A formal approach to autonomic systems programming: The SCEL Language

ROCCO DE NICOLA, IMT Institute for Advanced Studies Lucca
MICHELE LORETI, Università degli Studi di Firenze
ROSARIO PUGLIESE, Università degli Studi di Firenze
FRANCESCO TIEZZI, IMT Institute for Advanced Studies Lucca

The autonomic computing paradigm has been proposed to cope with size, complexity and dynamism of contemporary software-intensive systems. The challenge for language designers is to devise appropriate abstractions and linguistic primitives to deal with the large dimension of systems, and with their need to adapt to the changes of the working environment and to the evolving requirements. We propose a set of programming abstractions that permit to represent behaviors, knowledge and aggregations according to specific policies, and to support programming context-awareness, self-awareness and adaptation. Based on these abstractions, we define SCEL (Software Component Ensemble Language), a kernel language whose solid semantic foundations lay also the basis for formal reasoning on autonomic systems behavior. To show expressiveness and effectiveness of SCEL's design, we present a Java implementation of the proposed abstractions and show how it can be exploited for programming a robotics scenario that is used as a running example for describing features and potentials of our approach.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Autonomic computing, Programming languages, Formal methods

ACM Reference Format:
Rocco De Nicola, Michele Loreti, Rosario Pugliese and Francesco Tiezzi, 2014. A formal approach to autonomic systems programming: The SCEL Language. *ACM Trans. Autonom. Adapt. Syst.* V, N, Article A (January YYYY), 29 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Modern software-intensive, cyber-physical systems have to deal with massive numbers of components, featuring complex interactions among components and with humans and other systems. Moreover, they have to operate in open and non-deterministic environments, and to dynamically adapt to new requirements, technologies and environmental conditions. They fall within none of the previously used classes of complex systems, therefore it has been proposed to refer to them using a new term, namely *ensembles* [Project InterLink 2007]. Sometimes ensembles are explicitly created by

This work has been partially supported by the EU projects ASCENS (257414) and QUANTICOL (600708) and by the MIUR PRIN project CINA (2010LHT4KM).
Authors' addresses: R. De Nicola and F. Tiezzi, IMT, Institute for Advanced Studies Lucca, Piazza San Francesco 19, I-55100, Lucca, Italy; Michele Loreti and Rosario Pugliese, Università degli Studi di Firenze, Viale Morgagni 65, I-50134, Firenze, Italy.

DOI: http://dx.doi.org/10.1145/0000000.0000000

design. Some other times they are assembled from systems that are independently controlled and managed, while their interaction "mood" might be cooperative or competitive; then one has to deal with systems coalitions or so-called *systems of systems*. Due to their inherent complexity, today's engineering methods and tools do not scale well with ensembles and new engineering techniques are needed to address the challenges of developing, integrating, and deploying them [Sommerville et al. 2012].

A possible answer to the problems posed by such complex systems is to make them able to self-manage by continuously monitoring their behavior and their working environment and by selecting the actions to perform for best dealing with the current status of affairs. Self-management could be exploited also to face situations in which humans intervention is limited or even absent and components have to collaborate to achieve specific goals. This requires increasing systems' self-management capabilities and guaranteeing what now are known as *self-** properties (self-configuration, self-healing, self-optimization, self-protection) of *autonomic computing* [Kephart and Chess 2003; IBM 2005].

The challenge for language designers is to devise appropriate abstractions and linguistic primitives to deal with the large dimension of systems, to guarantee adaptation to (possibly unpredicted) changes of the working environment, to take into account evolving requirements, and to control the emergent behaviors resulting from complex interactions. In this paper, we propose facing this challenge by taking as starting point the notions of *autonomic components* (ACs) and *autonomic-component ensembles* (ACEs) and defining programming abstractions to model their evolutions and their interactions. Building on these notions, we define SCEL (Software Component Ensemble Language), a kernel language that takes a holistic approach to programming autonomic computing systems and aims at providing programmers with a complete set of linguistic abstractions for programming the behavior of ACs and the formation of ACEs, and for controlling the interaction among different ACs. As shown in Section 2, these abstractions permit describing autonomic systems in terms of *Behaviors*, *Knowledge* and *Aggregations*, by complying with specific *Policies*.

SCEL is, somehow, minimal; its syntax fully specifies only constructs for modeling Behaviors and Aggregations and is parametric with respect to Knowledge and Policies. This choice permits integrating different approaches to policies specifications or to knowledge handling within our language and to easily superimpose ACEs on top of heterogeneous ACs. Indeed, we see SCEL as a 'kernel' language based on which different full-blown languages can be designed. Later in this paper, we will present a simple, yet expressive, language for defining access control policies. We will also consider a SCEL's dialect where knowledge repositories are implemented as multiple distributed *tuple-spaces*. This dialect is supported by a Java runtime environment, to be used for developing autonomic and adaptive systems according to the SCEL paradigm. Specifically, the runtime environment provides a library that permits using SCEL constructs in Java programs for controlling computations and interactions of ACs, and for defining the architecture of ACs and ACEs.

We consider our work as the blending of different concepts that have emerged in different fields of Computer Science and Engineering. Indeed, we have learnt from software engineering the importance of component-based design and of separation of concerns [McKinley et al. 2004], from multi-agent systems the relevance of knowledge handling and of spatial representation [Rao 1996; Bordini et al. 2005; Winikoff 2005; Bellifemine et al. 2007; Dastani 2008], from middleware and network architectures the importance of flexibility in communication [Mottola and Picco 2006; Costa et al. 2009; Mamei and Zambonelli 2009; Nordström et al. 2009; Mottola and Picco 2012], from distributed systems' security the role of policies [NIST 2009], from actors and process algebras the importance of minimality and formality [Agha 1990; Milner 1989]. What

we consider as our main contribution is the actual choice of the specific programming abstractions and their reconciliation under a single roof with a uniform formal semantics. What we offer then is a new language with appropriate programming abstractions for autonomic computing.

This work is an extended and revisited version of [De Nicola et al. 2012]. Here, ACEs are dynamically 'synthesized' via group-oriented, attribute-based communication and used as target of actions, while in [De Nicola et al. 2012] they are explicitly created by an ensemble coordinator that exploits specific interface attributes. This change, apart for permitting a more dynamic characterization of ensembles, avoids centralization and single point of failures. It also simplifies the actual semantics, because interactions do not require intervention of a third party (the ensemble coordinator). Another key difference between the two contribution is that now policies can be dynamically modified to take into account new requirements and to adapt to changing environments. Moreover, in [De Nicola et al. 2012] no specific policy language was considered and there was no description of the supporting Java runtime environment.

Structure of the paper. The rest of the paper is organized as follows. Section 2 introduces the design principles at the basis of SCEL; this section contains also a simple, yet illustrative, scenario of autonomic computing borrowed from the robotics domain, which will be used as a running example for describing the different features of the language. Syntax and operational semantics of SCEL are presented in Section 3 and Section 4, respectively. Section 5 outlines a language for defining access control policies and shows how it can be integrated with SCEL. Section 6 illustrates expressiveness and potentialities of SCEL by presenting the complete specification of the robotics scenario. Section 7 describes the main features of the Java runtime environment and shows how it can be used to execute, as Java code, the SCEL specification of the scenario. Section 8 reviews more strictly related work. Finally, Section 9 concludes by also touching upon directions for future work.

2. DESIGN PRINCIPLES

Autonomic Components (ACs) and Autonomic-Component Ensembles (ACEs) are our means to structure systems into well-understood, independent and distributed building blocks that interact and adapt.

ACs are entities with dedicated knowledge units and resources; awareness is guaranteed by providing them with information about their state and behavior via their knowledge repositories. These repositories can be also used to store and retrieve information about ACs working environment, and thus can be exploited to adapt their behavior to the perceived changes. Each AC is equipped with an *interface*, consisting of a collection of *attributes*, describing component's features such as identity, functionalities, spatial coordinates, group memberships, trust level, response time, etc.

Attributes are used by the ACs to dynamically organize themselves into ACEs. Indeed, one of the main novelties of our approach is the way groups of partners are selected for interaction and thus how ensembles are formed. Individual ACs can single out communication partners by using their identities, but partners can also be selected by taking advantage of the attributes exposed in the interfaces. Predicates over such attributes are used to specify the targets of communication actions, thus permitting a sort of *attribute-based* communication. In this way, the formation rule of ACEs is endogenous to ACs: members of an ensemble are connected by the interdependency relations defined through predicates. An ACE is therefore not a rigid fixed network but rather a highly flexible structure where ACs' linkages are dynamically established.

A typical scenario that gives rise to ACEs is reported in Figure 1. It suggests that ACEs can be thought of as logical layers (built on top of the physical ACs network) that

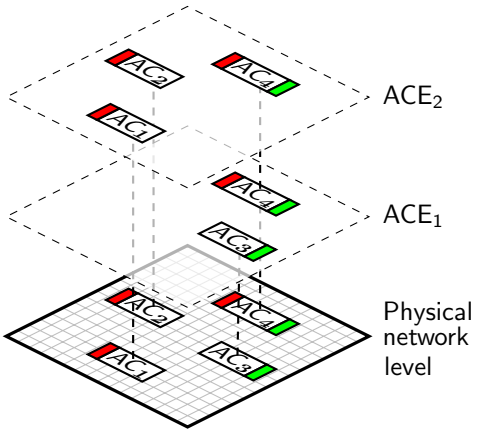

Fig. 1. Autonomic Component Ensembles

identify dynamic (overlay) subnetworks of ACs by exploiting specific attributes; in the picture, these are the different colours associated to individual ACs.

We have thus identified some linguistic abstractions for uniformly programming the evolution and the interactions of ACs and the architecture of ACEs. These abstractions permit describing autonomic systems in terms of Behaviors, Knowledge and Aggregations, according to specific Policies.

— *Behaviors* describe how computations may progress and are modeled as processes executing actions, in the style of process calculi.
— *Knowledge* repositories provide the high-level primitives to manage pieces of information coming from different sources. Each knowledge repository is equipped with operations for *adding*, *retrieving*, and *withdrawing* knowledge items.
— *Aggregations* describe how different entities are brought together to form ACs and to construct the software architecture of ACEs. Composition and interaction are implemented by exploiting the attributes exposed in ACs' interfaces.
— *Policies* control and adapt the actions of the different ACs for guaranteeing accomplishment of specific tasks or satisfaction of specific properties.

By accessing and manipulating their own knowledge repository or the repositories of other ACs, components acquire information about their status (*self-awareness*) and their environment (*context-awareness*) and can perform *self-adaptation*, initiate *self-healing* actions to deal with system malfunctions, or install *self-optimizing* behaviors. All these *self-** properties, as well as *self-configuration*, can be naturally expressed by exploiting SCEL's higher-order features, namely the capability to store/retrieve (the code of) processes in/from the knowledge repositories and to dynamically trigger execution of new processes (as shown by the example in Section 6). Moreover, by implementing appropriate security policies, e.g. limiting information flow or external actions, components can set up *self-protection* mechanisms against different threats, such as unauthorised access or denial-of-service attacks.

Our aim is to provide a common semantic framework for describing meaning and interplay of the abstractions above, while minimizing overlaps and incompatibilities. We shall illustrate the main features of SCEL in a step-by-step fashion by using a running example from the swarm robotics domain that is described below.

A swarm robotics scenario. We consider a robot swarm where robots are distributed over a physical area and have to reach different target zones according to the tasks assigned to them, such as rescue people in danger, help other robots, reach

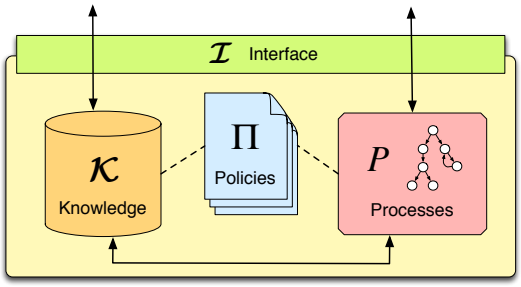

Fig. 2. SCEL component

a safe area, clear a minefield, etc. In the considered scenario, each robot of the swarm has to fulfill one of two different tasks. Moreover, robots have limited battery lifetime, hence the battery's state of charge must be monitored during the course of robots activities. If the state of charge drops to value low, *self-healing* actions are required, e.g. reaching a charging station or sending a distress signal.

Robots are not informed about the position of the two target zones. Thus, to discover the location of the target, each robot follows a *random walk*. As soon as a robot reaches the assigned zone, it 'publishes' its location within the local knowledge repository. In this way, robots with the same task can get informed about the location of the corresponding target. Notice that, by relying on *group-oriented* queries, the identity of the robot publishing the target location can be ignored. Informed robots can then move directly towards the target, by saving time with respect to random walking (i.e., they *self-optimise* their behaviour).

3. A FORMAL LANGUAGE FOR AUTONOMIC COMPUTING

In this section we introduce the constructs of our language, while their precise semantics will be presented in the next one. We would like to stress that we have taken a minimal approach and SCEL syntax fully specifies only constructs for modeling Behaviors and Aggregations and is parametric with respect to Knowledge and Policies.

Concretely, an AC in SCEL is rendered as the term $\mathcal{I}[\mathcal{K}, \Pi, P]$. This is graphically illustrated in Figure 2 and consists of:

— An *interface* $\mathcal{I}$ publishing and making available structural and behavioral information about the component itself in the form of *attributes*, i.e. names acting as references to information stored in the component's knowledge repository. Among them, attribute id is mandatory and is bound to the name of the component. Component names are not required to be unique; this allows us to easily model replicated service components.
— A *knowledge repository* $\mathcal{K}$ managing both *application data* and *awareness data*, together with the specific handling mechanism. Application data are used for enabling the progress of ACs' computations, while awareness data provide information about the environment in which the ACs are running (e.g. monitored data from sensors) or about the status of an AC (e.g. its current location). 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 between the different parts of a single component and the interaction between components. Interaction policies and Service Level Agreement policies provide two standard examples of policy abstractions. Other examples are security policies, such as access control and reputation.

Table I. SCEL syntax

SYSTEMS:	S	$::=$	$C \mid S_1 \parallel S_2 \mid (\nu n)S$
COMPONENTS:	C	$::=$	$\mathcal{I}[\mathcal{K}, \Pi, P]$
PROCESSES:	P	$::=$	$\mathbf{nil} \mid a.P \mid P_1 + P_2 \mid P_1[\,P_2\,] \mid X \mid A(\bar{p})$
ACTIONS:	a	$::=$	$\mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c \mid \mathbf{fresh}(n) \mid \mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$
TARGETS:	c	$::=$	$n \mid x \mid \mathsf{self} \mid \mathcal{P} \mid p$

Note: KNOWLEDGE $\mathcal{K}$, POLICIES Π, TEMPLATES T, and ITEMS t are parameters of the language.

— A *process* P, together with a set of process definitions that can be dynamically activated. Some of the processes in P execute local computations, while others may coordinate interaction with the knowledge repository or perform adaptation and reconfiguration. *Interaction* is obtained by allowing ACs to access knowledge in the repositories of other ACs.

SCEL syntax is presented in Table I. Its basic category is the one defining PROCESSES that are used to build up COMPONENTS that in turn are used to define SYSTEMS. PROCESSES specify the flow of the ACTIONS that can be performed. ACTIONS can have a TARGET to determine the other components that are involved in that action. As stated in the Introduction, SCEL is parametric with respect to some syntactic categories, namely KNOWLEDGE, POLICIES, TEMPLATES and ITEMS (with the last two determining the part of KNOWLEDGE to be retrieved/removed or added, respectively).

Systems and components. SYSTEMS aggregate COMPONENTS through the *composition* operator $\_ \parallel \_$. It is also possible to restrict the scope of a name, say n, by using the *name restriction* operator $(\nu n)\_$. In a system of the form $S_1 \parallel (\nu n)S_2$, the effect of the operator is to make name n invisible from within S_1. Essentially, this operator plays a role similar to that of a *begin* ... *end* block in sequential programming and limits visibility of specific names. Additionally, restricted names can be exchanged in communications thus enabling the receiving components to use those "private" names.

Running example (step 1/7). The robotics scenario can be expressed in SCEL as a system S defined as follows

$$S \triangleq \mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \parallel \mathcal{I}_3[\mathcal{K}_3, \Pi_3, P_3] \parallel \mathcal{I}_4[\mathcal{K}_4, \Pi_4, P_4] \parallel \ldots$$

The robots are rendered as components, identified by $\mathcal{I}_i.id$ (i.e., the values of attribute id exposed in their interfaces $\mathcal{I}_i$), that concurrently execute and interact. □

Processes. PROCESSES are the active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* ($a.P$), *nondeterministic choice* ($P_1 + P_2$), *controlled composition* ($P_1[\,P_2\,]$), *process variable* (X), and *parameterized process invocation* ($A(\bar{p})$). The construct $P_1[\,P_2\,]$ abstracts the various forms of parallel composition commonly used in process calculi. 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 assume that A ranges over a set of parameterized *process identifiers* that are used in recursive process definitions. We also assume that each process identifier A has a *single* definition of the form $A(\bar{f}) \triangleq P$. Lists of actual and formal parameters are denoted by $\bar{p}$ and $\bar{f}$, respectively.

Running example (step 2/7). The process P_1 running on the first robot, i.e. component $\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1]$, has the form $a_1.\, a_2.\, P_1'$, meaning that actions a_1 and a_2 are sequentially executed and thereafter the process continues as P_1'. □

Actions and targets. Processes can perform five different kinds 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 as patterns to select knowledge items t in the repositories. They heavily rely on the used knowledge repository and are implemented by invoking the handling operations it provides. Action $\mathbf{fresh}(n)$ introduces a scope restriction for the name n so that 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 $\mathcal{I}[\mathcal{K}, \Pi, P]$.

Action **get** may cause the process executing it to wait for the wanted element if it is not (yet) available in the knowledge repository. Action **qry**, exactly like **get**, may suspend the process executing it if the knowledge repository does not (yet) contain or cannot 'produce' the wanted element. The two actions differ for the fact that **get** removes the found item from the knowledge repository while **qry** leaves the target repository unchanged. Actions **put**, **fresh** and **new** are instead immediately executed (provided that their execution is allowed by the policies in force).

Different entities may be used as the target c of an action. Component names are denoted by n, n', ..., while variables for names are denoted by x, x', ... The distinguished variable self can be used by processes to refer to the name of the component hosting them. The possible targets could, however, be also singled out via predicates expressed as boolean-valued expression obtained by logically combining the evaluation of relations between attributes and expressions. Thus targets could also be an explicit *predicate* $\mathcal{P}$ or the name p of a predicate that is exposed as an attribute of a component interface whose value may dynamically change. We adopt the following conventions about attribute names within predicates. If an attribute name occurs in a predicate without specifying (via prefix notation) the corresponding interface, it is assumed that this name refers to an attribute within the interface of the *object* component (i.e., a component that is a target of the communication action). Instead, if an attribute name occurring in a predicate is prefixed by the keyword this, then it is assumed that this name refers to an attribute within the interface of the *subject* component (i.e., the component hosting the process that performs the communication action). Thus, for example, the predicate this.$status$ = "$sending$" $\wedge$ $status$ = "$receiving$" is satisfied when the status of the subject component is $sending$ and that of the object is $receiving$.

In actions using a predicate $\mathcal{P}$ to indicate the target (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 $\mathcal{P}$ 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 target of a communication action are considered as the *ensemble* with which the process performing the action intends to interact. Indeed, in spite of the stress we put on ensembles, SCEL does not have any specific syntactic category or operator for forming ACEs. For example, the names of the components that can be members of an ensemble can be fixed via the predicate $id \in \{n, m, o\}$. When an action has this predicate as target, it will act on all components named n, m or o, if any. Instead, to dynamically characterize the members of an ensemble that are active and have a battery whose level is higher than low, by assuming that attributes $active$ and $batteryLevel$ belong

Table II. Semantics of processes

$$a.P \downarrow_a P \qquad P \downarrow_\circ P$$

$$\frac{P \downarrow_\alpha P'}{P + Q \downarrow_\alpha P'} \qquad \frac{Q \downarrow_\alpha Q'}{P + Q \downarrow_\alpha Q'} \qquad \frac{P\{\bar{p}/\bar{f}\} \downarrow_\alpha P'}{A(\bar{p}) \downarrow_\alpha P'}\ A(\bar{f}) \triangleq P$$

$$\frac{P \downarrow_\alpha P' \quad Q \downarrow_\beta Q'}{P[Q] \downarrow_{\alpha[\beta]} P'[Q']}\ \mathrm{bv}(\alpha) \cap \mathrm{bv}(\beta) = \emptyset \qquad \frac{P' \downarrow_\alpha P''}{P \downarrow_\alpha P''}\ P \equiv P'$$

to the interface of any component willing to be part of the ensemble, one can write $active = \text{“}yes\text{”} \wedge batteryLevel > \text{“}low\text{”}$.

Running example (step 3/7). By specifying actions a_1 and a_2 as a **qry** and a **put** action, respectively, the process P_1 becomes

$$\mathbf{qry}(\text{“}targetLocation\text{”}, ?x, ?y)@(task = \text{“}task_1\text{”}).$$
$$\mathbf{put}(\text{“}targetLocation\text{”}, x, y)@\mathsf{self}.\, P_1'$$

This process retrieves the target location from one of the informed robots in charge of doing $task_1$, binds the location's coordinates to variables x and y, and publishes such information in the local repository. □

4. SCEL OPERATIONAL SEMANTICS

The operational semantics of SCEL is defined in two steps. First, the semantics of processes specifies *commitments*, i.e. the actions that processes can initially perform and the continuation process obtained after each such action; issues like process allocation, available data, regulating policies are ignored at this level. Then, by taking process commitments and system configuration into account, the semantics of systems provides a full description of systems behavior.

4.1. Semantics of processes

Process commitments are generated by the following production rule

$$\alpha, \beta \quad ::= \quad a \quad \mid \quad \circ \quad \mid \quad \alpha[\,\beta\,]$$

meaning that a commitment is either an action a as defined in Table I, or the symbol $\circ$, denoting *inaction*, or the composition $\alpha[\,\beta\,]$ of the two commitments α and β. We use P and Q to range over processes and write $P \downarrow_\alpha Q$ to mean that "P can commit to perform α and become Q after doing so".

The relation $\downarrow$ defining the semantics of processes is the least relation induced by the inference rules in Table II. The first rule says that a process of the form $a.P$ is committed to do a and then to continue as process P. The second rule allows any process to stay idle. The third and fourth rules state that $P + Q$ non-deterministically behaves as P or Q. The fifth rule says that a process invocation $A(\bar{p})$ behaves as the invoked process P , where the formal parameters $\bar{f}$ have been replaced by the actual parameters $\bar{p}$. The sixth rule, defining the semantics of $P[\,Q\,]$, states that a commitment $\alpha[\,\beta\,]$ is exhibited when P commits to α and Q commits to β. However, P and Q are not forced to actually commit to a meaningful action. Indeed, thanks to the second rule, which allows any process to commit to $\circ$, α and/or β may always be $\circ$. The semantics of $P[\,Q\,]$ at the level of processes is indeed very permissive and generates all possible compositions of the commitments of P and Q. This semantics is then specialized at the level of systems by means of interaction predicates that take also policies into account. Condition $\mathrm{bv}(\alpha) \cap \mathrm{bv}(\beta) = \emptyset$ ensures that the variables used by the two processes P

and Q are different, to avoid improper name captures. In fact, $\mathrm{bv}(\alpha)$ denotes the sets of *bound* variables occurring in α, with **get** and **qry** being the only binding constructs for variables. Similarly, the action **fresh** is a binding construct for names. The last rule states that *alpha-equivalent* ($\equiv$) processes, i.e. processes differing only for bound variables and names, can guarantee the same commitments.

Running example (step 4/7). The process P_1 running on the first robot, apart for the trivial case $P_1 \downarrow_\circ P_1$, produces only the following meaningful commitment

$$P_1 \downarrow_{\mathbf{qry}(\text{“}targetLocation\text{”},?x,?y)@(task=\text{“}task_1\text{”})} P_1''$$

with $P_1'' \triangleq \mathbf{put}(\text{“}targetLocation\text{”}, x, y)@\mathsf{self}.\, P_1'$. □

4.2. Semantics of systems

The operational semantics of systems is defined in two steps. First, the possible behaviors of systems without occurrences of the name restriction operator are defined. This is done in the SOS style [Plotkin 2004] by relying on the notion of Labeled Transition System (LTS). Then, by exploiting this LTS, the semantics of generic systems is provided by means of a (unlabelled) Transition System (TS) only accounting for systems' computation steps. This approach allows us to avoid the notational intricacies arising when dealing with name mobility in computations (e.g. when opening and closing the scopes of name restrictions).

The labeled transition relation of the LTS defining the semantics of systems without restricted names is induced by the inference rules in Tables III and V. We write $S \xrightarrow{\lambda} S'$ to mean that "S can perform a transition labeled λ and become S' in doing so". Transition labels are generated by the following production rule

$$\begin{array}{rcl} \lambda & ::= & \tau \;\mid\; \mathcal{I} : \mathbf{fresh}(n) \;\mid\; \mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P) \\ & \mid & \mathcal{I} : t \triangleleft \gamma \;\mid\; \mathcal{I} : t \blacktriangleleft \gamma \;\mid\; \mathcal{I} : t \triangleright \gamma \;\mid\; \mathcal{I} : t \bar{\triangleleft} \mathcal{J} \;\mid\; \mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J} \;\mid\; \mathcal{I} : t \bar{\triangleright} \mathcal{J} \end{array}$$

where γ is either the name n of a component or a predicate $\mathcal{P}$ indicating a set of components, and $\mathcal{I}$ and $\mathcal{J}$ range over interfaces[1]. The meaning of labels is as follows: τ denotes an internal computation step, $\mathcal{I} : \mathbf{fresh}(n)$ denotes the willingness of component $\mathcal{I}$ to restrict visibility of name n, $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$ denotes the willingness of component $\mathcal{I}$ to create the new component $\mathcal{J}[\mathcal{K}, \Pi, P]$, $\mathcal{I} : t \triangleleft \gamma$ (resp. $\mathcal{I} : t \blacktriangleleft \gamma$) denotes the intention of component $\mathcal{I}$ to withdraw (resp. retrieve) item t from the repositories at γ, $\mathcal{I} : t \triangleright \gamma$ denotes the intention of component $\mathcal{I}$ to add item t to the repositories at γ, $\mathcal{I} : t \bar{\triangleleft} \mathcal{J}$ (resp. $\mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J}$) denotes that component $\mathcal{I}$ is allowed to withdraw (resp. retrieve) item t from the repository of component $\mathcal{J}$, $\mathcal{I} : t \bar{\triangleright} \mathcal{J}$ denotes that component $\mathcal{I}$ is allowed to add item t to the repository of component $\mathcal{J}$. Moreover, in the rules, we use $\mathcal{I}.\pi$ to denote the policy in force at the component $\mathcal{I}$ and $S[\mathcal{I}.\pi := \Pi']$ to denote the replacement of the policy in force at the component $\mathcal{I}$ with the policy Π'.

The labeled transition is parameterised with respect to the following two predicates:

— The *interaction predicate*, $\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi'$, means that under policy Π and interface $\mathcal{I}$, process commitment α yields system label λ, substitution σ (i.e., a partial function from variables to values) and, possibly new, policy Π'. Intuitively, λ identifies

[1] Actually, the names of the attributes of a component are just pointers to the real values contained in the knowledge repository associated to the component. This amounts to saying that in terms of the form $\mathcal{I}[\mathcal{K}, \Pi, P]$, $\mathcal{I}$ only includes the names of the attributes, as their corresponding values can be easily retrieved from $\mathcal{K}$. However, when $\mathcal{I}$ is used in isolation, we assume that it also includes the attributes' values. For example, given the component $\mathcal{I}[\mathcal{K}, \Pi, P]$, we can use the condition $n = \mathcal{I}.id$ to check if the value associated to the attribute id in the repository $\mathcal{K}$ is equal to the name n.

Table III. Systems' labeled transition relation

$$\frac{P \downarrow_\alpha P' \qquad \Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\lambda} \mathcal{I}[\mathcal{K}, \Pi', P'\sigma]} \textit{(pr-sys)}$$

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:\mathbf{fresh}(n)} \mathcal{I}[\mathcal{K}, \Pi', P'] \qquad n \notin \mathrm{n}(\mathcal{I}[\mathcal{K}, \Pi, \mathbf{nil}]) \qquad \Pi' \vdash \mathcal{I} : \mathbf{fresh}(n), \Pi''}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} (\nu n)\, \mathcal{I}[\mathcal{K}, \Pi'', P']} \textit{(freshn)}$$

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:\mathbf{new}(\mathcal{J}, \mathcal{K}'', \Pi'', P'')} \mathcal{I}[\mathcal{K}, \Pi', P'] \qquad \Pi' \vdash \mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}'', \Pi'', P''), \Pi'''}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}, \Pi''', P'] \parallel \mathcal{J}[\mathcal{K}'', \Pi'', P'']} \textit{(newc)}$$

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \lhd n} \mathcal{I}[\mathcal{K}, \Pi', P'] \qquad n = \mathcal{I}.id \qquad \Pi' \vdash \mathcal{I} : t \bar{\lhd} \mathcal{I}, \Pi'' \qquad \mathcal{K} \ominus t = \mathcal{K}'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}', \Pi'', P']} \textit{(lget)}$$

$$\frac{\Pi \vdash \mathcal{I} : t \bar{\lhd} \mathcal{J}, \Pi' \qquad \mathcal{K} \ominus t = \mathcal{K}'}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \bar{\lhd} \mathcal{J}} \mathcal{J}[\mathcal{K}', \Pi', P]} \textit{(accget)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \lhd n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \bar{\lhd} \mathcal{J}} S_2' \qquad \mathcal{J}.id = n \qquad \mathcal{I}.\pi \vdash \mathcal{I} : t \bar{\lhd} \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1'[\mathcal{I}.\pi := \Pi'] \parallel S_2'} \textit{(ptpget)}$$

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \blacktriangleleft n} \mathcal{I}[\mathcal{K}, \Pi', P'] \qquad n = \mathcal{I}.id \qquad \Pi' \vdash \mathcal{I} : t \bar{\blacktriangleleft} \mathcal{I}, \Pi'' \qquad \mathcal{K} \vdash t}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}, \Pi'', P']} \textit{(lqry)}$$

$$\frac{\Pi \vdash \mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J}, \Pi' \qquad \mathcal{K} \vdash t}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \bar{\blacktriangleleft} \mathcal{J}} \mathcal{J}[\mathcal{K}, \Pi', P]} \textit{(accqry)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \blacktriangleleft n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \bar{\blacktriangleleft} \mathcal{J}} S_2' \qquad \mathcal{J}.id = n \qquad \mathcal{I}.\pi \vdash \mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1'[\mathcal{I}.\pi := \Pi'] \parallel S_2'} \textit{(ptpqry)}$$

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \rhd n} \mathcal{I}[\mathcal{K}, \Pi', P'] \qquad n = \mathcal{I}.id \qquad \Pi' \vdash \mathcal{I} : t \bar{\rhd} \mathcal{I}, \Pi'' \qquad \mathcal{K} \oplus t = \mathcal{K}'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}', \Pi'', P']} \textit{(lput)}$$

$$\frac{\Pi \vdash \mathcal{I} : t \bar{\rhd} \mathcal{J}, \Pi' \qquad \mathcal{K} \oplus t = \mathcal{K}'}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \bar{\rhd} \mathcal{J}} \mathcal{J}[\mathcal{K}', \Pi', P]} \textit{(accput)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \rhd n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \bar{\rhd} \mathcal{J}} S_2' \qquad \mathcal{J}.id = n \qquad \mathcal{I}.\pi \vdash \mathcal{I} : t \bar{\rhd} \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1'[\mathcal{I}.\pi := \Pi'] \parallel S_2'} \textit{(ptpput)}$$

the effect of α at the level of components, while σ associates values to the variables occurring in α and is used to capture the changes induced by communication. Π' is the policy in force after the transition; in principle it may differ from that in force before the transition. This predicate is used to determine the effect of the simultaneous execution of actions by processes concurrently running within a component that, e.g., exhibit commitments of the form $\alpha[\,\beta\,]$.

— The *authorization predicate*, $\Pi \vdash \lambda, \Pi'$, means that under policy Π, the action generating the system label λ (which can be thought of as an *authorization request*) is allowed and the policy Π' is produced. Labels λ taken as argument by the au-

Table IV. The interaction predicate interleaving

$$\Pi, \mathcal{I} : \mathbf{fresh}(n) \succ \mathbf{fresh}(n), \{\}, \Pi \qquad \frac{\mathcal{E}v[\![\, T \,]\!]_{\mathcal{I}} = T' \quad \mathcal{E}v[\![\, c \,]\!]_{\mathcal{I}} = \gamma \quad match(T', t) = \sigma}{\Pi, \mathcal{I} : \mathbf{get}(T)@c \succ \mathcal{I} : t \triangleleft \gamma, \sigma, \Pi}$$

$$\frac{\mathcal{E}v[\![\, T \,]\!]_{\mathcal{I}} = T' \quad \mathcal{E}v[\![\, c \,]\!]_{\mathcal{I}} = \gamma \quad match(T', t) = \sigma}{\Pi, \mathcal{I} : \mathbf{qry}(T)@c \succ \mathcal{I} : t \blacktriangleleft \gamma, \sigma, \Pi} \qquad \frac{\mathcal{E}v[\![\, t \,]\!]_{\mathcal{I}} = t' \quad \mathcal{E}v[\![\, c \,]\!]_{\mathcal{I}} = \gamma}{\Pi, \mathcal{I} : \mathbf{put}(t)@c \succ \mathcal{I} : t' \triangleright \gamma, \{\}, \Pi}$$

$$\Pi, \mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P) \succ \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, \mathcal{E}v[\![\, P \,]\!]_{\mathcal{I}}), \{\}, \Pi$$

$$\frac{\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi}{\Pi, \mathcal{I} : \alpha[\circ] \succ \lambda, \sigma, \Pi} \qquad \frac{\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi}{\Pi, \mathcal{I} : \circ[\alpha] \succ \lambda, \sigma, \Pi}$$

thorization predicate are system labels of the form $\mathcal{I} : \mathbf{fresh}(n)$, $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$, $\mathcal{I} : t \bar{\triangleleft} \mathcal{J}$, $\mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J}$, or $\mathcal{I} : t \bar{\triangleright} \mathcal{J}$. This predicate is used to determine the actions allowed by specific policies, and the (possibly new) policy to be enforced. The authorization to perform an action is checked when a computation step can potentially take place, i.e. when it becomes known which is the component target of the action.

Many different interaction predicates can be defined to capture well-known process computation and interaction patterns such as interleaving, monitoring, asynchronous communication, synchronous communication, full synchrony, broadcasting, etc. In fact, depending on the considered class of systems, one can prefer a communication model with respect to the others.

A specific interaction predicate is defined in Table IV; it is obtained by interpreting controlled composition as the *interleaved* parallel composition of the two involved processes. In the table, function $\mathcal{E}v[\![\, \cdot \,]\!]_{\mathcal{I}}$ denotes the evaluation of terms with respect to interface $\mathcal{I}$ with attributes occurring therein being replaced by the corresponding value in $\mathcal{I}$. Moreover, $match(T, t)$ denotes a partial function performing matching between a template T and an item t; when they do match, the function returns a substitution σ for the variables in T (we use $\{\}$ to denote the empty substitution), otherwise it is undefined. We have a rule for each different kind of process action; for example, the third rule states that, once the target γ of the action and an item t matching the template T' through a substitution σ have been determined (by also exploiting the interface $\mathcal{I}$ for evaluating c and T), an action **qry** at the level of processes corresponds to a proper transition label at the level of systems semantics. The last two rules ensure that in case of controlled composition of multiple processes only one process at a time can perform an action (the other stays still).

Likewise the interaction predicate, many different reasonable authorization predicates can be defined, possibly resorting to specific policy languages. One of such languages inspired by, but simpler than, the OASIS standard for policy-based access control XACML [OASIS-TC 2005], will be presented in Section 5. There, we will stress also how the actual semantics of this policy language is intertwined and integrated with SCEL semantics.

The labeled transition relation also relies on the following three operations that each knowledge repository's handling mechanism must provide:

— $\mathcal{K} \ominus t = \mathcal{K}'$: the *withdrawal* of item t from the repository $\mathcal{K}$ returns $\mathcal{K}'$;
— $\mathcal{K} \vdash t$: the *retrieval* of item t from the repository $\mathcal{K}$ is possible;
— $\mathcal{K} \oplus t = \mathcal{K}'$: the *addition* of item t to the repository $\mathcal{K}$ returns $\mathcal{K}'$.

We now briefly comment the rules in Table III. Rule *(pr-sys)* transforms process commitments into system labels by exploiting the interaction predicate. In particular, it generates the following system labels: τ, $\mathcal{I} : \mathbf{fresh}(n)$, $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$, $\mathcal{I} : t \triangleleft \gamma$, $\mathcal{I} : t \blacktriangleleft \gamma$ and $\mathcal{I} : t \triangleright \gamma$. As a consequence, a substitution σ is applied to the continuation P' of the process that committed to α. When α contains a $\mathbf{get}(T)$ or a $\mathbf{qry}(T)$, σ replaces in P' the variables occurring in T with the corresponding values. The application of the rule also replaces, in the generated label, self with the corresponding name. Moreover, due to the evaluation of the interaction predicate, the policy in force at the component performing the action may change.

Actions **fresh** and **new** are decided by using the information within a single component. However, since they affect the system, as they either create a name restriction or a new component, their execution by a process is indicated by a specific system label $\mathcal{I} : \mathbf{fresh}(n)$ or $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$ (generated by rule *(pr-sys)*) carrying enough information for the authorization request to perform the action to be checked according to the local policy and for the modification of the system to take place (rules *(freshn)* and *(newc)*). Notably, the authorization predicate is evaluated under the policy produced by the interaction predicate (rule *(pr-sys)*); thus, the component performing the action will enforce the (possibly new) policy so generated. Moreover, the scope of a new name n is put in place (rule *(freshn)*) only if the name is not already used in the creating component, possibly except for the process part (notation $\mathrm{n}(E)$ is used here to denote the sets of names occurring in a syntactic term E); this condition can be always made true by exploiting alpha-equivalence among processes.

The successful execution of the remaining three actions requires, at system level, appropriate synchronization. For this reason, we have a pair of complementary labels corresponding to each action. The rules in Table III model the variants of these actions implementing point-to-point communication (the rules for group-oriented communication are shown in Table V).

Action **get** can withdraw an item either from the local repository *(lget)* or from a specific repository with a point to point access *(ptpget)*. In any case, this transition corresponds to an internal computation step. The label $\mathcal{I} : t \bar{\triangleleft} \mathcal{J}$, generated by rule *(accget)*, denotes the willingness of component $\mathcal{J}$ to provide the item t to component $\mathcal{I}$. Notably, the label is generated only if such willingness is authorized by the policy in force at the component $\mathcal{J}$ (by means of the authorization predicate $\Pi \vdash \mathcal{I} : t \bar{\triangleleft} \mathcal{J}, \Pi'$) and if withdrawing item t from the repository of $\mathcal{J}$ is possible ($\mathcal{K} \ominus t = \mathcal{K}'$). Thus, when the target of the action denotes a specific remote repository *(ptpget)*, the action is only allowed if n is the name of the component $\mathcal{J}$ simultaneously willing to provide the wanted item and if the request to perform the action at $\mathcal{J}$ is authorized by the local policy (identified by notation $\mathcal{I}.\pi$).

The semantics of action **qry** is modeled by rules *(lqry)*, *(accqry)*, and *(ptpqry)*. This action behaves similarly to **get**, the only difference being that it invokes the retrieval operation of the repository's handling mechanism ($\mathcal{K} \vdash t$), rather than the withdrawal operation. Therefore, if the action succeeds, after the computation step all repositories remain unchanged.

Action **put** adds item t to one or more repositories. Its behavior is modeled by rules *(lput)*, *(accput)*, and *(ptpput)*, that are similar to those of actions **get** and **qry**, with the major difference being that the addition operation of the repository's handling mechanism is invoked.

As we already said, SCEL also provides a form of group-oriented communication. It is modeled by the rules in Table V. Thus, when the target of action **get** denotes a set of repositories satisfying a given predicate *(grget)*, the action is only allowed if one of these repositories, say that of component $\mathcal{J}$, is willing to provide the wanted item and if the request to perform the action at $\mathcal{J}$ is authorized by the policy in force at the component

Table V. Systems' labeled transition relation (cnt.): rules for group communication

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \lhd \mathcal{P}} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \barwedge \mathcal{J}} S_2' \qquad \mathcal{J} \models \mathcal{P} \qquad \mathcal{I}.\pi \vdash \mathcal{I} : t \barwedge \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1'[\mathcal{I}.\pi := \Pi'] \parallel S_2'} \textit{(grget)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \blacktriangleleft \mathcal{P}} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \barwedge \mathcal{J}} S_2' \qquad \mathcal{J} \models \mathcal{P} \qquad \mathcal{I}.\pi \vdash \mathcal{I} : t \barwedge \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1'[\mathcal{I}.\pi := \Pi'] \parallel S_2'} \textit{(grqry)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \rhd \mathcal{P}} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \barvee \mathcal{J}} S_2' \qquad \mathcal{J} \models \mathcal{P} \qquad \mathcal{I}.\pi \vdash \mathcal{I} : t \barvee \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\mathcal{I}:t \rhd \mathcal{P}} S_1'[\mathcal{I}.\pi := \Pi'] \parallel S_2'} \textit{(grput)}$$

$$\frac{S \xrightarrow{\mathcal{I}:t \rhd \mathcal{P}} S' \qquad (\mathcal{J} \not\models \mathcal{P} \quad \vee \quad \Pi \not\vdash \mathcal{I} : t \barvee \mathcal{J}, \Pi')}{S \parallel \mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \rhd \mathcal{P}} S' \parallel \mathcal{J}[\mathcal{K}, \Pi, P]} \textit{(engrput)}$$

$$\frac{S_1 \xrightarrow{\lambda} S_1' \qquad \lambda \notin \{\mathcal{I} : t \rhd \mathcal{P}, \mathcal{I} : t \barvee \mathcal{J}\}}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2} \textit{(async)}$$

performing the action. Relation $\mathcal{J} \models \mathcal{P}$ states that the attributes of $\mathcal{J}$ satisfy predicate $\mathcal{P}$; the definition of such relation depends on the kind of the used predicates. In any case, if the action succeeds, this transition corresponds to an internal computation step (denoted by τ) that changes the repository of component $\mathcal{J}$. Rule *(grqry)* is similar, but in the case of action **qry** the item is not removed from the repository. Differently from the two previous actions that only interact with one target component arbitrarily chosen among those satisfying the predicate $\mathcal{P}$ and willing to provide the wanted item, $\mathbf{put}(t)@\mathcal{P}$ can interact with all components satisfying $\mathcal{P}$ and willing to accept the item t. In fact, rule *(grput)* permits the execution of a **put** for group-oriented communication when there is a parallel component, say $\mathcal{J}$, satisfying the target of the action and whose policy authorizes this remote access. Of course, the action must be authorized to use $\mathcal{J}$ as a target also by the policy in force at the component performing the action. Notably, the resulting action is still a **put** for group-oriented communication, thus further authorization actions performed by other parallel components satisfying the target of the action can be simultaneously executed.

The capability of a component to perform a **put** for group-oriented communication is not affected by those system components not satisfying predicate $\mathcal{P}$, i.e. not belonging to the ensemble, or not authorising the action (rule *(engrput)*). Therefore, when there is a system component able to perform a **put** for group-oriented communication, by repeatedly applying rules *(grput)* and *(engrput)* it is possible to infer that the whole system can perform such an action (which in fact means that a component produces an item which is added to the repository of all the ensemble components that simultaneously are willing to receive the item). Instead, rule *(async)* states that all actions different from a **put** for group-oriented communication and an authorization for a **put** can be performed by involving only some of the system's components. Therefore, if there is a system component able to perform the authorization for a **put**, there is no way to infer that such component in parallel with any other one (hence the system as a whole) can perform the action. This ensures that when a system component is going to execute a **put** for group-oriented communication all potential receivers are taken into account.

Table VI. Systems' transition relation

$$\frac{S \xrightarrow{\tau} S'}{(\nu\bar{n})S \rightarrowtail (\nu\bar{n})S'}\ \textit{(tau)} \qquad \frac{S \xrightarrow{\mathcal{I}:t\triangleright\mathcal{P}} S'}{(\nu\bar{n})S \rightarrowtail (\nu\bar{n})S'}\ \textit{(put)}$$

$$\frac{(\nu\bar{n}, n'')(S_1 \parallel S_2\{n''/n'\}) \rightarrowtail S' \qquad n'' \textit{ fresh}}{(\nu\bar{n})(S_1 \parallel (\nu n')S_2) \rightarrowtail S'}\ \textit{(top)}$$

$$\frac{(\nu\bar{n})(S_2 \parallel S_1) \rightarrowtail S'}{(\nu\bar{n})(S_1 \parallel S_2) \rightarrowtail S'}\ \textit{(comm)} \qquad \frac{(\nu\bar{n})((S_1 \parallel S_2) \parallel S_3) \rightarrowtail S'}{(\nu\bar{n})(S_1 \parallel (S_2 \parallel S_3)) \rightarrowtail S'}\ \textit{(assoc)}$$

Running example (step 5/7). Let us suppose that $\mathcal{I}_2.task = \mathcal{I}_3.task =$ "$task_1$" while $\mathcal{I}_4.task =$ "$task_2$" and that $\mathcal{K}_3$ contains an item indicating that the $targetLocation$ has position $(3, 5)$. Now, by exploiting the operational rule *(accqry)*, the third component can generate the following labelled transition

$$\mathcal{I}_3[\mathcal{K}_3, \Pi_3, P_3] \xrightarrow{\mathcal{I}_1:\langle \text{"}targetLocation\text{"},3,5\rangle \bar{\blacktriangleleft} \mathcal{I}_3} \mathcal{I}_3[\mathcal{K}_3, \Pi_3, P_3]$$

while, by exploiting the operational rule *(pr-sys)*, the first component can generate the following labelled transition

$$\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \xrightarrow{\mathcal{I}_1:\langle \text{"}targetLocation\text{"},3,5\rangle \blacktriangleleft (task=\text{"}task_1\text{"})} \mathcal{I}_1[\mathcal{K}_1, \Pi_1', (\mathbf{put}(\text{"}targetLocation\text{"}, 3, 5)@\mathsf{self}.P_1')]$$

Hence, by exploiting the operational rule *(grqry)*, the overall system can perform the transition

$$S \xrightarrow{\tau} \mathcal{I}_1[\mathcal{K}_1, \Pi_1', (\mathbf{put}(\text{"}targetLocation\text{"}, 3, 5)@\mathsf{self}.P_1')] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \parallel \mathcal{I}_3[\mathcal{K}_3, \Pi_3, P_3] \parallel \mathcal{I}_4[\mathcal{K}_4, \Pi_4, P_4] \parallel \ldots \qquad \square$$

The unlabeled transition relation ($\rightarrowtail$) of the TS providing the semantics of generic systems is defined on top of the labeled one by the inference rules in Table VI. As a matter of notation, $\bar{n}$ denotes a (possibly empty) sequence of names and $\bar{n}, n'$ is the sequence obtained by composing $\bar{n}$ and n'. $(\nu\bar{n})S$ abbreviates $(\nu n_1)((\nu n_2)(\cdots(\nu n_m)S\cdots))$, if $\bar{n} = n_1, n_2, \cdots, n_m$ with $m > 0$, and S, otherwise. $S\{n'/n\}$ denotes the system obtained by replacing any free occurrence in S of n with n'. When considering a system S, a name is deemed *fresh* if it is different from any name occurring in S.

Rule *(tau)* of Table VI accounts for the computation steps of a system where all (possible) name restrictions are at top level. Rule *(put)* states that, besides those labeled by τ, computation steps may additionally be labeled by $\mathcal{I} : t \triangleright \mathcal{P}$, corresponding to group-oriented communication triggered by an action $\mathbf{put}(t)@\mathcal{P}$ performed by component $\mathcal{I}$, and thus transforms them into transitions of the form $\rightarrowtail$. Rule *(top)* permits to manipulate the syntax of a system, by moving all name restrictions at top level, thus putting it into a form to which one of the first two rules can be possibly applied. This manipulation may require the renaming of a restricted name with a freshly chosen one, thus ensuring that the name moved at top level is different both from the restricted names already moved at top level (to avoid name clashes) and from the names occurring free in the other (sub-)systems in parallel (to avoid improper name captures). Rules *(comm)* and *(assoc)* state that systems' composition is a commutative and associative operator. Notably, by exploiting these two rules, we can manipulate systems and avoid adding analogous rules to those defining the labeled transition relation.

Running example (step 6 / 7). The robotics system can thus evolve as follows

$$\begin{aligned} S \succ\!\!\longrightarrow\ & \mathcal{I}_1[\mathcal{K}_1, \Pi'_1, (\mathbf{put}(\text{“}targetLocation\text{”}, 3, 5)@\mathsf{self}.P'_1)] \\ & \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \parallel \mathcal{I}_3[\mathcal{K}_3, \Pi_3, P_3] \parallel \mathcal{I}_4[\mathcal{K}_4, \Pi_4, P_4] \parallel \ldots \\ \succ\!\!\longrightarrow\ & \mathcal{I}_1[(\mathcal{K}_1 \oplus \langle \text{“}targetLocation\text{”}, 3, 5\rangle), \Pi'_1, P'_1] \\ & \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \parallel \mathcal{I}_3[\mathcal{K}_3, \Pi_3, P_3] \parallel \mathcal{I}_4[\mathcal{K}_4, \Pi_4, P_4] \parallel \ldots \end{aligned}$$

Notably, the group-oriented **qry** action involves the robots belonging to the ensemble in charge of $task_1$ (which includes the components identified by $\mathcal{I}_2$ and $\mathcal{I}_3$), while the subsequent point-to-point **put** action only involves the first robot (i.e., the component identified by $\mathcal{I}_1$). Recall that $\mathcal{K}_1 \oplus \langle \text{“}targetLocation\text{”}, 3, 5\rangle$ means that the information about $targetLocation$ is added to the knowledge repository $\mathcal{K}_1$. The fourth robot (i.e., the component identified by $\mathcal{I}_4$) is never involved in such communications, because it is in charge of doing $task_2$. □

5. AN ACCESS CONTROL POLICY LANGUAGE FOR SCEL

Access control is a fundamental mechanism for restricting the operations users can perform on protected resources. Many *models* of access control have been defined in the literature. Here, we focus on the Policy Based Access Control (PBAC) model [NIST 2009], that is by now the de-facto standard model for enforcing access control policies in service-oriented architectures. In this model, a request to access a protected resource is evaluated with respect to one or more policies that define which requests are authorized. An authorization decision is based on attribute values required to allow access to a resource according to policies stored in system's components. Component attributes are used to describe the entities that must be considered for authorization purposes; they might concern:

— the *subject* who is demanding access: e.g., identity, role, age, zip code, IP address, group memberships, citizenships, company, management level, certifications;
— the *action* that the user wants to perform: e.g., write, read, withdrawn;
— the *object* (or resource) impacted by the action: e.g., identity, location, size, value;
— the *environment* identifying the context in which access is requested: e.g., time, date, location, battery level, system load, available memory, communication channel type.

In this section we instantiate the parameter of SCEL that deals with policies and specifically with those for access control. Such policies refine components behavior to guarantee accomplishment of specific tasks or satisfaction of specific properties (e.g., protection of private information, management of resource usage, activation of adaptation procedures, etc.). As an example of policy language for SCEL, we shall consider SACPL (Simple Access Control Policy Language), a simple, yet expressive, language for defining access control policies that has been much influenced by the OASIS standard for policy-based access control XACML [OASIS-TC 2005]. In the following, we briefly present SACPL by focussing especially on its integration with SCEL.

5.1. SACPL syntax and semantics

According to the PBAC model, SACPL policies are evaluated to decide if authorisation requests are granted or forbidden. A *request* ρ can be thought of as a function mapping (attribute) names to elements, and is generated from a label produced by the SCEL operational semantics in correspondence of a given action. For example, the request corresponding to the tentative of executing action $\mathbf{put}(t)@n$ provides information about the attributes of the request's subject, i.e. the component performing the action, the attributes of the request's object, i.e. the component identified by n, the exchanged item

Table VII. SACPL policy syntax

POLICIES:	Π	::=	$\langle Decision \,;\, \mathsf{target:}\{\, Target \,\}\rangle \mid \Pi$ p-o $\Pi \mid \Pi$ d-o Π
DECISIONS:	$Decision$	::=	permit | deny
TARGETS:	$Target$	::=	$MatchF(Designator, Expr) \mid Target$ or $Target \mid Target$ and $Target$
DESIGNATORS:	$Designator$	::=	action | item | subject.$attr$ | object.$attr$

t and, of course, the type of the action, i.e. **put**. Each SCEL action is executed only if it is authorized by the policies in force at the component willing to perform the action and at the target component(s). In particular, when the target of an action **put** denotes a set of repositories satisfying a given target predicate, each insertion of the item in these repositories must be authorized separately by the policy in force at the corresponding component; such policy evaluation, however, does not affect the authorization of the insertions in the other target repositories. Instead, in case of a group-oriented action **get** or **qry**, only one authorization is required from the target side, since only one repository is selected for the interaction. Thus, SACPL policies regulate (intra- or inter-components) interactions by simply enabling or disabling behaviours.

SACPL syntax is presented in Table VII. Policies are hierarchically structured as trees. Indeed, a *policy* is either an atomic policy or a pair of simpler policies combined through one of the decision-combining operators p-o (*permit override*) and d-o (*deny override*). To match a composed policy (Π_1 p-o Π_2), an authorization request is only required to match one of Π_1 and Π_2, while it must match both Π_1 and Π_2, in order to match the policy (Π_1 d-o Π_2). An *atomic policy* is a pair consisting of a decision and a target. The target defines the set of requests to which the policy applies. If the target is empty, any request matches the policy. The *decision* — permit or deny — is the effect returned when the policy is 'applicable', i.e. the request belongs to the target. Otherwise, i.e. when a request does not belong to the policy's target, then the policy is not-applicable (this is a third kind of decision that can be returned by the semantics).

A *target* is either an atomic target or a pair of simpler targets combined using the standard logic operators or and and. To match a composed target ($Target_1$ or $Target_2$), a request is only required to match one of $Target_1$ and $Target_2$, while it must match both $Target_1$ and $Target_2$, in order to match the target ($Target_1$ and $Target_2$).

An *atomic target* is a triple denoting the application of a matching function $MatchF$ to values from the request and the policy, like e.g. greater-than(subject.$skill, 30 -$ object.$dependability$). To base an authorization decision on some characteristics of the request, e.g. subjects' or objects' identity, atomic targets use *designators* (i.e. *attribute names*) to point to specific values contained in the request. Specifically, the designator action refers to the action to be performed (such as **get**, **qry**, **put**, etc.). E.g., a request matches an atomic target of the form equal(action,**qry**) if the request's action corresponds to the action **qry** identified by the target. Similarly, item permits referring to the item exchanged in the considered interaction and, hence, an atomic target pattern-match(item, ("$targetLocation$", $?x, ?y$)) is matched by all requests whose item matches the template ("$targetLocation$", $?x, ?y$). Designators subject.$attr$ and object.$attr$ refer to the specific attribute $attr$ provided, respectively, by the request's subject or object (like, e.g., subject.$skill$ and object.$dependability$).

Finally, $Expr$essions are built from basic values, e.g. integers and strings, and attributes through standard operators. The evaluation of an atomic target involving a subject (resp. object) designator consists in obtaining the subject (resp. object) interface from the request, retrieving the value of the attribute from the interface, evaluating the expression by possibly retrieving other attribute values from the request elements and, finally, calling the corresponding match function.

In practice, the semantics of SACPL can be formalised in terms of a judgement $\Pi \vdash \rho$, defined by a set of inference rules and meaning that the authorization decision returned by a policy Π in response to a request ρ is permit, i.e. access to the resource requested in ρ is granted by Π.

5.2. Integration with SCEL

We now demonstrate how SACPL policies and requests, as well as the related evaluation mechanism, integrate with SCEL. The fact that SCEL is parametric with respect to the language used to specify the policies permits to regulate orthogonal aspects of components' behavior by means of different kinds of policies, which should be *enforced together* but *evaluated separately*. Hence, the policy Π specified within a component $\mathcal{I}[\mathcal{K}, \Pi, P]$ can be better thought of as a tuple of policies. For example, Π can be of the form (Π_i, Π_{ac}), where Π_i is one of the policies discussed in Section 4 (e.g., interleaving, monitoring) for regulating the interaction among processes inside a component, while Π_{ac} is a SACPL policy for regulating the access to the knowledge and resources of a component.

The policy tuple is used as a whole in the definition of SCEL's operational semantics, while it is decomposed in its constituent elements, which are then used in different ways, in the definition of the interaction and the authorization predicates. In particular, the interaction predicate over the policy tuple (Π_i, Π_{ac}) can be simply defined as the interaction predicate over the interaction policy Π_i. Similarly, the authorization predicate over the policy tuple (Π_i, Π_{ac}) can be defined in terms of a judgement $\Pi_{ac} \vdash \rho$ by means of the following rule

$$\frac{\Pi_{ac} \vdash \lambda 2\rho(\lambda)}{(\Pi_i, \Pi_{ac}) \vdash \lambda, (\Pi_i, \Pi_{ac})}$$

which also implies that the policy in force does never change owing to evaluation of a request. The definition of the authorization predicate relies on the function $\lambda 2\rho(\cdot)$ that maps (a subset of) the SCEL labels to SACPL requests. For example, the label $\mathcal{I} : t \blacktriangleleft \mathcal{J}$ is converted into the authorization request $\{(\mathsf{subject}, \mathcal{I}), (\mathsf{item}, t), (\mathsf{action}, \mathbf{qry}), (\mathsf{object}, \mathcal{J})\}$. Hence, the authorization of a SCEL request λ over the policy tuple (Π_i, Π_{ac}) corresponds to establishing the authorization decision returned by the policy Π_{ac} in response to the SACPL request $\rho = \lambda 2\rho(\lambda)$, which is exactly the judgement $\Pi \vdash \rho$ explained in Section 5.1.

Running example (step 7/7). In the robotics scenario, a robot component (with identifier n) could regulate the access to its knowledge by remote retrieving actions through the use of the SACPL policy resulting from the composition, by means of the d-o (deny override) operator, of the following policies:

$\langle$permit ; target: { } $\rangle$ // `permit all` //

$\langle$deny ; target: { not-equal(subject.id,n) and equal(object.id,n) and (equal(action,**qry**) or equal(action,**get**)) and not-in(subject.id,object.$ListOfTrusted$) } $\rangle$ // `deny remote` **qry** and **get** //

The composed policy says that all actions are permitted apart for those **qry** and **get** actions whose target is the considered robot and whose subject is a robot that is not trusted. □

Table VIII. Tuple-space-based SCEL

KNOWLEDGE:	ITEMS:	TEMPLATES:
$\mathcal{K} ::= \emptyset \mid \langle t \rangle \mid \mathcal{K}_1 \parallel \mathcal{K}_2$	$t ::= e \mid c \mid P \mid t_1, t_2$	$T ::= e \mid c \mid ?x \mid ?X \mid T_1, T_2$

Note: e is an EXPRESSION.

Table IX. Tuple-space operations ($\ominus$, $\vdash$, $\oplus$)

$$\langle t \rangle \ominus t = \emptyset \qquad \frac{\mathcal{K}_1 \ominus t = \mathcal{K}'}{(\mathcal{K}_1 \parallel \mathcal{K}_2) \ominus t = \mathcal{K}' \parallel \mathcal{K}_2} \qquad \frac{\mathcal{K}_2 \ominus t = \mathcal{K}'}{(\mathcal{K}_1 \parallel \mathcal{K}_2) \ominus t = \mathcal{K}_1 \parallel \mathcal{K}'}$$

$$\langle t \rangle \vdash t \qquad \frac{\mathcal{K}_1 \vdash t}{(\mathcal{K}_1 \parallel \mathcal{K}_2) \vdash t} \qquad \frac{\mathcal{K}_2 \vdash t}{(\mathcal{K}_1 \parallel \mathcal{K}_2) \vdash t} \qquad \mathcal{K} \oplus t = \mathcal{K} \parallel \langle t \rangle$$

6. A SWARM ROBOTICS SCENARIO IN SCEL

In this section, we describe how the SCEL language can be used to model the autonomic computing scenario introduced in Section 3. To this aim we rely on a SCEL's dialect obtained by instantiating knowledge repositories as multiple distributed *tuple-spaces* à la KLAIM [De Nicola et al. 1998], as shown in Table VIII. Specifically, knowledge items are sequences of values, i.e. *tuples*, while templates are sequences of values and variables (the latter ones are preceded by the symbol '?'). A value can be a target c or a process P, or can result from the evaluation of some given expression e. We assume that expressions contain *boolean*, *integer*, *float* and *string* values and variables, together with the corresponding standard operators. A knowledge repository $\mathcal{K}$ is thus a *tuple space*, i.e. a multiset of stored tuples $\langle t \rangle$ and empty tuples $\emptyset$ composed by operator $\_ \parallel \_$.

The three operations provided by the knowledge repository's handling mechanism, namely *withdrawal* ($\mathcal{K} \ominus t$), *retrieval* ($\mathcal{K} \vdash t$) and *addition* ($\mathcal{K} \oplus t$) of an item t from/to repository $\mathcal{K}$, are inductively defined in Table IX. Notably, when a matching tuple is removed from $\mathcal{K}$, it is replaced by $\emptyset$.

Finally, concerning policies, in the considered dialect the interaction predicate is the *interleaving* one (see Section 4) for any component, while the authorization predicate is always satisfied, in other words we assume that the interaction among components is always authorized.

The autonomic behaviour of each robot in the swarm is implemented by means of an *autonomic manager* controlling the execution of a *managed element*. The autonomic manager monitors, in a self-aware fashion, the state of charge of the robot's battery and verifies whether the target area has been reached or not. The managed element can be seen as an empty "executor" which retrieves from the knowledge repository the activities to be performed at the current control step.

Thus, each robot is a component $\mathcal{I}[\mathcal{K}, \Pi, (AM[\,ME\,])\,]$, where the managed element ME is as follows:

$$ME \triangleq \mathbf{qry}(\text{“}controlStep\text{”}, ?X)@\mathsf{self}.\ (\mathbf{get}(\text{“}terminated\text{”})@\mathsf{self}.ME)[\,X\,] \qquad (1)$$

This process retrieves from the local knowledge repository the process implementing the current control step and bounds it to variable X, executes the retrieved process and waits until it terminates.

Therefore, *self-adaptation* is naturally expressed by exploiting SCEL's higher-order features, namely the capability to store/retrieve (the code of) processes in/from the knowledge repositories and to dynamically trigger execution of new processes. As shown in [Gjondrekaj et al. 2012], this form of higher-order communication enables a

straightforward implementation of adaptive behaviours. The autonomic manager AM can then replace the control step code from the knowledge repository to implement the adaptation logic and therefore to change the managed element's behavior. For example, when a robot becomes informed, it self-adapts (i.e., *self-configure*) its behaviour through its autonomic manager in order to directly move towards the target area.

We assume that robots publish within their interface the task that they have to fulfill through attribute $task$. In this way, the predicate $(task = \text{“}task_i\text{”})$, with $i = 1, 2$, identifies all robots in charge of doing $task_i$. We also assume that robots are equipped with a GPS sensor, with a sensor that permits verifying whether the target area has been reached, and with a sensor that monitors the level of battery. These sensors publish their values directly within the knowledge repository. For example, a tuple of the form $\langle \text{“}batteryLevel\text{”}, l \rangle$ in a robot's repository indicates that the state of charge of robot's battery is l. The information stored in these tuples represents the *awareness data* (called *control data* in [Bruni et al. 2012]) and is used to regulate the system's *adaptation*. Specifically, the autonomic manager detects run-time modifications of awareness data and appropriately adapts the robot's behaviour to deal with such changes, by simply replacing the process stored in the “*controlStep*” tuple.

The autonomic manager AM is defined as follows:

$$AM \triangleq P_{batteryMonitor}[\, P_{dataSeeker}[\, P_{targetSeeker}\,]\,]$$

where $P_{batteryMonitor}$ monitors the state of charge of the robot's battery, $P_{dataSeeker}$ tries to retrieve data from the ensemble of robots with the same task in order to obtain the actual position of the target area, and $P_{targetSeeker}$ checks the awareness data and properly sets the “*controlStep*” tuple with either $P_{randomWalk}$, $P_{informed}$, P_{found} or $P_{lowBattery}$. Due to lack of space, we describe below only some of the processes above.

Process $P_{dataSeeker}$ is defined as follows:

$$\begin{aligned} P_{dataSeeker} \triangleq\ & \mathbf{qry}(\text{“}targetLocation\text{”}, ?x, ?y)@(task = \text{“}task_i\text{”}). \\ & \mathbf{put}(\text{“}targetLocation\text{”}, x, y)@\mathsf{self}. \\ & \mathbf{get}(\text{“}informed\text{”}, \mathsf{false})@\mathsf{self}.\ \mathbf{put}(\text{“}informed\text{”}, \mathsf{true})@\mathsf{self} \end{aligned}$$

This process corresponds to the process P_1 described in the running example shown in Section 3. After it has retrieved the target location from the other robots doing the same task, the process publishes such information within the local repository and sets the $informed$ tuple to true.

Process $P_{randomWalk}$, executed by the managed element, randomly selects a direction followed by the robot to search the target area, while process $P_{informed}$ calculates a direction towards a given location. Hence, they both add a tuple $\langle \text{“}direction\text{”}, \theta \rangle$ to the local repository, which will be retrieved by the actuator governing the robot wheels. The two processes are defined as follows:

$$P_{randomWalk} \triangleq \mathbf{put}(\text{“}direction\text{”}, random() \cdot 2\pi)@\mathsf{self}.\ \mathbf{put}(\text{“}terminated\text{”})@\mathsf{self}$$

$$\begin{aligned} P_{informed} \triangleq\ & \mathbf{qry}(\text{“}targetLocation\text{”}, ?x, ?y)@\mathsf{self}. \\ & \mathbf{put}(\text{“}direction\text{”}, towards(x, y))@\mathsf{self}.\ \mathbf{put}(\text{“}terminated\text{”})@\mathsf{self} \end{aligned}$$

7. A RUNTIME ENVIRONMENT FOR SCEL PROGRAMS

In this section we present jRESP[2], 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

[2]jRESP (Java Run-time Environment for SCEL Programs) website: http://jresp.sourceforge.net/.

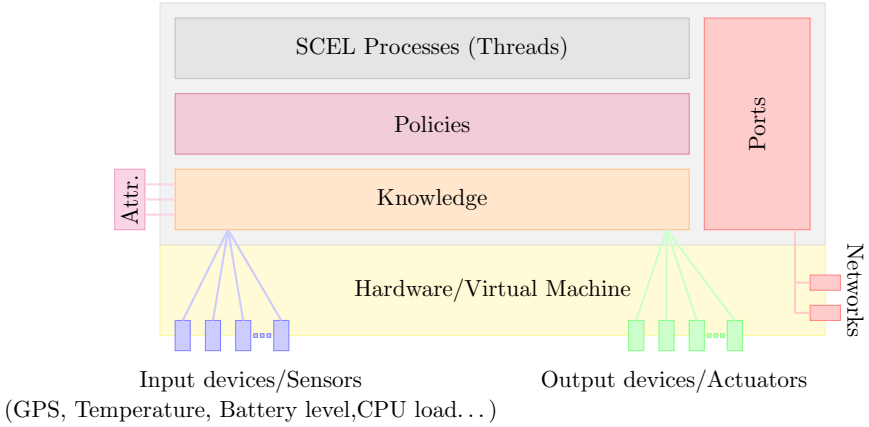

Fig. 3. Node architecture

linguistic constructs for controlling the computation and interaction of autonomic components, and for defining the architecture of systems and ensembles.

The implementation of jRESP fully relies on the SCEL formal semantics. This close correspondence enhances confidence on the behaviour of the jRESP implementation of SCEL programs, once the latter have been analysed through the formal methods made possible by the formal operational semantics.

We have already explained in the previous sections that SCEL is parametric with respect to some aspects, e.g. knowledge representation, that may change to tailor to different application domains. For this reason, also jRESP is designed to accommodate alternative instantiations of the above mentioned features. Indeed, thanks to the large use of design patterns, the integration of new features in jRESP is greatly simplified.

SCEL operational semantics abstracts from a specific communication infrastructure. A SCEL *program* typically consists of a set of (possibly heterogeneous) components, each of which is equipped with its own knowledge repository. These components concur and cooperate in a highly dynamic environment to achieve a set of *goals*. In this kind of systems the underlying communication infrastructure can change dynamically as the result of local component interactions. To cope with this dynamicity, jRESP communication infrastructure has been designed to avoid *centralized control*. Moreover, to facilitate interoperability with other tools and programming frameworks, jRESP relies on JSON[3]. This is an open data interchange technology that permits simplifying the interactions between heterogeneous network components and provides the basis on which SCEL programs can cooperate with external services or devices.

7.1. jRESP's main features

Components. SCEL components are implemented via the class Node. The architecture of a node is shown in Figure 3. Nodes are executed over virtual machines or physical devices providing access to input/output devices and network connections. A node aggregates a knowledge repository, a set of running processes, and a set of policies. Structural and behavioral 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.

[3]JSON (JavaScript Object Notation) website: http://www.json.org/.

Knowledge. The interface Knowledge identifies a generic knowledge repository and indicates the high-level primitives to manage pieces of relevant information coming from different sources. This interface contains the methods for withdrawing/retrieving/adding piece of knowledge from/to a repository. Currently, a single implementation of the Knowledge interface is available in jRESP, which relies on the KLAIM [De Nicola et al. 1998] notion of tuple space. Thus, 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.

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 SCEL processes to control exogenous devices that identify logical/physical actuators.

The interface associated to a node is computed by exploiting *attribute collectors*. Each of this 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[4].

Network Infrastructure. Each Node 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. 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, ...).

Currently, four kinds of port are available: InetPort, P2PPort, ServerPort and VirtualPort. The first one implements point-to-point and group-oriented interactions via TCP and UDP, respectively. In particular, InetPort implements group-oriented interactions in terms of a UDP broadcast. Unfortunately, this approach does not scale when the size of involved components increases. To provide a more efficient and reliable support to group-oriented interactions, jRESP provides the class P2PPort. This class realises interactions in terms of the *P2P* and *multicast* protocols provided by Scribe[5] [Castro et al. 2003] and FreePastry[6] [Rowstron and Druschel 2001]. A more centralized implementation is provided by ServerPort. All messages sent along this kind of port pass through a centralize server that dispatches all the received messages to each of the managed ports. Finally, VirtualPort implements a port where interactions are performed via a buffer stored in memory. A VitualPort is used to *simulate* nodes in a single application without relying on a specific network infrastructure.

[4]This mechanism is implemented via the *Observer/Observable* pattern.
[5]Scribe is a generic, scalable and efficient system for group communication and notification.
[6]FreePastry is a substrate for peer-to-peer applications.

Behaviors. SCEL processes are implemented as threads via the abstract class Agent, which provides the methods implementing the SCEL actions. In fact, they can be used for generating fresh names, for instantiating new components and for withdrawing/retrieving/adding information items from/to shared knowledge repositories. The latter methods extend the ones 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.

Policies. Like in SCEL, in jRESP policies can be used to regulate the interaction between the different internal parts of components and their mutual interactions. When a method of an instance of 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 generating an exception when some access right has been violated) and, possibly, of related extra actions. By default, each node is instantiated with the policy allowing any operation. Different kinds of policies can be easily integrated in jRESP by implementing the interface PolicyI.

7.2. The swarm robotics scenario in jRESP

We report here the relevant code[7] of the jRESP implementation of the SCEL specification, presented in Section 6, of the swarm robotics scenario. The Java classes reported in this section permit appreciating how close SCEL processes are to their implementation in jRESP. Indeed, since programmers can directly use SCEL communication primitives, the resulting classes are very compact. The jRESP code of the swarm robotic scenario consists of only eight classes, each of which represents a specific behaviour. The average number of lines per class, including class and method declarations, is 12.

Process ME, defined in Equation (1) at page 18, can be rendered as the agent ManagedElement defined below:

```
public class ManagedElement extends Agent {
   public ManagedElement() {
      super("ManagedElement");
   }
   protected void doRun() throws Exception {
      while (true) {
         Tuple t = query(new Template(new ActualTemplateField("controlStep"),
                                      new FormalTemplateField(Agent.class)),
                         Self.SELF);
         Agent X = t.getElementAt(Agent.class,1);
         X.call ();
      }
   }
}
```

When an instance of class Agent is executed, it is invoked the method doRun() that defines the agent behaviour. In the case of ManagedElement, the method consists of an infinite loop where, at each iteration, the *control step* is first retrieved from the local knowledge repository and then executed. The method query(), used to retrieve data from a knowledge repository, is defined in the base class Agent and implements the

[7]The source code for the complete scenario, together with a simulation environment, can be downloaded from http://jresp.sourceforge.net/.

SCEL action **qry**[8]. It takes as parameters an instance of class Template and a target, and returns a matching tuple. In the case above, the target is the local component (referred by Self.SELF) while the retrieved tuple is one consisting of two fields: the first field is the constant "controlStep" and the second field is an instance of class Agent. The latter is executed (via method call()) once the tuple is read from the tuple space.

The *autonomic manager* is implemented by the three Java classes corresponding to processes $P_{batteryMonitor}$, $P_{dataSeeker}$ and $P_{targetSeeker}$. We report below, as an example, the code of DataSeeker.

```
public class DataSeeker extends Agent {
   public DataSeeker() {
      super("DataSeeker");
   }
   protected void doRun() throws Exception {
      Tuple t = query(new Template(new ActualTemplateField("targetLocation"),
                                   new FormalTemplateField(Double.class),
                                   new FormalTemplateField(Double.class)),
                      new Group(new HasValue( "task",1)));
      double x = t.getElementAt(Double.class,1);
      double y = t.getElementAt(Double.class,2);
      put(new Tuple("targetLocation",x,y), Self.SELF);
      get(new Template(new ActualTemplateField("informed"),
                       new ActualTemplateField(false)),
          Self.SELF);
      put(new Tuple("informed",true), Self.SELF);
   }
}
```

Method doRun() of DataSeeker implements exactly the same behaviour as $P_{dataSeeker}$. This is witnessed by the clear correspondence between the code listed above and the definition of $P_{dataSeeker}$ given in Section 6. In particular, the method query() is performed by contacting the components satisfying the predicate HasValue("task",1), i.e. those components whose interface associates the value 1 to the task attribute. The ports associated to each jRESP component will provide specific protocols that permit discovering the nodes satisfying the target predicate. For instance, in the case of InetPort this task is performed by relying on UDP broadcast.

Finally, the jRESP code of the processes $P_{randomWalk}$ and $P_{informed}$, executed by the managed element, is the following:

```
public class RandomWalk extends Agent {
   Random r = new Random();
      public RandomWalk() {
         super("RandomWalk");
      }
      protected void doRun() throws Exception {
         put(new Tuple("direction",r.nextDouble()*2*Math.PI), Self.SELF);
      }
}

public class Informed extends Agent {
   public Informed() {
      super("Informed");
   }
   protected void doRun() throws Exception {
```

[8]Class Agent also provides methods put() and get() that implements actions **put** and **get**, respectively.

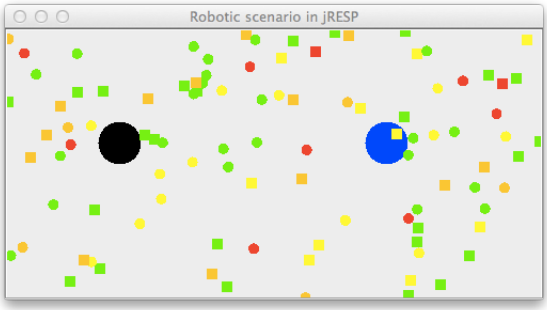

Fig. 4. Simulation of the swarm robotic scenario in jRESP

```
        Tuple t = query(new Template(new ActualTemplateField("targetLocation"),
                                     new FormalTemplateField(Double.class),
                                     new FormalTemplateField(Double.class)),
                        Self.SELF);
        double x = t.getElementAt(Double.class,1);
        double y = t.getElementAt(Double.class,2);
        put(new Tuple("direction",towards(x,y)), Self.SELF);
    }
}
```

The jRESP code presented in this section can be executed both on real robots and within a simulation environment. In the first case, jRESP nodes would be directly executed on robots where a *Java Virtual Machine* is running. However, this may not be always possible. For this reason, jRESP also provides modules that can be used to simulate SCEL programs. Simulation modules enable the execution of *virtual components* within a simulation environment that controls component interactions and collects relevant simulation data. A screenshot of a simulation run of the robotic scenario is reported in Figure 4, where blue and black circles represent the locations of the two target zones. In the figure, the two groups of robots are represented via squares and circles while their color denotes the battery level: green if the battery is completely charged, red if the battery is empty.

By relying on the jRESP simulation environment, a prototype framework for *statistical model-checking* has been also developed. Following this approach, 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* ε and *error probability* p. The used algorithm guarantees that the difference between the value computed by the algorithm and the exact one is greater than ε with a probability that is less than p.

The model-checker included in jRESP can be used to verify *reachability properties*. These permit evaluating, e.g., the probability to reach a configuration where a given predicate on collected data is satisfied within a given deadline. Figure 5 shows the probability that "at least 25% of robots reach the target when the time varies from 0 to $4000s$". The diagram shows that this goal can be reached only after $500s$ and that after $2500s$ with probability 1 at least $\frac{1}{4}$ of robots reach their target.

8. RELATED WORK

Our proposal combines the notion of ensemble with concepts that have emerged from different research fields, such as autonomic computing, multi-agent systems, component-based design, context-oriented programming, network architectures and

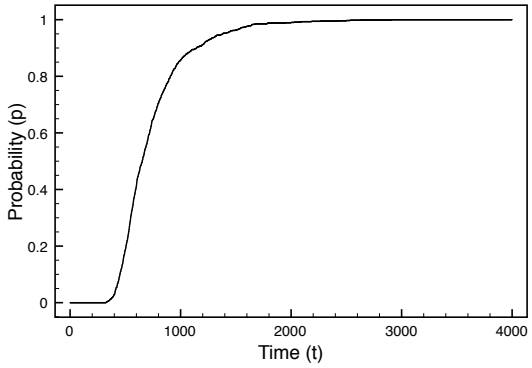

Fig. 5. Statistical model-checking of robotic scenario in jRESP

concurrency theory. Below, we list some of the closely related works developed within these areas and stress their relationships with SCEL.

Declarative programming has been proposed to program ensembles, see e.g. Meld [Ashley-Rollman et al. 2007; Ashley-Rollman et al. 2009] and Declarative Networking [Loo et al. 2009]. The underlying idea is that ensembles can be programmed as a unified whole from a global perspective and then compiled automatically into fully distributed local behaviors. In this context, SCEL could be used as the actual target language.

Among the many works focussing on the self-adaptation capability of autonomic systems, we want to mention [Khakpour et al. 2012]. This work proposes a policy-based formalism that combines an actor-based model, for specifying the computational aspects of system elements, and a configuration algebra, for expressing autonomous managers in charge of enforcing 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. SCEL, instead, is parametric with respect to the policy language and, hence, more appropriate for dealing with heterogenous systems and different application domains.

Multi-agent systems (as e.g. [Rao 1996; Bordini et al. 2005; Winikoff 2005; Bellifemine et al. 2007; Dastani 2008]) share with SCEL the stress on knowledge representation and the way single agents can handle it. However, SCEL components can directly access knowledge repositories of other components (provided this is allowed by the related policies), while in agents system this requires additional message exchanges. In general, the SCEL's communication model, and its tuple-spaces-based implementation of the knowledge repositories, is more flexible and better suitable to, e.g., support adaptive context-aware activities in pervasive and mobile computing scenarios (as those considered in [Mamei and Zambonelli 2009]).

Being our notion of ensemble based on components equipped with interfaces, our work is also related to component-based design, that has been indicated as a key approach for adaptive software design [McKinley et al. 2004]. A relevant example in this field is FRACTAL [Bruneton et al. 2006], a hierarchical component model with sharing. This latter feature permits defining components whose boundaries are not completely fixed, which can be used to form systems with a less rigid structure than that obtained with the standard component-based paradigm. However, communication between components is still defined via bindings (i.e. component connectors) and system adaptation is obtained by adding, removing or modifying components and/or bindings. These forms of communication and adaptation are therefore less flexible and expres-

sive then the corresponding mechanisms used in SCEL and not adequate to deal with highly dynamic ensembles.

Context-Oriented Programming (COP) [Hirschfeld et al. 2008; Salvaneschi et al. 2012] has been advocated to program autonomic systems [Salvaneschi et al. 2011]. It exploits ad-hoc explicit language-level abstractions to express context-dependent behavioral variations and their run-time activation. So far, most efforts have been directed towards the design and implementation of concrete languages, as e.g. Erlang, Java, JavaScript, Python, Ruby, and Smalltalk (a comparison can be found in [Appeltauer et al. 2009]). Only few works provide a foundational account of programming languages extended with COP facilities like, e.g., the object-oriented ones of [Clarke et al. 2009; Hirschfeld et al. 2011; Aotani et al. 2011] and the functional one of [Degano et al. 2012]. 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.

A few programming abstractions that are related to the ones provided by SCEL have been recently proposed in the field of network architectures for mobile opportunistic applications and for wireless sensor networks. For example, the Haggle network architecture [Nordström et al. 2009] provides a push-based data dissemination service that notifies applications when data matching their interests is received. Applications need not themselves implement essential mechanisms for opportunistic communication, such as neighbour discovery and data dissemination, but only to register with Haggle their interest. This, and other similar forms of communication adopted in publish-subscribe architectures, can be easily rendered in SCEL by exploiting attributes for registering components' interests and by using predicates on those attributes for disseminating data to the registered components. [Mottola and Picco 2006; Mottola and Picco 2012] introduces the concept of *logical neighbourhoods* and the SPIDEY declarative language for defining them. Logical neighbourhood replaces the physical neighbourhood –i.e., the set of nodes in the communication range of a given device– provided by wireless broadcast with a higher-level notion of proximity determined by applicative information. Application programmers still reason in terms of neighbourhood relations and broadcast messages, but can now specify declaratively which nodes to consider as neighbours and, therefore, the span of communication. The communication mechanism enabled by the notion of logical neighbourhood is similar to the SCEL's one: predicates can indeed be thought of as a way of singling out the logical neighbours of a given node according to the features indicated by the attributes used in the predicates themselves. However, in [Mottola and Picco 2006; Mottola and Picco 2012] neighbourhood relations are statically defined through templates, while SCEL allows processes to form and use new predicates on-demand. Moreover, the SPIDEY language is specific for Wireless Sensor Networks (WSNs), while SCEL constructs are aimed at coordinating a larger class of systems/applications. For example, the interface of a SCEL component permits abstracting from the specific data source while it synthesises all the relevant part of the (state of) component's knowledge in a set of valued attributes. In the SCEL approach, WSNs are no longer stand-alone sense-only systems but can be easily integrated into a general framework where multiple concurrent applications coexist and cooperate.

Finally, in the area of concurrency theory, calculi such as those defined in [Banâtre et al. 2004] and in [Andrei and Kirchner 2009], relying on the (bio)chemical programming paradigm, have been proposed for the specification of autonomic systems. Some other formalisms, like e.g. those introduced in [Mezzetti and Sangiorgi 2006] and in [Singh et al. 2010], aiming at modelling dynamically changing network topologies (a feature common to many types of distributed systems and to ensembles) can also be source of inspiration for linguistic primitives for specifying autonomic systems. Com-

pared to these proposals, SCEL allows one to provide high-level abstract descriptions of systems that nevertheless have a direct correspondence with their implementation.

9. CONCLUSIONS AND FUTURE WORK

We have presented the kernel language SCEL, i.e. a set of linguistic abstractions for programming autonomic systems, and its Java implementation. Our holistic approach to programming autonomic computing systems permits to govern systems complexity by providing flexible abstractions, by permitting transparent monitoring of the involved entities and by supporting adaptation. Besides, the solid semantic ground of SCEL lays the basis for developing logics, tools and methodologies for formally reasoning on systems behavior in order to establish qualitative and quantitative properties of both the individual components and their ensembles.

To assess to which extent SCEL meets our expectations, we have used it to tackle a number of case studies from the robotics and service provision domains (see, e.g., [De Nicola et al. 2013]). We plan to extend the above experimentation to other application domains, such as Cloud-computing (transiently available computers) and e-Mobility (cooperative e-vehicles).

We also want to develop a methodology that enables components to take decisions about possible alternative behaviors by choosing among the best possibilities while being aware of the consequences. By relying on an abstract model of the evolving environment, each component will be able to locally verify the possibility (or the probability) of guaranteeing the wanted properties or of achieving the wanted goals by analyzing the possible outcome of its interactions with the abstract model. This information will be then used to take decisions about the choices that the component has to face.

Along the same lines we have started investigating the integration of SCEL with "reasoners" to be invoked by processes when facing choices. Having two different languages, one for computation and coordination the other for "reasoning", does guarantee *separation of concerns*. Also, it may be beneficial to have a methodology for integrating with a given programming language different reasoners or meta-reasoners designed and optimised for specific purposes. What we envisage is having SCEL processes that whenever need to take decisions have the possibility of invoking a reasoner by providing it with information about the relevant knowledge they have access to and receiving in exchange informed suggestions about how to proceed.

Moreover, we plan to define a *high-level* programming language that, by enriching SCEL with standard constructs (e.g. control flow constructs such as **while** or **if-then-else**), simplifies the programming task. We intend to implement an integrated environment for supporting the development of adaptive systems at different levels of abstraction: from a high-level perspective, based on SCEL, to a more concrete one, based on jRESP. (Semi-)Automatic analysis tools, based on the SCEL's formal semantics, will be integrated in this toolchain.

ACKNOWLEDGMENTS

We would like to thank Martin Wirsing and all friends of the ASCENS project, without their contributions and stimuli SCEL would not have been conceived. We would also like to thank GianLuigi Ferrari for former collaborations and discussions, and Guido Salvaneschi for comments on a previous version of this paper.

References

Gul A. Agha. 1990. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press. I–IX, 1–144 pages.

Oana Andrei and Hélène Kirchner. 2009. A Higher-Order Graph Calculus for Autonomic Computing. In *Graph Theory, Computational Intelligence and Thought*. Springer, 15–26.

Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. 2011. Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition. In *COP*. ACM, 1:1–1:7.

Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. 2009. A comparison of context-oriented programming languages. In *COP*. ACM, 6:1–6:6.

Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, and Padmanabhan Pillai. 2007. Meld: A declarative approach to programming ensembles. In *IROS*. IEEE, 2794–2800.

Michael P. Ashley-Rollman, Peter Lee, Seth Copen Goldstein, Padmanabhan Pillai, and Jason Campbell. 2009. A Language for Large Ensembles of Independently Executing Nodes. In *ICLP (LNCS 5649)*. Springer, 265–280.

Jean-Pierre Banâtre, Yann Radenac, and Pascal Fradet. 2004. Chemical Specification of Autonomic Systems. In *IASSE*. ISCA, 72–79.

Fabio L. Bellifemine, Giovanni Caire, and Dominic Greenwood. 2007. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons.

Rafael H. Bordini, Jomi F. Hübner, and Renata Vieira. 2005. Jason and the Golden Fleece of Agent-Oriented Programming. In *Multi-Agent Programming*. Multiagent Systems, Artificial Societies, and Simulated Organizations, Vol. 15. Springer, 3–37.

Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. 2006. The FRACTAL component model and its support in Java. *Softw., Pract. Exper.* 36, 11-12 (2006), 1257–1284.

Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. 2012. A Conceptual Framework for Adaptation. In *FASE (LNCS 7212)*. Springer, 240–254.

Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony I. T. Rowstron. 2003. Scalable Application-Level Anycast for Highly Dynamic Groups. In *ICQT (LNCS 2816)*. Springer, 47–57.

Dave Clarke, Pascal Costanza, and Éric Tanter. 2009. How should context-escaping closures proceed?. In *COP*. ACM, 1:1–1:6.

Paolo Costa, Luca Mottola, Amy L. Murphy, and GianPietro Picco. 2009. Tuple Space Middleware for Wireless Networks. In *Middleware for Network Eccentric and Mobile Applications*. Springer, 245–264.

Mehdi Dastani. 2008. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems* 16, 3 (2008), 214–248.

Rocco De Nicola, GianLuigi Ferrari, Michele Loreti, and Rosario Pugliese. 2012. A Language-based Approach to Autonomic Computing. In *FMCO 2011 (LNCS 7542)*. Springer, 25–48. http://rap.dsi.unifi.it/scel/.

Rocco De Nicola, GianLuigi Ferrari, and Rosario Pugliese. 1998. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. Software Eng.* 24, 5 (1998), 315–330.

Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. 2013. SCEL: a Language for Autonomic Computing. Technical Report. (July 2013). http://rap.dsi.unifi.it/scel/pdf/SCEL-TR.pdf.

Pierpaolo Degano, GianLuigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. 2012. Types for Coordinating Secure Behavioural Variations. In *COORDINATION (LNCS 7274)*. Springer, 261–276.

Edmond Gjondrekaj, Michele Loreti, Rrosario Pugliese, and Francesco Tiezzi. 2012. Modeling adaptation with a tuple-based coordination language. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, Sascha Ossowski and Paola Lecca (Eds.). ACM, 1522–1527.

Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented Programming. *Journal of Object Technology* 7, 3 (2008), 125–151.

Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. 2011. ContextFJ: a minimal core calculus for context-oriented programming. In *FOAL*. ACM, 19–23.

IBM. 2005. *An architectural blueprint for autonomic computing*. Technical Report. Third edition.

Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50.

Narges Khakpour, Saeed Jalili, Carolyn L. Talcott, Marjan Sirjani, and Mohammad Reza Mousavi. 2012. Formal modeling of evolving self-adaptive systems. *Sci. Comput. Program.* 78, 1 (2012), 3–26.

Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative networking. *Commun. ACM* 52, 11 (2009), 87–95.

Marco Mamei and Franco Zambonelli. 2009. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. Softw. Eng. Methodol.* 18, 4 (2009).

P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B. H C Cheng. 2004. Composing adaptive software. *Computer* 37, 7 (2004), 56–64.

Nicola Mezzetti and Davide Sangiorgi. 2006. Towards a Calculus For Wireless Systems. *Electr. Notes Theor. Comput. Sci.* 158 (2006), 331–353.

Robin Milner. 1989. *Communication and concurrency*. Prentice Hall. I–XI, 1–260 pages.

Luca Mottola and Gian Pietro Picco. 2006. Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks. In *DCOSS (LNCS 4026)*. Springer, 150–168.

Luca Mottola and Gian Pietro Picco. 2012. Middleware for wireless sensor networks: an outlook. *J. Internet Services and Applications* 3, 1 (2012), 31–39.

NIST. 2009. A survey of access control models. (2009). http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf.

Erik Nordström, Per Gunningberg, and Christian Rohner. 2009. *A Search-based Network Architecture for Mobile Devices*. Technical Report 2009-003. Uppsala University, Computer Systems.

OASIS-TC. 2005. eXtensible Access Control Markup Language (XACML) version 2.0. (2005). http://docs.oasis-open.org/xacml/2.0/XACML-2.0-OS-NORMATIVE.zip.

Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebr. Program.* 60-61 (2004), 17–139.

Project InterLink. 2007. http://interlink.ics.forth.gr. (2007).

Anand S. Rao. 1996. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *MAAMAW (LNCS 1038)*. Springer, 42–55.

Antony I. T. Rowstron and Peter Druschel. 2001. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware (LNCS 2218)*. Springer, 329–350.

Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2011. Context-Oriented Programming: A Programming Paradigm for Autonomic Systems. *CoRR* abs/1105.0069 (2011).

Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2012. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software* 85, 8 (2012), 1801 – 1817.

Anu Singh, C. R. Ramakrishnan, and Scott A. Smolka. 2010. A process calculus for Mobile Ad Hoc Networks. *Sci. Comput. Program.* 75, 6 (2010), 440–469.

Ian Sommerville, Dave Cliff, Radu Calinescu, Justin Keen, Tim Kelly, Marta Z. Kwiatkowska, John A. McDermid, and Richard F. Paige. 2012. Large-scale complex IT systems. *Commun. ACM* 55, 7 (2012), 71–77.

Michael Winikoff. 2005. JACK^{TM} Intelligent Agents: An Industrial Strength Platform. In *Multi-Agent Programming*. Multiagent Systems, Artificial Societies, and Simulated Organizations, Vol. 15. Springer, 175–193.