SCEL: a Language for Autonomic Computing*

Rocco De Nicola¹, Michele Loreti², Rosario Pugliese², and Francesco Tiezzi¹

¹ IMT, Institute for Advanced Studies Lucca, Italy
² Università degli Studi di Firenze, Italy

WORKING DRAFT – October 8, 2013

Abstract. 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. In this document, we first present the syntax and operational semantics of SCEL. We then demonstrate that adaptation can be naturally modeled in SCEL, and present a simple policy language and its integration with SCEL. Finally, to show expressiveness and effectiveness of SCEL’s design, we present a Java implementation of the SCEL’s programming abstractions and illustrate their use by specifying two case studies from the service provision and robotics domains.

* This work has been partially sponsored by the EU project ASCENS (257414).
Table of Contents

1 Introduction .......................................................... 3
2 SCEL Syntax .......................................................... 6
3 SCEL Operational Semantics ........................................... 11
  3.1 Semantics of processes .............................................. 11
  3.2 Semantics of systems ............................................... 13
4 Interaction Predicates .................................................. 21
5 Self-adaptation in SCEL ............................................... 24
6 SCEL at work .......................................................... 27
  6.1 A SCEL dialect based on tuple-spaces ............................. 27
  6.2 A service provision scenario in SCEL_TS ......................... 29
  6.3 A swarm robotics scenario in SCEL_TS ......................... 31
7 SACPL: a SCEL access control policy language ...................... 34
  7.1 SACPL syntax .................................................... 35
  7.2 SACPL Semantics .................................................. 37
  7.3 Integration with SCEL .............................................. 39
8 SCEL and SACPL at work on a cloud computing scenario ............. 41
  8.1 Scenario specification in SCEL .................................... 42
  8.2 The first approach: use an additional attribute .................. 43
  8.3 The second approach: use obligations ............................. 44
  8.4 The third approach: use automatons ............................... 47
9 A runtime environment for SCEL programs ............................ 48
  9.1 jRESP’s main features ............................................. 49
  9.2 The swarm robotics scenario in jRESP ............................. 51
10 Related work .......................................................... 56
11 Concluding remarks and future directions ........................... 58
1 Introduction

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 be designed to operate in open and non-deterministic environments, and to dynamically adapt to new requirements, technologies and environmental conditions. This class of systems has been named ensembles \[1\]. Sometimes, ensembles 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 such systems. Therefore, new engineering techniques are needed to address the challenges of developing, integrating, and deploying these large-scale, complex software-intensive systems \[2\].

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 to best deal 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 \[3,4\].

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. ACs and ACEs will be our means to structure systems into well-understood, independent and distributed building blocks that interact and adapt in different ways.

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 to adapt their behavior to the perceived changes. Each AC is equipped with an interface, consisting of a collection of attributes, describing different component’s features such as its 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 sets of partners are selected for interaction and thus how ensembles are formed. Individual ACs can single out communication partners by using their identities, but they can also select them by exploiting the attributes in their 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 identify dynamic (overlay) subnetworks of ACs by exploiting specific attributes; in the picture, these are the different colours (or grey shades) associated to individual ACs.

In this work, we present 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. These abstractions permit describing autonomic systems in terms of *Behaviors*, *Knowledge* and *Aggregations*, according to specific *Policies*.

- *Behaviors* describe how computations 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 in Section 5). Moreover, by implementing appropriate security policies, e.g. limiting information flow or external actions, components can set up *self-protection* mechanisms.

Our aim is providing a common semantic framework for describing the meaning of these abstractions and their interplay, while minimizing overlaps and incompatibilities. We have thus identified linguistic constructs for uniformly programming the evolution and the interactions of ACs and the architecture of ACEs. The language is however, somehow, minimal; SCEL syntax fully specifies only constructs for modeling Behaviors and Aggregations and is parametric with respect to Knowledge and Policies. This choice enables us to integrate different approaches to knowledge handling or to policies specifications 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 consider a SCEL’s dialect where knowledge repositories are implemented as multiple distributed *tuple-spaces* and policies are written in a simple, yet expressive, language for defining access control policies. This version of the language is now 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 in Java programs the SCEL’s linguistic constructs 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 separation of concerns and the importance of component-based design [5], from multi-agent systems the relevance of knowledge handling and of spatial representation [6,7,8,9,10], from middleware and network architectures the importance of flexibility in communication [11,12,13,14,15], from distributed systems’ security the role of policies [16], from
actors and process algebras the importance of minimality and formality [17][18]. 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 [19]. Here, ACEs are dynamically ‘synthesized’ via group-oriented, attribute-based communication and used as target of actions, while in [19] 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 [19] 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. Sections 2 and 3 formally introduce the syntax and operational semantics of SCEL, respectively. Section 4 reports the definitions of three examples of interaction predicate, one of the parameters on which the operational semantics relies. Section 5 shows that (self-)adaptation can be naturally rendered in SCEL. Section 6 shows SCEL at work on two case studies in order to illustrate the expressiveness and effectiveness of the approach. Section 7 introduces a language for defining access control policies, and shows its integration with SCEL, while Section 8 illustrates the use of the integrated language on a cloud computing scenario. Section 9 presents the Java runtime environment for developing autonomic and adaptive systems according to the SCEL paradigm. Finally, Section 10 surveys related work, while Section 11 concludes by touching upon directions for future work.

2 SCEL Syntax

SCEL relies on the notion of autonomic component \(\mathcal{I}[K, \Pi, P]\). 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 \(K\) 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
Fig. 2. SCEL component

(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 $\Pi$ regulating the interaction between the different internal parts of the component and the interaction of the component with the others. 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.

- 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.

The syntax of SCEL is presented in Table 1. There, different syntactic categories are defined that constitute the main ingredients of our language. The 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).

In the rest of this section, we consider one by one the explicitly defined categories of the SCEL’s syntax and describe them in detail. We illustrate the features of the language in a step-by-step fashion using a running example\footnote{The running example will end in Section 3. We refer to Section 6.3 for a more complete account of the swarm robotics scenario from which the running example is taken.} from
Systems:
\[ S ::= C \mid S_1 \parallel S_2 \mid (\nu n)S \]

Components:
\[ C ::= I[K, \Pi, P] \]

Processes:
\[ P ::= \text{nil} \mid a.P \mid P_1 + P_2 \mid P_1[P_2] \mid X \mid A(p) \]

Actions:
\[ a ::= \text{get}(T)@c \mid \text{qry}(T)@c \mid \text{put}(t)@c \mid \text{fresh}(n) \mid \text{new}(I, K, \Pi, P) \]

Targets:
\[ c ::= n \mid x \mid \text{self} \mid P \mid p \]

Table 1. SCEL syntax (Knowledge \( K \), Policies \( \Pi \), Templates \( T \), and Items \( t \) are parameters of the language)

The robotics domain. In the example, we consider a swarm of robots that have to reach different zones according to the tasks that they have to do (for the sake of simplicity, either task \(_1\) or task \(_2\)). Robots are not informed about the position of the two target zones. Thus, when a robot reaches its target area, it ‘publishes’ the location within its local knowledge repository in order to make it available to robots with the same task.

**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 1/6).** The robotics scenario can be expressed in SCEL as a system \( S \) defined as follows

\[
S \triangleq I_1[K_1, \Pi_1, P_1] \parallel I_2[K_2, \Pi_2, P_2] \\
\parallel I_3[K_3, \Pi_3, P_3] \parallel I_4[K_4, \Pi_4, P_4] \parallel \ldots
\]

The robots are rendered as components, identified by \( I_i.id \) (i.e., the values of attribute \( id \) exposed in their interfaces \( I_i \)), 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 \((A(p))\). We will omit trailing occurrences of nil, writing e.g. \(a\) instead of \(a.nil\). 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. As shown in [20], this form of higher-order communication enables a straightforward implementation of adaptive behaviors. 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 \(A(f) \triangleq P.\bar{p}\) and \(\bar{f}\) denote lists of actual and formal parameters, respectively.

Running example (step 2/6). The process \(P_1\) running on the first robot, i.e. component \(I_1[K_1,\Pi_1,P_1]\), 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 \(get(T)@c, qry(T)@c\) and \(put(t)@c\) 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 \(fresh(n)\) 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 \(new(I,K,\Pi,P)\) creates a new component \(I[K,\Pi,P]\).

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 \(n, n', \ldots\), while variables for names are denoted by \(x, x', \ldots\). 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 \texttt{this.status = “sending” \lor 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 \( P \) 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 \( P \) to be the target of the action. Thus, actions \texttt{put(t)@n} and \texttt{put(t)@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 (see also Remark 2 at page 19).

The set of components satisfying a given predicate \( P \) 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 \( \text{id} \in \{n,m,o\} \). 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 \( \text{low} \), by assuming that attributes \text{active} and \text{batteryLevel} belong to the interface of any component willing to be part of the ensemble, one can write \( \text{active} = \text{“yes”} \land \text{batteryLevel} > \text{“low”} \).

\textbf{Running example (step 3/6).} By specifying actions \( a_1 \) and \( a_2 \) as a \texttt{qry} and a \texttt{put} action, respectively, the process \( P_1 \) becomes

\begin{align*}
\text{\texttt{qry}}(\text{“targetLocation”,?x,?y}@\text{task} = \text{“task}_1\text{”}). \\
\text{\texttt{put}}(\text{“targetLocation”,x,y}@\text{self. } P_1^').
\end{align*}

This process retrieves the target location from one of the informed robots in charge of doing \text{task}_1, binds\(^4\) the location’s coordinates to variables \( x \) and \( y \), and publishes such information in the local repository.

\( \Box \)

\textbf{Remark 1 (On dynamically determined communication partners).} Differently from the original version of SCEL \[19\], attribute \text{ensemble} is no longer part of components’ interface. The choice of dynamically determining an ensemble

\(^4\) We use here the symbol ‘?’ to indicate a variable binder in a template. We refer to Section 6.1 for more details on how such a binding can be implemented.
as a target of an action, rather than having it necessarily characterized by an interface attribute, has many important consequences. Firstly, it avoids to have a single component acting as the coordinator of the ensemble (the coordinating component would be a single point of centralisation and, potentially, of failure). Secondly, it permits a more dynamic characterisation of ensembles, since the target ensemble can potentially differ from one action to the next one. Finally, it simplifies the operational semantics, since an interaction between two components does not require a third party, i.e. the ensemble coordinator. Also attribute membership is no longer part of components’ interface; its role can indeed be held by the authorisation predicate (see the operational semantics rules for systems).

3 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.

To define the semantics, we use the sets of bound variables \( \text{bv}(E) \) and free variables \( \text{fv}(E) \), and the sets of names \( \text{n}(E) \), bound names and free names occurring in a syntactic term \( E \). These sets, as usual, can be defined inductively on the syntax of actions, processes, components, and systems by taking into account that the only binding constructs are actions \text{get} and \text{qry} as concerns variables and action \text{fresh} and the restriction operator as concerns names. More precisely, actions \text{get}(T)@c and \text{qry}(T)@c bind the variables occurring in the template \( T \), while action \text{fresh}(n) binds the name \( n \); the scope of these binders is the process \( P \) syntactically following the action in a prefix form \( a.P \). The restriction operator \((\nu n)\_\) binds \( n \) in the scope \__. A term without free variables is deemed closed (notably, it may contain free names).

The semantics is only defined for closed systems. Indeed, we consider the binding of a variable as its declaration (and initialization), therefore free occurrences of variables at the outset in a system must be prevented since they are similar to uses of variables before their declaration in programs (which are considered as programming errors).

3.1 Semantics of processes

The semantics of processes specifies process commitments, namely given a process \( P \), its semantics points out all the actions that \( P \) can initially perform and the continuation process \( P' \) obtained after each such action. To simplify the rules, we do not restrict them (and the semantics) to the subset of closed processes, although when defining the semantics of systems we only consider the commitments from closed processes. Moreover, we only consider processes that
are such that their bound names are pairwise distinct and different from their free names (to avoid improper name captures).

The relation \( \downarrow \) defining the semantics of processes is the least relation induced by the inference rules in Table 2. We use \( P \) and \( Q \), possibly indexed, to range over processes and write \( P \downarrow_\alpha Q \) instead of \( \langle P, \alpha, Q \rangle \in \downarrow \), to mean that “\( P \) can commit to perform \( \alpha \) and become \( Q \) after doing so.” Process commitments are generated by the following production rule

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

meaning that a commitment is either an action \( a \) as defined in Table 1, or the symbol \( \circ \), denoting inaction, or the composition \( \alpha[\beta] \) of two commitments \( \alpha \) and \( \beta \).

The rules defining the relation are straightforward. In particular, a process of the form \( a.P \) is committed to do \( a \) and then to continue as process \( P \). Process \( P + Q \) nondeterministically behaves as \( P \) or as \( Q \), while a process invocation \( A(\bar{p}) \) behaves as the invoked process \( P \), where the formal parameters \( f \) have been replaced by the actual parameters \( \bar{p} \). The rule defining the semantics of \( P[Q] \) states that a commitment \( \alpha[\beta] \) is exhibited when \( P \) commits to \( \alpha \) and \( Q \) commits to \( \beta \). 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 \( \circ \) (i.e. to stay idle), \( \alpha \) and/or \( \beta \) may always be \( \circ \). The semantics of \( P[Q] \) at the level of processes is indeed absolutely 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 for taking policies into account (as we will see in Section 3.2). Condition \( \text{bv}(\alpha) \cap \text{bv}(\beta) = \emptyset \) means that the variables freed by the commitment \( \alpha[\beta] \) in the two processes \( P \) and \( Q \) must be different: this because they correspond to bound variables that were intended to be different (although they might have had the same identity) and, once they get free, should be subject to possibly different substitutions. Substitutions are (partial) functions from variables to values, which are generated by rule \((pr-sys)\) in Table 3. We will use \( \sigma \) to denote a generic substitution and \( \{\} \) to denote the empty one. Notably, also condition \( \text{bv}(\alpha) \cap \text{bv}(\beta) = \emptyset \) is not strict: it can be always made true by application of the last rule stating that alpha-equivalent processes, i.e. processes only differing in the identity of bound variables (this equivalence relation is denoted by \( \equiv \)), can perform the same commitments.
Running example (step 4/6). 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

$$P_1 \downarrow \text{qry}("targetLocation",?x,?y)@\{\text{task}="task_1"\} \rightarrow P''_1$$

with $P''_1 \triangleq \text{put}("targetLocation",x,y)@\text{self}.P_1$.

3.2 Semantics of systems

The operational semantics of systems is defined in two steps. First, we derive the possible behaviors of systems without occurrences of the name restriction operator. This is done in the SOS style [21] by relying on the notion of Labeled Transition System (LTS), which is a triple $\langle S, L, \rightarrow \rangle$ made of a set of states $S$, a set of transition labels $L$, and a labeled transition relation $\rightarrow \subseteq S \times L \times S$ accounting for the actions that can be performed from each state and the new state reached after each such transition. Then, by exploiting this LTS, we provide the semantics of generic systems by means of a (unlabeled) transition system (TS), that is a pair $\langle S, \rightarrow\rightarrow \rangle$ made of a set of states $S$ and a (unlabeled) transition relation $\rightarrow\rightarrow \subseteq S \times S$ accounting for the computation steps that can be performed from each state and the new state reached after each such transition. This approach allows us to avoid the intricacies, also from a notational point of view, arising when dealing with name mobility in computations (e.g. when opening and closing the scopes of name restrictions).

We will use $I$ and $J$ to range over interfaces. Recall that the names of the attributes of a component are just pointers to the actual values contained in the knowledge repository associated to the component. This amounts to saying that in terms of the form $I[K, \Pi, P]$, $I$ only includes the names of the attributes, as their corresponding values can be easily retrieved from $K$. However, when $I$ is used in isolation, we assume that it also includes the attributes’ values. For example, given the component $I[K, \Pi, P]$, we can use the condition $n = I.id$ to check if the value associated to the attribute $id$ in the repository $K$ is equal to the name $n$.

The LTS defining the semantics of systems without restricted names is $\langle S, L, \rightarrow \rangle$ where

- the set of states $S$ includes all the systems defined in Table 1
- $L$ is the set of transition labels generated by the following production rule

$$\lambda ::= \tau \mid I : \text{fresh}(n) \mid I : \text{new}(J, K, \Pi, P)$$

$$\mid I : t \triangleleft \gamma \mid I : t \triangleleft_{\circ} \gamma \mid I : t \triangleright \gamma \mid I : t \triangleright J \mid I : t \bigtriangleup J \mid I : t \bigtriangleup J$$

where

$$\gamma ::= n \mid P$$

is either the name $n$ of a component or a predicate $P$ indicating a set of components. The meaning of labels is as follows: $\tau$ denotes an internal computation step, $I : \text{fresh}(n)$ denotes the willingness of component $I$ to restrict
visibility of name \(n\), \(I : \text{new}(J, K, II, P)\) denotes the willingness of component \(I\) to create the new component \(J[K, II, P]\), \(I : t \triangleleft \gamma\) (resp. \(I : t \blacklozenge \gamma\)) denotes the intention of component \(I\) to withdraw (resp. retrieve) item \(t\) from the repositories at \(\gamma\), \(I : t \triangleright J\) (resp. \(I : t \blacktriangleleft J\)) denotes that component \(I\) is allowed to withdraw (resp. retrieve) item \(t\) from the repository of component \(J\), \(I : t \triangleright J\) denotes that component \(I\) is allowed to add item \(t\) to the repository of component \(J\):

- \(\rightarrow\) is the labeled transition relation induced by the inference rules in Tables 3 and 4. We write \(S \xrightarrow{\lambda} S'\), instead of \(\langle S, \lambda, S' \rangle \in \rightarrow\), to mean that “\(S\) can perform a transition labeled \(\lambda\) and become \(S'\) in doing so”. Moreover, we use \(I.\pi\) to denote the policy in force at the component \(I\) and \(S[I.\pi := \Pi']\) to denote the replacement of the policy in force at the component \(I\) with policy \(\Pi'\).

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

- The interaction predicate, \(\Pi, I : \alpha \succ \lambda, \sigma, \Pi'\), means that under policy \(\Pi\) and interface \(I\), process commitment \(\alpha\) yields system label \(\lambda\), substitution \(\sigma\) and the (possibly new) policy \(\Pi'\);
- The authorization predicate, \(\Pi \vdash \lambda, \Pi'\), means that under policy \(\Pi\), the action generating the system label \(\lambda\) (which can be thought of as an authorization request) is allowed and the policy \(\Pi'\) is produced. Notably, labels \(\lambda\) taken as argument by the authorization predicate are the subset of system labels of the form \(I : \text{fresh}(n)\), \(I : \text{new}(J, K, II, P)\), \(I : t \triangleleft J\), \(I : t \blacklozenge J\), or \(I : t \triangleright J\).

The interaction predicate establishes a relation between process commitments and system labels and thus determines the system label \(\lambda\) to exhibit, the substitution \(\sigma\) to apply and the (possibly new) policy \(\Pi'\) to be enforced when a process exhibits a commitment \(\alpha\). It is called interaction predicate because its main role is determining the effect of the simultaneous execution of actions by processes concurrently running within a component that, e.g., exhibit commitments of the form \(\alpha[\beta]\). Many different interaction predicates can thus be defined to capture well-known process computation and interaction patterns such as interleaving, asynchronous communication, synchronous communication, full synchrony,

---

Footnote: It is worth noticing that, to establish the authorization of an interaction, it is not necessary to distinguish between labels of type \(\equiv\), \(\triangleleft\) and \(\triangleright\), and labels of type \(<\), \(\blacklozenge\) and \(\triangleright\). Indeed, when the authorization predicate is verified during a transition inference (see operational rules in Tables 3 and 4), all information about the subject and object of the interaction is already known (via the interfaces \(I\) of the acting component and \(J\) of the target component). Hence, by examining such data, it is possible to recognize if it is under evaluation either the authorization to perform a local/remote action or the authorization to accept a remote access. This permits avoiding extra notations of the form \(I : t \triangleleft J\), \(I : t \blacklozenge J\) and \(I : t \triangleright J\), which actually are not system labels.
broadcasting, etc. We detail three such predicates in Section 4. For example, the
interleaving interaction predicate, whose defining rules are reported in Table 3, is
obtained by interpreting controlled composition as the interleaved parallel composition of the two involved processes. Despite the several interaction predicates

```
\begin{align*}
\text{Table 3. Systems’ labeled transition relation}
\end{align*}
```
that can be defined, we expect anyway that a well-defined interaction predicate satisfies some obvious criteria. For example, a process commitment of the form $\text{get}(T)@c$ should be related to system labels of the form $I : t \triangleright \gamma$, where $t$ is any item ‘matching’ the template $T$ and $\gamma$ results from the evaluation of target $c$, while a process commitment of the form $\text{put}(t)@c$ should be related to system labels of the form $I : t' \triangleright \gamma$, where $t'$ is any item resulting from the evaluation of $t$.

The authorization predicate is used to determine the actions allowed by specific policies, and the (possibly new) policy to be enforced. By resorting to different policies, components can protect themselves against different threats, such as unauthorised access or denial-of-service attacks, hence behaving in a self-protecting way. 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. Likewise the interaction predicate, many different reasonable authorization predicates can be defined (several examples are presented in Section 7).

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

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

Some comments about the rules in Table 3 follow. Rule $(\text{pr-sys})$ transforms process commitments into system labels by exploiting the interaction predicate. In particular, it generates the following system labels: $\tau$, $I : \text{fresh}(n)$, $I : \text{new}(J, K, II, P)$, $I : t \triangleright \gamma$, $I : t \triangleleft \gamma$ and $I : t \triangleright \gamma$. As a consequence of this transformation, a substitution $\sigma$ is generated and applied to the continuation of the process that committed to $\alpha$. This is necessary when $\alpha$ contains a get or a qry, because, due to the way the semantics of processes is defined, the continuation $P'$ may contain free variables even if $P$ is closed. It is worth noting that the domain of $\sigma$ is the set of variables that are bound in $\alpha$, thus, since $\text{fv}(P') \subseteq \text{bv}(\alpha)$, the process $P'\sigma$ is closed. 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.

In SCEL the process of evaluating an interaction/authorization predicate may cause some side effects. These side effects can range, e.g., from the modification of the policy in force at the component to the production of some obligations. These are additional process actions to be performed in conjunction with the enforcement of an authorization decision and correspond to e.g. updating a log file, sending a message, generating an event, setting an attribute. We refer to Section 8.3 for more details on how to deal with obligations.

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 $I : \text{fresh}(n)$ or $I : \text{new}(J, K, II, P)$ (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 (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 3 model the variants of these actions implementing point-to-point communication (the rules for group-oriented communication are shown in Table 4).

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 $I : t \bowtie \mathcal{J}$, generated by rule $(accget)$, denotes the willingness of component $\mathcal{J}$ to provide the item $t$ to component $I$. 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 $II \vdash I : t \bowtie \mathcal{J}, II'$) and if withdrawing an item $t$ from the repository of $\mathcal{J}$ is possible ($K \ominus t = K'$). 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 $I.\pi$).

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, 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 communications. It is modeled by the rules in Table 4. Thus, when the target of action get denotes a set of repositories satisfying a given target 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 performing the action. Relation $\mathcal{J} \models P$ states that (the attributes of) the component $\mathcal{J}$ satisfy the predicate $P$; the definition of such relation depends on which kind of predicates is used. If the action succeeds, this transition corresponds to an internal com-
Table 4. Systems’ labeled transition relation (cnt.): rules for group communication

The capability of a component to perform a put for group-oriented communication is not affected by those system components not satisfying predicate \( P \), 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 that 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 \texttt{put} for group-oriented communication all potential receivers are taken into account.

**Running example (step 5/6).** Let us suppose that $I_2.task = I_3.task = \text{"task1"}$ while $I_4.task = \text{"task2"}$ and that $K_3$ 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

$$I_3[K_3, H_3, P_3] \xrightarrow{	ext{I_3:\("targetLocation",3,5\)}} I_3[K_3, H_3, P_3]$$

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

$$I_1[K_1, H_1, P_1] \xrightarrow{	ext{I_1:\("targetLocation",3,5\)}} I_1[K_1, H_1, \text{put("targetLocation",3,5)@self.P'_1}]$$

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

$$S \xrightarrow{T} I_1[K_1, H'_1, \text{put("targetLocation",3,5)@self.P'_1}]$$
$$\parallel I_2[K_2, H_2, P_2] \parallel I_3[K_3, H_3, P_3] \parallel I_4[K_4, H_4, P_4] \parallel \ldots \square$$

**Remark 2 (On different forms of \texttt{put}).** The two actions \texttt{put(t)@n} and \texttt{put(t)@((id \in \{n\}}) have not the same meaning. Indeed, the former is a point-to-point communication and succeeds only whenever there is a component named \textit{n} willing to receive the item \textit{t}. The latter is a sort of group-oriented communication over a channel without message loss and can also succeed whenever a component named \textit{n} does not exist or exists but does not authorize the action (i.e. is not willing to receive \textit{t}). In the second case above, \texttt{put(t)@n} would get stuck, while \texttt{put(t)@((id \in \{n\})} would terminate successfully (but \textit{t} would not be added to the repository at \textit{n}). Another way of writing the above group-oriented communication action is \texttt{put(t)@p}, where \textit{p} is an attribute, exposed in the interface of the subject component, associated to the predicate $id \in \{n\}$. While the former two \texttt{put} actions will try to interact always with the component named \textit{n}, the latter action \texttt{put(t)@p} may also interact with other components, because the association for the attribute \textit{p} may dynamically change (and refer, e.g., to the predicate $id \in \{n, o\}$).

**Remark 3 (On systems with name restrictions).** It is worth noticing that, although the inference rules in Tables 3 and 4 are defined on top of all the systems produced by the syntax in Table 1, no transition can be derived from a system...
containing name restrictions. That is, in a transition $S \xrightarrow{\lambda} S'$, $S$ may not contain name restrictions (instead, because of rule \(\text{newc}\), $S'$ may do). This account for our statement at the beginning of this section, i.e. that we first define an LTS to derive the transitions enabled from systems without occurrences of name restrictions.

Now, the TS defining the semantics of generic systems is defined on top of the LTS as

- the set of states $S$ includes all the systems defined in Table 1;
- the transition relation $\Rightarrow$ is the least relation induced by the inference rules in Table 5. As a matter of notation, we will write $S \Rightarrow S'$ instead of $(S, S') \in \Rightarrow$. Moreover, $\bar{n}$ 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 $S$.

The rules in Table 5 defining the transition relation are straightforward. Rule \(\text{tau}\) accounts for the computation steps of a system where all (possible) name restrictions are at top level. Rule \(\text{put}\) states that, besides to those labeled by $\tau$, computation steps may additionally be labeled by $I : t \triangleright P$, corresponding to group-oriented communication, and thus transforms them into transitions of the form $\Rightarrow$. Rule \(\text{top}\) 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 \(\text{comm}\) and \(\text{assoc}\) state that systems’ composition is a commutative and associative operator. Notably, by exploiting these two rules we can manipulate systems so that we do not need analogous rules to be added to those defining the labeled transition relation.
Running example (step 6/6). The robotics system can thus evolve as follows

\[ S \rightarrow I_1 ([K_1, \Pi_1', \text{put} ("\text{targetLocation}", 3, 5)@self.\Pi_1']) \]
\[ \parallel I_2 [K_2, \Pi_2, P_2] \parallel I_3 [K_3, \Pi_3, P_3] \parallel I_4 [K_4, \Pi_4, P_4] \parallel \ldots \]
\[ \rightarrow I_1 ([K_1 \oplus \langle "\text{targetLocation}", 3, 5 \rangle, \Pi_1', \Pi_1']) \]
\[ \parallel I_2 [K_2, \Pi_2, P_2] \parallel I_3 [K_3, \Pi_3, P_3] \parallel I_4 [K_4, \Pi_4, P_4] \parallel \ldots \]

Notably, the group-oriented \textbf{qry} action involves the robots belonging to the ensemble in charge of \textit{task}_1 (which includes the components identified by \( I_2 \) and \( I_3 \)), while the subsequent point-to-point \textbf{put} action only involves the first robot (i.e., the component identified by \( I_1 \)). Recall that \( K_1 \oplus \langle "\text{targetLocation}", 3, 5 \rangle \) means that the information about \textit{targetLocation} is added to the knowledge repository \( K_1 \). The fourth robot (i.e., the component identified by \( I_4 \)) is never involved in such communications, because it is in charge of doing \textit{task}_2.

4 Interaction Predicates

The operational semantics introduced in Section 3 relies on the interaction predicate\(^6\)

\[ \Pi, I : \alpha \succ \lambda, \sigma, \Pi' \]

This predicate establishes a relation between a given triple, consisting of a policy \( \Pi \), an interface \( I \) and a process commitment \( \alpha \), and a triple, consisting of a system label \( \lambda \), a substitution \( \sigma \) and a (possibly new) policy \( \Pi' \). Intuitively, \( \lambda \) identifies the effect of \( \alpha \) at the level of components, while \( \sigma \) associates values to the variables occurring in \( \alpha \) and is used to capture the changes induced by communication. \( \Pi' \) is the policy in force after the transition; in principle it may differ from that in force before the transition. An interaction predicate then permits defining sophisticated policies for regulating the interaction among processes of a component, while possibly taking other policies (e.g. for access control) into account. For the sake of simplicity, in this report we will be only concerned with policies \( \Pi \) controlling process interaction and refer the interested reader to [23] for more general situations.

Below, we present three possible instances, that we call \textit{interleaving}, \textit{monitoring} and \textit{limited monitoring}, of the above predicate. In all cases, the interaction predicate is defined by a set of inference rules. The three instances of the interaction predicate are somehow reminiscent of the three variants of parallel composition in process algebras where composed processes either never interact, or are forced to interact on all actions, or interact only on a specific set of actions.

The following notations will be used:

\(^6\) Interaction predicates are reminiscent of \emph{synchronization algebras} introduced by Glynn Winskel in a seminal paper on Event Structures [22] as a device to specify how events from parallel processes do synchronize, thus associating with any synchronization algebra a particular parallel composition.
$Π \oplus I : \text{fresh}(n) \succ \text{fresh}(n), \{\}, Π_⊕$

\[
\begin{align*}
\mathcal{E}[T]_I = T' & \quad \mathcal{N}[c]_I = γ \quad \text{match}(T', t) = σ \\
\Pi_⊕, I : \text{get}(T)@c \succ I : t @ γ, σ, Π_⊕ \\
\mathcal{E}[t]_I = t' & \quad \mathcal{N}[c]_I = γ \\
\Pi_⊕, I : \text{put}(t)@c \succ I : t' @ γ, \{\}, Π_⊕ \\
\Pi_⊕, I : \text{new}(J, K, Π, P)[P][x] & \succ \text{new}(J, K, Π, P)[P][x], \{\}, Π_⊕ \\
\Pi_⊕, I : α @ γ, σ, Π_⊕ & \succ \Pi_⊕, I : α[α] @ γ, σ, Π_⊕ \\
\Pi_⊕, I : α @ γ, σ, Π_⊕ & \succ \Pi_⊕, I : α[α] @ γ, σ, Π_⊕ \\
\Pi_⊕, I : α @ γ, σ, Π_⊕ & \succ \Pi_⊕, I : α[α] @ γ, σ, Π_⊕ \\
\end{align*}
\]

**Table 6.** The interaction predicate interleaving

- $\mathcal{E}[t]_I$ (resp. $\mathcal{E}[T]_I$) denotes the evaluation of item $t$ (resp. template $T$) with respect to interface $I$: attributes occurring in $t$ (resp. $T$) are replaced by the corresponding value in $I$;
- $\mathcal{N}[c]_I$ denotes the evaluation of target $c$ according to interface $I$, thus variables are replaced by the corresponding component names and predicate names are replaced by the corresponding predicates;
- $\mathcal{P}[P]_I$ denotes the evaluation of $P$ according to interface $I$: functionalities in $P$ are replaced by the corresponding code in $I$;
- $\text{match}(T, t)$ 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 $T$, otherwise it is undefined.

**Interleaving.** The interaction predicate *interleaving*, denoted by $Π_⊕$, is obtained by interpreting controlled composition as the *interleaved* parallel composition of the two involved processes. The inference rules defining predicate $Π_⊕ : \Pi_⊕, I : α @ γ, σ, Π_⊕$ are reported in Table 6. We have a rule for each different kind of process action, plus two additional rules (the last ones) ensuring that in case of controlled composition of multiple processes only one process can perform an action (the other stays still). The (five) rules for process actions basically state that, at the level of the operational semantics of systems, all process actions correspond to properly labeled transitions.

**Monitoring.** The interaction predicate *monitoring*, denoted by $Π_⊗[Π_1, Π_2]$, can be used to ensure that in a controlled composition $P[Q]$, $P$ can actually control the actions performed by $Q$. This makes controlled composition a non-commutative operator, differently from the interaction predicate interleaving $Π_⊕$ described before.

The inference rules defining the interaction predicate monitoring $Π_⊗[Π_1, Π_2]$ are reported in Table 7. The rules managing basic labels (i.e. not of the form $α[β]$) are omitted since they are similar to the first five rules of Table 6 (but
for \( \Pi_\otimes[\Pi_1, \Pi_2] \) replacing \( \Pi_\otimes \). Assume that, in a controlled composition \( P[Q] \), process interactions in \( P \) and \( Q \) are regulated by policies \( \Pi_1 \) and \( \Pi_2 \), respectively. Then, \( \Pi_\otimes[\Pi_1, \Pi_2] \) prescribes that \( Q \) can evolve with a transition labeled \( \beta \), which is mapped by \( \Pi_2 \) to a \textit{put} or \textit{get} of item \( t \) at component \( n \) (label \( \mathcal{I} : t \triangleright n \)), only when \( P \) can evolve with a transition labeled \( \alpha \), which is mapped by \( \Pi_1 \) to either a \textit{get} or a \textit{retrieve} of item \( t \) at component \( n \) (labels \( \mathcal{I} : t \triangleleft n \) or \( \mathcal{I} : t \bowtie n \)). In the former case the overall outcome is a \( \tau \) while in the latter case the label \( \mathcal{I} : t \triangleright n \) is propagated to the rest of the system. In all other cases, \( \Pi_\otimes[\Pi_1, \Pi_2] \) works similarly to the interleaving predicate, i.e., all other labels of the form \( \alpha[\beta] \) are mapped to a system label \( \lambda \) only if either \( \alpha = \circ \) or \( \beta \) is mapped to \( \lambda \) with \( \lambda \notin \mathcal{I} : t \triangleright n \) or \( \beta = \circ \) and \( \alpha \) is mapped to \( \lambda \) with \( \lambda \notin \{ \mathcal{I} : t \triangleleft n, \mathcal{I} : t \bowtie n \} \).

Predicate \( \Pi_\otimes[\Pi_1, \Pi_2] \) is actually a predicate 'schema' as it is parametric with respect to predicates \( \Pi_1 \) and \( \Pi_2 \) defining the interaction policies of processes \( P \) and \( Q \). The monitoring predicate could be combined with the interleaving predicate to obtain more refined interaction patterns for processes.

\textit{Limited monitoring.} The interaction predicate limited monitoring, denoted by \( \Pi_N[\Pi_1, \Pi_2] \), where \( N \) is a set of components names, constrains the behavior of processes of the form \( P[Q] \) in such way that:

- \( P \) and \( Q \) interact according to \( \Pi_\otimes[\Pi_1, \Pi_2] \) for any system label whose target is in \( N \);
- \( P \) and \( Q \) can freely execute actions whose target is not in \( N \).

The inference rules defining the predicate \( \Pi_N[\Pi_1, \Pi_2] \), are reported in Table 8. The rules extend those of \( \Pi_\otimes[\Pi_1, \Pi_2] \) by exploiting additional side conditions. The first two rules guarantee that synchronization only occur on labels involving names in \( N \), while the last two rules model the case where processes \( Q \) and \( P \), in a process of the form \( P[Q] \), can evolve independently. Again, the
basic labels are dealt with similarly to the case of the interaction predicate interleaving, that is by the first five rules of Table 8.

Remark 4 (On the expressive power of interaction predicates). The interaction predicates monitoring and limited monitoring provide just a few examples of the expressive power of interaction predicates. It is not difficult to envisage more general situations where, e.g., actions performed by \( Q \) in a controlled composition of the form \( P[Q] \) are intercepted by suitable actions of \( P \) and appropriately transformed into labels at the level of systems. This allows \( P \) to act as a sort of ‘execution monitoring’ for \( Q \) and is somehow reminiscent of the approach for enforcing security policies that relies on the so called security automata [24].

Remark 5 (On static vs. dynamically changing policies). All the interaction policies defined by the interaction predicates we have considered are static (if also the parameters \( \Pi_1 \) and \( \Pi_2 \) of the monitoring and limited monitoring predicates are static), as they do never change while systems progress. More sophisticated interaction policies could be defined by changing the predicate after any its evaluation, thus exploiting the full power of the judgement \( \Pi, I : \alpha > \lambda, \sigma, \Pi' \) where \( \Pi' \), the predicate to be applied next time, can differ from \( \Pi \).

5 Self-adaptation in SCEL

In this section we argue that adaptation can be naturally expressed in SCEL. As we have seen in Section 2, the knowledge repository of components can contain both application data and awareness data. At this level of abstraction, we are not concerned with the way data are actually represented, we only assume that they can be appropriately tagged to distinguish awareness data from application data. This distinction is indeed at the basis of a tangible notion of adaptation [25], which is defined as the run-time modification of the so-called awareness data (which correspond to control data of [25]). A component is then deemed adaptive if it has a precisely identified collection of awareness data that are modified at
run-time, at least in some of its computations. Besides, it is self-adaptive if it is able to modify its own awareness data at run-time.

In general, a component in SCEL is adaptive (and, hence, autonomic) because its awareness data can be dynamically modified by means of the actions put/get/qry. Moreover, a component is self-adaptive as the hosted process can trigger modifications of its awareness data by interacting with the local knowledge handler. So-called feedback-loops, that adapt the behavior of autonomic components to changing contexts, can thus be easily implemented.

The one outlined above is perhaps the simplest form of adaptation, but we can envisage more sophisticated forms by taking the nature of the awareness data into account. Suppose, for example, that the process part of a component is split into an autonomic manager controlling execution of a managed element. The autonomic manager monitors the state of the component, as well as the execution context, and identifies relevant changes that may affect the achievement of its goals or the fulfillment of its requirements. It also plans adaptations in order to meet the new functional or non-functional requirements, executes them, and monitors that its goals are achieved, possibly without any interruption. In practice, the autonomic manager implements the rules for adaptation. Now, by exploiting SCEL 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, it is e.g. possible to dynamically replace (part of) the managed element process or even the autonomic manager process. In this case, we are also changing the rules, i.e. processes, with which the awareness data are manipulated, since these rules are represented as awareness data themselves.

A managed element can be seen as an empty “executor” which retrieves from the knowledge repository the process implementing a required functionality id and bounds it to a variable X, sends the retrieved process for execution and waits until it terminates (this coordination can be worked out by exchanging appropriate synchronization items). Also actual parameters for the process to be executed can be stored as knowledge items and retrieved by the executor (or by the process itself) when needed, as shown by the code fragment below

\[
ME \triangleq \text{qry}("\text{required\_functionality\_id}\), X)@self.
\]

\[
\text{get}("\text{required\_functionality\_id\_args}\), y, z)@self.
\]

\[
\text{fresh}(n), \text{put}("\text{required\_functionality\_id\_params}\), n, y, z)@self.
\]

\[
\text{(get("\text{required\_functionality\_id}\), \text{terminated}\), n)@self.ME}[X]
\]

where the fresh name n is used for coordination purposes. We assume that each process implementing a required functionality starts by reading the parameters of the call (i.e. the call identifier n and the functionality arguments), by means of an action get("\text{required\_functionality\_id\_params}\), id, y, z)@self, and terminates by performing an action put("\text{required\_functionality\_id}\), \text{terminated}\), n)@self.

\footnote{The whole body of activities mentioned above has been named MAPE-K loop (Monitoring, Analyzing, Planning, and Executing, through the use of Knowledge) \cite{4}.}
Items containing processes or parameters can be thought of as awareness data. Autonomic managers can add/remove/replace these data from the knowledge repositories thus implementing the adaptation logic and therefore changing the managed element’s behavior. For example, different versions of the process providing a requested service may exist. While managed elements could only read these data, the autonomic manager could dynamically change the association between the service request and the service process by simply performing:

```plaintext
get("required_functionality.id", ?X)@self.
p(put("required_functionality.id", Q)@self
```

which has the effect of replacing the ‘old’ service implementing the functionality id with a possibly new one Q. The two actions above have target self since we are assuming that the autonomic manager process is co-located with the managed element process ME. Of course, if this would not be the case, the target of the actions would be the name of the component hosting ME.

The autonomic manager can also add a new service or even remove an existing one. Besides, it is a process just like the managed element, thus it is very well suited to be itself subject to adaptation. In this way, we can build up hierarchical adaptations and cover a wide range of adaptation mechanisms.

One issue with SCEL is that it does not have any specific mechanism for stopping or killing processes. However, exploiting knowledge and higher-order features, the application designer can specify when to terminate processes by following suitable patterns. For example, in the code fragment below, the managed element can ask the autonomic manager for the authorization to proceed as process P (indicated by the item (pid, “ok”) in the local knowledge repository, where pid is the process identifier) and, in the negative case, signal its termination (by producing the item (pid, “dead”)):

```plaintext
qry(pid, “ko”)@self.put(pid, “dead”)@self
+ qry(pid, “ok”)@self.P
```

The following code fragment would allow an autonomic manager to send a termination request to the process with identifier pid and wait for its termination, assuming that both items (pid, “ok”) and (pid, “ko”) are used for coordination purposes:

```plaintext
get(pid, “ok”)@self.put(pid, “ko”)@self get(pid, “dead”)@self
```

As we have seen, it is the autonomic manager that chooses which adaptation to use. The decision about when to perform adaptation is jointly taken by the autonomic manager and the application designer. This is reminiscent of another approach, named context-oriented programming (COP) [26]. COP is a novel programming paradigm introduced to manage and control adaptivity of programs. It allows developers to define behavioral variations, that is chunks of code that can be activated depending on the current working environment (the context), to dynamically modify program execution and thus adapt to its environment.
In this approach, the application designer has to insert adaptation hooks in the application code and is thus able to control when adaptation can take place. Leaving the designer to specify where and when to adapt has its advantages, because adaptation would be explicit in the code and thus more visible, and the application designer could better plan some adaptations. However, not being transparent to the application designer has significant disadvantages, because only adaptations planned at design phase could be exploited. When the autonomic computing approach is used, the autonomic manager, which continuously monitors awareness data or event occurrences, reacts to changes of contexts or of goals.

Other than language-level adaptation, as e.g. used in COP, another approach to adaptation focuses on the architectural-level. It consists in dynamically reshaping the structure of the system, e.g. by exchanging a specific component with one that provides similar functionalities, but behaves better in a new context. SCEL supports also this coarse-grained approach since component’s membership to ensembles is dynamic and can be parametric w.r.t. to some information controlled by an autonomic manager.

Finally, in case of distributed applications one can plan to have (i) awareness data residing at autonomic elements and the autonomic managers performing the adaptation for all controlled elements, or (ii) all autonomic elements reading from a single knowledge repository that contains both awareness data and global autonomic processes. The distributed approach may cause consistency problems between autonomic elements during the adaptation procedure, because the autonomic managers of different elements may not be synchronized. The centralized approach may lead to efficiency loss and relies too much on the communication between autonomic elements, that can have considerable latencies or be unreliable. However, both approaches may be useful. For example, at ensemble level, adaptation can be partly centralized, controlled by an autonomic manager, and partly distributed in each component. At system level, the distributed approach better supports the dynamic structures and loosely-coupled components.

6 SCEL at work

In this section, we first introduce a dialect of SCEL based on tuple-spaces and then we show how this language can be used to model two application scenarios, the former from the service provision domain and the latter from the robotics one.

6.1 A SCEL dialect based on tuple-spaces

We show here how dialects of SCEL can be easily defined by appropriately specifying the parameters of the language. As a concrete example, we define a SCEL dialect, named SCEL_TS, where knowledge repositories are implemented as multiple distributed tuple-spaces à la KLAIM [27]. Another SCEL dialect, specifically devised for concurrent constraint programming, is described in [28].
In order to define a dialect with specific features, one has to fix the parameters SCEL depends on, that is

1. the languages for representing knowledge items and the templates to be used to retrieve these items from the repositories;
2. the language for representing knowledge repositories, together with the three operations, i.e. withdrawal, retrieval and addition, that we assume provided by each knowledge repository’s handling mechanism;
3. the language for expressing policies, together with an interaction predicate and an authorisation predicate.

In SCEL_TS, knowledge items are sequences of values, i.e. tuples, while templates are sequences of values and variables. More generally, a value can result from the evaluation of some given expression $e$ belonging to an appropriate language of EXPRESSIONS. We assume that expressions contain boolean, integer, float and string values and variables, together with the corresponding standard operators.

Knowledge repositories are multisets of tuples, i.e. tuple spaces, providing the three operations of withdrawal, retrieval and addition. The first two operations use pattern-matching w.r.t. a given template to pick a tuple from a tuple space: a tuple matches a template if they have the same number of elements and corresponding elements have matching values or variables; variables match any value of the same type ($?x$ and $?X$ are used to bind variables to values and processes, respectively), and two values match only if they are identical. In case more tuples match a given template, one of them is arbitrarily chosen.

In practice, we can complete the syntax of knowledge, items and templates as shown in Table 9. A knowledge repository $\mathcal{K}$ is thus a multiset of stored tuples $\langle t \rangle$ and empty tuples $\emptyset$, composed by operator $\parallel$.

Now, the three operations provided by the knowledge repository’s handling mechanism and considered in Section 3.2, 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 10. Notably, when a matching tuple is removed from $\mathcal{K}$, it is replaced by $\emptyset$.

Finally, we have to instantiate policies for SCEL_TS. In the previous sections, we have seen that policies can regulate the interaction between the different internal parts of a component and the interaction of a component with the others. In SCEL_TS, the interaction policy is $\Pi_{\ominus}$ and, hence, the interaction
\( \langle t \rangle \oplus t = \emptyset \)

\( (K_1 \parallel K_2) \oplus t = K' \parallel K_2 \)

\( (K_1 \parallel K_2) \oplus t = K_1 \parallel K' \)

\( \langle t \rangle \vdash t \)

\( (K_1 \parallel K_2) \vdash t \)

\( (K_1 \parallel K_2) \vdash t \)

\( K \oplus t = K \parallel \langle t \rangle \)

Table 10. Tuple-space operations (\( \oplus, \vdash, \oplus \))

Predicate is the *interleaving* one, formally defined in Table 6. Instead, the authorization predicate in SCEL_TS is always satisfied, that is we assume that the interaction among components is always authorized.

### 6.2 A service provision scenario in SCEL_TS

In this application scenario, we consider a collection of *service components*, all offering the same services. These services are published in the component interfaces through suitable attributes defining the signature of the services.

Each component manages and elaborates service requests with different requirements, roughly summarized by the following three quality levels: *gold*, *silver* and *base*. These requirements are defined via a combination of predicates on the hardware configuration and the runtime state. For example, the runtime state can give a measure of the number of service requests currently handled locally. The parameters of the different requirements are identified by suitable attributes of the component interfaces. In particular, we assume that the attributes ("\( \text{hw} \), i") and ("\( \text{load} \), j") are provided by the interface of each component. For example, value i (an integer in \([0, 10]\)) in ("\( \text{hw} \), i") gives an indication of the capacity of the hardware configuration of the component; while value j (an integer in \([0, 100]\)) in ("\( \text{load} \), j") estimates the actual computational load of the component. Notice that the hardware measure is static while the load estimate is dynamically updated whenever a component receives or completes a service request.

We recall that, in SCEL, attribute names are thought of as pointers to the actual values contained in the knowledge repository associated to the components. Hence, in SCEL_TS, where data tuples are used as knowledge items, this amounts to saying that the two attributes above corresponds to the following tuples

\( \langle \text{"hw"}, i \rangle \)

\( \langle \text{"load"}, j \rangle \)

that are stored in the local tuple-space of each component. Thus, the values of such attributes can be dynamically changed through actions *get* and *put*.

Each service component also publishes in its interface the signature of the available services through suitable attributes. Here we assume that *aService* is the name of the available service and requires a *string* value as input parameter and yields a *string* value as a result. Besides, additional information about the client and the session has to be provided when the service is invoked.
Service components constitute three ensembles depending on the quality of service they can provide. Ensembles are common pools where service components remain alive to be dynamically allocated to handle new requests. Since ensemble aggregation is attribute-based, the following predicates are used to identify the ensembles of gold, silver and base service providers:

- \( P_g = hw \geq 7 \)
- \( P_s = (hw \geq 4) \land (P_g \rightarrow load < 40) \)
- \( P_b = (P_s \rightarrow load < 40) \land (P_g \rightarrow load < 20) \)

Thus, the gold ensemble identifies a gold component by the high measure of its hardware configuration (value greater or equal to 7). The silver ensemble is less demanding: a component has to provide a hardware configuration that is valued more than 7, the computational load must be less than 40% (operator \( \rightarrow \) stands for logical implication). This last condition guarantees that gold components can handle requests at silver level only when their computational load is under 40%. The same schema is used to define the base ensemble. Of course, all the components, independently of their hardware level, can be part of this ensemble. However, gold and silver components are involved only when their computational load is under 20% and 40%, respectively.

Notice that components dynamically and transparently leave or enter an ensemble when their computational load changes. For instance, a gold component (i.e., a component with attribute \( hd \) that is greater or equal to 7) leaves a silver ensemble whenever its computational load becomes higher than 40%.

The process running at the client component taking care of the interaction with the service, let us call it \( P_c \), performs the following code fragment:

\[
\begin{align*}
\text{fresh}(n). \\
\text{qry}(\text{"service"}, \text{"aService"}, ?u)@P. \\
\text{put}(\text{"invoke"}, \text{"aService"}, v, \text{self}, n)@u. \\
\text{get}(\text{"result"}, \text{"aService"}, ?x, n)@\text{self}.P'_{c}.
\end{align*}
\]

This process first searches among the components belonging to the ensemble identified by predicate \( P \) (where \( P \) is one among \( P_b, P_s \) or \( P_g \)) an item matching the template ("service", "aService", \( ?u \)). Indeed, we assume that the knowledge located at a component \( c \) contains the tuple ("service", "aService", \( c \)) whenever \( c \) is able to handle requests for service \( aService \). Notice that, by taking advantage of group-oriented communication, a client is able to dynamically identify a component that exposes the service \( aService \) at the wanted service level (thanks to the predicate used to identify the target of the \text{qry} operation). Then, the client posts the actual parameters to the selected component. Value \( v \) is the required input string, while the pair \( \text{self}, n \) provides the bookkeeping information: \text{self} evaluates to the client name and the fresh name \( n \) is used to characterize the working session. After issuing the invocation, the client waits for the result (recall that action \text{get} is blocking in SCELTS). Whenever the result of the service invocation is made available, the client can withdraw it from the local repository and continue as process \( P'_{c} \).
The processes $P_s$ running at service components is defined as follows:

\[
P_s \triangleq \text{//Retrieve a request from the local repository} \\
\text{get("invoke", "aService", \text{?param, ?client, ?session})@self.} \\
\text{//Update the computational load} \\
\text{get("load", \text{?x})@self.} \\
\text{put("load", (x + 5))@self.} \\
\text{//The request is processed and the process restart} \\
\text{P_s\[Q(param, client, session]\]}
\]

The process is triggered by a client request. Whenever this happens, the computational load is updated, service $aService$ becomes again ready to serve other client requests, and the process $Q$, which actually computes the result of the invoked service “$aService$” for the current request, is executed. We assume that, before its termination, process $Q$ updates the value of attribute $load$ and puts the result of the computation into the repository of the requesting client.

Notice that the application scenario discussed above exploits different forms of communication. First, the invoking client uses group-oriented communication to identify the component that is able to handle specific service request. Then, point-to-point communication is used for client-server interaction.

### 6.3 A swarm robotics scenario in SCEL$_{TS}$

We now use SCEL$_{TS}$ to model a swarm robotics scenario where robots are distributed over a physical area and have to reach different zones according to the tasks that they have to do, such as rescue people in danger, help other robots, reach a safe area, clear a minefield, etc. In particular, in the considered scenario, each robot of the swarm has to fulfill one of two different tasks, i.e. either $task_1$ or $task_2$. 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. For this reason, to discover the location of the target, each robot follows a random walk. As soon as a robot reaches the area, 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-optimize their behaviour).

The autonomic behaviour of each robot in the swarm is implemented by means of an autonomic manager controlling the execution of a managed element (following the self-adaptation approach discussed in Section 5). The autonomic manager monitors, in a self-aware fashion, the state of charge of the robot’s resources.

---

8 Here we assume each service instance uses 5% of the component computational resources.
battery and verifies if 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.

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

$$ME \triangleq \text{qry}("controlStep",?X)@self.
\text{(get("terminated")}@self.ME)[X]$$

This process retrieves from the local knowledge repository the process implementing the current control step and bounds it to a 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. The autonomic manager $AM$ can then replace the control step code from the knowledge repository thus implementing the adaptation logic and therefore changing 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 $\text{task}_i$. 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 ("batteryLevel", $l$) 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 [25]) 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 $P_{lowBattery}$.
The processes composing the autonomic manager $AM$ are as follows:

$$P_{\text{batteryMonitor}} \triangleq \text{qry}(\text{"batteryLevel"}, \text{"low"})@self. $$

get(\text{"lowBattery"}, \text{false})@self.

put(\text{"lowBattery"}, \text{true})@self.

$$P_{\text{dataSeeker}} \triangleq \text{qry}(\text{"targetLocation"}, x, y)@\text{task} = \text{"task"}. $$

get(\text{"informed"}, \text{false})@self.

put(\text{"informed"}, \text{true})@self.

$$P_{\text{targetSeeker}} \triangleq \text{qry}(\text{"lowBattery"}, \text{?low})@self. $$

if (\text{low}) then {

get(\text{"controlStep"}, \text{?X})@self.

put(\text{"controlStep"}, \text{P\text{lowBattery}})@self.

$$P_{\text{dataSeeker}} \triangleq \text{qry}(\text{"targetLocation"}, x, y)@\text{task} = \text{"task"}. $$

get(\text{"informed"}, \text{false})@self.

put(\text{"informed"}, \text{true})@self.

P_{\text{targetSeeker}}$$

The process $P_{\text{batteryMonitor}}$ becomes active when the state of charge of the robot’s battery reaches the level low. To deal with this issue, the process sets the lowBattery tuple to true. Then, when the battery is recharged, the process brings the value of lowBattery back to false and restarts. The process $P_{\text{dataSeeker}}$ corresponds to the process $P_1$ described in the running example shown in Section 2. 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. Finally, the process $P_{\text{targetSeeker}}$ sets the content of the “controlStep” tuple according to the values of tuples lowBattery, informed and target (i.e. the awareness data). Notably, until the robot is not informed and the battery level is high, the default control step is P_randomWalk. When the
target is found, the process \( P_{\text{doTask}} \) is installed. This is an autonomic manager process that deals with the new requirements concerning the execution of the robot’s task; we do not specify this process here since it is out of scope in our scenario.

For the sake of readability, in the definition of process \( P_{\text{targetSeeker}} \), we have exploited an \textbf{if}−\textbf{then}−\textbf{else} construct, which however can be easily rendered in SCEL. For example, the term

\[
\text{qry}(“lowBattery”, \?low)@\text{self}. \text{ if } (\text{low}) \text{ then } \{ P_{\text{then}} \} \text{ else } \{ P_{\text{else}} \}
\]

can be rewritten as follows:

\[
\text{qry}(“lowBattery”, \text{true})@\text{self}. P_{\text{then}} + \text{ qry}(“lowBattery”, \text{false})@\text{self}. P_{\text{else}}
\]

The processes executed by the managed element \( ME \) at each control step are as follows:

\[
P_{\text{lowBattery}} \triangleq \text{put} (“\text{stop}”)@\text{self}. \text{ qry}(“gps”, \?x, \?y)@\text{self}. \\
\quad \text{ put} (“\text{sos}, x, y”)@(\text{task} = “\text{taskk}”). \\
\quad \text{ got (“rescued”)@\text{self}. put (“terminated”)@\text{self}}
\]

\[
P_{\text{found}} \triangleq \text{put} (“\text{stop}”)@\text{self}. \text{ qry}(“gps”, \?x, \?y)@\text{self}. \\
\quad \text{ put} (“\text{targetLocation}, x, y”)@\text{self}. \ldots \text{ execute task } i \ldots
\]

\[
P_{\text{informed}} \triangleq \text{qry}(“\text{targetLocation}, \?x, \?y)@\text{self}. \\
\quad \text{ put (“direction”, \text{towards}(x, y))@\text{self}. put (“terminated”)@\text{self}}
\]

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

When a low battery’s level is detected (process \( P_{\text{lowBattery}} \)) or the target area is found (process \( P_{\text{found}} \)), the wheel actuator is notified to stop the robot’s movement and the GPS position is retrieved. Then, in the former case a distress signal is sent to all robots performing the same task, while in the latter case the execution of the robot’s task starts. Of course, different solutions to deal with the low battery’s level issue can be devised, perhaps by taking into account working environment conditions, such as the presence of near charging stations. Notably, when the battery is recharged, the process \( P_{\text{targetSeeker}} \) reinstalled in the managed element either the process \( P_{\text{informed}} \) or the process \( P_{\text{randomWalk}} \), according to whether the robot is informed or not. Finally, process \( P_{\text{informed}} \) specifies a direction towards a given location, while process \( P_{\text{randomWalk}} \) randomly selects a direction followed by the robot to search the target area. Given a computed direction \( \theta \), these operations are performed by adding the tuple (“\text{direction}, \theta”) in the local repository, which will be retrieved by the actuator governing the robot wheels.

7 SACPL: a SCEL access control policy language

In this section, we present SACPL (SCEL Access Control Policy Language), a simple, yet expressive, language for defining access control policies and access
requests, and its integration with SCEL. SACPL is inspired to, but much simpler and less expressive than, the OASIS standard for policy-based access control XACML [29].

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 model [16], 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.

It is worth noticing that, other than as a policy language for regulating the access to component resources, SACPL can also be used for controlling adaptation, as the specification of the High Load Scenario presented in the next section shows.

7.1 SACPL syntax

SACPL syntax is presented in Table 7.1. A policy is

- either an atomic policy,
- or a pair of simpler policies combined through one of the decision-combining operators permit override p-o and deny override d-o.

An atomic policy is a pair made of a decision and a target. The target defines the set of access requests to which the policy applies. The decision — permit or deny — is the effect returned when the policy is ‘applicable’, i.e. the access 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). Notably, policies are hierarchically structured as trees: the leaf nodes return a ‘conclusive’ decision permit or deny, while the intermediate nodes combine the decisions returned by the evaluation of the policies of their child nodes through the corresponding decision combining operator.
\[ \Pi ::= \langle \text{Decision}; \text{target: \{ Targets \}} \rangle \] (Policies)
\[ | \Pi \text{ p-o } \Pi | \Pi \text{ d-o } \Pi \] (atomic policy)
\[ \Pi p-o \Pi \text{ | } \Pi d-o \Pi \] (policy combination)

Decision ::= permit | deny (Decisions)

Targets ::= MatchF(Designator, Expr) (atomic target)
\[ | \text{Targets or Targets | Targets and Targets} \] (target combination)

MatchF ::= equal | pattern-match (Matching functions)
\[ | \text{greater-than | ...} \]

Designator ::= action | item | subject.attr | object.attr (Designators)

Expr ::= get | qry | put | fresh | new | value | T | not Expr | Expr or Expr | Expr and Expr
\[ | \text{Expr + Expr | Expr x Expr | ...} \]
\[ | \text{Expr < Expr | Expr = Expr | ...} \]

Table 11. SACPL Policy Syntax

A target is either an atomic target or a pair of simpler targets combined using the standard logic operators and and or. An atomic target is a triple denoting the application of a matching function 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.), item permits referring to the item exchanged in the considered interaction via function pattern-match and template T, while subject.attr and object.attr refer to the specific attribute attr provided, respectively, by the request’s subject or object (like, e.g., subject.id, subject.skill, object.trust_level).

Finally, Expressions are built from values and attributes through various operators.

An example from the robotics scenario. In the robotics scenario illustrated in Section 6.3, 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:
\( \langle \text{permit}; \text{target:} \{ \} \rangle \) // permit all //
\( \langle \text{deny}; \text{target:} \{ \text{not-equal} (\text{subject.id}, n) \text{ } \text{and} \text{ } \text{equal} (\text{object.id}, n) \text{ } \text{and} \text{ } (\text{equal} (\text{action}, \text{qry}) \text{ } \text{or} \text{ } \text{equal} (\text{action}, \text{get})) \text{ } \text{and} \text{ } \text{not-in} (\text{subject.id}, \text{object.ListOfTrusted}) \} \rangle \)

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

**Remark 6 (On expressible policies).** In many access control policy languages (as, e.g., XACML) a combined policy has also a target that identifies the set of requests to which the policy applies. However, policies of the form \( \langle \Pi; \text{target:} \{ \text{Targets} \} \rangle \) can be easily expressed in terms of SACPL policies. Indeed, by induction on the syntax of \( \Pi \), we have that:

\[
\langle \langle \text{Decision}; \text{target:} \{ \text{Targets}_1 \} \rangle ; \text{target:} \{ \text{Targets}_2 \} \rangle \\
\text{can be rewritten as} \\
\langle \text{Decision}; \text{target:} \{ \text{Targets}_1 \text{ and } \text{Targets}_2 \} \rangle
\]

\[
\langle \Pi_1 \text{ p-o } \Pi_2 ; \text{target:} \{ \text{Targets} \} \rangle \\
\text{can be rewritten as} \\
\langle \Pi_1 ; \text{target:} \{ \text{Targets} \} \rangle \text{ p-o } \langle \Pi_2 ; \text{target:} \{ \text{Targets} \} \rangle
\]

\[
\langle \Pi_1 \text{ d-o } \Pi_2 ; \text{target:} \{ \text{Targets} \} \rangle \\
\text{can be rewritten as} \\
\langle \Pi_1 ; \text{target:} \{ \text{Targets} \} \rangle \text{ d-o } \langle \Pi_2 ; \text{target:} \{ \text{Targets} \} \rangle
\]

SACPL requests, ranged over by \( \rho \), are functions mapping names to elements and are written as collections of pairs of the form \((\text{name}, \text{element})\). A request’s element can be a knowledge item, a component’s interface, the type of an action, etc. In its turn, an interface provides a set of attributes characterizing the corresponding component, which can be either the subject or the object of the request. A typical example of request is as follows:

\[
\rho = \{ (\text{subject}, \mathcal{I}), (\text{item}, t), (\text{action}, \text{get}), (\text{object}, \mathcal{J}) \}
\]

Here, the subject identified by the interface \( \mathcal{I} \) requires the authorization to withdraw the item \( t \) from component \( \mathcal{J} \). For example, the request’s subject is obtained by calling \( \rho(\text{subject}) \), which returns \( \mathcal{I} \).

**7.2 SACPL Semantics**

The semantics of SACPL is given in terms of a judgement \( II, \rho \vdash d \) inferred through the rules in Table 12 (\( \rho \) stands for permit or deny, \( \neg \text{permit} \) stands for
\[
\begin{array}{ll}
\langle \varepsilon ; \text{target: } \{ \} \rangle, \rho \vdash - & \langle \varepsilon ; \text{target: } \{ \} \rangle, \rho \vdash - \\
\text{Targets, } \rho \vdash \text{match} & \text{Targets, } \rho \vdash \text{no-match} \\
\langle \varepsilon ; \text{target: } \{ \text{Targets} \} \rangle, \rho \vdash - & \langle \varepsilon ; \text{target: } \{ \text{Targets} \} \rangle, \rho \vdash - \\
\text{Targets, } \rho \vdash \text{no-match} & \text{Targets, } \rho \vdash \text{no-match} \\
\Pi_1, \rho \vdash \text{permit } \lor \Pi_2, \rho \vdash \text{permit} & \Pi_1, \rho \vdash \text{not-applicable } \Pi_2, \rho \vdash \text{not-applicable} \\
\Pi_1, \rho \vdash \text{permit } \land \Pi_2, \rho \vdash \text{permit} & \Pi_1, \rho \vdash \text{not-applicable } \Pi_2, \rho \vdash \text{not-applicable} \\
\Pi_1, \rho \vdash \text{not-permit } \Pi_2, \rho \vdash \text{deny} & \Pi_1, \rho \vdash \text{not-permit } \Pi_2, \rho \vdash \text{deny} \\
\Pi_1, \rho \vdash \text{deny } \land \Pi_2, \rho \vdash \text{deny} & \Pi_1, \rho \vdash \text{not-applicable } \Pi_2, \rho \vdash \text{not-applicable} \\
\Pi_1, \rho \vdash \text{permit } \lor \Pi_2, \rho \vdash \text{deny} & \Pi_1, \rho \vdash \text{not-permit } \Pi_2, \rho \vdash \text{permit} \\
\Pi_1, \rho \vdash \text{deny } \lor \Pi_2, \rho \vdash \text{permit} & \Pi_1, \rho \vdash \text{not-applicable } \Pi_2, \rho \vdash \text{permit} \\
\Pi_1, \rho \vdash \text{permit } \land \Pi_2, \rho \vdash \text{deny} & \Pi_1, \rho \vdash \text{not-applicable } \Pi_2, \rho \vdash \text{deny} \\
\Pi_1, \rho \vdash \text{deny } \lor \Pi_2, \rho \vdash \text{deny} & \Pi_1, \rho \vdash \text{not-applicable } \Pi_2, \rho \vdash \text{not-applicable} \\
\end{array}
\]

**Table 12. SACPL Semantics (where Act is any of get, qry, put, fresh, and new)**

deny or not-applicable, and ~deny stands for permit or not-applicable. It means that the authorization decision \( d \) is returned by policy \( \Pi \) in response to request \( \rho \). The decision \( d \) can be either of permit, deny and not-applicable. Thus, access to the resource requested in \( \rho \) is granted by \( \Pi \) if \( d = \text{permit} \) and it is disallowed if \( d = \text{deny} \); instead, if \( d = \text{not-applicable} \), the policy \( \Pi \) does not apply to the request \( \rho \).
The meaning of the rules is straightforward. The composed policy \((\Pi_1 \text{ p-o } \Pi_2)\) returns \textit{permit} if at least one of the component policies returns \textit{permit} and \textit{not-applicable} if this is the decision returned by both component policies; otherwise, it returns \textit{deny}. The policy \((\Pi_1 \text{ d-o } \Pi_2)\) has a similar interpretation. As for atomic policies, if the target is empty, the request matches the policy, otherwise the matching between request and target is checked. If the request matches the policy’s target, then the policy’s decision (i.e. \textit{permit} or \textit{deny}) is returned; otherwise, it is returned the value \textit{not-applicable}. The matching of a request with a policy’ target is checked by means of the judgement \(\Pi, \rho \vdash r\), where \(r\) can be either \textsc{match} or \textsc{no-match}. To match the composed target of the form \((\text{Targets}_1 \text{ or } \text{Targets}_2)\), a request is only required to match one of \text{Targets}_1 and \text{Targets}_2, while it must match both \text{Targets}_1 and \text{Targets}_2, in order to match the target \((\text{Targets}_1 \text{ and } \text{Targets}_2)\). A request matches an atomic target of the form \text{equal}(\text{action, } \text{Act}) if the request’s action corresponds to the action \text{Act} identified by the target. An atomic target of the form \text{pattern-match}(\text{item, } T) is matched by all requests whose item matches the template \(T\); this is checked by means of the pattern-matching function \text{match} (see Sections 4 and 6.1). Before checking the match, the template is evaluated, since it may contain expressions (this is not the case for the item, which indeed has been retrieved from the request): anyway, this depends on the considered notion of items and templates. Finally, when an atomic target contains a subject’s (resp. object’s) attribute as designator, the evaluation 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. This evaluation relies on a few auxiliary functions:

- The evaluation function \(E[Expr]_\rho\) that evaluates the expression \(Expr\) after replacing the attributes occurring in \(Expr\) by the corresponding values in the subject/object interfaces in \(\rho\). For example, given the request \(\rho = \{(\text{subject, } I), (\text{object, } J), \ldots\}\) with \(I\text{.attr} = 3\) and \(J\text{.attr} = 4\), we have \(E[\text{subject.attr + object.attr}]_\rho = \rho(\text{subject}).\text{attr} + \rho(\text{object}).\text{attr} = I\text{.attr} + J\text{.attr} = 3 + 4 = 7\).
- The definition of the matching functions specified in the target, like e.g.

\[
\begin{align*}
\text{equal}(n, m) &= \text{true} & \text{equal}(n, m) &= \text{false} & \text{greater-than}(\text{int}_1, \text{int}_2) &= \text{false} & \ldots
\end{align*}
\]

Thus, for example, the atomic target \text{equal}(\text{subject.status,} \text{“on”}) matches all authorization requests issued by a component whose \text{status} attribute is set to \text{on}.

### 7.3 Integration with SCEL

We want now to demonstrate how SACPL policies and requests, as well as the related evaluation mechanism, integrate with SCEL. SCEL is indeed parametric with respect to the language used to specify the policies regulating the behavior.
of system components, as it is shown by the definition of its operational semantics. Orthogonal aspects of components’ behavior can be regulated by means of different kinds of policies, which should be enforced together but evaluated separately. Hence, the policy \( \Pi \) specified within a component \( I[K, \Pi, P] \) can be better thought of as a tuple of policies. To illustrate, in the sequel we explicitly consider the case when \( \Pi \) is of the form \((\Pi_i, \Pi_{ac})\), where:

- \( \Pi_i \) is an interaction policy regulating the interaction among processes inside a component (three such policies are shown in Section 4);
- \( \Pi_{ac} \) is an access control policy regulating the access to the knowledge and resources of a component and is defined as a SACPL policy.

Of course, if other kinds of policies are defined to deal with further aspects of components’ behavior, the policy tuple is extended accordingly.

The specific composition of the policy tuple is explicitly taken into account in the definition of the interaction predicate and of the authorization predicate, while it is ignored in the definition of SCEL’s operational semantics (that, anyway, is indirectly affected by it). Indeed, 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. Let us now see in details what happens when the policy tuple \( \Pi \) is \((\Pi_i, \Pi_{ac})\).

In SCEL’s operational semantics, the authorization to perform an action is checked when a computation step can potentially take place, i.e. when it becomes known which are the components target of the action, by resorting to evaluation of the authorization predicate \( \Pi \vdash \lambda, \Pi' \), where labels \( \lambda \) passed as a parameter to the predicate have either the form \( I : \text{fresh}(n) \), \( I : \text{new}(J, K, \Pi, P) \), \( I : t \circ J \), \( I : t \cdot J \), or \( I : t \triangleright J \). ‘Authorization’ actions indicating the willingness of a component to accept a remote interaction are checked as well by resorting to evaluation of the authorization predicate.

Now, as for the specific policy tuple \((\Pi_i, \Pi_{ac})\), we can define the interaction predicate over the policy tuple \((\Pi_i, \Pi_{ac})\) simply as the interaction predicate over the interaction policy \( \Pi_i \) by means of the following rule:

\[
(P_i, \Pi_{ac}) \vdash \lambda, \sigma, \Pi'_i
\]

We refer the interested reader to Section 4 for the exact definition of predicate \( P_i, \Pi : \alpha \succ \lambda, \sigma, \Pi'_i \).

Similarly, we define the authorization predicate over the policy tuple \((\Pi_i, \Pi_{ac})\) simply in terms of the access control policy \( \Pi_{ac} \) by means of the following rules:

\[
\Pi_{ac}, \lambda 2\rho(\lambda) \vdash \text{permit}
\]

\[
(\Pi_i, \Pi_{ac}) \vdash \lambda, (\Pi_i, \Pi_{ac})
\]

which means that in our setting not-applicable has the same effect as deny. The authorization predicate definition relies on the function \( \lambda 2\rho(\cdot) \) that maps (a
subset of) the SCEL labels to SACPL requests. It is defined as follows:

\[
\lambda_2 \rho(I : t \triangle J) = \{(\text{subject}, I), (\text{item}, t), (\text{action}, \text{put}), (\text{object}, J)\}
\]

Intuitively, the authorization predicate acts as a Policy Enforcement Point in the XACML architecture [29], because it converts authorization requests written in the SCEL’s labels format into requests in the SACPL format. Hence, the authorization of a SCEL request \( \lambda \) over the policy \( \Pi_{ac} \) corresponds to establishing the authorization decision returned by the policy \( \Pi_{ac} \) in response to the corresponding SACPL request \( \rho = \lambda_2 \rho(\lambda) \), which is exactly the judgement \( \Pi_{ac} \vdash \rho \) defined by the rules in Table 12.

Notably, the use of different formats for the SCEL’s requests (identified by a subset of the system labels) and the SACPL’s requests permits decoupling the two languages, in order to make them as much independent as possible. The obvious benefit of this approach is that policies and requests may be written and evaluated independently of the specific environment in which they have to be enforced. In other words, SACPL can be easily integrated with other languages different from SCEL.

It is worth noticing that the evaluation of the authorization predicate above does not cause a change of the policy used to perform the check. More general cases will be presented in the next section.

8 SCEL and SACPL at work on a cloud computing scenario

We consider here a scenario from the ASCENS Cloud Computing Case Study defined in [30]. The science cloud is a collection of notebooks, desktops, servers, or virtual machines running the Science Cloud Platform (SCP). Each (virtual) machine is running (usually) one instance of the Science Cloud Platform; we call such instances Science Cloud Platform instances (SCPi). Each SCPi is considered to be a service component in the SCEL sense. Multiple SCPis communicate over the Internet (IP protocol), thus forming a cloud. In particular, we consider the setup of the science cloud depicted in Figure 3: there are seven SCPis in total, three at LMU Munich, three at IMT Lucca, and one mobile device (in the English Garden in Munich).

The considered High Load Scenario scenario is as follows. A singleton application currently runs on one of the VMs at IMT Lucca. This application runs alone on its node and experiences consistently high CPU load. Since the application is a singleton, no additional instances can be spawned. All other machines feature the same amount of CPU power.
The adaptivity decision is made to spawn a new VM with more processing power and run the application there. The SCPi currently running the application instructs the Zimory platform to start a new VM. Once the VM is up, an SCPi is injected into the machine and run. Once the instance joins the cloud, it is recognized and the application is moved there.

8.1 Scenario specification in SCEL

The SCPi where the application is running is the SCEL component

$$\mathcal{I}[\mathcal{K}, \mathcal{II}, AM[ME]]$$

where

$$ME \triangleq \texttt{qry}(\text{“key”}, ?x)@\texttt{self} \cdot \texttt{put}(\text{“key”}, f(x))@m.ME$$

is the application (i.e. the managed element), while

$$AM \triangleq \texttt{new}(\mathcal{J}, \mathcal{K}, \mathcal{II}, AM[ME])$$

is the autonomic manager. The managed element cyclically reads a local datum (e.g. a key), elaborates it and sends the result to the component m.

We assume that, other than id, the interfaces I and J provide the attribute CPUload. Its value is considered to be ‘high’ when it is greater than a given threshold, e.g. 80%. CPUload stores a context information, updated by the underlying infrastructure and ‘sensed’ by the managed element. It is also an awareness datum, since modification of its value triggers the autonomic manager.
The policy $\Pi$ in force at the component results from the composition, by means of the p-o (permit override) operator, of the following policies:

\[
\langle \text{deny}; \text{target: \{ \} } \rangle \quad \text{* deny all *}
\]
\[
\langle \text{permit}; \text{target: \{ equal(subject.id,n) and equal(object.id,n) and equal(action.qry) and less-or-equal-than(subject.CPUload,threshold) \} } \rangle \quad \text{* permit local qry *}
\]
\[
\langle \text{permit}; \text{target: \{ equal(subject.id,n) and equal(object.id,m) and equal(action.put) \} } \rangle \quad \text{* permit remote put *}
\]
\[
\langle \text{permit}; \text{target: \{ equal(action.new) and greater-than(subject.CPUload,threshold) \} } \rangle \quad \text{* enable new *}
\]

Basically, $\Pi$ says that

- action \texttt{qry} may only be performed until \texttt{subject.CPUload} is less than, or equal to, \texttt{threshold};
- action \texttt{put} may always be performed, leaving the value of \texttt{subject.CPUload} out of consideration;
- action \texttt{new} may not be performed until \texttt{subject.CPUload} is not greater than \texttt{threshold};
- all other actions that differs from those above are denied.

The rationale underlying this policy is that a \texttt{qry} may be computationally heavy (because it requires examination of the repository), while a \texttt{put} is a light operation (because it only requires addition of an item to the repository). Of course, different choices are possible.

Notably, $\Pi$ depends on the run-time value of attribute \texttt{CPUload}. This means that SACPL can express policies that may depend on the value of some parameters and can thus dynamically change according to the context. This is already a (limited) form of dynamism (we will see more expressive forms afterwards).

We must however ensure that whenever a new component (i.e. SCPi) is created and the application is moved there, if the run-time value of attribute \texttt{CPUload} of the ‘old’ component (i.e. SCPi) decreases and becomes less than \texttt{threshold}, the application instance running there cannot resume its execution (because the considered application is required to behave as a singleton).

Three ways of ensuring this are shown in the next subsections.

### 8.2 The first approach: use an additional attribute

The simpler approach is to use an additional attribute \texttt{status} initialized to \texttt{on}. It is set to \texttt{off} by \texttt{AM} whenever the value of attribute \texttt{CPUload} becomes greater than \texttt{threshold}. 
We need to modify the autonomic manager, that now becomes

\[
AM \triangleq \text{get}("attr", \text{status}, "on")@self.
\]

\[
\text{put}("attr", \text{status}, "off")@self.
\]

\[
\text{new}(J, K, \Pi_J, AM[ME])
\]

We also need to modify the policy \(\Pi\) by adding

- a check on the value of attribute \text{status} in the targets of its component policies;
- a new policy allowing to change the value of attribute \text{status} thus storing the information that the active instance of the application has been moved as a consequence of an high CPU load (thus the local instance has been blocked).

The new policy \(\Pi_1\) in force at the component results from the composition, by means of the \(p\)-o operator, of the following policies:

\[
\langle \text{deny}; \text{target}: \{ \} \rangle
\]

\[
\langle \text{permit}; \text{target}: \{ \text{equal}(\text{subject}.\text{id}, n) \text{ and equal}(\text{action}.\text{qry}) \text{ and less-or-equal-than}(\text{subject}.\text{CPUload}, \text{threshold}) \text{ and equal}(\text{subject}.\text{status}, \text{on}) \} \rangle
\]

\[
\langle \text{permit}; \text{target}: \{ \text{equal}(\text{subject}.\text{id}, n) \text{ and equal}(\text{action}.\text{put}) \text{ and equal}(\text{subject}.\text{status}, \text{on}) \} \rangle
\]

\[
\langle \text{permit}; \text{target}: \{ \text{equal}(\text{action}.\text{new}) \text{ and greater-than}(\text{subject}.\text{CPUload}, \text{threshold}) \text{ and equal}(\text{subject}.\text{status}, \text{on}) \} \rangle
\]

\[
\langle \text{permit}; \text{target}: \{ \text{equal}(\text{action}.\text{get}) \text{ and greater-than}(\text{subject}.\text{CPUload}, \text{threshold}) \text{ and pattern-match}(\text{item}.("attr", \text{status}, "on")) \} \rangle
\]

8.3 The second approach: use obligations

In order to manage the value of attribute \text{status}, instead of explicitly modifying \(AM\), we now use policies producing \textit{obligations}. Obligations define operations that should be performed in conjunction with the enforcement of an authorization decision. They correspond to e.g. updating a log file, sending a message, generating an event, setting an attribute.

In SACPL, we assume that obligations are possibly empty sequences (ranged over by \(s\)) of SCEL process actions acting on shared repositories and that they can be associated to a policy and returned as result of policy evaluation. Hence, to deal with obligations, we need a \textit{smooth} extension of SACPL syntax, as shown in Table 13 (where \(\epsilon\) denotes the empty sequence). We call SACPL+ the obtained policy language.

The advantage of using obligations consists in the fact that they permit to separate the operations needed for managing the modification of the policy from
Π ::= \langle Decision; \text{target}: \{\text{Targets}\}; \text{obligation}: \{s\}\rangle \quad \text{(atomic policy)}
| Π \text{p-o} Π \quad Π \text{d-o} Π \quad \text{(policy combination)}

s ::= ϵ \mid a.s \quad \text{(Obligations)}

a ::= \text{get}(T)@c \mid \text{qry}(T)@c \mid \text{put}(t)@c \quad \text{(SCEL Actions)}

Table 13. SACPL+ Syntax (excerpt)

the operations needed to manage the adaptation of the application. In fact, in our scenario, the autonomic manager could (again) be defined as

\[ AM \triangleq \text{new}(\mathcal{J}, \mathcal{K}, \Pi_{\mathcal{J}}, AM[ME]) \]

The policy is modified so that, whenever the value of CPUload is high, policy evaluation generates a sequence of actions that performs the needed update of attribute status. In fact, only the policy for action new changes and becomes:

\[ \langle \text{permit}; \text{target}: \{ \text{equal(action,new)} \text{ and}
\text{greater-than(subject.CPUload,threshold)} \text{ and}
\text{equal(status,on)} \}\rangle; \]
\[ \text{obligation}: \{ \text{get("attr",status,"on")}@self.
\text{put("attr",status,"off")}@self \} \]

Therefore, the policy \( \Pi_2 \) in force at the component results from the composition, by means of the p-o operator, of the following policies:

\[ \langle \text{deny}; \text{target}: \{ \} \rangle \]
\[ \langle \text{permit}; \text{target}: \{ \text{equal(subject.id,n)} \text{ and}
\text{equal(action, qry)} \text{ and}
\text{less-or-equal-than(subject.CPUload,threshold)} \text{ and}
\text{equal(status, on)} \}\rangle \]
\[ \langle \text{permit}; \text{target}: \{ \text{equal(subject.id,n)} \text{ and}
\text{equal(action, put)} \text{ and}
\text{equal(status, on)} \}\rangle \]
\[ \langle \text{permit}; \text{target}: \{ \text{equal(action,new)} \text{ and}
\text{greater-than(subject.CPUload,threshold)} \text{ and}
\text{equal(status, on)} \}\rangle; \]
\[ \text{obligation}: \{ \text{get("attr",status,"on")}@self.
\text{put("attr",status,"off")}@self \} \]
\[ \langle \text{permit}; \text{target}: \{ \text{equal(action, get)} \text{ and}
\text{equal(status, on)} \text{ and}
\text{pattern-match(item,("attr",status,"on")}) \}\rangle \]

The use of obligations has an impact on SACPL+ and SCEL semantics so that, after a transition, the obligations returned by the evaluation of the authorization predicate are enforced. Indeed, a request evaluation produces also some
obligations, say \( s \), which will be composed with the continuation of the SCEL process, say \( P \), running at the component so that the new process will be \( s.P \).

This ensures that all obligations will be executed before the process resumes.

When using obligations, the policy semantics judgements are of the form \( \Pi \vdash \rho, s \). Some significant rules are

\[
\langle \text{permit}; \text{target: } \{}; \text{ obligation: } \{ s \} \rangle \vdash \rho, s
\]

\[
\text{Targets} \vdash \rho, s'
\]

\[
\text{Targets}_1 \vdash \rho, s \lor \text{Targets}_2 \vdash \rho, s
\]

\[
(\text{Targets}_1 \text{ and Targets}_2) \vdash \rho, s
\]

The evaluation of the authorization predicate, differently from that of the interaction predicate that remains unchanged, additionally produces some (possibly empty) obligations and is defined by the following rule:

\[
\Pi_{ac} \vdash \lambda 2p(\lambda), s
\]

\[
(\Pi_1, \Pi_{ac}) \vdash \lambda, s, (\Pi_1, \Pi_{ac})
\]

SCEL’s operational semantics uses the obligations returned by the evaluation of an authorization predicate to ‘control’ execution of the running process, as e.g. in the rules shown below.
As a matter of notation, \( I.p \) indicate the process part of component \( I \), and, when \( s = \epsilon \), \( s.P \) stands for \( P \).

Let us now come back to the High Load Scenario. Whenever the value of attribute \( \text{CPUload} \) becomes greater than \( \text{threshold} \), action \textbf{new} is authorized and the obligations

\[
\begin{align*}
&\text{get}(\text{"attr"}, \text{status}, \text{"on"})@\text{self} \\
&\text{put}(\text{"attr"}, \text{status}, \text{"off"})@\text{self}
\end{align*}
\]

are returned. When action \textbf{new} is performed, the returned obligations are prefixed to the continuation process that becomes

\[
\begin{align*}
&\text{get}(\text{"attr"}, \text{status}, \text{"on"})@\text{self} \\
&\text{put}(\text{"attr"}, \text{status}, \text{"off"})@\text{self}.(\text{nil}[ME])
\end{align*}
\]

This ensures that attribute \text{status} is set to \text{off} before any action of \( ME \) can be performed (i.e. execution of \( ME \) can no longer resume).

### 8.4 The third approach: use automata

The value of attribute \text{status} represents two different policies that are in force within the component at different times during the execution of the application. The initial policy, characterized by attribute \text{status} set to \text{on}, is maintained until checking the value of attribute \text{CPUload} returns a value greater than \text{threshold}. After which \text{status} is set to \text{off} by \( AM \) and the policy changes accordingly (it becomes a deny-all policy). This observation leads us to the last approach that explicitly represents the fact that the policies can dynamically change.

We use sort of automata where states are policies and state transitions represent occurrence in the system of security relevant events triggering policy modification. These automata are the most dynamic forms of policies we consider and are somehow reminiscent of the security automata [24]. An automaton represents all the policies that can be in force within a given component at different times and the events that can cause their modification. Concretely, automata states are \( \text{SACPL} \) policies, while state transitions are labeled by \( \text{SACPL} \) targets.

The use of automata has a certain impact on the semantics of \( \text{SACPL} \) and \( \text{SCEL} \), as we sketch in the rest of this section.

The access control policy \( \Pi_{ac} \) in force within a component at a given time is represented by a pair \( \langle A, \Pi \rangle \), where \( A \) is an automaton and \( \Pi \) is its current state. A request \( \rho \) (obtained by applying function \( \lambda_2(\cdot) \) to labels generated by \( \text{SCEL} \) operational semantics) is allowed if \( \Pi \vdash \rho \); moreover, if for some target \( \text{Targets} \), such that \( \text{Targets} \vdash \rho \), there is a transition \( \Pi \xrightarrow{\text{Targets}} \Pi' \), then the state of the automaton \( A \) after the request evaluation becomes \( \Pi' \).

The new policy semantics, written \( \langle A, \Pi \rangle \vdash \rho, \langle A, \Pi' \rangle \), can be easily expressed in terms of the previous one \( \Pi \vdash \rho \) by means of the following rule

\[
\begin{align*}
&\Pi \vdash \rho \\
&\Pi' = \begin{cases} \\
&\Pi'' \quad \text{if } \text{Targets} \vdash \rho \land \Pi \xrightarrow{\text{Targets}} \Pi'' \\
&\Pi \quad \text{otherwise}
\end{cases}
\end{align*}
\]

\( \langle A, \Pi \rangle \vdash \rho, \langle A, \Pi' \rangle \)
Notably, the current state $\Pi$ does not change unless there is a target $\text{Targets}$ matching the request $\rho$ and producing a transition in the security automaton.

The evaluation of the authorization predicate, differently from that of the interaction predicate that remains unchanged, can cause a change of the state of the automaton and is defined by the following rule:

$$\langle A, \Pi \rangle \vdash \lambda 2\rho(\lambda), \langle A, \Pi' \rangle$$

$$(\Pi_i, \langle A, \Pi \rangle) \vdash \lambda, (\Pi_i, \langle A, \Pi' \rangle)$$

Let us now come back to the High Load Scenario. The automaton has two states, the initial (active) state $\Pi$ and the passive state $\Pi_{off}$ given by the deny-all policy

$$(\text{deny}; \text{target}: \{\})$$

and only one transition

$$\Pi \xrightarrow{\text{equal(action,new)}} \Pi_{off}$$

9 A runtime environment for SCEL programs

In this section we present jRESP\footnote{JRESP (Java Run-time Environment for SCEL Programs) website: http://jresp.sourceforge.net/}, a Java runtime environment providing a framework for developing autonomic and adaptive systems according to the SCEL paradigm. Specifically, jRESP provides an API that permits using in Java programs the SCEL’s linguistic constructs for controlling the computation and interaction of autonomic components, and for defining the architecture of systems and ensembles.

The implementation of jRESP fully relies on the SCEL’s formal semantics. This close correspondence enhances confidence on the behaviour of the jRESP implementation of SCEL programs, once the latter have been analysed through 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’s 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\(^{10}\). 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.

9.1 jRESP’s main features

**Components.** SCEL components are implemented via the class *Node*. The architecture of a node is shown in Figure \(\text{4}\). 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.

**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 \cite{27} notion of tuple space (see the dialect SCEL\(_{TS}\) described in Section 6.2). 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*.

\(^{10}\) JSON (JavaScript Object Notation) website: \url{http://www.json.org/}
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.

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 and FreePastry. 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 VirtualPort is used to simulate nodes in a single application without relying on a specific network infrastructure.

---

11 This mechanism is implemented via the Observer/Observable pattern.
12 Scribe is a generic, scalable and efficient system for group communication and notification.
13 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 Policy.

9.2 The swarm robotics scenario in jRESP

We report here the code of the jRESP implementation of the SCEL specification, presented in Section 6.3, of the swarm robotics scenario. The Java classes reported in this section permit appreciating how close the SCEL processes are to their implementation in jRESP.

Process ME is rendered as the agent ManagedElement defined below:

```java
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, the method doRun() is invoked. This method defines the agent behaviour. In the case of ManagedElement, it consists

---

14 The complete source code for the scenario, together with a simulation environment, can be downloaded from [http://jresp.sourceforge.net/](http://jresp.sourceforge.net/)
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 SCEL’s action \texttt{qry}. The method 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 \texttt{Self.SELF}) while the retrieved tuple is one consisting of two fields: the first field is the constant “controlStep” while 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 \texttt{P_{batteryMonitor}}, \texttt{P_{dataSeeker}} and \texttt{P_{targetSeeker}}, respectively:

```java
public class BatteryMonitor extends Agent {
    public BatteryMonitor() {
        super("BatteryMonitor");
    }
    protected void doRun() throws IOException, InterruptedException {
        while (true) {
            query( new Template(
                new ActualTemplateField("batteryLevel") ,
                new ActualTemplateField("low") ) ,
                Self.SELF );
            get( new Template(
                new ActualTemplateField("lowBattery") ,
                new ActualTemplateField(false) ) ,
                Self.SELF );
            put( new Tuple( "lowBattery", true ), Self.SELF );
            query( new Template(
                new ActualTemplateField("batteryLevel") ,
                new ActualTemplateField("high") ) ,
                Self.SELF );
            get( new Template(
                new ActualTemplateField("lowBattery") ,
                new ActualTemplateField(true) ) ,
                Self.SELF );
            put( new Tuple( "lowBattery", false ), Self.SELF );
        }
    }
}

public class DataSeeker extends Agent {
    public DataSeeker() {
        super("DataSeeker");
    }
    protected void doRun() throws IOException, InterruptedException {
        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.elementAt(Double.class,1);
double y = t.elementAt(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);
}
}

public class TargetSeeker extends Agent {
    public TargetSeeker() {
        super("TargetSeeker");
    }
    protected void doRun() throws IOException, InterruptedException {
        while (true) {
            Tuple t = query(new Template(
                new ActualTemplateField("lowBattery") ,
                new FormalTemplateField(Boolean.class)) ,
            Self.SELF);
            boolean low = t.elementAt(Boolean.class,1);
            if (low) {
                get(new Template(
                    new ActualTemplateField( "controlStep" ),
                    new FormalTemplateField( Agent.class ) ) ,
                Self.SELF);
                put( new Tuple( "controlStep" , new LowBattery() ) , Self.SELF);
                query( new Template(
                    new ActualTemplateField("lowBattery") ,
                    new ActualTemplateField(false)) ,
                Self.SELF );
            } else {
                t = query(new Template(
                    new ActualTemplateField("target") ,
                    new FormalTemplateField(Boolean.class)) ,
                Self.SELF);
                boolean found = t.elementAt(Boolean.class, 1);
                if (found) {
                    get( new Template(
                        new ActualTemplateField( "controlStep" ) ,
                        new FormalTemplateField( Agent.class ) ) ,
                    Self.SELF);
                    put( new Tuple( "controlStep" , new Found() ) , Self.SELF );
                    doTask();
                } else {
t = query(new Template(
    new ActualTemplateField("informed") ,
    new FormalTemplateField(Boolean.class) ),
    Self.SELF);
boolean informed = t.getElementAt(Boolean.class, 1);
if (informed) {
    get( new Template(
        new ActualTemplateField( "controlStep" ),
        new FormalTemplateField( Agent.class ) ),
        Self.SELF);
    put( new Tuple("controlStep", new Informed() ),
        Self.SELF);
} else {
    get( new Template(
        new ActualTemplateField( "controlStep" ),
        new FormalTemplateField( Agent.class ) ),
        Self.SELF);
    put( new Tuple("controlStep", new RandomWalk() ),
        Self.SELF);
}
}
}

private void doTask() {
    //This method implements the manger for task i ...
}

The correspondence between the code within the method doRun() of the three classes listed above and the definitions of the three SCEL processes given in Section 6.3 is straightforward. It is worth noticing the code for the group-oriented communication in the class DataSeeker. 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 executed by the managed element, namely P_lowBattery, P_found, P_informed and P_randomWalk, is as follows:

public class LowBattery extends Agent {
    public LowBattery() {
        super("LowBattery");
    }
    protected void doRun() throws IOException, InterruptedException {
        put( new Tuple( "stop" ), Self.SELF );
        Tuple t = query(new Template( new ActualTemplateField( "gps" ) ,
            new FormalTemplateField(Double.class) ,
            new FormalTemplateField(Boolean.class) ,
            Self.SELF));
        boolean informed = t.getElementAt(Boolean.class, 1);
        if (informed) {
            get( new Template(
                new ActualTemplateField("controlStep"),
                new FormalTemplateField(Agent.class) ),
                Self.SELF);
            put( new Tuple("controlStep", new Informed() ),
                Self.SELF);
        } else {
            get( new Template(
                new ActualTemplateField( "controlStep" ),
                new FormalTemplateField(Agent.class) ),
                Self.SELF);
            put( new Tuple("controlStep", new RandomWalk() ),
                Self.SELF);
        }
    }
}

private void doTask() {
    //This method implements the manger for task i ...
}

new FormalTemplateField(Double.class),

Self.SELF);
double x = t.getElementAt(Double.class, 1);
double y = t.getElementAt(Double.class, 2);
put(new Tuple("sos", x, y), new Group(new HasValue("task", 1)));
get(new Template(new ActualTemplateField("rescued")), Self.SELF);
}
}

public class Found extends Agent {
    public Found() {
        super("Found");
    }
    protected void doRun() throws IOException, InterruptedException {
        put(new Tuple("stop"), Self.SELF);
        Tuple t = query(new Template(new ActualTemplateField("gps"),
                                      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("targetLocation", x, y), Self.SELF);
        //Execute task i ...
    }
}

public class Informed extends Agent {
    public Informed() {
        super("Informed");
    }
    protected void doRun() throws IOException, InterruptedException {
        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);
    }
}

public class RandomWalk extends Agent {
    Random r = new Random();
    public RandomWalk() {
        super("RandomWalk");
    }
    protected void doRun() throws IOException, InterruptedException {

put( new Tuple("direction", r.nextDouble() * 2 * Math.PI) , Self.SELF );

10 Related work

Our proposal combines the notion of ensemble with concepts emerged from different research fields, like autonomic computing, multi-agent systems, component-based design, context-oriented programming, network architectures and concurrency theory. Below, we review some of the closely related works concerning these fields.

Declarative programming has been proposed to program ensembles, see e.g. Meld [33,34] and Declarative Networking [35]. 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. SCEL, instead, is a formal language that could be used as the kernel of a programming language for ensembles. Its operational semantics describes the computational steps that a system can perform and lays the basis for developing implementations (indeed, it has been used for jRESP) and tools for program analysis.

Among the many works focussing on the self-adaptation capability of autonomic systems, we want to mention [36]. 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. [6,7,8,9,10]) share with SCEL the importance assigned to knowledge representation and to the way single agents can handle it. However, SCEL components can directly access the knowledge repositories of other components (provided this is allowed by the policies in force at the involved components), 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 [13]).

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 [5]. A relevant example in this field is Fractal [37], 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 expressive then the corresponding mechanisms used in SCEL and not adequate to deal with highly dynamic ensembles.

Context-Oriented Programming (COP) \cite{26,38} has been advocated to program autonomic systems \cite{39}. 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 \cite{40}). Only few works provide a foundational account of programming languages extended with COP facilities like, e.g., the object-oriented ones of \cite{41,42,43} and the functional one of \cite{44}. 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 \cite{14} 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. \cite{11,15} introduces the concept of \textit{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 \cite{11,15} 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 [45] and [46], relying on the (bio)chemical programming paradigm, have been proposed for the specification of autonomic systems. Some other formalisms, like e.g. [47] and [48], aiming at modeling 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. Compared to these proposals, SCEL allows one to provide high-level abstract descriptions of systems that nevertheless have a direct correspondence with their implementation.

11 Concluding remarks and future directions

We have presented the kernel language SCEL, i.e. a set of linguistic abstractions specifically devised 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 case studies from the robotics and service provision domains. 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 then be 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 and 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.

References

46. Andrei, O., Kirchner, H.: A higher-order graph calculus for autonomic computing. In: Graph Theory, Computational Intelligence and Thought, Springer (2009) 15–26