sirjani, M., Mousavi, M.R.: Formal modeling of evolving self-adaptive systems. Sci. Comput. Program. **78**(1) (2012) 3–26
7. Lanese, I., Bucchiarone, A., Montesi, F.: A framework for rule-based dynamic adaptation. In: TGC. LNCS 6084, Springer (2010) 284–300
8. Banâtre, J.P., Radenac, Y., Fradet, P.: Chemical Specification of Autonomic Systems. In: IASSE, ISCA (2004) 72–79
9. Margheri, A., Masi, M., Pugliese, R., Tiezzi, F.: A Formal Software Engineering Approach to Policy-based Access Control. Technical report, Univ. Firenze (2013) `http://rap.dsi.unifi.it/facpl/research/Facpl-TR.pdf`.
10. Masi, M., Pugliese, R., Tiezzi, F.: Formalisation and implementation of the XACML access control mechanism. In: ESSoS. LNCS 7159, Springer (2012) 60–74
11. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. In: POLICY. LNCS 1995, Springer (2001) 18–38
12. IBM: Autonomic Computing Policy Language - ACPL `http://www.ibm.com/developerworks/tivoli/tutorials/ac-spl/`.
13. Margheri, A., Pugliese, R., Tiezzi, F.: Linguistic Abstractions for Programming and Policing Autonomic Computing Systems. Technical report, Univ. Firenze (2013) `http://rap.dsi.unifi.it/scel/pdf/PSCEL-TR.pdf`.
14. N. Serbedzija et al.: Integration and simulation report for the Ascens case studies. D7.3 (2013) `http://www.pst.ifi.lmu.de/~mayer/papers/2013-11-30_D73.pdf`.
15. EU project ASCENS: `http://www.ascens-ist.eu/`.
16. Mamei, M., Zambonelli, F., Leonardi, L.: Co-fields: A physically inspired approach to motion coordination. IEEE Pervasive Computing **3**(2) (2004) 52–61
17. FACPL website: `http://rap.dsi.unifi.it/facpl`
18. Winikoff, M.: Jack$^{\text{tm}}$ intelligent agents: An industrial strength platform. In: Multi-Agent Programming. Volume 15. Springer (2005) 175–193

19. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley Series in Agent Technology. John Wiley & Sons (2007)
20. McKinley, P., Sadjadi, S., Kasten, E., Cheng, B.H.C.: Composing adaptive software. Computer **37**(7) (2004) 56–64
21. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The FRACTAL component model and its support in Java. Softw., Pract. Exper. **36**(11-12) (2006) 1257–1284
22. Salvaneschi, G., Ghezzi, C., Pradella, C.: Context-Oriented Programming: A Programming Paradigm for Autonomic Systems. CoRR **abs/1105.0069** (2011)
23. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology **7**(3) (2008) 125–151
24. Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., Perscheid, M.: A comparison of context-oriented programming languages. In: COP, ACM (2009) 6:1–6:6
25. Hirschfeld, R., Igarashi, A., Masuhara, H.: ContextFJ: a minimal core calculus for context-oriented programming. In: FOAL, ACM (2011) 19–23