2013 IEEE 10th International Conference on Ubiquitous Intelligence & Computing and 2013 IEEE 10th International Conference on Autonomic & Trusted Computing

Linguistic Abstractions for Programming and Policing Autonomic Computing Systems

Andrea Margheri
Università degli Studi di Firenze, Università di Pisa
andrea.margheri@unifi.it, margheri@di.unipi.it

Rosario Pugliese
Università degli Studi di Firenze
rosario.pugliese@unifi.it

Francesco Tiezzi
IMT Advanced Studies Lucca
francesco.tiezzi@imtlucca.it

***Abstract*—We introduce PSCEL, a new language for developing autonomic software components capable of adapting their behaviour to react to external stimuli and environment changes. The application logic generating the computational behaviour of systems 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. The interplay between these two kinds of constructs permits to dynamically produce and enforce adaptation actions. To show PSCEL practical applicability and effectiveness, we employ it in a Cloud Computing case study.**

I. Introduction

Nowadays, many computing systems include massive numbers of components, featuring complex interactions in open and non-deterministic environments. Moreover, they are more and more integrated with a variety of other heterogeneous and interactive systems, also connected to humans and to the external environment. To address the challenges of developing, integrating, and deploying these large-scale, complex software-intensive systems, *self-adaptation* has been advocated as a key feature. This is the ability of a system to autonomously adapt its behaviour and/or structure to dynamic operating conditions [1]. It is at the basis of self-management capabilities like self-configuration, self-healing, self-optimization and self-protection (also known as *self-** properties) typical of *autonomic computing systems* [2].

To enable the systematic and principled development of self-adaptive systems, a high level, linguistic description of how the different components are brought together to form the overall system architecture, together with a clear identification of the adaptation logic is worthwhile. Therefore, different linguistic abstractions have been proposed in the literature (see e.g. [3], [4], [5], [6], [7]). Here we rely on SCEL (Software Component Ensemble Language) [8], [9], a language expressly designed for programming autonomic computing systems in terms of the constituent components and their reciprocal interactions. In SCEL, systems and components result from the aggregation of knowledge and behaviours, according to some policies. Self-adaptation is enabled by knowledge acquisition and behaviour manipulation, and can be implemented through appropriate regulating policies. *Ensembles* of components are dynamically formed and referred to in communication actions by means of predicates over components' *attributes*. These describe components' public features such as identity, functionalities, spatial coordinates, trust level, etc. that may dynamically change. By using these abstractions, the behaviour of a whole system, that emerges from the behaviour of its individual components, adapts itself to new requirements or environment conditions.

SCEL main aim is to identify linguistic constructs for uniformly modeling the architecture of autonomic systems, the control of computation, and the interaction among possibly heterogeneous components. Therefore, to enhance flexibility and better support self-adaptation in different application domains, SCEL is parametric with respect to the policy language although its semantics is defined so as to take policies into account and to be compatible with many different ways to define them.

Recently, *policy languages* (see e.g. [10], [5], [11]) are receiving much attention in different research fields. In fact, their declarative nature makes policy specifications intuitive and easy to maintain. Here we consider FACPL (Formal Access Control Policy Language) [12], a simple, yet expressive, language for defining access control, resource usage and adaptation policies, which is inspired to the XACML [13] standard for access control. In FACPL, policies are sets of rules specifying strategies, requirements, constraints, guidelines, etc. about the behaviour of systems and their components.

By integrating SCEL and FACPL we have designed a new language for programming and policing autonomic software components capable of adapting their behaviour to react to external stimuli and environment changes. This language, that we call PSCEL (Policed SCEL), takes advantage of the features of the two inspiring languages and appropriately integrates their linguistic abstractions. In PSCEL, it is for example possible to define policies implementing adaptation strategies by exploiting actions that are produced at run-time as an effect of policy evaluation and are used to modify the behaviour of systems components. Furthermore, policies can depend on the values of components attributes (reflecting the status of components and their environment) and can be dynamically replaced for better reacting to system changes.

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, the adaptation actions generated by policy evaluation will be executed as part of components' behaviour.

The rest of the paper is organised as follows. In Section II we review the related work and highlight the main differences

978-1-4799-2481-3/13 $31.00 © 2013 IEEE
DOI 10.1109/UIC-ATC.2013.33

with respect to PSCEL. Syntax and informal semantics of our linguistic abstractions are presented in Sections III. In Section IV we use a Cloud Computing case study for illustrating the effectiveness of our approach. Finally, Section V concludes the paper by touching upon directions for future work.

II. Related work

Autonomic computing systems are currently studied within many research communities. To deal with such systems different approaches are used, like multi-agent systems, component-based design and context-oriented programming. Below, we mention some of the most closely related works. These proposals however usually concern programming, rather than regulating, the behavior of such systems.

Multi-agent systems (as e.g. [14], [15], [3]) 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 [16]. A relevant example in this field is FRACTAL [17], 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 [18] is Context-Oriented Programming (COP) [19]. It exploits ad-hoc linguistic constructs to express 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 [20]). Only few works provide a foundational account, like e.g. [21], focussing on an object-oriented language extended with COP facilities, and [22], focussing on a functional one. 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 [11], which, however, comes without a precise syntax and semantics. [5] 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 expressing 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 expressing policies, being strictly integrated with an expressive 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 run-time 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 [10]. The language borrows the idea introduced in [23] of using 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.

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	$::=$	$n \mid x \mid \mathsf{self} \mid \mathsf{P} \mid \mathsf{p}$
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 I. PROGRAMMING CONSTRUCTS (POLICIES Π ARE IN TABLE II)

III. PSCEL: Programming and Policing Autonomic Computing Systems

We present PSCEL in two steps: we first introduce the constructs for programming autonomic computing systems and then those for policing their behaviour. We refer the interested reader to [24] for a full account of the formal semantics.

A. Constructs for programming autonomic computing systems

The constructs are presented in Table I. 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. Among them, attribute id is mandatory and is bound to the component's name.

- A *knowledge repository* $\mathcal{K}$ managing application data, internal status data (supporting *self-awareness*) and environmental data (supporting *context awareness*). The knowledge repository of a component stores also the information associated to its interface, which therefore can be dynamically manipulated by means of the operations provided by the knowledge repositories' handling mechanisms.
- A set of *policies* Π regulating the interaction among the component and the others. By exploiting policies, on the one hand, components can protect themselves against unauthorised access, hence behaving in a *self-protecting* way. On the other hand, they can detect specific conditions regarding themselves and their (execution) context, and trigger appropriate adaptation actions, hence behaving in a *(self-)adaptive* way.
- A *process* P, together with a set of process definitions that can be dynamically activated. Processes in P execute local computations, coordinate interaction with the knowledge repository and with the other components.

It is worth noticing that the normal computational behaviour of a component is defined in P, while the adaptation logic is defined in Π. At run-time adaptation actions will be generated by policy evaluation and become part of the process run by the component. We describe below the syntactic categories of the programming constructs.

SYSTEMS aggregate components through the *composition* operator, as in $S_1 \parallel S_2$. It is 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.

In PSCEL, 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. Notationally, n ranges over component names, while x ranges over variables for names. The distinguished variable self can be used by processes to refer to the name of the component hosting them. The destination can also be a *predicate* 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 the results returned by the evaluation of relations between components' attributes and expressions.

In actions using a predicate P to indicate the destination (directly or via p), predicates act as 'guards' specifying *all* components that may be affected by the execution of the action, i.e. a component must satisfy P to be the target of the action. Thus, actions $\mathbf{put}(t)@n$ and $\mathbf{put}(t)@\mathsf{P}$ 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 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 characterise the members of an ensemble that are located in the same area, say IMT, by assuming that attribute $location$ belong to the interface of any component willing to be part of the ensemble, one can write $location = IMT$.

In PSCEL, when an action is considered for execution, a corresponding *authorisation request* is generated. This request is evaluated with respect to the policies in force at the component willing to perform the action and at the destination component(s). In particular, when the destination of an action **put** denotes multiple repositories satisfying the action predicate, each insertion in these repositories must be authorised separately by each component; such evaluation, however, does not affect the authorisation of the insertions in the other destination repositories. Instead, in case of actions **get** or **qry**, only one authorisation is required from the destination side, since only one repository is selected for the interaction. Thus, PSCEL policies regulate (intra- or inter-components) interactions by simply enabling or disabling behaviours and by

POLICY AUTOMATA:	Π	::=	$\langle\, A,\, \pi \,\rangle$
POLICIES:	π	::=	$\langle \alpha$ target : $\tau^?$ rules : r^+ obl : $o^* \,\rangle$
		|	$\{ \alpha$ target : $\tau^?$ policies : π^+ obl : $o^* \}$
COMBINING ALGORITHMS:	α	::=	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:	τ	::=	$f(pv,sn) \mid \tau \wedge \tau \mid \tau \vee \tau$
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	::=	$\epsilon \mid a.s$

TABLE II. POLICING CONSTRUCTS

dynamically adding new actions to components' behaviours as result of policies evaluation.

B. Constructs for policing autonomic computing systems

The policing constructs are presented in Table II. As a matter of notation, ? 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 $(d$ target : $\tau\,)$ in place of rule $(d$ target : τ condition : obl : $)$.

To explicitly represent the fact that the policies in force at any given component can dynamically change while the component evolves, we use a sort of automata somehow reminiscent of the *security automata* [25]. Thus, a POLICY AUTOMATON Π is a pair $\langle\, A,\, \pi \,\rangle$, where:

- A is an automaton of the form $\langle Policies, Targets, \mathcal{T} \rangle$ 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 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 authorisation 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 the *policy value*[1] pv from the policy and to policy values from the evaluation context identified by the *attribute (structured) name*[2] 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 authorisation decision can be based on some characteristics of the request, e.g. subjects' or objects' identity, or of the environment, e.g. CPU load. For example, the target greater-than$(90\%, subject/CPUload)$ matches whenever the CPU load of the subject component is greater than 90%. Similarly, the structured name $action/action\text{-}id$ refers to the identifier of the action to be performed (such as **get**, **qry**, **put**, etc.) and, thus, the target equal(**get**,$action/action\text{-}id$) matches whenever such action is the withdrawing 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 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 I, extended with policy values and structured names.

A COMBINING ALGORITHM computes the authorisation 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 IV (and relegate the descriptions of the other algorithms to [24]). 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 (ϵ denotes the empty one) of actions that should be performed in conjunction with the enforcement of an authorisation decision. It is returned when the authorisation decision for the enclosing element, i.e. rule, policy or policy set, is the same as the one attached to the obligation. These actions correspond to, e.g., updating a log file, sending a message, generating an event, setting an attribute. For example, if an execution path is forbidden due to unavailable resources, it can be needed to execute some other actions to reconfigure systems resources. 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 of Table I. E.g., the obligation action

$$\mathbf{put}(\text{“}taskEnd\text{”}, \mathsf{self}, env/time)@(role = gateway)$$

could be fullfilled, w.r.t. a given request, as follows

$$\mathbf{put}(\text{“}taskEnd\text{”}, \mathsf{self}, Aug\ 26\ 14:42:28)@(role = gateway)$$

It can be used to notify the completion time of a task to all components playing the gateway role.

To sum up, policies, and their evaluation, are hierarchically structured as trees: the evaluation of leaf nodes, i.e. rules, return a 'starting' decision, permit, deny or not-applicable, while the intermediate nodes, i.e. policy or policy sets, combine the decisions and obligations returned by the evaluation of their

[1] The set of policy values, besides the values that can be used within evaluated knowledge items, also contains action identifiers (i.e., **get**, **qry**, **put**, **fresh** and **new**), items and templates.

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

child nodes through the chosen combining algorithm. Policy evaluation terminates when the root is reached producing a decision and a sequence of obligations. This sequence consists of fulfilled actions that will enforce the consequences resulting from the authorisation process.

IV. PSCEL AT WORK ON A CLOUD CASE STUDY

We consider the *High Load* scenario from the *Science Cloud* case study [26] defined in the ASCENS project [27]. The Science Cloud is a collection of notebooks, desktops, servers, or virtual machines running the Science Cloud Platform. Each (virtual) machine is running (usually) one Science Cloud Platform instance (SCPi). Each SCPi is considered to be a service component. Multiple SCPis communicate over the Internet, thus forming a Cloud. We consider a small-size setup of the Science Cloud where a group of SCPis is located at IMT Lucca, another group at LMU Munich, and another one (running on top of a mobile device) at the English Garden in Munich. For each group, one stable member plays the role of gateway, consisting in collecting information about the whole Cloud and, if necessary, notifying the other members.

In the High Load scenario, a singleton application currently runs on one of the SCPis at IMT Lucca, so no additional instances of the application can be spawned in the Cloud. This application runs alone on its node and experiences consistently high CPU load. When the processing power is not enough to fulfil the application requirements, an adaptivity decision is made to instantiate a new Virtual Machine (VM) running a SCPi. Once this VM is up and the enclosed SCPi has joined the Science Cloud, the singleton application is moved there.

To illustrate the main features of PSCEL, we assume that the considered singleton application behaves as follows: it cyclically retrieves a task from the ensemble of SCPis located in the same area, say IMT Lucca, executes it and sends the result back to the task's owner. Moreover, when it is moved to the new VM, it sends a message to the ensemble made of all those SCPis having a gateway role for communicating its new locality to every nodes.

The whole scenario can be rendered as the following PSCEL system

$$\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \;\parallel\; \ldots \;\parallel\; \mathcal{I}_n[\mathcal{K}_n, \Pi_n, P_n]$$

where each component represents an SCPi of the Cloud. Let us consider the component $\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1]$ corresponding to the SCPi where the considered singleton application is running. The application is modelled as the following process[3]:

$$P_1 \triangleq \mathbf{get}(\text{“}task\text{”}, ?taskId, ?owner, ?X)@(location{=}IMT).\\ (\; X \;\mid\; \mathbf{get}(\text{“}result\text{”}, ?res)@\mathsf{self}.\\ \mathbf{put}(\text{“}result\text{”}, taskId, res)@owner.\, P_1 \;)$$

The process performs a *group-oriented* **get** to (non-deterministically) retrieve a task from a member of the ensemble of SCPis within the IMT area. This ensemble is dynamically determined when the action is executed and consists of all components that expose in their interface the $location$ attribute with value IMT. A task is any process Q, stored in a tuple of the form $\langle \text{“}task\text{”}, taskId, owner, Q\rangle$, that terminates its execution by locally producing a tuple of the form $\langle \text{“}result\text{”}, res\rangle$ storing the task result. Then, the process P_1 sends the retrieved task process (bound to the process variable X) for execution and waits until it terminates. The result is retrieved via a local **get** action and sent to the task's owner via a *point-to-point* **put** action, before restarting.

We assume that, other than id and $location$, the components interface exposes the attribute $CPUload$. Its value is considered to be high when it is greater than a given threshold, i.e. 90%. $CPUload$ stores a *context information*, updated by the underlying infrastructure and 'sensed' by the component, whose modification can trigger a self-adaptive behaviour.

The policy Π_1 in force at the component is the pair $\langle A, \pi\rangle$, where π is defined as follows:

$$\begin{array}{l}\langle\,\mathsf{permit\text{-}unless\text{-}deny}\\ \quad \mathsf{rules} : (\mathsf{deny}\;\; \mathsf{target} : \mathsf{equal}(\mathbf{get}, action/action\text{-}id)\\ \qquad\qquad\qquad\qquad \wedge\, \mathsf{pattern\text{-}match}((\text{“}task\text{”}, \_, \_, \_), action/item)\\ \qquad\qquad\qquad\qquad \wedge\, \mathsf{greater\text{-}than}(90\%, subject/CPUload)\\ \qquad\qquad\quad \mathsf{obl} : [\,\mathsf{deny}\;\; \mathbf{fresh}(n').\,\mathbf{new}(\mathcal{J}, \mathcal{K}_1, \Pi_1, P_1)\,]\,)\\ \quad \mathsf{obl} : [\,\mathsf{deny}\;\; \mathbf{put}(\text{“}newVM\text{”}, \mathsf{self}, n')@(role = gateway)\,]\,\rangle\end{array}$$

where interface $\mathcal{J}$ is like $\mathcal{I}_1$ but for $\mathcal{J}.id = n'$. Basically, π says that a new task can be retrieved (via the first **get** action) until $subject/CPUload$ is less than, or equal to, the threshold 90%. All other actions, including those performed by the retrieved tasks, are always allowed (algorithm permit-unless-deny). Note that choosing another algorithm means that a different combining strategy is enforced, thus the choice is application-dependent. Moreover, when a **get** action attempts to retrieve a new task and the threshold is exceeded, i.e. the policy evaluates to deny, three obligation actions are returned by the policy evaluation. The first one is a **fresh** action that generates a new identifier. This is used by the second action, a **new**, that creates a new component $\mathcal{J}$ (corresponding to the new VM) storing the same knowledge as $\mathcal{I}_1$, enforcing the same policies and running the same singleton application. The third one is a **put** action that notifies all SCPi having a gateway role that a new VM machine has been created (the identifiers of the old and new components are also specified).

Notably, π depends on the run-time value of attribute $CPUload$. This means that PSCEL can express policies that may depend on the value of some parameters and can thus dynamically change according to the context. This is already a form of dynamism; a more expressive form is obtained by exploiting the fact that π is a state of the policy automaton A. Indeed, A has two states: the initial 'active' state π, and the 'passive' state π_{off} defined as

$$\begin{array}{l}\langle\,\mathsf{permit\text{-}unless\text{-}deny}\\ \quad \mathsf{rules} : (\mathsf{deny}\;\mathsf{target} : \mathsf{equal}(\mathbf{get}, action/action\text{-}id)\\ \qquad\qquad\qquad \wedge\, \mathsf{pattern\text{-}match}((\text{“}task\text{”}, \_, \_, \_), action/item)\;)\;\rangle\end{array}$$

and only one transition

$$\pi \xrightarrow{\begin{array}{l}\mathsf{equal}(\mathbf{get}, action/action\text{-}id)\\ \wedge\, \mathsf{pattern\text{-}match}((\text{“}task\text{”}, \_, \_, \_), action/item)\\ \wedge\, \mathsf{greater\text{-}than}(90\%, \mathsf{subject/CPUload})\end{array}} \pi_{off}$$

The policy automaton ensures that whenever a new component is created and the application is moved there, if the run-time value of attribute $CPUload$ of the 'old' component decreases and becomes less than 90%, the application instance running there cannot resume its execution (this is necessary because the considered application is required to behave as a singleton).

[3]For the sake of simplicity, the first component is running only the considered singleton application, while the others may run different applications.

More specifically, when the target of the rule specified within π is matched, i.e. when the **get** action is blocked and the new VM is created, the state of the automaton evolves to π_{off}, which always disallows the old component to retrieve new tasks, regardless the value of attribute $CPUload$.

To sum up, when a task is retrieved and the threshold is exceeded, the system performs a computation step according to the obligations produced; thus the component $\mathcal{I}_1$ evolves to:

$$\mathcal{I}_1[\mathcal{K}_1, \langle A, \pi_{off} \rangle, \\ (\mathbf{fresh}(n').\, \mathbf{new}(\mathcal{J}, \mathcal{K}_1, \Pi_1, P_1). \\ \mathbf{put}(\text{“}newVM\text{”}, \mathsf{self}, n')@(role = gateway).\, P_1)]$$

The three process actions produced as obligations by the evaluation are prefixed to the continuation P_1 of the process running at the component. This ensures that the actions will be executed before the execution of the process may resume (in our case, this latter event is prevented by the new enforced policy π_{off}). We refer to [24] for the PSCEL specification of a variant of this scenario.

V. Concluding remarks and future directions

We have presented PSCEL, a novel language for programming and policing autonomic computing systems, that relies not only on programming constructs but also on declarative policies to easily express adaptation strategies. In PSCEL the policy-based concepts, emerged from different research fields, are integrated with the notion of ensemble *à la* SCEL. PSCEL with respect to SCEL specifies a particular policy language and introduces the concept of obligations for enforcing adaptation. One advantage of the proposed approach is the 'separation of concerns': components functionalities are taken apart from adaptation rules, thus providing flexible system abstractions and various modelling strategies. Moreover, the formal semantics lays the basis for developing logics, tools and methodologies to formally reason, on the one hand, on systems behaviour for establishing qualitative and quantitative properties and, on the other hand, on adaptation policies for predicting and validating adaptation actions to perform.

As future work, we plan to improve the practical applicability of PSCEL by introducing an UML-profile for developing policies and interactions among autonomic components. To assess the potentialities of PSCEL we also plan to consider other application domains and case studies among those developed within the ASCENS project, concerning cooperative e-vehicles and robot swarms. We also want to develop a run-time environment for PSCEL, by relying on the jRESP framework [28] that already provides a SCEL implementation.

Acknowledgements

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

References

[1] D. Weyns and T. Holvoet, "An Architectural Strategy for Self-Adapting Systems," in *SEAMS*. IEEE, 2007, p. 3.

[2] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, pp. 41–50, 2003.

[3] M. Dastani, "2APL: a practical agent programming language," *Autonomous Agents and Multi-Agent Systems*, vol. 16, no. 3, pp. 214–248, 2008.

[4] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *IROS*. IEEE, 2007, pp. 2794–2800.

[5] N. Khakpour, S. Jalili, C. L. Talcott, M. Sirjani, and M. R. Mousavi, "Formal modeling of evolving self-adaptive systems," *Sci. Comput. Program.*, vol. 78, no. 1, pp. 3–26, 2012.

[6] I. Lanese, A. Bucchiarone, and F. Montesi, "A framework for rule-based dynamic adaptation," in *TGC*, ser. LNCS 6084. Springer, 2010, pp. 284–300.

[7] J.-P. Banâtre, Y. Radenac, and P. Fradet, "Chemical Specification of Autonomic Systems," in *IASSE*. ISCA, 2004, pp. 72–79.

[8] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, "SCEL: a language for autonomic computing," Univ. Firenze, Tech. Rep., 2013, http://rap.dsi.unifi.it/scel/pdf/SCEL-TR.pdf.

[9] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese, "A Language-based Approach to Autonomic Computing," in *FMCO 2011*, ser. LNCS 7542. Springer, 2012, pp. 25–48.

[10] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," in *POLICY*, ser. LNCS 1995. Springer, 2001, pp. 18–38.

[11] IBM, "Autonomic Computing Policy Language - ACPL," http://www.ibm.com/developerworks/tivoli/tutorials/ac-spl/.

[12] A. Margheri, M. Masi, R. Pugliese, and F. Tiezzi, "A Formal Software Engineering Approach to Policy-based Access Control," Univ. Firenze, Tech. Rep., 2013, http://rap.dsi.unifi.it/facpl/research/Facpl-TR.pdf.

[13] OASIS XACML TC, "eXtensible Access Control Markup Language (XACML) version 3.0," September 2012.

[14] M. Winikoff, "Jack$^{\text{tm}}$ intelligent agents: An industrial strength platform," in *Multi-Agent Programming*, ser. Multiagent Systems, Artificial Societies, and Simulated Organizations 15. Springer, 2005, pp. 175–193.

[15] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE*, ser. Wiley Series in Agent Technology. John Wiley & Sons, 2007.

[16] P. McKinley, S. Sadjadi, E. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[17] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The FRACTAL component model and its support in Java," *Softw., Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[18] G. Salvaneschi, C. Ghezzi, and C. Pradella, "Context-Oriented Programming: A Programming Paradigm for Autonomic Systems," *CoRR*, vol. abs/1105.0069, 2011.

[19] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented programming," *Journal of Object Technology*, vol. 7, no. 3, pp. 125–151, 2008.

[20] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid, "A comparison of context-oriented programming languages," in *COP*. ACM, 2009, pp. 6:1–6:6.

[21] R. Hirschfeld, A. Igarashi, and H. Masuhara, "ContextFJ: a minimal core calculus for context-oriented programming," in *FOAL*. ACM, 2011, pp. 19–23.

[22] P. Degano, G. Ferrari, L. Galletta, and G. Mezzetti, "Types for coordinating secure behavioural variations," in *COORDINATION*, ser. LNCS 7274. Springer, 2012, pp. 261–276.

[23] M. Sloman, "Policy driven management for distributed systems," *J. Network Syst. Manage.*, vol. 2, no. 4, pp. 333–360, 1994.

[24] A. Margheri, R. Pugliese, and F. Tiezzi, "Linguistic abstractions for programming and policing autonomic computing systems," Univ. Firenze, Tech. Rep., 2013, http://rap.dsi.unifi.it/scel/pdf/PSCEL-TR.pdf.

[25] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.

[26] P. Mayer, C. Kroiss, and J. Velasco, "Specification: The Science Cloud Case Study," ASCENS Tech. Rep., 2012, http://svn.pst.ifi.lmu.de/trac/scp/raw-attachment/wiki/WikiStart/CloudCaseStudySpecV1.pdf.

[27] EU project ASCENS, "Autonomic service-component ensembles," http://www.ascens-ist.eu/.

[28] M. Loreti, "jRESP," http://jresp.sourceforge.net.