The SCEL Language: Design, Implementation, Verification

Rocco De Nicola, Rocco x; Latella, Diego ; Lluch Lafuente, Alberto; Loreti, Michele ; Margheri, Andrea Margheri; Massink, Mieke ; Morichetta, Andrea ; Pugliese, Rosario ; Tiezzi, Francesco ; Vandin, Andrea

Published in:
Software Engineering for Collective Autonomic Systems

Link to article, DOI:
10.1007/978-3-319-16310-9_1

Publication date:
2015

Document Version
Peer reviewed version

The SCEL Language: Design, Implementation, Verification*

Rocco De Nicola[1], Diego Latella[2], Alberto Lluch Lafuente[1,3], Michele Loreti[4], Andrea Margheri[4], Mieke Massink[2], Andrea Morichetta[1], Rosario Pugliese[4], Francesco Tiezzi[1], and Andrea Vandin[1,5]

[1] IMT Institute for Advanced Studies Lucca, Italy
[2] Istituto di Scienza e Tecnologie dellInformazione 'A. Faedo', CNR, Italy
[3] DTU Compute, The Technical University of Denmark, Denmark
[4] Università degli Studi di Firenze, Italy
[5] University of Southampton, UK

Abstract. SCEL (Service Component Ensemble Language) is a new language specifically designed to rigorously model and program autonomic components and their interaction, while supporting formal reasoning on their behaviors. SCEL brings together various programming abstractions that allow one to directly represent aggregations, behaviors and knowledge according to specific policies. It also naturally supports programming interaction, self-awareness, context-awareness, and adaptation. The solid semantic grounds of the language is exploited for developing logics, tools and methodologies for formal reasoning on system behavior to establish qualitative and quantitative properties of both the individual components and the overall systems.

Keywords: Autonomic computing, Programming languages, Adaptation policies, Formal methods, Verification.

* Research supported by the European projects IP 257414 ASCENS and the Italian PRIN 2010LHT4KM CINA.

Table of Contents

1 Introduction . . . 3
2 The Parametric Language SCEL . . . 5
  2.1 Design Principles . . . 6
  2.2 Syntax . . . 7
  2.3 Operational Semantics . . . 11
3 Knowledge Management . . . 20
  3.1 Tuple Spaces . . . 20
  3.2 Constraints . . . 21
  3.3 External Reasoners . . . 23
4 A Policy Language . . . 26
  4.1 Policies and their Syntax . . . 27
  4.2 Semantics of the Policy Language . . . 29
5 A Full-fledged SCEL Instance . . . 34
  5.1 PSCEL: Policed SCEL . . . 34
  5.2 PSCEL at Work . . . 39
6 A Runtime Environment for SCEL . . . 43
  6.1 Programming Constructs . . . 44
  6.2 Policing Constructs . . . 46
  6.3 Exploitation . . . 47
7 Quantitative Variants of SCEL . . . 50
  7.1 StocS: Stochastic SCEL . . . 51
  7.2 Semantics of a StocS Fragment . . . 53
8 Verification . . . 57
  8.1 Simulation and Analysis via jRESP . . . 57
  8.2 Maude-based Verification . . . 59
  8.3 Spin-based Verification . . . 64
9 Concluding Remarks . . . 67

1 Introduction

Nowadays much attention is devoted to software-intensive cyber-physical systems. These are systems possibly made of a massive numbers of components, featuring complex intercommunications and interactions both with humans and other systems and operating in open and unpredictable environments. It is therefore necessary that such systems dynamically adapt to new requirements, technologies and contextual conditions. Such classes of systems include the so-called *ensembles* [44]. Sometimes ensembles are explicitly created by design, while some other other times they are assembled from systems that are independently controlled and managed, while their interaction "mood" may be cooperative or competitive; then one has to deal with systems coalitions, also called *systems of systems*. Due to their inherent complexity, today's engineering methods and tools do not scale well to ensembles and new engineering techniques are needed to address the challenges of developing, integrating, and deploying them [53]. The design of such systems, their implementation and the verification that they meet the expectations of their users pose big challenges to language designers and software engineers. It is of paramount importance 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.

It is thus important to look for methodologies and linguistic constructs that can be used to build ensembles while combining traditional software engineering approaches, techniques from autonomic, adaptive, knowledge-based and self-aware systems, and formal methods, in order to guarantee compositionality, expressiveness and verifiability. It has to be said that most of the basic properties of the class of systems we have outlined above are already guaranteed by current service-oriented architectures; the novelties come from the need of self-awareness and context-awareness. Indeed, self-management is a key challenge of modern distributed IT infrastructures spanning almost to all levels of computing. Self-managing systems are designed to continuously monitor their behaviors in order to select the optimal meaningful operations to match the current status of affairs. After [30], the term *autonomic computing* has been used to identify the self-managing features of computing systems. A variety of inter-disciplinary proposals has been launched to deal with autonomic computing. We refer to [47] for a detailed survey.

In this chapter, we propose facing the challenge of engineering autonomic systems by taking as starting point the notions of *autonomic components* (ACs) and *autonomic-component ensembles* (ACEs) and defining programming abstractions to model their evolutions and their interactions. Building on these notions, we define SCEL (Software Component Ensemble Language). This is a kernel language that takes a holistic approach to model and program autonomic computing systems. SCEL aims at providing programmers with an appropriate set of linguistic abstractions for programming the behavior of ACs and the formation of ACEs, and for controlling the interaction among different ACs.

SCEL permits governing the complexity of such systems by providing flexible abstractions, by enabling transparent monitoring of the involved entities and by supporting the implementation of self-* mechanisms such as self-adaptation. The key concepts of the language are those of *Behaviors*, *Knowledge*, *Aggregations* and *Policies* that have proved fruitful in modelling autonomic systems from different application domains such as, e.g., collective robotic systems, cloud-computing, and cooperative e-vehicles.

One of the distinguishing features of SCEL is the use of flexible, *group-oriented*, communication primitives that allows one to implicitly select the set of components (the ensemble) to communicate with, by evaluating a given predicate $\mathcal{P}$ used as the target. When a communication action has predicate $\mathcal{P}$ as a target, it will involve all components that satisfy $\mathcal{P}$. For example, if a system contains elements that export attributes such as *serviceProvided* and *QoS* and one would like to program a component willing to interact with the ensemble of all the components that provide a service s and offer a QoS above q, (s)he can use the predicate $serviceProvided = s \wedge QoS > q$ to select the component's partners.

We would like to add that SCEL is, somehow, minimal; its syntax fully specifies only constructs for modeling Behaviors and Aggregations and is parametric with respect to Knowledge and Policies. This choice permits integrating different approaches to policies specifications or to knowledge handling within our language and to easily superimpose ACEs on top of heterogeneous ACs. Indeed, we see SCEL as a *kernel* language based on which different full-blown languages can be designed. Afterwards, we will present a simple, yet expressive, SCEL's dialect that is equipped with a specific language for defining access control policies and that relies on knowledge repositories implemented as distributed *tuple spaces*. The small set of basic constructs and their *solid semantic grounds* permits us to develop logics, tools and methodologies for formal reasoning on systems behavior in order to establish qualitative and quantitative properties of both the individual components and the ensembles.

In this chapter, we will present most of the work that has been done within the ASCENS project on the SCEL language. We shall introduce the main linguistic abstractions for components specification and interaction together with different alternatives for modeling knowledge and for the operations for knowledge handling (we refer to Chapter II.3 [54]for more sophisticated forms of knowledge and to Chapter II.4 [27]for other knowledge-based reasoning techniques). We shall also discuss different possibilities for describing interaction and authorization policies. We shall describe a Java runtime environment, to be used for developing autonomic and adaptive systems according to the SCEL paradigm and thus for the deployment of SCEL specifications (we refer to Chapter III.5 [1]for other software tools for supporting the development of this class of systems). Finally, we shall introduce tools and methodologies for the verification of qualitative and quantitative properties of SCEL programs (we refer to Chapter I.3 [17]for other verification techniques and tools).

The main features of SCEL will be presented in a step-by-step fashion by using, in most of the following sections, a running example from the swarm

robotics domain described below (we refer to Chapter IV.2 [42]for a comprehensive presentation of the swarm robotics case study). A complete account of the specification of this scenario is given in Section 5.2.

A swarm robotics scenario. We consider a scenario where a swarm of robots spreads throughout a given area where some kind of disaster has happened. The goal of the robots is to locate and rescue possible victims. As common in swarm robotics, all robots playing the same role execute the same code. According to the separation of concerns principle fostered by SCEL, this code consists of two parts: *(i)* a process, defining the functional behaviour; and *(ii)* a collection of policies, regulating the interactions among robots and with their environment and generating the (adaptation) actions necessary to react to specific (internal or environmental) conditions. This combination permits a convenient design and enacts a collaborative swarm behaviour aiming at achieving the goal of rescuing the victims.

A robot initially plays the *explorer* role in order to look in the environment for the victims' positions. When a robot finds a victim, it changes to the *rescuer* role starting the victim rescuing and indicating the victim's position to the other robots. As soon as another robot receives the victim's position, it changes to the *helpRescuer* role going to help other rescuers. During the exploration, in case of critical battery level, a robot changes to the *lowBattery* role to activate the battery charging. Notably, the role changes according to the sensors and data values, e.g. when the robot is close to a victim that needs help.

Outline of the Chapter. The rest of this chapter is structured as follows. Section 2 introduces the key principles underlying the design of SCEL together with the syntax and the operational semantics of the language. Section 3 presents two different knowledge handling mechanisms, i.e. *tuple spaces* and *constraint stores*, and illustrates how components can exploit external *reasoners* for taking decisions. Section 4 introduces a language for defining access control, resource usage and adaptation policies. Section 5 presents a full instantiation of SCEL: it uses tuple spaces as knowledge handling mechanism and the language presented in Section 4 as policy language. Section 6 describes a Java runtime environment that provides an API for using SCEL's linguistic constructs in Java programs. Section 7 deals with the issue of enriching SCEL with information about action duration, by providing a stochastic semantics for the language. Section 8 deals with verification of qualitative and quantitative properties of SCEL specifications via the analysis tools provided by the runtime environment illustrated in Section 6, the MAUDE framework [16], and the SPIN model checker [28]. Section 9 concludes by also touching upon directions for future work.

2 The Parametric Language SCEL

In this section we first introduce the key principles underlying the design of the SCEL language. Then, we formally present its syntax and operational semantics.

2.1 Design Principles

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

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

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

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

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

By accessing and manipulating their own knowledge repository or the repositories of other ACs, components acquire information about their status (*self-awareness*) and their environment (*context-awareness*) and can perform *self-adaptation*, initiate *self-healing* actions to deal with system malfunctions, or

install *self-optimizing* behaviors. All these *self-** properties, as well as *self-configuration*, can be naturally expressed by exploiting SCEL's higher-order features, namely the capability to store/retrieve (the code of) processes in/from the knowledge repositories and to dynamically trigger execution of new processes. Moreover, by implementing appropriate security policies, e.g. limiting information flow or external actions, components can set up *self-protection* mechanisms against different threats, such as unauthorised access or denial-of-service attacks.

Our aim is to provide a common semantic framework for describing meaning and interplay of the abstractions above, while minimizing overlaps and incompatibilities. In the subsection below we introduce the constructs of SCEL, while their precise semantics will be presented in the next one.

2.2 Syntax

We present here the syntax of SCEL. We would like to stress that we have taken a minimal approach and SCEL syntax specifies only constructs for modeling Behaviors and Aggregations and is parametric with respect to Knowledge and Policies.

Concretely, an AC in SCEL is rendered as the term $\mathcal{I}[\mathcal{K}, \Pi, P]$. This is graphically illustrated in Figure 1 and consists of:

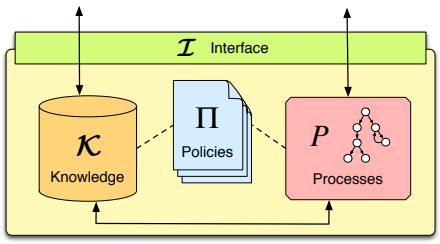

Fig. 1. SCEL component

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

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

Table 1. SCEL syntax (KNOWLEDGE $\mathcal{K}$, POLICIES Π, TEMPLATES T, and ITEMS t are parameters of the language)

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

SCEL syntax is reported in Table 1. Its basic category is the one defining PROCESSES that are used to build up COMPONENTS that in turn are used to define SYSTEMS. PROCESSES specify the flow of the ACTIONS that can be performed. ACTIONS can have a TARGET to determine the other components that are involved in that action. As stated in the Introduction, SCEL is parametric with respect to some syntactic categories, namely KNOWLEDGE, POLICIES, TEMPLATES and ITEMS (with the last two determining the part of KNOWLEDGE to be retrieved/removed or added, respectively).

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

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

$$S \triangleq \textsc{Robot}_1 \parallel \ldots \parallel \textsc{Robot}_n$$

Each robot is rendered as a SCEL component $\textsc{Robot}_i$, which has the form $\mathcal{I}_{R_i}[\mathcal{K}_{R_i}, \Pi_R, P_R]$. These components concurrently execute and interact. Each interface $\mathcal{I}_{R_i}$ specifies the attribute *role*, which can assume values *explorer*, *rescuer*, etc. according to the current role played by the robot. □

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* $(P_1 \mid P_2)$, *process variable* (X), and *parameterized process invocation* $(A(\bar{p}))$. The construct $P_1 \mid P_2$ abstracts the various forms of parallel composition commonly used in process calculi. Process variables support *higher-order* communication, namely the capability to exchange (the code of) a process and possibly execute it. This is realized by first adding an item containing the process to a knowledge repository and then retrieving/withdrawing this item while binding the process to a process variable. We assume that A ranges over a set of parameterized *process identifiers* that are used in recursive process definitions. We also assume that each process identifier A has a *single* definition of the form $A(\bar{f}) \triangleq P$. Lists of actual and formal parameters are denoted by $\bar{p}$ and $\bar{f}$, respectively.

Running example (step 2/7) The process P_R running on a robot has the form $(a_1.\, P_1 + a_2.\, P_2) \mid P_3$ meaning that it is a parallel composition of two subprocesses, where the one on the left-hand side of the controlled composition can either execute the action a_1 and thereafter continue as P_1, or execute the action a_2 and thereafter continue as P_2. □

Actions and targets. Processes can perform five different kinds of ACTIONS. Actions $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$ and $\mathbf{put}(t)@c$ are used to manage shared knowledge repositories by withdrawing/retrieving/adding information items from/to the knowledge repository identified by c. These actions exploit templates T as patterns to select knowledge items t in the repositories. They heavily rely on the used knowledge repository and are implemented by invoking the handling operations it provides. Action $\mathbf{fresh}(n)$ introduces a scope restriction for the name n so that this name is guaranteed to be *fresh*, i.e. different from any other name previously used. Action $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ creates a new component $\mathcal{I}[\mathcal{K}, \Pi, P]$.

Action **get** may cause the process executing it to wait for the expected 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 expected 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 expressions 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 $\mathsf{this}.status = \text{“}sending\text{”} \wedge status = \text{“}receiving\text{”}$ is satisfied when the status of the subject component is *sending* and that of the object is *receiving*.

In actions using a predicate $\mathcal{P}$ to indicate the target (directly or via p), the predicate acts as a 'guard' specifying *all* components that may be affected by the execution of the action, i.e. a component must satisfy $\mathcal{P}$ to be the target of the action. Thus, actions $\mathbf{put}(t)@n$ and $\mathbf{put}(t)@\mathcal{P}$ give rise to two different primitive forms of communication: the former is a *point-to-point* communication, while the latter is a sort of *group-oriented* communication. The set of components satisfying a given predicate $\mathcal{P}$ used as the target of a communication action are considered as the *ensemble* with which the process performing the action intends to interact. Indeed, in spite of the stress we put on ensembles, SCEL does not have any specific syntactic category or operator for forming ACEs. For example, the names of the components that can be members of an ensemble can be fixed via the predicate $id \in \{n, m, o\}$. When an action has this predicate as target, it will act on all components named n, m or o, if any. Instead, to dynamically characterize the members of an ensemble according to the role they are currently playing in the system, by assuming that attribute *role* belongs to the interface of any component willing to be part of the ensemble, one can write $role=\text{“}rescuer\text{”} \vee role=\text{“}helpRescuer\text{”}$ to refer to the ensemble of components playing either the role *rescuer* or *helpRescuer*.

It is worth noticing that the group-oriented variant of action **put** is used to insert a knowledge item in the repositories of all components belonging to the ensemble identified by the target predicate. Differently, group-oriented actions **get** and **qry** withdraw and retrieve, respectively, an item from *one* of the components satisfying the target predicate, non-deterministically selected.

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

$$(\mathbf{qry}(\text{“}victimPerceived\text{”}, \mathsf{true})@\mathsf{self}.\, P_1 \\ + \mathbf{get}(\text{“}victim\text{”}, ?x, ?y, ?c)@(role=\text{“}rescuer\text{”} \vee role=\text{“}helpRescuer\text{”}).\, P_2) \mid P_3$$

The sub-process on the left-hand side of the controlled composition allows the robot to recognise the presence of a victim, by means of the **qry** action, or to help other robots to rescue a victim, by means of the **get** action. In the latter case, the action binds the victim's coordinates to variables x and y, and the number of other robots needed for rescuing the victim to variable c. □

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

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

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

Table 2. Semantics of processes

2.3 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.

Semantics of processes. Process commitments are generated by the following production rule

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

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 the two commitments α and β. We write $P \downarrow_\alpha Q$ to mean that "P can commit to perform α and become Q after doing so".

The relation $\downarrow$ defining the semantics of processes is the least relation induced by the inference rules in Table 2.

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

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

Running example (step 4/7) The process P_R running on the robots, apart for the trivial case $P_R \downarrow_{\circ[\,\circ\,]} P_R$ and the commitments of P_3 (not specified here), produces the following meaningful commitments

$$P_R \downarrow_{\mathbf{qry}(\text{“}victimPerceived\text{”},\mathsf{true})@\mathsf{self}[\,\circ\,]} (P_1 \mid P_3)$$

$$P_R \downarrow_{\mathbf{get}(\text{“}victim\text{”},?x,?y,?c)@(role=\text{“}rescuer\text{”}\vee role=\text{“}helpRescuer\text{”})[\,\circ\,]} (P_2 \mid P_3)$$

□

Semantics of systems. The operational semantics of systems is defined in two steps. First, the possible behaviors of systems without occurrences of the name restriction operator are defined. This is done in the SOS style [43] by relying on the notion of Labeled Transition System (LTS). Then, by exploiting this LTS, the semantics of generic systems is provided by means of a (unlabelled) Transition System (TS) only accounting for systems' computation steps. This approach allows us to avoid the notational intricacies arising when dealing with name mobility in computations (e.g. when opening and closing the scopes of name restrictions).

The labeled transition relation of the LTS defining the semantics of systems without restricted names is induced by the inference rules in Tables 4, 5 and 6. We write $S \xrightarrow{\lambda} S'$ to mean that "S can perform a transition labeled λ and become S' in doing so". Transition labels are generated by the following production rule

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

where γ is either the name n of a component or a predicate $\mathcal{P}$ indicating a set of components, and $\mathcal{I}$ and $\mathcal{J}$ range over interfaces[6]. The meaning of labels is as follows: τ denotes an internal computation step; $\mathcal{I} : \mathbf{fresh}(n)$ denotes the willingness of component $\mathcal{I}$ to restrict visibility of name n; $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$ denotes the willingness of component $\mathcal{I}$ to create the new component $\mathcal{J}[\mathcal{K}, \Pi, P]$; $\mathcal{I} : t \triangleleft \gamma$ (resp. $\mathcal{I} : t \blacktriangleleft \gamma$) denotes the intention of component $\mathcal{I}$ to withdraw

[6] The names of the attributes of a component are pointers to the real values contained in the knowledge repository associated to the component. This amounts to saying that in terms of the form $\mathcal{I}[\mathcal{K}, \Pi, P]$, $\mathcal{I}$ only includes the names of the attributes, as their corresponding values can be retrieved from $\mathcal{K}$. However, when $\mathcal{I}$ is used in isolation (e.g., within a label), we assume that it also includes the attributes' values; we then use, for example, $\mathcal{I}.id$ to denote the value associated to the attribute id in the corresponding repository.

(resp. retrieve) item t from the repositories at γ; $\mathcal{I} : t \triangleright \gamma$ denotes the intention of component $\mathcal{I}$ to add item t to the repositories at γ; $\mathcal{I} : t \bar{\triangleleft} \mathcal{J}$ (resp. $\mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J}$) denotes that component $\mathcal{I}$ is allowed to withdraw (resp. retrieve) item t from the repository of component $\mathcal{J}$; $\mathcal{I} : t \bar{\triangleright} \mathcal{J}$ denotes that component $\mathcal{I}$ is allowed to add item t to the repository of component $\mathcal{J}$. Moreover, in the rules, we use $\mathcal{I}.\pi$ to denote the policy in force at the component $\mathcal{I}$, $\mathcal{I}[\Pi/\mathcal{I}.\pi]$ to denote the update of the policy of the component $\mathcal{I}$ with the policy Π, $\bullet$ to denote a placeholder for the policy of a component and $S[\Pi/\bullet]$ to denote the replacement of the placeholder $\bullet$ with a policy Π in a system S.

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

- The *interaction predicate*, $\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi'$, means that under policy Π and interface $\mathcal{I}$, process commitment α yields system label λ, substitution σ (i.e., a partial function from variables to values) and, possibly new, policy Π'. Intuitively, λ identifies the effect of α at the level of components, while σ associates values to the variables occurring in α and is used to capture the changes induced by communication. The generated system label λ must be one among τ, $\mathcal{I} : \mathbf{fresh}(n)$, $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$, $\mathcal{I} : t \triangleleft \gamma$, $\mathcal{I} : t \blacktriangleleft \gamma$ and $\mathcal{I} : t \triangleright \gamma$. Π' is the policy in force after the transition; in principle it may differ from the one in force before the transition. This predicate is used to determine the effect of the simultaneous execution of actions by processes concurrently running within a component that, e.g., exhibit commitments of the form $\alpha[\,\beta\,]$.
- The *authorization predicate*, $\Pi \vdash \lambda, \Pi'$, means that under policy Π, the action generating the system label λ (which can be thought of as an *authorization request*) is allowed and the policy Π' is produced. Labels λ taken as argument by the authorization predicate are system labels of the form $\mathcal{I} : \mathbf{fresh}(n)$, $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$, $\mathcal{I} : t \bar{\triangleleft} \mathcal{J}$, $\mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J}$, or $\mathcal{I} : t \bar{\triangleright} \mathcal{J}$. This predicate is used to determine the actions allowed by specific policies, and the (possibly new) policy to be enforced. The authorization to perform an action is checked when a computation step can potentially take place, i.e. when it becomes known which is the component target of the action.

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

A specific interaction predicate is given in Table 3; it is obtained by interpreting controlled composition as the *interleaved* parallel composition of the two involved processes. Notably, this simple predicate does never modify the policy currently in force. Notice also that process commitments corresponding to inaction ($\circ$, $\circ[\,\circ\,]$, etc.) are disallowed. In the table, function $[\![\,\cdot\,]\!]_{\mathcal{I}}$ denotes the evaluation of terms with respect to interface $\mathcal{I}$ with attributes occurring therein being replaced by the corresponding value in $\mathcal{I}$. Moreover, $match(T, t)$ denotes a partial function performing matching between a template T and an item t;

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

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

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

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

Table 3. The interleaving interaction predicate

when they do match, the function returns a substitution σ for the variables in T (we use $\{\}$ to denote the empty substitution), and is otherwise undefined. We have a rule for each different kind of process action; for example, the third rule states that, once the target γ of the action and an item t matching the template T' through a substitution σ have been determined (by also exploiting the interface $\mathcal{I}$ for evaluating c and T), an action **qry** at the level of processes corresponds to a proper transition label at the level of systems semantics. The last two rules ensure that in case of controlled composition of multiple processes only one process at a time can perform an action (the other stays still).

Like the interaction predicate, many different reasonable authorization predicates can be defined, possibly resorting to specific policy languages. One of such languages inspired by, but simpler than, the OASIS standard for policy-based access control XACML [39], will be presented in Section 4. There, we will stress also how the actual semantics of this policy language is intertwined and integrated with SCEL semantics.

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

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

We now briefly comment the rules in Table 4. Rule *(pr-sys)* transforms process commitments into system labels by exploiting the interaction predicate. As a consequence, a substitution σ is applied to the continuation P' of the process that committed to α. When α contains a $\mathbf{get}(T)$ or a $\mathbf{qry}(T)$, σ replaces in P' the variables occurring in T with the corresponding values. The application of the rule also replaces, in the generated label, self with the corresponding name. When α is a **fresh**, it is checked if the name is not already used in the creating component, except for the process part that will likely use n as, e.g., an information to be added to some knowledge repository (notation $\mathrm{n}(E)$ is used here to

$$\frac{P \downarrow_\alpha P' \quad \alpha = \mathcal{I} : \mathbf{fresh}(n) \Rightarrow n \notin \mathrm{n}(\mathcal{I}[\mathcal{K}, \Pi, \mathbf{nil}]) \quad \Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\lambda[\Pi'/\mathcal{I}.\pi]} \mathcal{I}[\mathcal{K}, \bullet, P'\sigma]} \textit{(pr-sys)}$$

$$\frac{C \xrightarrow{\mathcal{I}:\mathbf{fresh}(n)} C' \qquad \mathcal{I}.\pi \vdash \mathcal{I} : \mathbf{fresh}(n), \Pi'}{C \xrightarrow{\tau} (\nu n)\, C'[\Pi'/\bullet]} \textit{(freshn)}$$

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

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

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

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

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

Table 4. Systems' labeled transition relation (1/3): base rules

denote the sets of names occurring in a syntactic term E); this condition can be always made true by exploiting alpha-equivalence among processes. Moreover, as a consequence of the evaluation of the interaction predicate, the policy in force at the component performing the action may change; this update is registered in the produced system label by applying $[\Pi'/\mathcal{I}.\pi]$ to the label λ generated by the interaction predicate. Notably, the component generated by this transition contains a placeholder $\bullet$ in place of the policy; it will be replaced by a (possibly new) policy during the rest of the derivation (see, e.g., the use of $[\Pi'/\bullet]$ in rule *(freshn)*).

The possibility of executing actions **fresh** and **new** is decided by using the information within a single component. However, since these actions affect the system, as they either create a name restriction or a new component, their execution by a process is indicated by a specific system label $\mathcal{I} : \mathbf{fresh}(n)$ or $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$ (generated by rule *(pr-sys)*) carrying enough information for the authorization request to perform the action to be checked according to the local policy and for the modification of the system to take place (rules *(freshn)* and *(newc)*). Notably, the authorization predicate is evaluated under the policy produced by the interaction predicate (rule *(pr-sys)*); thus, the component per-

forming the action will enforce the (possibly new) policy so generated. Notably, rule *(freshn)* relies on the condition checked in rule *(pr-sys)*, about the freshness of the new name n in the creating component, in order to put in place its scope.

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. Rules *(accget)*, *(accqry)* and *(accput)* are used to generate the labels denoting the willingness of components to accept the execution of an action. More specifically, rule *(accget)* generates the label $\mathcal{I} : t \triangleleft \mathcal{J}$ indicating the willingness of component $\mathcal{J}$ to provide the item t to component $\mathcal{I}$. Notably, the label is generated only if such willingness is authorized by the policy in force at the component $\mathcal{J}$ (by means of the authorization predicate $\Pi \vdash \mathcal{I} : t \triangleleft \mathcal{J}, \Pi'$) and if withdrawing item t from the repository of $\mathcal{J}$ is possible ($\mathcal{K} \ominus t = \mathcal{K}'$). An effect of this transition is also the update of policy Π in Π' (both in the resulting component and in the produced label). Rules *(accqry)* and *(accput)* are similar to *(accget)*, the only difference being that they invoke the retrieval ($\mathcal{K} \vdash t$) and the addition ($\mathcal{K} \oplus t = \mathcal{K}'$) operations of the repository's handling mechanism, respectively, rather than the withdrawal one. Finally, rule *(async)* states that all actions different from a **put** for group-oriented communication and an authorization for a **put** can be performed by involving only some of the system's components. Therefore, if there is a system component able to perform the authorization for a **put**, there is no way to infer that such component in parallel with any other one (hence the system as a whole) can perform the action. This ensures that when a system component is going to execute a **put** for group-oriented communication all potential receivers are taken into account.

The rules in Table 5 model the variants of the three communication actions implementing point-to-point interaction, while the rules for group-oriented communication are shown in Table 6.

In case of point-to-point interaction, 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 transition labelled by $\mathcal{I} : t \triangleleft \mathcal{I}$ in the premise of *(lget)* can only be produced by rule *(accget)*; it ensures that the component $\mathcal{I}$ authorizes the local access to item t and that the component's knowledge and policy are updated accordingly. When the target of the action denotes a specific remote repository *(ptpget)*, the action is only allowed if n is the name of the component $\mathcal{J}$ simultaneously willing to provide the wanted item and if the request to perform the action at $\mathcal{J}$ is authorized by the local policy (identified by notation $\mathcal{I}.\pi$). Of course, if there are multiple components with the same name, one of them is non-deterministically chosen as the target of the action. Action **qry** behaves similarly to **get**, the only difference being that, if the action succeeds, after the computation step all repositories remain unchanged. Its semantics is modeled by rules *(lqry)* and *(ptpqry)*. Finally, action **put** adds item t to a repository. Its behavior is modeled by rules *(lput)* and *(ptpput)*, that are similar to those of actions **get** and **qry**, with the major difference being that, if the action succeeds, after the computation step an item is added to the target repository.

$$\frac{C \xrightarrow{\mathcal{I}:t\lhd n} C' \quad n = \mathcal{I}.id \quad C'[\mathcal{I}.\pi/\bullet] \xrightarrow{\mathcal{I}:t\,\bar{\lhd}\,\mathcal{I}} C''}{C \xrightarrow{\tau} C''}\ \textit{(lget)}$$

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

$$\frac{C \xrightarrow{\mathcal{I}:t\blacktriangleleft n} C' \quad n = \mathcal{I}.id \quad C'[\mathcal{I}.\pi/\bullet] \xrightarrow{\mathcal{I}:t\,\bar{\blacktriangleleft}\,\mathcal{I}} C''}{C \xrightarrow{\tau} C''}\ \textit{(lqry)}$$

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

$$\frac{C \xrightarrow{\mathcal{I}:t\rhd n} C' \quad n = \mathcal{I}.id \quad C'[\mathcal{I}.\pi/\bullet] \xrightarrow{\mathcal{I}:t\,\bar{\rhd}\,\mathcal{I}} C''}{C \xrightarrow{\tau} C''}\ \textit{(lput)}$$

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

Table 5. Systems' labeled transition relation (2/3): point-to-point communication rules

Let us now comment the rules for group-oriented communication that are shown in Table 6. When the target of action **get** denotes a set of repositories satisfying a given predicate *(grget)*, the action is only allowed if one of these repositories, say that of component $\mathcal{J}$, is willing to provide the wanted item and if the request to perform the action at $\mathcal{J}$ is authorized by the policy in force at the component performing the action. Relation $\mathcal{J} \models \mathcal{P}$ states that the attributes of $\mathcal{J}$ satisfy predicate $\mathcal{P}$; the definition of such relation depends on the kind of the used predicates. In any case, if the action succeeds, this transition corresponds to an internal computation step (denoted by τ) that changes the repository of component $\mathcal{J}$. Rule *(grqry)* is similar, but in the case of action **qry** the item is not removed from the repository. Differently from the two previous actions that only capture the interaction with one target component arbitrarily chosen among those satisfying the predicate $\mathcal{P}$ and willing to provide the wanted item, $\mathbf{put}(t)@\mathcal{P}$ can interact with all components satisfying $\mathcal{P}$ and willing to accept the item t. In fact, rule *(grput)* permits the execution of a **put** for group-oriented communication when there is a parallel component, say $\mathcal{J}$, satisfying the target of the action and whose policy authorizes this remote access. Of course, the action must be authorized to use $\mathcal{J}$ as a target also by the policy in force at the component performing the action (which is updated after each evaluation of the authorization predicate). Notably, the resulting action is still a **put** for group-oriented communication, thus further authorization actions performed by other

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

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

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

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

Table 6. Systems' labeled transition relation (3/3): group communication rules

parallel components satisfying the target of the action can be simultaneously executed.

The capability of a component to perform a **put** for group-oriented communication is not affected by those system components not satisfying predicate $\mathcal{P}$, i.e. not belonging to the ensemble, or not authorising the action according to the policy in force at the sending component or at the target ones (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).

Running example (step 5/7) Let us suppose that $\mathcal{I}_{R_2}.role$="*rescuer*" and $\mathcal{I}_{R_3}.role$="*helpRescuer*", while $\mathcal{I}_{R_i}.role$="*explorer*" for $4 \leq i \leq n$. Suppose also that $\mathcal{K}_{R_3}$ contains an item indicating that the victim has position $(3, 5)$ and that 3 additional robots are needed for rescuing it.

Now, by exploiting the operational rule *(accget)*, the third component can generate the following labelled transition

$$\mathcal{I}_{R_3}[\mathcal{K}_{R_3}, \Pi_R, P_R] \xrightarrow{\mathcal{I}_{R_1}:\langle\text{“}victim\text{”},3,5,3\rangle\,\triangleleft\,\mathcal{I}_{R_3}} \mathcal{I}_{R_3}[\mathcal{K}_{R_3} \ominus \langle\text{“}victim\text{”}, 3, 5, 3\rangle, \Pi_R, P_R]$$

Recall that $\mathcal{K}_{R_3} \ominus \langle\text{“}victim\text{”}, 3, 5, 3\rangle$ means that the information about the victim is withdrawn from the knowledge repository $\mathcal{K}_{R_3}$.

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

$$\mathcal{I}_{R_1}[\mathcal{K}_{R_1}, \Pi_R, P_R] \xrightarrow{\lambda} \mathcal{I}_{R_1}[\mathcal{K}_{R_1}, \bullet, (P_2\{3/x, 5/y, 3/c\} \mid P_3)]$$

$$\frac{S \xrightarrow{\tau} S'}{(\nu\bar{n})S \succ\!\!\rightarrow (\nu\bar{n})S'} \textit{ (tau)} \qquad \frac{S \xrightarrow{\mathcal{I}:t\triangleright\mathcal{P}} S'}{(\nu\bar{n})S \succ\!\!\rightarrow (\nu\bar{n})S'[\mathcal{I}.\pi/\bullet]} \textit{ (put)}$$

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

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

Table 7. Systems' transition relation

where λ is

$$\mathcal{I}_{R_1}[\Pi'_R/\Pi_R] : \langle \text{“}victim\text{”}, 3, 5, 3\rangle \triangleleft (role{=}\text{“}rescuer\text{”} \vee role{=}\text{“}helpRescuer\text{”})\,.$$

Hence, by exploiting the operational rule *(grget)* and assuming $\Pi'_R \vdash \lambda, \Pi''_R$, the overall system can perform the transition

$$\begin{aligned} S \xrightarrow{\tau} {} & \mathcal{I}_{R_1}[\mathcal{K}_{R_1}, \Pi''_R, (P_2\{3/x, 5/y, 3/c\} \mid P_3)] \parallel \textsc{Robot}_2 \\ & \parallel \mathcal{I}_{R_3}[\mathcal{K}_{R_3} \ominus \langle \text{“}victim\text{”}, 3, 5, 3\rangle, \Pi_R, P_R] \parallel \textsc{Robot}_4 \parallel \ldots \parallel \textsc{Robot}_n \triangleq S' \end{aligned}$$

□

The unlabeled transition relation ($\succ\!\!\rightarrow$) of the TS providing the semantics of generic systems is defined on top of the labeled one by the inference rules in Table 7. As a matter of notation, $\bar{n}$ denotes a (possibly empty) sequence of names and $\bar{n}, n'$ is the sequence obtained by composing $\bar{n}$ and n'. $(\nu\bar{n})S$ abbreviates $(\nu n_1)((\nu n_2)(\cdots(\nu n_m)S\cdots))$, if $\bar{n} = n_1, n_2, \cdots, n_m$ with $m > 0$, and S, otherwise. $S\{n'/n\}$ denotes the system obtained by replacing any free occurrence in S of n with n'. When considering a system S, a name is deemed *fresh* if it is different from any name occurring in S.

Rule *(tau)* of Table 7 accounts for the computation steps of a system where all (possible) name restrictions are at top level. Rule *(put)* states that, besides those labeled by τ, computation steps may additionally be labeled by $\mathcal{I} : t \triangleright \mathcal{P}$, corresponding to group-oriented communication triggered by an action $\mathbf{put}(t)@\mathcal{P}$ performed by component $\mathcal{I}$, and thus transforms them into transitions of the form $\succ\!\!\rightarrow$. This rule also takes care of updating the policy in force at the sending component with the policy produced by the last evaluation of the authorization predicate in the inference of transition $S \xrightarrow{\mathcal{I}:t\triangleright\mathcal{P}} S'$. Rule *(top)* permits to manipulate the syntax of a system, by moving all name restrictions at top level, thus putting it into a form to which one of the first two rules can be possibly applied. This manipulation may require the renaming of a restricted name with a freshly chosen one, thus ensuring that the name moved at top level is different both from the restricted names already moved at top level (to avoid name

clashes) and from the names occurring free in the other (sub)systems in parallel (to avoid improper name captures). Rules *(comm)* and *(assoc)* state that systems' composition is a commutative and associative operator. Notably, by exploiting these two rules, we can manipulate systems and avoid adding analogous rules to those defining the labeled transition relation.

Running example (step 6/7) The robotics system can thus evolve by performing the reduction $S \rightarrowtail S'$. □

3 Knowledge Management

As we have seen in the previous section, the SCEL language definition abstracts from a few ingredients of the language. In this section, we show two different knowledge mechanisms that can be used to instantiate the knowledge parameter. We start presenting the simplest, yet effective, instantiation of knowledge repositories based on multiple distributed *tuple spaces* à la KLAIM [20]. Then, we consider *constraints*, which are suitable to represent partial knowledge, to deal with multi-criteria optimization, to express preferences, fuzziness, and uncertainty. Finally, we show how knowledge can be exploited by external *reasoners* for taking decisions according to (a partial perception of) the context.

3.1 Tuple Spaces

Table 8 shows how to instantiate knowledge repositories, items and templates to deal with tuple spaces. Knowledge ITEMS are *tuples*, i.e. sequences of values, while TEMPLATES are sequences of values and variables. KNOWLEDGE repositories are then *tuple spaces*, i.e. (possibly empty) multisets of stored tuples $\langle t \rangle$. We use $\emptyset$ to denote an empty 'place' and the operator $\_ \parallel \_$ to aggregate items in multisets. Values within tuples can either be targets c, or processes P or, more generally, can result from the evaluation of some given expression e. We assume that expressions may contain attribute names, *boolean*, *integer*, *float* and *string* values and variables, together with the corresponding standard operators. To pick a tuple out from a tuple space by means of a given template, the *pattern-matching* mechanism is used: a tuple matches a template if they have the same number of elements and corresponding elements have matching values or variables; variables match any value of the same type ($?\,x$ and $?\,X$ are used to bind variables to values and processes, respectively), and two values match only if they are identical. If more tuples match a given template, one of them is arbitrarily chosen.

This form of knowledge representation has been already used in the running examples shown in the previous section. For instance, the template ("victim", $?x, ?y, ?c$) is used as argument of a **get** action to withdraw a 4-element tuple from the repository of one of the robots that knows the victim position. The first element of such a tuple must be the string "victim"; the other three values will be bound to variables x, y, and c, respectively.

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

Table 8. Tuple space syntax (e is an EXPRESSION)

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

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

Table 9. Tuple space operations ($\ominus$, $\vdash$, $\oplus$)

The three operations provided by the knowledge repository's handling mechanism, namely *withdrawal* ($\mathcal{K} \ominus t$), *retrieval* ($\mathcal{K} \vdash t$) and *addition* ($\mathcal{K} \oplus t$) of an item t from/to repository $\mathcal{K}$, are inductively defined by the inference rules shown in Table 9. Notably, when a matching tuple is withdrawn from $\mathcal{K}$, it is replaced by the empty place $\emptyset$.

3.2 Constraints

In this section, we report some basic definitions concerning the concept of (soft) constraints. Among the many available formalizations, hereafter we refer to the one based on *c-semirings* [7, 45], which generalizes many of the others.

Intuitively, a constraint is a relation that gives information on the possible values that the variables of a specified set may assume. We adopt a functional formulation. Hence, given a set V of variables and a domain D of values that the variables may assume, assignments and constraints are defined as follows.

Definition 1 (Assignments). *An* assignment η *of values to variables is a function* $\eta : V \to D$.

Definition 2 (Constraints). *A* constraint χ *is a function* $\chi : (V \to D) \to \{\mathsf{true}, \mathsf{false}\}$.

A constraint is then represented as a function that, given an assignment η, returns a truth value indicating if the constraint is satisfied by η. An assignment that satisfies a constraint is called a *solution*.

When SCEL's knowledge repositories are instantiated as multiple distributed constraint stores, D could be taken as the set of SCEL *basic values* (e.g., integers and strings). Variables in V, that we call *constraint variables* to take them apart from those of SCEL processes, could be written as pairs of names of the

form $n@n'$ (e.g., $batteryLevel@robot_i$), where n is the variable name and n' the name of the component that *owns* the variable. Different components may own variables with the same name; such variables are distinct and may thus store different values.

We denote an assignment as a collection of pairs of the form $n@n' \mapsto v$, where $n@n'$ and v range over variables and values, respectively. Such pairs explicitly specify the associations for only the variables relevant for the considered constraint; these variables form the so-called *support* [8] of the constraint, which is assumed to be finite. For example, given the constraints $batteryLevel@robot_i \geq 20\%$ and $lifetime@robot_i = batteryLevel@robot_i \cdot 3000$, the assignment $\{batteryLevel@robot_i \mapsto 25\%,\ lifetime@robot_i \mapsto 1000\}$ satisfies the first constraint (i.e., returns true) but does not satisfy the second one (i.e., returns false).

The constraints introduced above are called *crisp* in the literature, because they can only be either satisfied or violated. A more general notion is represented by the *soft constraints*. These constraints, given an assignment, return an element of an arbitrary constraint semiring (*c-semiring* [7]). C-semirings are partially ordered sets of 'preference' values equipped with two suitable operations for comparison ($+$) and combination ($\times$) of (tuples of) values and constraints.

Definition 3 (C-semiring). *A* c-semiring *is an algebraic structure* $\langle S, +, \times, 0, 1\rangle$ *such that:* S *is a set and* $0, 1 \in S$*;* $+$ *is a binary operation on* S *that is commutative, associative, idempotent,* 0 *is its unit element and* 1 *is its absorbing element;* $\times$ *is a binary operation on* S *that is commutative, associative, distributes over* $+$*,* 1 *is its unit element and* 0 *is its absorbing element. Operation* $+$ *induces a partial order* $\leq$ *on* S *defined by* $a \leq b$ *iff* $a + b = b$*, which means that* a *is more constrained than* b *or, equivalently, that* b *is better than* a*. The minimal element is thus* 0 *and the maximal* 1*.*

Definition 4 (Soft constraints). *Let* $\langle S, +, \times, 0, 1\rangle$ *be a c-semiring. A* soft constraint χ *is a function* $\chi : (V \to D) \to S$*.*

In particular, crisp constraints can be understood as soft constraints on the c-semiring $\langle\{\mathsf{true}, \mathsf{false}\}, \vee, \wedge, \mathsf{false}, \mathsf{true}\rangle$.

By lifting the c-semiring operators to constraints, we get the operators

$$(\chi_1 + \chi_2)(\eta) \ = \ \chi_1(\eta) + \chi_2(\eta) \qquad\qquad (\chi_1 \times \chi_2)(\eta) \ = \ \chi_1(\eta) \times \chi_2(\eta)$$

(their n-ary extensions are straightforward). We can formally define the notions of consistency and entailment. The *consistency* condition $\chi \neq 0$ stands for

$$\exists\, \eta \ : \ \chi(\eta) \neq 0$$

i.e. a constraint is consistent if it has at least a solution; the *entailment* condition $\chi_1 \leq \chi_2$ stands for

$$\forall\, \eta,\ \chi_1(\eta) \leq \chi_2(\eta)$$

$\mathcal{K} \ominus \chi = \begin{cases} \mathcal{K}' \\ \mathcal{K} \end{cases}$	*if* $\mathcal{K} \equiv \mathcal{K}' \parallel \chi$ *otherwise*
$\mathcal{K} \vdash \chi$	*if* $\mathcal{K} \equiv (\chi_1 \parallel \ldots \parallel \chi_m)$ *and* $(\chi_1 \times \ldots \times \chi_m) \leq \chi$
$\mathcal{K} \oplus \chi = \mathcal{K} \parallel \chi$	*if* $\mathcal{K} \equiv (\chi_1 \parallel \ldots \parallel \chi_m)$ *and* $(\chi_1 \times \ldots \times \chi_m \times \chi) \neq 0$

Table 10. Constraint store operations ($\ominus$, $\vdash$, $\oplus$)

When constraints are used as the argument of actions **put**, **qry** and **get**, these actions play the role of actions **tell**, **ask** and **retract**, respectively, commonly used in the CCP paradigm [48] to add a constraint to a store, to check entailment of a constraint by a store and to remove a constraint from a store. These constraints may only involve constraint variables whose owner is the component target of the action. This ensures that all the constraints stored in the same repository only involve variables owned by the same component, which is the owner of the repository. Thus, for example, it will never happen that the $robot_2$'s repository stores a constraint like $batteryLevel@robot_1 < 100\%$.

The three operations provided by the knowledge repository's handling mechanism, namely *withdrawal* ($\mathcal{K} \ominus \chi$), *retrieval* ($\mathcal{K} \vdash \chi$) and *addition* ($\mathcal{K} \oplus \chi$) of a constraint χ from/to repository $\mathcal{K}$, are inductively defined by the inference rules shown in Table 10. We use $\mathcal{K}_1 \equiv \mathcal{K}_2$ to denote that $\mathcal{K}_1$ and $\mathcal{K}_2$ are equal up to commutation of items. In the definition of $\mathcal{K} \vdash \chi$ and $\mathcal{K} \oplus \chi$, if the constraint store is empty (i.e. $m = 0$), then it suffices to verify that χ is a tautology (i.e., it is a constant function returning the c-semiring value 1 for any assignment) and that χ has at least a solution (i.e., it differs from the c-semiring value 0), respectively.

As an example of use in our robotics scenario of the constraint-based interaction, $robot_1$ could perform the action $\mathbf{qry}(lifetime@robot_2 > 1000)@robot_2$ to check if the lifetime of $robot_2$ is at least 1000 seconds (which could be, e.g., the minimum time to transport the victim to a safe area). Assuming that the $robot_2$'s repository stores the constraints $batteryLevel@robot_2 = 50\%$ and $lifetime@robot_2 = batteryLevel@robot_2 \cdot 3000$, the entailment of constraint $lifetime@robot_2 > 1000$ is satisfied and, hence, the execution of the robot behaviour can proceed with the continuation process of the **qry** action.

3.3 External Reasoners

As discussed, SCEL is sufficiently powerful for dealing with coordination and interaction issues. However, it does not provide explicit machineries for specifying components that take decisions about the action to perform based on their context. Obviously, the language could be extended in order to encompass such possibilities, and one could have specific reasoning phases, or dedicated SCEL components, triggered by the perception of changes in the context.

The general perspective. In our view, it is however preferable to have separate reasoning components specified in another language, that SCEL programs can invoke at need. Having two different languages for computation and coordination, and for *reasoning*, does guarantee *separation of concerns*, a fundamental property to obtain reliable and maintainable specifications. Also, it may be beneficial to have a methodology for integrating different reasoners designed and optimised for specific purposes.

What we envisage is having SCEL programs that whenever have to take decisions have the possibility of invoking an external reasoner by providing it information about the relevant knowledge they have access to, and receiving in exchange informed suggestions about how to proceed. In a scenario like the robot rescue one, reasoners could for example be exploited by robots to "improve" their random walk phase, e.g. trying to minimise collisions in an environment densely populated by robots moving in an unexpected way. Intuitively, the current robot's perception of the surrounding environment should be provided to a reasoner, which would return the "best" movement direction according to the probability of colliding with other robots and, possibly, to other criteria.

As a matter of fact, in [5] we provided a general methodology to enrich SCEL components with reasoning capabilities by resorting to explicit *reasoner integrators*, we instantiated the methodology for MISSCEL[7], a SCEL interpreter, and we discussed the integration of MISSCEL with the PIRLO reasoner [4]. This permits to specify *reasoning service component ensembles*, and also paves the way towards the exploitation of tools and techniques for analysing their behaviour, allowing thus to *reason on* reasoning service component ensembles. An example is the collision avoidance scenario considered in [5], which has been analysed exploiting MULTIVESTA [51], a recently proposed statistical model checker. More details about the scenario and its analysis are provided in Section 8.2.

In the following we present our approach to enrich SCEL components with external reasoning capabilities, in particular focusing on a SCEL instance where repositories are implemented as multisets of tuples (as in Section 3.1), while we refer to [5] for details about its instantiation for MISSCEL.

The methodology. We aim at enriching SCEL components with an external reasoner to be *invoked* when necessary (e.g. by a robot before performing a movement). Ideally, this should be done by minimally extending SCEL. In Figure 1 we depicted the constituents of a SCEL component: interfaces, policies, processes and repositories. Interfaces will not be involved in the extension, as they only expose the local knowledge to other components. Moreover, we currently restrict ourselves to not explicitly consider policies in the extension. Since, in the considered dialect, processes store and retrieve tuples in repositories, the interaction between a process and its local repository is a natural choice where to plug-in a reasoner: we can use special data (*reasoning request tuples*) whose addition to the local knowledge (i.e. via a **put** at `self`) triggers the reasoner. For example, assuming we have a reasoner offering the capability of computing

[7] `http://sysma.lab.imtlucca.it/tools/misscel/`

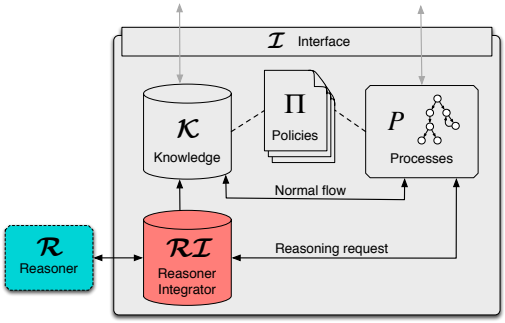

Fig. 2. Enriched SCEL component

the best direction where to move so as to minimize the probability of collisions, a robot may invoke the reasoner before performing a movement by resorting to an action like **put**("*reasoningRequest*", "*computeDirection*", *perceivedEnv*)@self, where *perceivedEnv* is the current perception that the robot has of the surrounding environment (e.g. the number and position of robots within a certain range). Reasoning results can then be stored in the knowledge as *reasoning result tuples*, allowing local processes to access them as any other data (e.g. via a **get** from `self`). For example, the direction "*dir*" generated by the reasoner can be accessed by resorting to an action like **get**("*reasoningResult*", *dir*)@self.

Figure 2 depicts such an *enriched* SCEL *component*, together with a generic external reasoner *R*. With respect to Figure 1, now local communications are filtered by *RI*, a *reasoner integrator*. As depicted by the grey arrow between *RI* and *R*, in case of reasoning requests *RI* invokes *R*, which evaluates the request and returns back the result of the reasoning phase. *RI* then stores the obtained result in the knowledge, allowing the local processes to access it via common **get** or **qry**. In case of normal data the flow goes instead directly to the knowledge. Note that only local **put** of reasoning request tuples trigger a reasoner.

Actually, *RI* has the further fundamental role of translating data among the internal representations used by SCEL and by the reasoner, acting hence as an adapter between them. For example, the reasoner may use a different representation for the space (and thus for the positions of the other robots) with respect to SCEL. To sum up, *RI* performs three tasks: it translates the parameters of the reasoning requests from SCEL's representation to the reasoner's one (*scel2reasoner*), it invokes the reasoner (*invokeReasoner*), and finally it translates back the results (*reasoner2scel*). Clearly, each reasoner requires its own implementation of the three operations. Hence, as depicted in Figure 3, we separate the *RI* component into an *Abstract Reasoning Interface* and a *Concrete Adapter*. The former is given just once and contains the definition of the three operations, while the latter is reasoner- and domain-specific, and provides the actual implementation of the three operations. In [5] we discussed the instantiation for MISSCEL of the Abstract Reasoning Interface, together with an example

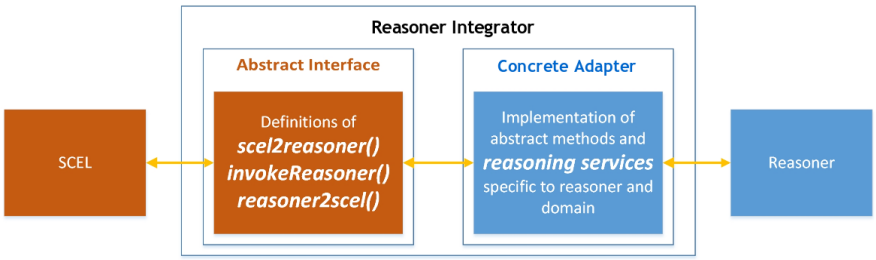

Fig. 3. An architectural perspective of the reasoner integrator

of a concrete adapter for the reasoner PIRLO in the context of the mentioned collision avoidance robotic scenario.

Note that the presented methodology is not restricted to a particular reasoner. Moreover, many reasoners could be used at the same time, each performing particular reasoning tasks for which they are best suited. To this end, particular *reasoning services* (like e.g. the *computeDirection* one) can be requested by a SCEL process according to the task at hand.

Finally, it may be worth to remark that, as mentioned, we did not investigate yet the role of policies in extending SCEL's components with reasoning capabilities. However, they already play an important role in our methodology, as they can manipulate the flow of data among processes and local repositories, and thus can intercept, modify or generate reasoning requests and results. Moreover, we can easily foresee a scenario in which complicated policies, possibly involving reasoning tasks, resort to a reasoner as well, following the proposed methodology. For example, in case of a group **get** like, e.g. the second action of the process presented in the *step 3/7* of our running example, which is used to help other robots for rescuing a victim, it may be useful to allow policies to use reasoners in order to select the *best* tuple among the many matching ones present in a distributed repository (e.g. different help requests) according to some specific criteria (e.g. the one regarding the nearest robot, or the most urgent one).

4 A Policy Language

The SCEL programming constructs presented in Section 2 define the computational behaviour of components in a procedural style. According to the SCEL design principles, the interaction and adaptation logics are defined separately by means of behavioural policies. These policies have to be intuitive and easy-to-maintain, therefore the use of a declarative paradigm for their specification is advocated. Recently, *policy languages* (see e.g. [18, 31, 29]) are receiving much attention in different research fields, varying, e.g., from access control to network management. In fact, policies can regulate multiple system's aspects and, by using a declarative approach, can be easily integrated with other programming languages.

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

Table 11. Policy constructs

Here, we present a simplified version[8] of FACPL (Formal Access Control Policy Language) [36], a simple, yet expressive, language for defining access control, resource usage and adaptation policies, which is inspired by the XACML [39] standard for access control. We refer the interested reader to [35] for a presentation of the full version of FACPL, which contains additional aspects that are not exploited in the integration with SCEL. Syntax and semantics of policy abstractions are presented in Section 4.1 and Section 4.2, respectively.

4.1 Policies and their Syntax

Policies are sets of rules that specify strategies, requirements, constraints, guidelines, etc. about the behaviour of a controlled system. The syntax is presented in Table 11. As a matter of notation, symbol ? stands for optional elements, $*$ for (possibly empty) sequences, and $+$ for non-empty sequences. For the sake of readability, whenever an element is missing, we also omit the possibly related keyword; thus, e.g., rule $(d \;\; \mathsf{target}:\tau \;\; \mathsf{condition}: \;\; \mathsf{obl}:)$ will be written as $(d \;\; \mathsf{target}:\tau)$.

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

[8] In the rest of the paper, unless when explicitly mentioned, we use the acronym FACPL for referring to this simplified version.

A TARGET indicates the authorisation requests to which a policy/rule applies. It is either an atomic target or a pair of simpler targets combined using the standard logic operators $\wedge$ and $\vee$. An *atomic target* $f(pv,sn)$ is a triple denoting the application of a matching function f to *policy values*[9] pv from the policy and to policy values from the evaluation context identified by *attribute (structured) names*[10] sn. In fact, an attribute name refers to a specific attribute of the request or of the environment, which is available via the evaluation context. In this way, an authorisation decision can be based on some characteristics of the request, e.g. subjects' or objects' identity, or of the environment, e.g. CPU load. For example, the target greater-than(90%,*subject*/*CPUload*) matches whenever 90% is greater then the CPU load of the subject component. Similarly, the structured name *action*/*action-id* refers to the identifier of the action to be performed (such as **get**, **qry**, **put**, etc.) and, thus, the target equal(**get**, *action*/*action-id*) matches whenever such action is the withdrawing one.

Rules are the basic elements for request evaluation. A RULE defines the tests that must be successfully passed by attributes for returning a positive or negative DECISION — i.e. permit or deny — to the enclosing policy. This decision is returned only if the target is 'applicable', i.e. the request matches the target; otherwise the evaluation of the rule returns not-applicable. In fact, as shown in Section 4.2, the semantics of the policies is defined over a three-valued decision δ that, in addition to permit and deny, can also assume the value not-applicable. Rule applicability can be further refined by the CONDITION expression *be*, which permits more complex calculations than those in target expressions. *be* is a boolean expression which acts on policy values and structured names. Notably, these expressions, as well as the matching functions, can be extended in order to properly deal with specific data types.

A COMBINING ALGORITHM computes the authorisation decision corresponding to a given request by combining a set of rules/policies' evaluation results. The language provides the following algorithms:

- deny-overrides: if any rule/policy in the considered list evaluates to deny, then the result of the combination is deny. In other words, deny takes precedence, regardless of the result of evaluating any of the other rules/policies in the list. Instead, if at least a rule/policy evaluates to permit and all others evaluate to not-applicable or permit, then the result of the combination is permit. If all policies are found to be not-applicable to the decision request, then the policy set evaluates to not-applicable.
- permit-overrides: this algorithm is the dual of the previous one, i.e. this time permit takes precedence over the other results.

[9] The set of policy values depends on the system where the policies are enforced. In case of SCEL systems, this set contains action identifiers (i.e., **get**, **qry**, **put**, **fresh** and **new**), items and templates, and all the other knowledge values that can be used within the evaluation.

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

– deny-unless-permit: this algorithm is similar to permit-overrides, because it is intended for those cases where a permit decision takes precedence over deny decisions; differently from permit-overrides, this algorithm never returns not-applicable, i.e. it is converted to deny.
– permit-unless-deny: this algorithm is the dual of the previous one, i.e. deny takes precedence over permit decisions and not-applicable is never returned.
– first-applicable: rules/policies are evaluated in the order of appearance in the considered list of rules/policies and the combined result is the same as the result of evaluating the first rule/policy in the list that is applicable to the decision request, if such result is either permit or deny. If all rules/policies evaluate to not-applicable, then the overall result is not-applicable.
– only-one-applicable: if one and only one rule/policy in the considered list is applicable by virtue of its target, the result of the combining algorithm is the result of evaluating the single applicable rule/policy. Otherwise, the result is not-applicable.

An OBLIGATION is a sequence (ϵ denotes the empty one) of actions that should be performed in conjunction with the enforcement of an authorisation decision. It is returned when the authorisation decision for the enclosing element, i.e. rule, policy or policy set, is the same as the one attached to the obligation. An OBLIGATION ACTION is a generic action that can be used for enforcing additional behaviours in the controlled system and whose arguments may also contain expressions and structured names that are fulfilled during request evaluation. Like policy values, the set of obligation actions depends on the system where the policies are enforced. For example, w.r.t. a given request, the obligation

$$[\,\mathsf{deny}\ \ \mathbf{put}(\text{“}goTo\text{”}, env/station.x, env/station.y)@\mathsf{self}\,]$$

returned when the authorisation decision for the enclosing element is deny, could be fullfilled as follows

$$\mathbf{put}(\text{“}goTo\text{”}, 5.45, 3.67)@\mathsf{self}$$

and could be used to set the robot movement's direction with respect to the coordinates of the closest (charging) station. Notably, the coordinates are retrieved at evaluation-time through the context, as indeed obligation actions can use context-dependent arguments.

4.2 Semantics of the Policy Language

Before presenting the semantics, we introduce the key notion of *authorisation requests*. They are functions, ranged over by ρ, mapping structured names to policy values and are written as collections of pairs of the form (sn, pv). As an example, consider the following request:

$$\begin{aligned}\rho\ = \{&(subject/subject\text{-}id, \text{“}cmp\text{”}), (subject/attr, 4),\ \ldots\\ &(action/action\text{-}id, \text{“}act\text{”}),\ \ldots\\ &(object/resource\text{-}id, \text{“}res\text{”}), (object/attr, 3),\ \ldots\}\end{aligned}$$

Here, the subject identified by the string "*cmp*" requires the authorisation to execute the action "*act*" on the object identified by string "*res*". Notably, authorisation requests contain all attributes needed to evaluate them, forming the so-called evaluation context, including environmental properties.

The language semantics permits, given a policy π and a request ρ, to obtain a decision $\delta \in \{\mathsf{permit}, \mathsf{deny}, \mathsf{not\text{-}applicable}\}$ and a (possibly empty) sequence s of (fulfilled) obligation actions. This is expressed by the judgement $\pi, \rho \vdash \delta, s$. When we only consider permit and deny as resulting decisions, we use d instead of δ. The semantics is defined by the inference rules for the evaluation of policy elements, reported in Table 12, and of policy combining algorithms, reported in Table 13. In the next two subsections we comment the rules of the two tables, respectively.

Semantics of policies' elements. The inference rules of Table 12 are grouped according to the type of element they refer to. Thus, from top to bottom, we have the inference rules concerning policies, rules, targets and obligations. To save space, we do not show the inference rules of policy sets, as they are similar to those of atomic policies, and of some matching functions.

Some inference rules use the evaluation function $[\![\cdot]\!]_\rho$ that first replaces each attribute occurring in its argument expression with the corresponding value retrieved from the request ρ and then makes possible (boolean, integer, float, string, etc.) calculations. For example, given the request shown before, the arithmetic expression $subject/attr + object/attr$ is evaluated as follows

$$[\![subject/attr + object/attr]\!]_\rho = \rho(subject/attr) + \rho(object/attr) = 3 + 4 = 7$$

In case of occurrence of run-time errors, e.g. a function is not defined for arguments of a certain type, the expression evaluation halts and the policy evaluation does not complete.

The judgement $\pi, \rho \vdash \delta, s$ defines the semantics of policies and is inferred via the inference rules for targets, obligations and rules, combined together using the inference rules for the combining algorithms. Specifically, when a policy π is applied to a request ρ, first it is checked if the policy's target matches the request. If this is the case the evaluation proceeds by applying the policy's combining algorithm to the (sequence of) enclosed rules, thus obtaining a policy decision d and a sequence of obligation actions s. The decision d is then used to fulfill the sequence of policy's obligations thus obtaining a sequence s' of obligation actions that, in the resulting authorisation statement, is appended to s. If the target is empty then it matches any request. Finally, if the target does not match the request or the decision obtained by the combining algorithm is not-applicable, the policy does not apply and the decision not-applicable is returned together with an empty sequence of obligations. A policy set is evaluated like a policy, the only difference is that the combining algorithm is applied to a sequence of policies and/or policy sets, rather than rules.

When a rule r is applied to a request ρ, first it is checked if the rule's target matches the request. Additionally, in this case, a condition expression, if present, is evaluated. If its evaluation returns true, the rule applies, otherwise the deci-

POLICIES

$$\frac{\tau, \rho \vdash \mathsf{applicable} \qquad \alpha(r^+), \rho \vdash d, s \qquad o^*, \rho, d \vdash s'}{\langle \alpha \;\; \mathsf{target}: \tau \;\; \mathsf{rules}: r^+ \;\; \mathsf{obl}: o^* \rangle, \rho \vdash d, s.s'}$$

$$\frac{\alpha(r^+), \rho \vdash d, s \qquad o^*, \rho, d \vdash s'}{\langle \alpha \;\; \mathsf{rules}: r^+ \;\; \mathsf{obl}: o^* \rangle, \rho \vdash d, s.s'} \qquad \frac{\alpha(r^+), \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}{\langle \alpha \;\; \mathsf{rules}: r^+ \;\; \mathsf{obl}: o^* \rangle, \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}$$

$$\frac{\tau, \rho \vdash \mathsf{not\text{-}applicable} \quad \vee \quad (\tau, \rho \vdash \mathsf{applicable} \;\wedge\; \alpha(r^+), \rho \vdash \mathsf{not\text{-}applicable}, \epsilon)}{\langle \alpha \;\; \mathsf{target}: \tau \;\; \mathsf{rules}: r^+ \;\; \mathsf{obl}: o^* \rangle, \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}$$

RULES

$$\frac{\tau, \rho \vdash \mathsf{applicable} \qquad [\![\, be \,]\!]_\rho = \mathsf{true} \qquad o^*, \rho, d \vdash s}{(d \;\; \mathsf{target}: \tau \;\; \mathsf{condition}: be \;\; \mathsf{obl}: o^*), \rho \vdash d, s} \qquad \frac{\tau, \rho \vdash \mathsf{applicable} \qquad o^*, \rho, d \vdash s}{(d \;\; \mathsf{target}: \tau \;\; \mathsf{obl}: o^*), \rho \vdash d, s}$$

$$\frac{[\![\, be \,]\!]_\rho = \mathsf{true} \qquad o^*, \rho, d \vdash s}{(d \;\; \mathsf{condition}: be \;\; \mathsf{obl}: o^*), \rho \vdash d, s} \qquad \frac{o^*, \rho, d \vdash s}{(d \;\; \mathsf{obl}: o^*), \rho \vdash d, s}$$

$$\frac{\tau, \rho \vdash \mathsf{not\text{-}applicable}}{(d \;\; \mathsf{target}: \tau \;\; \mathsf{condition}: be^? \;\; \mathsf{obl}: o^*), \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}$$

$$\frac{[\![\, be \,]\!]_\rho = \mathsf{false}}{(d \;\; \mathsf{condition}: be \;\; \mathsf{obl}: o^*), \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}$$

$$\frac{\tau, \rho \vdash \mathsf{applicable} \qquad [\![\, be \,]\!]_\rho = \mathsf{false}}{(d \;\; \mathsf{target}: \tau \;\; \mathsf{condition}: be \;\; \mathsf{obl}: o^*), \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}$$

TARGETS

$$\frac{\tau_1, \rho \vdash \mathsf{applicable} \quad \vee \quad \tau_2, \rho \vdash \mathsf{applicable}}{(\tau_1 \;\vee\; \tau_2), \rho \vdash \mathsf{applicable}} \qquad \frac{\tau_1, \rho \vdash \mathsf{applicable} \quad \tau_2, \rho \vdash \mathsf{applicable}}{(\tau_1 \;\wedge\; \tau_2), \rho \vdash \mathsf{applicable}}$$

$$\frac{\tau_1, \rho \vdash \mathsf{not\text{-}applicable} \quad \tau_2, \rho \vdash \mathsf{not\text{-}applicable}}{(\tau_1 \;\vee\; \tau_2), \rho \vdash \mathsf{not\text{-}applicable}} \qquad \frac{\tau_1, \rho \vdash \mathsf{not\text{-}applicable} \;\vee\; \tau_2, \rho \vdash \mathsf{not\text{-}applicable}}{(\tau_1 \;\wedge\; \tau_2), \rho \vdash \mathsf{not\text{-}applicable}}$$

$$\frac{f([\![\, pv \,]\!]_\rho, \rho(sn)) \vdash \mathsf{true}}{f(pv{,}sn), \rho \vdash \mathsf{applicable}} \qquad \frac{sn \notin dom(\rho) \quad \vee \quad f([\![\, pv \,]\!]_\rho, \rho(sn)) \vdash \mathsf{false}}{f(pv{,}sn), \rho \vdash \mathsf{not\text{-}applicable}}$$

$$\frac{pv > pv'}{\mathsf{greater\text{-}than}(pv{,}pv) \vdash \mathsf{true}} \qquad \frac{pv \not> pv'}{\mathsf{greater\text{-}than}(pv{,}pv') \vdash \mathsf{false}}$$

...

OBLIGATIONS

$$\frac{}{\epsilon, \rho, d \vdash \epsilon} \qquad \frac{d' = d \qquad [\![\, s \,]\!]_\rho = s' \qquad o^*, \rho, d \vdash s''}{[\; d' \;\; s \;]\, o^*, \rho, d \vdash s'.s''} \qquad \frac{d' \neq d \qquad o^*, \rho, d \vdash s'}{[\; d' \;\; s \;]\, o^*, \rho, d \vdash s'}$$

Table 12. Semantics of policies' elements

sion not-applicable is returned. If the condition expression is absent, then it is considered true. When the rule's target matches the request and the condition expression returns true, the rule's effect, i.e. permit or deny, is returned, together with the sequence of obligation actions resulting from fulfilling the sequence of rule's obligations.

A target τ matches a request ρ, i.e. its evaluation returns applicable, if the combination of its atomic targets matches ρ. A composed target of the form $(\tau_1 \ \vee \ \tau_2)$ matches ρ, if one between τ_1 and τ_2 matches the request, while in case of target $(\tau_1 \ \wedge \ \tau_2)$, both τ_1 and τ_2 must match the request. An atomic target of the form $f(pv,sn)$ matches a request ρ if the matching function f, applied to (the evaluation of) the policy value pv and the value identified by the structured name sn in the request, i.e. $\rho(sn)$, returns true. Before applying the matching function, the policy value must be evaluated, since it may be a template containing expressions. This is not necessary for the structured name; indeed, although it may be an item, it will not contain expressions since it has been retrieved from the request. Instead, the evaluation of an atomic target returns not-applicable either if the structured name does not identify any value in the request, i.e. it does not belong to the request's domain ($sn \notin dom(\rho)$), or if the matching function returns false. The rules for matching functions are straightforward. For example, greater-than(pv,pv') returns true only when pv is greater than pv'.

The last three rules account for fulfilment of a sequence of policy's obligations when the decision for the enclosing element is d, i.e. permit or deny. If the sequence is empty, then an empty sequence of obligation actions is returned. Otherwise, the obligations in the sequence are fulfilled sequentially and the resulting sequences of obligation actions are linked together preserving the same order. The fulfilment of an obligation with attached effect equal to the decision d of the enclosing element consists in evaluating all argument expressions by using function $[\![\cdot]\!]_\rho$. Instead, if the attached effect differs from d, the last rule permits to continue the fulfilment while ignoring the current obligation.

Semantics of policy combining algorithms. The rules of Table 13 rewrite formally the combining algorithms descriptions presented in Section 4.1. For each algorithm, we have separate, but quite similar, rules that deal with the case that the algorithm is applied to a sequence of rules or of policies. Therefore, to save space, we only report and comment the first type of rules and, for dual algorithms (e.g. permit-overrides and deny-overrides), we avoid to report both set of rules.

In the inference rules, given a non-empty sequence r^+ of rules, notation $\exists\ r, \rho \vdash \delta, s$ ($\nexists\ r, \rho \vdash \delta, s$, resp.) means that there is (not, resp.) a rule in the sequence satisfying the judgement. The variants with universal quantifier or where the index of rules is explicitly considered have a similar meaning.

The rules for the algorithm permit-overrides (resp. deny-overrides) are straightforward: if there is a rule in the sequence to which the algorithm is applied returning decision permit (resp. deny) then the algorithm returns permit; instead, if all rules in the sequence are not-applicable, then the algorithm returns

PERMIT-OVERRIDES

$$\frac{\exists\ r, \rho \vdash \mathsf{permit}, s}{\mathsf{permit\text{-}overrides}(r^+), \rho \vdash \mathsf{permit}, s} \qquad \frac{\nexists\ r', \rho \vdash \mathsf{permit}, s' \qquad \exists\ r, \rho \vdash \mathsf{deny}, s}{\mathsf{permit\text{-}overrides}(r^+), \rho \vdash \mathsf{deny}, s}$$

$$\frac{\forall\ r, \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}{\mathsf{permit\text{-}overrides}(r^+), \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}$$

DENY-UNLESS-PERMIT

$$\frac{\exists\ r, \rho \vdash \mathsf{permit}, s}{\mathsf{deny\text{-}unless\text{-}permit}(r^+), \rho \vdash \mathsf{permit}, s} \qquad \frac{\nexists\ r', \rho \vdash \mathsf{permit}, s' \qquad \exists\ r, \rho \vdash \mathsf{deny}, s}{\mathsf{deny\text{-}unless\text{-}permit}(r^+), \rho \vdash \mathsf{deny}, s}$$

$$\frac{\forall\ r, \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}{\mathsf{deny\text{-}unless\text{-}permit}(r^+), \rho \vdash \mathsf{deny}, \epsilon}$$

FIRST-APPLICABLE $\quad i \in \{1, 2, \ldots, |r|\}$

$$\frac{r_i, \rho \vdash d, s \qquad \forall\ 1 \leq j < i : r_j, \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}{\mathsf{first\text{-}applicable}(r^+), \rho \vdash d, s}$$

$$\frac{\forall\ i : r_i, \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}{\mathsf{first\text{-}applicable}(r^+), \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}$$

ONLY-ONE-APPLICABLE $\quad i \in \{1, 2, \ldots, |r|\}$

$$\frac{r_i, \rho \vdash d, s \qquad \forall\ j \neq i : r_j, \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}{\mathsf{only\text{-}one\text{-}applicable}(r^+), \rho \vdash d, s}$$

$$\frac{\exists\ i, j : \quad i \neq j \quad \wedge \quad r_i, \rho \vdash d_i, s_i \quad \wedge \quad r_j, \rho \vdash d_j, s_j}{\mathsf{only\text{-}one\text{-}applicable}(r^+), \rho \vdash \mathsf{not\text{-}applicable}, \epsilon}$$

Table 13. Semantics of policy combining algorithms

not-applicable; otherwise, i.e. no rule returns permit and there is at least one rule returning deny (resp. permit), then the algorithm returns deny.

The rules for the algorithm deny-unless-permit (resp. permit-unless-deny) are similar to those for permit-overrides (resp. deny-overrides) but in this case the decision not-applicable is never returned. Thus, the algorithm returns permit (resp. deny) if there is a rule in the sequence returning it; otherwise, it returns deny (resp. permit).

In the previous inference rules, the sequence of obligation actions returned by an algorithm together with a decision d is any of those returned by one of the rules, in the sequence to which the algorithm is applied, returning the same decision d (if there is no such rule, then it is the empty sequence). However, no assumption is made about the evaluation order of the rules in the sequence. Thus, when more rules return permit or deny, the returned sequence of obligation actions is somehow nondeterministically chosen. Namely, id the execution halts

$\dfrac{match(T,t) = \sigma}{\mathsf{pattern\text{-}match}(T,t) \vdash \mathsf{true}}$	$\dfrac{\nexists \sigma : match(T,t) = \sigma}{\mathsf{pattern\text{-}match}(T,t) \vdash \mathsf{false}}$

Table 14. pattern-match function

at the first permit or deny result, the resulting obligation sequence s is composed by only the actions returned together with such a result.

Differently from the previous cases, the inference rules for the algorithm first-applicable ensure that the rules to which the algorithm is applied are evaluated sequentially. The decision returned is then the first one differing from not-applicable, if any, or not-applicable, otherwise.

Finally, the rules for the algorithm only-one-applicable check if only one of the rules to which the algorithm is applied returns a decision differing from not-applicable: if this is the case, then such decision is returned (together with the associated sequence of obligation actions), otherwise not-applicable is returned.

To sum up, policies, and their evaluation, are hierarchically structured as trees: the evaluation of leaf nodes, i.e. rules, return a 'starting' decision, permit, deny or not-applicable, while the intermediate nodes, i.e. policies, combine the decisions and obligations returned by the evaluation of their child nodes through the chosen combining algorithm. Policy evaluation terminates when the root is reached producing a decision and a sequence of obligations. This sequence consists of fulfilled actions that will enforce the consequences resulting from the authorisation process.

5 A Full-fledged SCEL Instance

In this section, we present a full instantiation of the SCEL language, called PSCEL (Policed SCEL). PSCEL uses as knowledge the tuple spaces presented in Section 3.1 and as policy language the version of FACPL presented in Section 4.1. Therefore, each PSCEL component has its own tuple repository and a collection of policies controlling the behaviour of such a component and consequently the interactions with others.

Section 5 introduces the formal integration of FACPL with SCEL by outlining the integration steps which must be followed in order to obtain fully-interoperable abstractions. Then, Section 5.2 shows PSCEL at work on the considered swarm robotics scenario.

5.1 PSCEL: Policed SCEL

We present the syntax refinement, followed by the formal integration in the SCEL operational semantics.

Syntax. FACPL policies are specialised by instantiating the obligation actions as the set of SCEL actions reported in Table 1, and by defining the matching function pattern-match, which aims at comparing knowledge values with policy ones. In particular, this function defines, by using the pattern-matching mechanism presented in Section 3.1, the matching of a template with a knowledge item. The formal rules for the new comparison function are reported in Table 14.

To explicitly represent the fact that the policies in force at any given component can dynamically change while the component evolves, we use a sort of automata somehow reminiscent of *security automata* [49]. Thus, a POLICY AUTOMATON Π is a pair $\langle\, A,\, \pi \,\rangle$, where

- A is an automaton of the form $\langle \mathit{Policies}, \mathit{Targets}, \mathcal{T} \rangle$ where the set of states *Policies* contains all the policies that can be in force at different times, the set of labels *Targets* contains the security relevant events (expressed as the TARGETS in Table 11) that can trigger policy modification and the set of transitions $\mathcal{T} \subseteq (\mathit{Policies} \times \mathit{Targets} \times \mathit{Policies})$ represents policy replacement.
- $\pi \in \mathit{Policies}$ is the current state of A.

Dynamically changing policies is a powerful mechanism that permits controlling, in a natural and clear way, the evolution of adaptive systems having a very high degree of dynamism, which in principle would be quite difficult to manage. In Section 5.2 a full application of such an automaton is provided.

Semantics. PSCEL specialises the SCEL operational semantics by connecting the evaluation of the authorization predicate with the inference rules of Tables 12 and 13. More specifically, the authorization predicate in PSCEL also considers the outcome of policy evaluation; this is an authorization decision δ and a sequence of actions s. The authorization predicate takes the form $\Pi \vdash^{\delta}_{s} \lambda, \Pi'$ meaning that the action generating the label λ is evaluated with respect to the policy automaton Π to a decision δ (i.e., permit, deny or not-applicable), along with a (possibly empty) sequence s of actions to perform, and a (possibly adapted) policy automaton Π' to enforce. To calculate an authorization decision, we need first to generate a request ρ from label λ and then to evaluate it with respect to the current policy state π of Π.

The authorization request is produced on demand when an action that needs to be authorised is going to be performed. The production is done by function $\lambda 2\rho(\cdot)$ that maps (a subset of) the SCEL labels to requests and is defined as

follows:

$$\begin{aligned}
\lambda 2\rho(\mathcal{I} : \mathbf{fresh}(n)) &= \{(subject/attr, val) \mid (attr, val) \in \mathcal{I}\} \\
&\quad \cup \{(action/action\text{-}id, \mathbf{fresh})\} \\
&\quad \cup \{(object/attr, val) \mid (attr, val) \in \mathcal{I}\} \\
\lambda 2\rho(\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)) &= \{(subject/attr, val) \mid (attr, val) \in \mathcal{I}\} \\
&\quad \cup \{(action/action\text{-}id, \mathbf{new})\} \\
&\quad \cup \{(object/attr, val) \mid (attr, val) \in \mathcal{J}\} \\
\lambda 2\rho(\mathcal{I} : t \bar{\triangleright} \mathcal{J}) &= \{(subject/attr, val) \mid (attr, val) \in \mathcal{I}\} \\
&\quad \cup \{(action/item, t), (action/action\text{-}id, \mathbf{put})\} \\
&\quad \cup \{(object/attr, val) \mid (attr, val) \in \mathcal{J}\} \\
\lambda 2\rho(\mathcal{I} : t \bar{\triangleleft} \mathcal{J}) &= \{(subject/attr, val) \mid (attr, val) \in \mathcal{I}\} \\
&\quad \cup \{(action/item, t), (action/action\text{-}id, \mathbf{get})\} \\
&\quad \cup \{(object/attr, val) \mid (attr, val) \in \mathcal{J}\} \\
\lambda 2\rho(\mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J}) &= \{(subject/attr, val) \mid (attr, val) \in \mathcal{I}\} \\
&\quad \cup \{(action/item, t), (action/action\text{-}id, \mathbf{qry})\} \\
&\quad \cup \{(object/attr, val) \mid (attr, val) \in \mathcal{J}\}
\end{aligned}$$

Each value for subject and object, retrieved from $\mathcal{I}$ and $\mathcal{J}$, respectively, is bound to an attribute identifier, e.g. the identifier of the subject component is bound to *subject/subject-id*. Notably, the *item* attribute identifies the exchanged item in a communication action, thus it is undefined in the case of **fresh** and **new**.

Finally, if we let $\Pi \triangleq \langle A, \pi \rangle$, $\Pi'' \triangleq \langle A, \pi' \rangle$ and $\rho \triangleq \lambda 2\rho(\lambda)$, the authorization predicate can be formally defined in terms of the semantics of policies by the following rule:

$$\frac{\pi, \rho \vdash \delta, s \quad \Pi' = \begin{cases} \Pi'' & \textit{if } \langle \pi, \tau, \pi' \rangle \in A \ \wedge \ \tau, \rho \vdash \mathsf{applicable} \\ \Pi & \textit{otherwise} \end{cases}}{\Pi \vdash_s^\delta \lambda, \Pi'}$$

Intuitively, an action λ is allowed if the corresponding request ρ satisfies $\pi, \rho \vdash \mathsf{permit}, s$; moreover, if for some target $\tau \in Targets$, such that $\tau, \rho \vdash \mathsf{applicable}$, the automaton A has a transition $\langle \pi, \tau, \pi' \rangle$, then the state of A after the request evaluation becomes π'. On the other hand, if we get $\pi, \rho \vdash \mathsf{deny}, s$, then the action is disallowed but, as a consequence of evaluation of the authorization predicate, and similarly to the previous case, the policy in force within the component can change. Notably, the current policy in Π does not change unless there is a target τ matching the request ρ and producing a transition in the policy automaton. Of course, if the automaton has a single state or an empty set of transitions, the policy in force at a component never changes.

The refinement of the authorization predicate forces a slight modification of the SCEL operational semantics, for appropriately dealing with the authorization decisions permit and deny, and the discharge of obligation actions. For the sake of simplicity, the PSCEL operational semantics does not take into account the decision not-applicable. This means that the rules do not explicitly deal with situations where none of the policies is applicable to a given request,

or the combining algorithms do not convert not-applicable into permit or deny. These situations are handled as runtime errors that induce the executing process to get stuck. They could be easily avoided by using an appropriate combining algorithm, as e.g. permit-unless-deny, at top level of each state of the policy automaton Π.

PSCEL operational rules are similar to those presented in Section 2.3, therefore in Table 15 we only report some significant ones[11]. As a matter of notation, $\mathcal{I}.p$ indicates the process part of component $\mathcal{I}$ and, when $s = \epsilon$, $s.P$ stands for P. Notably, all labels taken as argument by the authorization predicate, i.e. $\mathcal{I} : \mathbf{fresh}(n)$, $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$, $\mathcal{I} : t \triangleleft \mathcal{J}$, $\mathcal{I} : t \blacktriangleleft \mathcal{J}$ and $\mathcal{I} : t \triangleright \mathcal{J}$, have a counterpart corresponding to decision deny which is obtained by application of functional notation $\oslash(\cdot)$ to the label. Thus, label $\oslash(\mathcal{I} : t \triangleleft \mathcal{J})$ indicates that action **get** is denied. Some comments on the rules in Table 15 follow.

The interaction predicate, used in rule *(pr-sys)*, is instantiated by the interleaving interaction predicate of Table 3. Moreover, the rule here is tailored for taking into account the discharge of the obligation actions generated by the inference. To this aim, the component obtained after the transition contains the placeholder $*$ in place of the process part; it will be replaced by the continuation of process P, possibly prefixed by a sequence of obligations, during the rest of the derivation (see, e.g., the use of $[s.(\mathcal{I}.p)/*]$ in the rule *(ptpget-p-p)*). Notably, this mechanism permits to apply the substitution σ also to the obligation actions, so that the variables possibly contained get instantiated.

The rules denoting the willingness of components to accept the execution of an action operating on their local repository are now split in two rules: one rule, e.g. *(accget-p)*, corresponding to the fact that the action is authorised, and one rule, e.g. *(accget-d)*, corresponding to the the fact the action is denied. In the former rule, the label of the transition is updated with the new policy (as in the SCEL corresponding rule) and with the the obligation actions s that have to be performed before the continuation process. In the latter rule, the transition label is updated similarly, although the action on the repository, i.e. the withdrawing of item t, is not performed.

When considering the transitions that a single component can perform, the main difference is that we have one rule, e.g. *(lget-p)*, for the case the action is allowed by the authorization predicate, and one rule, e.g. *(lget-d)*, for the case it is denied. Notably, in this latter case, even though the action is not performed (indeed the last premise of *(lget-d)* starts from C and not from C'), the new policies and the obligation actions produced by evaluation of the authorisation predicate are installed (indeed, in the conclusion of the rule, C evolves to C''); they in fact may adapt the system to allow a subsequent successful execution of the action or to enable an alternative execution path.

Similarly, in case of synchronisation between two components, for each different type of action we have four cases to consider, corresponding to the pairs consisting of the values permit and deny. For example, the action **get** can with-

[11] We refer the interested reader to the technical report [37] for a complete account of the operational rules.

$$\frac{P \downarrow_\alpha P' \quad \alpha = \mathcal{I} : \mathbf{fresh}(n) \Rightarrow n \notin \mathrm{n}(\mathcal{I}[\mathcal{K}, \Pi, \mathbf{nil}]) \quad \Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\lambda[\Pi'/\mathcal{I}.\pi\,,\, P'/\mathcal{I}.p]} \mathcal{I}[\mathcal{K}, \bullet, *\sigma]} \textit{ (pr-sys)}$$

$$\frac{\Pi \vdash^{\mathsf{p}}_s \mathcal{I} : t \triangleleft \mathcal{J}, \Pi' \quad \mathcal{K} \ominus t = \mathcal{K}'}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \triangleleft \mathcal{J}[\Pi'/\mathcal{J}.\pi\,,\, s.P/\mathcal{J}.p]} \mathcal{J}[\mathcal{K}', \Pi', s.P]} \textit{ (accget-p)}$$

$$\frac{\Pi \vdash^{\mathsf{d}}_s \mathcal{I} : t \triangleleft \mathcal{J}, \Pi'}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\oslash(\mathcal{I}:t \triangleleft \mathcal{J}[\Pi'/\mathcal{J}.\pi\,,\, s.P/\mathcal{J}.p])} \mathcal{J}[\mathcal{K}, \Pi', s.P]} \textit{ (accget-d)}$$

$$\frac{C \xrightarrow{\mathcal{I}:t \lhd n} C' \quad n = \mathcal{I}.id \quad C'[\mathcal{I}.\pi/\bullet, \mathcal{I}.p/*] \xrightarrow{\mathcal{I}:t \triangleleft \mathcal{I}} C''}{C \xrightarrow{\tau} C''} \textit{ (lget-p)}$$

$$\frac{C \xrightarrow{\mathcal{I}:t \lhd n} C' \quad n = \mathcal{I}.id \quad C \xrightarrow{\oslash(\mathcal{I}:t \triangleleft \mathcal{I})} C''}{C \xrightarrow{\tau} C''} \textit{ (lget-d)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \lhd n} S_1' \quad S_2 \xrightarrow{\mathcal{I}:t \triangleleft \mathcal{J}} S_2' \quad \mathcal{J}.id = n \quad \mathcal{I}.\pi \vdash^{\mathsf{p}}_s \mathcal{I} : t \triangleleft \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1'[\Pi'/\bullet, s.(\mathcal{I}.p)/*] \parallel S_2'} \textit{ (ptpget-p-p)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \lhd n} S_1' \quad S_2 \xrightarrow{\oslash(\mathcal{I}:t \triangleleft \mathcal{J})} S_2' \quad \mathcal{J}.id = n \quad \mathcal{I}.\pi \vdash^{\mathsf{p}}_s \mathcal{I} : t \triangleleft \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1 \parallel S_2'} \textit{ (ptpget-p-d)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \lhd n} S_1' \quad S_2 \xrightarrow{\mathcal{I}:t \triangleleft \mathcal{J}} S_2' \quad \mathcal{J}.id = n \quad \mathcal{I}.\pi \vdash^{\mathsf{d}}_s \mathcal{I} : t \triangleleft \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1[\Pi'/\bullet, s.(\mathcal{I}.p)/*] \parallel S_2} \textit{ (ptpget-d-p)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \lhd n} S_1' \quad S_2 \xrightarrow{\oslash(\mathcal{I}:t \triangleleft \mathcal{J})} S_2' \quad \mathcal{J}.id = n \quad \mathcal{I}.\pi \vdash^{\mathsf{d}}_s \mathcal{I} : t \triangleleft \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1[\Pi'/\bullet, s.(\mathcal{I}.p)/*] \parallel S_2} \textit{ (ptpget-d-d)}$$

Table 15. PSCEL (excerpt of) operational semantics (p stands for permit in $\vdash^{\mathsf{p}}_s$, while d stands for deny in $\vdash^{\mathsf{d}}_s$)

draw an item from a specific repository with a point to point access according to the rule *(ptpget-p-p)* of Table 15. The label $\mathcal{I} : t \triangleleft \mathcal{J}$, generated by rule *(accget-p)*, denotes the willingness of component $\mathcal{J}$ to provide the item t to component $\mathcal{I}$. The label is generated only if such willingness is authorised by the authorization predicate in force at component $\mathcal{J}$ and if withdrawing an item t from the repository of $\mathcal{J}$ is possible ($\mathcal{K} \ominus t = \mathcal{K}'$). The target component $\mathcal{J}$ is modified by removing t from the repository and by installing the policy and the sequence of obligation actions produced by the evaluation of the authorization predicate. Thus, when the target of the action denotes a specific remote repository *(ptpget-p-*

p), 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 authorised by the policy at the source component $\mathcal{I}$ (identified by notation $\mathcal{I}.\pi$). The authorization to perform the action could be denied by the local policy (rule *(ptpget-d-p)* for remote ones) or by the policy of the target component (rules *(accget-d)* and *(ptpget-p-d)*) or by both policies (rule *(ptpget-d-d)*). Note that the policy and the sequence of obligation actions produced by evaluation of the authorisation predicate are always installed on the source component, except when the action is authorised by the local policy but not by the policy of the target component, i.e. as in the case of rule *(ptpget-p-d)*. In the target component, the installation occurs only when the action has been authorised by the source component.

The rules for group-oriented communication rely on the same basic ideas described above (i.e., there are four rules for each kind of action). Thus, due to space limitations, they are not reported here.

5.2 PSCEL at Work

We show here the effectiveness of the PSCEL approach by providing a complete model of the robot swarm scenario used as a running example in the previous sections and informally presented in Section 1. Notably, this model exploits the fact that a process, which represents the behaviour of a robot, can read tuples produced by sensors, e.g. the tuple $\langle$"*collision*", $\mathsf{true}\rangle$ indicating that an imminent collision with a wall of the arena has been detected, and can add tuples that trigger the activation of actuators, e.g. the tuple $\langle$"*goTo*", $4.34, 3.25\rangle$ forcing the robot to reach a specific position. Therefore, as these tuples are produced (resp., consumed) by sensors (resp., actuators), no additional data/assumptions on the initial state are needed. It is also worth noticing that sensors and actuators are not explicitly modelled in PSCEL, as they are robot's internal devices while the PSCEL model represents the programmable behaviour of the robot, i.e. its running code. We clarify the practical role of sensors and actuators in Section 6.

The scenario is modelled as a set of components $(\textsc{Robot}_1 \parallel \ldots \parallel \textsc{Robot}_n)$ where each $\textsc{Robot}_i$ has the form $\mathcal{I}_{R_i}[\mathcal{K}_{R_i}, \Pi_R, P_R]$. The behaviour of a single robot corresponds to the following PSCEL process

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

A robot follows a random walk to explore the disaster area. To this aim, the process *RandomWalk* randomly selects a direction that is followed until either a wall is hit or a *stop* signal is sent to the wheels actuator. The robot recognises the presence of a victim by means of the **qry** action, while it helps other robots to rescue a victim by means of the **get** action and according to the *HelpingRescuer* process definition. When a victim is found, information about his position

(retrieved by the attributes x and y of the robot's interface) and the number of other robots needed for rescuing him (3 robots in our case, but a solution with a varying number can be easily accommodated) is locally published. Then, the tuple $\langle$"*rescue*"$\rangle$ is locally inserted to start the rescuing procedure.

The *RandomWalk* process calculates the random direction followed by the robot to explore the arena. The robot starts moving as soon as the first direction is calculated. When the proximity sensor signals a possible collision, by means of the tuple $\langle$"*collision*", $\mathsf{true}\rangle$, a new random direction is calculated. This behaviour corresponds to the following PSCEL process

$$\begin{aligned} \mathit{RandomWalk} \triangleq\ & \mathbf{put}(\text{"}\mathit{direction}\text{"}, 2\pi\mathsf{rand}())@\mathsf{self}. \\ & \mathbf{qry}(\text{"}\mathit{collision}\text{"}, \mathsf{true})@\mathsf{self}.\mathit{RandomWalk} \end{aligned}$$

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

The *HelpingRescuer* process is defined as follows

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

This process is triggered by a *victim* tuple retrieved from the rescuers ensemble (see P_R). The tuple indicates that additional robots (whose number is stored in *count*) are needed at position (x_v, y_v) to rescue a victim. If more than one robot is needed, a new *victim* tuple is published (with decremented counter). Then, the robot, which becomes a helper of the rescuer, goes towards the victim position. Once it reaches him (i.e., its current position coincides with the victim's one), it becomes a rescuer and starts the rescuing procedure. It is worth noting that, if more victims are in the scenario, different groups of rescuers will be spontaneously organised to rescue them. To avoid that more than one group is formed for the same victim, we assume that the sensor used to perceive the victims is configured so that a victim that is already receiving assistance by some rescuers is not detected as a victim by further robots. This assumption is also feasible in a real scenario, where a light-based message communication among robots can be used [40]. Thus, once a robot has reached the victim, by using a specific light color, it signals not to "discover" the victim next to it (see Chapter IV.2 [42]).

Notably, the effectiveness of this approach relies on the assumption that robots cannot fail. In fact, when a robot that knows the victim's position fails, it cannot be ensured that such position is correctly communicated. Specific handling can be used in such a case, e.g. by enabling the perception of a victim if the robots already taking care of the victim are not active.

Finally, in order to check the level of the battery during the exploration, and possibly halting the robot when the battery is low, we need to capture the movement status. This information is represented by the tuple $\langle$"*isMoving*"$\rangle$, which is produced by the wheels sensor, and monitored by the following process

$$\mathit{IsMoving} \triangleq \mathbf{qry}(\text{"}\mathit{isMoving}\text{"})@\mathsf{self}.\mathit{IsMoving}$$

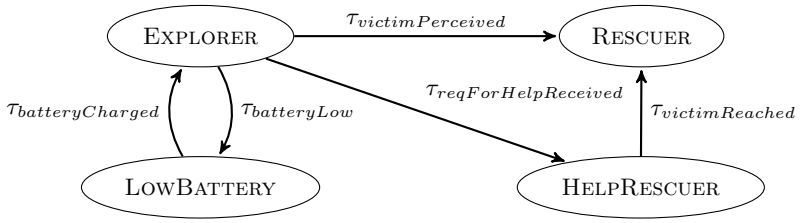

Fig. 4. Swarm robotics scenario: policy automaton

Condition	PSCEL *action*	*Additional Constraint*
$\tau_{victimPerceived}$	**qry**("*victimPerceived*", true)@self	-
$\tau_{victimReached}$	**qry**("*position*",_,_)@self	-
$\tau_{reqForHelpReceived}$	**get**("*victim*",_,_,_)@(*role*="*rescuer*" $\vee \ldots$)	-
$\tau_{batteryLow}$	**qry**("*isMoving*")@self	*subject*/*batteryLevel* $< 20\%$
$\tau_{batteryCharged}$	**qry**("*charged*")@self	-

Table 16. Conditions of the Policy Automaton Transitions

The reading of this datum is exploited by a rule of the authorisation policy.

Each robot dynamically adapts its role, as well as the enforced policies, according to external conditions and stimuli. Thus, each role corresponds to a different enforced policy. The transitions triggering the policy changes are defined by the policy automaton shown in Figure 4.

Before presenting the policies of some of the automaton states, we briefly outline the conditions of the policy automaton transitions, whose details are reported in Table 16. These transitions, which mimic the role changing previously described, define the conditions on the action the process wants to execute, and (if needed) some additional constraints on environmental values. For instance, the Explorer state evolves to the Rescuer one when the condition $\tau_{victimPerceived}$ holds, that is as soon as the perception of a victim has to be authorised (i.e., the action **qry**("*victimPerceived*", true)@self can complete). To move from Explorer to LowBattery, it is required that the robot is moving (i.e., the action **qry**("*isMoving*")@self can complete), and the battery level is less then 20%. All the other conditions are defined in the same way.

In the EXPLORER state, to stop the robot as soon as a victim is perceived and to diagnose a critical level of the battery, we define the following policy

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

The first positive rule has the only purpose of returning the obligation action $\mathbf{put}(\text{“}\mathit{stop}\text{”})@\mathsf{self}$ when the corresponding **qry** is executed. This obligation instructs the wheels actuator to stop the movement. The negative rule checks the battery level of the robot, and when the level is critical (i.e., lower then or equal to 20%), the obligation $\mathbf{put}(\text{“}\mathit{goTo}\text{”}, \mathit{env/station.x}, \mathit{env/station.y})@\mathsf{self}$ is returned in order to change the robot direction. Notably, the position of the charging station is provided by the evaluation context during the obligation fulfilment.

The policy enforced in the HELPRESCUER state is defined similarly: it controls the robot movement towards the previously received victim's position. In particular, the policy halts the robot as soon as the victim's position is reached and forbids unexpected direction changes during the movement.

In the case of LOWBATTERY state, we define instead the following policy

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

When the position of the charging station is reached, the first rule halts the movement and returns the actions needed for enacting the charging behaviour. In particular, the battery charging process is started by the $\mathbf{put}(\text{“}\mathit{charge}\text{”})@\mathsf{self}$ action, while the $\mathbf{qry}(\text{“}\mathit{charged}\text{”})@\mathsf{self}$ blocks the robot until the end of the charging process. Note that the transition condition $\tau_{\mathit{batteryCharged}}$ holds when the latter action requests for authorisation, and therefore in the continuation the robot will play the EXPLORER state.

Finally, the policy enforced in the RESCUER state is as follows

$$
\begin{array}{l}
\langle\, \mathsf{permit\text{-}unless\text{-}deny} \\
\quad \mathsf{rules} : (\mathsf{permit}\ \ \mathsf{target} : \mathsf{equal}(\mathbf{put}, \mathit{action/action\text{-}id}) \\
\qquad\qquad\qquad\qquad \wedge\ \mathsf{pattern\text{-}match}((\text{“}\mathit{rescue}\text{”}), \mathit{action/item})) \\
\qquad\qquad\qquad\quad \wedge\ \mathsf{less\text{-}than}(40\%, \mathit{subject/batteryLevel}) \\
\qquad\qquad\quad \mathsf{obl} : [\,\mathsf{permit}\ \ \mathbf{put}(\text{“}\mathit{camera}\text{”}, \text{“}\mathit{on}\text{”})@\mathsf{self}\,]\,)\,\rangle
\end{array}
$$

This policy does not forbid any actions, it is only used for turning on the robot's camera if there is enough battery; other functionalities could be activated as well.

As shown in this section, the design of a PSCEL specification involves processes, policies, and obligations. In order to decide which design approach, e.g. defining a process action or an obligation, is more appropriate, we can follow the *separation of concerns* principle. According to it, we decouple the functional aspects of the components behaviour from the adaptation ones. Thus, the application logic generating the computational behaviour of components is defined in a procedural style, in the form of processes, while the adaptation logic is defined in a declarative style, in the form of policies enclosing obligations. Processes and policies have indeed different features:

- a process contains the actions that *should* be executed. Thus, when an action is not authorised, the process is blocked until a positive authorisation for such action is received;
- a policy decides whether to authorise a process action and can force processes to perform additional actions which can depend on contextual information or on the authorisation of remote actions.

Of course, a process could decide by itself whether to execute an action or not, without resorting to a policy. However, this would lead to a specification where application and adaptation logics are mixed up, which is more difficult to develop and maintain. Moreover, it would require, at least, some additional efforts to introduce: (i) conditional choices for checking contextual information; (ii) actions monitoring the knowledge items for, e.g., discovering when a remote **get** is performed. These additional tasks significantly affect the burden of specifying a process. Indeed, the use of policies and obligations is advocated not only to decide the authorisation of process actions but also to define actions that are not executed all the times. Furthermore, by means of the policy automaton, policies can dynamically change to react to external conditions, while processes cannot. On the other hand, policy evaluation is triggered by a process action, therefore additional demon processes, such as the *isMoving* process, could be needed.

6 A Runtime Environment for SCEL

In this section we present jRESP[12], 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. The close correspondence between the two languages enhances confidence on the

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

behaviour of the jRESP implementation of SCEL programs, once the latter have been analysed via formal methods, which is possible given there is a formal operational semantics.

The SCEL language, as explained in Section 2, is parametric with respect to some aspects, e.g. knowledge representation and policy language, that may be tailored to better fit 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, the jRESP communication infrastructure has been designed to avoid *centralized control*. Moreover, to facilitate interoperability with other tools and programming frameworks, jRESP relies on JSON[13]. 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.

The overall environment and the programming constructs are presented in Section 6.1, while the integration of FACPL is detailed in Section 6.2. Finally, Section 6.3 reports the jRESP implementation of the robot swarm scenario.

6.1 Programming Constructs

Components. SCEL components are implemented via the class Node. The architecture of a node is shown in Figure 5. 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 (whose implementation is described in the *Network Infrastructure* paragraph below).

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 a piece of knowledge from/to a repository. Currently, a single implementation of the Knowledge interface is available in jRESP, which relies on the notion of tuple space presented in Section 3.1.

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

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

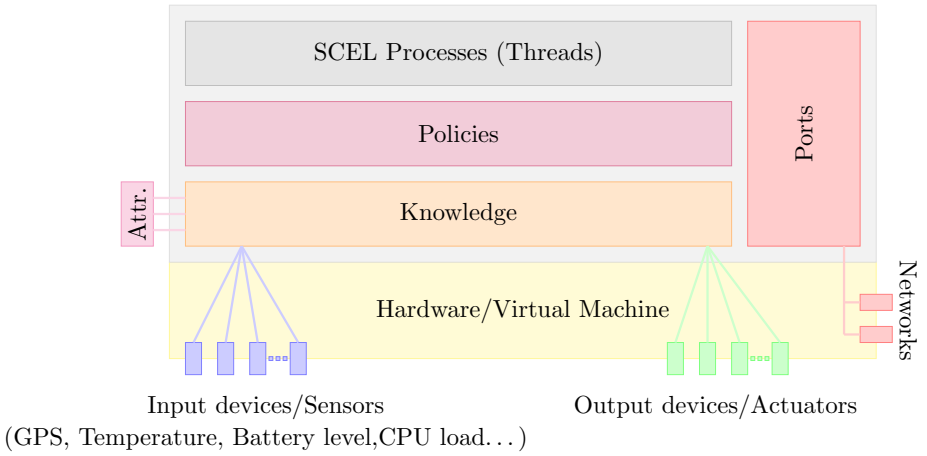

Fig. 5. Node architecture

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 one of these collectors 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[14].

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 as to use different underlying network infrastructures (e.g., Internet, Ad-hoc networks, . . .).

[14] This mechanism is implemented via the *Observer*/*Observable* pattern.

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[15] [14] and FreePastry[16] [46]. A more centralized implementation is provided by ServerPort. All messages sent along this kind of port pass through a centralized server that dispatches all the received messages to each of the managed ports. Finally, VirtualPort implements a port where interactions are performed via a buffer stored in memory. A VitualPort is used to *simulate* nodes in a single application without relying on a specific network infrastructure.

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 provided by 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. In jRESP, like in SCEL, policies can be used to authorise local actions and to regulate the interactions among components. When a method of an instance of class Agent is invoked, its execution is delegated to the policy in force at the node where the agent is running. The policy can authorise or not the execution of the action (e.g., according to some contextual information) and, possibly, adapt the agent behaviour by returning additional actions to be executed. The interface IPolicy permits to easily integrate different kinds of policies in jRESP Nodes. When a Node is instantiated, if no policy is provided, a default policy is used, which allows any operation to be executed.

6.2 Policing Constructs

The interface IPolicy is currently implemented by two different classes: DefaultPermitPolicy and PolicyAutomaton. The former is the default policy of each node; it allows any action by directly delegating its execution to the corresponding node. The latter policy implements a generic POLICY AUTOMATON Π (like the one presented in Section 5) which triggers policy changes according to the execution of agent actions. In particular, a PolicyAutomaton consists of a set of

[15] Scribe is a generic, scalable and efficient system for group communication and notification.

[16] FreePastry is a substrate for peer-to-peer applications.

IPolicyAutomatonStates, each of which identifies the possible policies enforced in the node, and of a reference to the current state, which is used to *authorise* agent actions with respect to the current policies.

When a PolicyAutomaton receives a request for the execution of a given action, first of all an AutorisationRequest representing the action (like the request ρ introduced in Section 5) is created. This request identifies the action an agent wants to perform, thus it provides the action name, its argument, its target and the list of attributes currently published in the node interface. The created AuthorizationRequest is then evaluated with respect to the current policy state via the (abstract) method evaluate(AutorisationRequest r) defined in the class IPolicyAutomatonState. The request evaluation can trigger an update of the current state of the PolicyAutomaton. Indeed, for each state, a sequence of *transitions* is stored in the automaton. The transitions are instances of the class PolicyAutomatonTransition which provides two methods: apply(AutorisationRequest r): boolean and nextState(): IPolicyAutomatonState. A transition is *enabled* if the first method returns true, while the next state is then obtained by invoking nextState() on the enabled transition. If no transitions are enabled, the current state is not changed.

Therefore, the full PSCEL implementation can be now achieved by defining the class FacplPolicyState, which extends IPolicyAutomatonState and wraps the Java-translated FACPL policies[17]. The overwritten method evaluate(AutorisationRequest r) delegates the authorisation to the referred FACPL policies, which return an instance of the class AuthorisationResponse containing a *decision*, i.e. permit or deny, and a set of *obligations*. The latter ones are rendered as a sequence of Actions that must be performed just after the completion of the requested action. Hence, if the decision is permit, the corresponding agent can continue as soon as all the obligations are executed. Instead, if the decision is deny, the requested action cannot be performed and the obligations possibly returned must be executed. After their completion, the action previously forbidden can be further evaluated.

6.3 Exploitation

We report here the code[18] of the jRESP implementation of the specification, presented in Section 5.2, of the robot swarm scenario.

In the previous sections we saw that jRESP, like SCEL, is parametric with respect to the knowledge representation and the policy language. The default implementations of these components provided with jRESP, i.e. the knowledge represented via *tuple space* and policies regulated according to the classes described in Section 6.2, allow a programmer to execute PSCEL specifications. The Java classes reported in this section permit appreciating how close the SCEL (resp. PSCEL) processes are to their implementation in jRESP.

[17] The Java-translated policies can be automatically obtained by using the FACPL Eclipse IDE available from the FACPL website [55].

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

For the considered scenario, in jRESP we have a Node for each robot operating in the arena[19]. Each node is equipped with the appropriate sensors and actuators that provide the machinery for interacting with the robots circuits/components. Sensors include the ones used to detect a *victim*, to check the *battery level*, to detect possible *collisions*, to access the robot position and to verify if a robot is *moving*. Actuators are used to set *robot direction*, to *stop* the movement and to start the *battery recharging activity*.

The current state of a robot is modelled via a tuple of the form ("role" , r), where r can be *explorer*, *rescuer*, *help rescuer* or *low battery*. This values correspond to the states of the policy automaton considered in Figure 4. The tuple identifying the robot state is stored in the local tuple space of each node and, together with the values read from the sensors, is used to infer the *node interface*. In the interface of each node, besides the role and the *id* of the corresponding robot, the current position is also published. The latter is identified by the attributes x and y.

Running at a node there are four agents: Explorer, HelpingRescuer, RandomWalk and isMoving. Agents Explorer and HelpingRescuer represent respectively the two branches of the non-deterministic choice[20] in process P_R defined in Section 5.2. Since there is almost a one-to-one correspondence between the class implementing an agent and its definition in PSCEL, here we only present the code of agent Explorer that is reported below. The interested reader can refer to the jRESP web site for a detailed description of the other classes.

```
1  public class Explorer extends Agent {
2    public Explorer() {
3      super("Explorer");
4    }
5    protected void doRun() throws Exception {
6      query(
7          new Template(
8              new ActualTemplateField("VICTIM_PERCEIVED"),
9              new ActualTemplateField(true)
10         ),
11         Self.SELF
12     );
13     // Pass to RESCUER state
14     put(new Tuple("role", Scenario.RESCUER), Self.SELF);
15     double x = getAttributeValue( "x" , Double.class );
16     double y = getAttributeValue( "y" , Double.class );
17     put(
18         new Tuple(
19             "victim",
20             x,
21             y,
22             3
23         ),
24         Self.SELF
25      );
26   }
27 }
```

[19] These nodes could be executed directly on physical robots assuming that these are able to execute java code

[20] Non-deterministic choice is rendered in jRESP in terms of concurrent execution of agents (which are implemented as Java threads) regulated by checks on the current status of the robot (corresponding to the state of the policy automaton).

When an instance of class Agent is executed, the method doRun() is invoked. This method defines the agent behaviour. In the case of Explorer, it consists of the sequence of steps needed to detect a victim and to broadcast its position to the other robots. The method query(), used to retrieve data from a knowledge repository, is defined in the base class Agent and implements the SCEL's action **qry**[21]. The method takes as parameters an instance of class Template and a target, and returns a matching tuple. In the code above, the target is the local component (referred by Self.SELF) while the retrieved tuple is one consisting of two fields: the first field is the constant "VICTIM_PERCEIVED" while the second field is the boolean value true. This tuple is not retrieved from the local knowledge but from the *victim sensor* when the robot is able to perceive the victim. After that, the agent retrieves the actual robot position via the attributes x and y stored in the node interface. To perform this operation method getAttributeValue is used. This method takes as parameter the name of the attribute to evaluate and its expected types and returns the collected value; null is obtained when the requested attribute is not published in the interface or when its value has not the requested type. Finally, method put is invoked to publish in the local knowledge repository the tuple witnessing that a victim has been perceived at position (x,y) and 3 robots are needed to rescue it.

The policies in force at each node are managed by an instance of the class PolicyAutomaton that implements the automaton reported in Figure 4. This automaton is instantiated as a list of FacplPolicyState, each of which contains the reference to a particular FACPL policy, and a list of transitions. For the sake of simplicity, the, straightforward, Java translation of the transition's conditions defined in Table 16 is not reported. In the following code, we show, instead, the Java implementation of the policy in page 42 that defines the automaton state RESCUER.

```
1  public class Policy_Rescuer extends Policy {
2
3    public Policy_Rescuer() {
4      addCombiningAlg(PermitUnlessDeny.class);
5      addRule(new RuleCameraOn());
6    }
7
8    class RuleCameraOn extends Rule{
9
10       RuleCameraOn(){
11         addEffect(RuleEffect.PERMIT);
12
13         addTarget(new TargetTreeRepresentation(TargetConnector.AND,
14                 new TargetTreeRepresentation(new TargetExpression(
15                         Equal.class, ActionID.PUT,
16                         new StructName("action", "action-id"))),
17                 new TargetTreeRepresentation(new TargetExpression(
18                         PatternMatch.class, new Template(
19                                 new ActualTemplateField("rescue"),
20                                 new FormalTemplateField(Double.class),
21                                 new FormalTemplateField(Double.class)),
```

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

```
22                          new StructName("action", "item"))),
23                  new TargetTreeRepresentation(new TargetExpression(
24                          LessThan.class, 40,
25                          new StructName("object", "battery_level")))
26          ));
27
28          // The PUT for adapting the process
29          addObligation(new ScelObligationExpression(RuleEffect.PERMIT,
30                  ActionID.PUT, new Tuple("cameraOn"), Self.SELF));
31      }
32    }
33  }
```

The policy is formed by the combining algorithm permit-unless-deny, which is passed as class reference, and the rule *CameraOn*. This rule is implemented as an inner class containing an authorization effect, which is `RuleEffect.PERMIT`, a target and an obligation. The target contains checks on: (i) the action's identifier (i.e., `ActionID.PUT`); (ii) the action's template (i.e., a tuple starting with the string "`rescue`" and followed by two formal double values; (iii) the battery level (i.e., more than 40). Finally, the instance of the class `ScelObligationExpression` defines the **put** action that triggers the activation of the robot's camera.

7 Quantitative Variants of SCEL

In this section we address the issue of enriching SCEL with information about action duration, by providing a stochastic semantics for the language. There exist various frameworks that support the systematic development of stochastic languages [22]. However, the main challenge in developing a stochastic semantics for SCEL is in making appropriate modeling choices, both taking into account the specific application needs and allowing to manage model complexity and size. Our contribution in this work is the proposal of four variants of StocS, a Markovian extension of a significant fragment of SCEL, that can be used to support quantitative analysis of adaptive systems composed of ensembles of cooperating components [32]. In this chapter, we focus on only one of the four variants, the so called *Network oriented* one (NET-OR, for short). The reader interested in the full spectrum of StocS semantics and their complete formal definition is referred to [32] and to the technical report [33]. In summary, StocS is essentially a *modeling language* which inherits the purpose and focus of SCEL. StocS extends SCEL by modeling the time of state-permanence as a *random variable* (r.v.) with negative exponential distribution and by replacing non-determinism by a probability distribution over outgoing transitions, thus adopting an operational semantics based on *continuous time Markov chains* (CTMC) [21]. Finally, an important aspect in a modelling language concerns the need of devising an appropriate syntax to express the environment model. In StocS, like in SCEL, the only point of contact with the environment is the knowledge base, which contains both internal information and externally-sensed events.

7.1 StocS: Stochastic SCEL

The syntax of StocS is essentially a subset of that of SCEL. In the presentation that follows, we deliberately omit to incorporate certain advanced features of SCEL, such as the presence and role of policies; we focus mainly on action durations and their stochastic modelling. Furthermore for the sake of simplicity, we consider only **put**, **get**, and **qry** actions and in $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$ and $\mathbf{put}(t)@c$ we restrict targets c to the distinguished variable self and to component *predicates* p. In order to be able to obtain a CTMC from a StocS model specification, all sources of non-determinism must be given a probabilistic interpretation. This is true also for knowledge repositories, where patterns may match different values. We push the probabilistic view a bit further, assuming that *all* repository operations have probabilistic behaviour, thus providing more flexibility to modellers (e.g. the possibility to model faulty/error outcomes and related probabilities). Letting $\mathbb{K}, \mathbb{I}$ and $\mathbb{T}$ denote the classes of all possible *knowledge states*, *knowledge items* and *knowledge templates* respectively, we require that the operator $\oplus : \mathbb{K} \times \mathbb{I} \to \mathsf{Dist}(\mathbb{K})$ for adding an item to a repository returns a *probabilistic distribution* over repositories as a result. Similarly, the withdraw operator $\ominus : \mathbb{K} \times \mathbb{T} \hookrightarrow \mathsf{Dist}(\mathbb{K} \times \mathbb{I})$ and the infer operator $\vdash\colon \mathbb{K} \times \mathbb{T} \hookrightarrow \mathsf{Dist}(\mathbb{I})$ are assumed to return a *probability distribution* over repositories paired with items, and over items, respectively. Functions $\ominus$ and $\vdash$ are partial: if no matching item is found the result is undefined. No further assumptions are required on knowledge repositories and, in fact, StocS is parametric w.r.t. to knowledge repository, like SCEL. Finally, it is assumed that an assignment of appropriate r.v. parameters is given which characterises the transmission and processing durations of the several phases of StocS action execution, as sketched below.

The semantics of SCEL does not consider any time related aspect of computation. More specifically, the execution of an action of the form $\mathbf{act}(T)@p\,.\,P$ (for **put**/**get**/**qry** actions) is described by a *single* transition of the underlying SCEL LTS semantics. In the system state reached by such a transition it is guaranteed that the process which executed the action is in its local state P and that the knowledge repositories of all components involved in the action execution have been modified accordingly. In particular, SCEL abstracts from details concerning: (a) when the execution of the action starts; (b) when the possible destination components are required to satisfy p; and (c) when the process executing the action resumes execution (i.e. becomes P); and their consequent time relationship. If we want to extend SCEL with an explicit notion of (stochastic) time, we need to take into account the time-related issues mentioned above. These issues can be addressed at different levels of abstraction, reflecting a different choice of details that are to be considered in modelling pobabilistic/timed aspects of SCEL actions.

Point (a) above does not require particular comments.

Point (b) requires to define *when* a component satisfies p with respect to a process executing an action, when time and possibly space are taken into consideration. We assume that source components are not aware of which are the components satisfying predicate p. Therefore, we define the notion of *observation*

of the component by the process, the result of which allows to establish whether the component satisfies the predicate or not. In the context of distributed systems this is very often realised by means of a message sent by the process to the component. According to this view, the check whether a component satisfies predicate p is performed *when the message reaches* the component. This means that a StocS action may require broadcast communication to be executed, even if its effect involves a few (and possibly no) components. In distributed systems, different components may have different response times depending on different network conditions and one can model explicitly the message delivery, taking into account the time required to reach the component.

Finally, point (c) raises the issue of when source component execution is to be resumed. In particular, it is necessary to identify how the source component is made aware that its role in the communication has been completed. Get/query actions are blocking and they terminate when the source receives a knowledge item from any component. A reasonable choice is that further responses received are ignored. We assume appropriate mechanisms that ensure no confusion arises between distinct actions and corresponding messages. Put actions are non-blocking, so it is sufficient that the source component is aware that the observation procedure of all components has started. Our choice is to make the source side set-up the transmission of one request of predicate evaluation for each component and then resume the execution of the source process immediately. The evaluation of the predicate against each component and the corresponding (possible) knowledge repository modification will take place at the target side(s).

In a *network-oriented* view of the system, the execution of the various phases sketched above is explicitly modelled in detail by the operational semantics, which entails that actions are *non-atomic*. Indeed, they are executed through several intermediate phases, or activities, each of which requires appropriate time duration modeling, as we illustrate by means of the following simple example. Let us consider three components, as illustrated in Fig. 6: $C_1 = \mathcal{I}_1\,[\,\mathcal{K}_1,\, P_1\,]$, $C_2 = \mathcal{I}_2\,[\,\mathcal{K}_2,\, P_2\,]$, and $C_3 = \mathcal{I}_3\,[\,\mathcal{K}_3,\, P_3\,]$ and let us assume process P_1 is defined as $\mathbf{put}(v)@p\,.\,Q$. Note that different components may be in different locations. The interaction we illustrate starts with process P_1 executing the first phase of $\mathbf{put}(v)@p$, i.e. creating two[22] copies of the special "envelope" message $\overline{\{v@p\}}$, one for component C_2 and one for component C_3, and sending these messages; they play the role of *observers*: each of them travels in the system and reaches the component it is associated with. The special message creation and message-component association phase has a duration, denoted in grey in the figure, which is determined by rate λ: this value is computed as a (given) function of several factors, among which (the size of) v. After message creation, P_1 can proceed without waiting for their arrival at the destination components—since **put** actions are non-blocking—behaving like Q (the light-grey stripe in

[22] For the sake of notational simplicity, here we assume that predicate p in process actions implicitly refers only to the *other* components, excluding the one where the process is in execution.

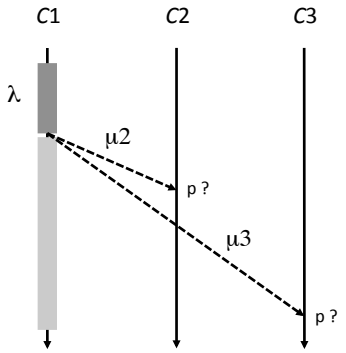

Fig. 6. Dynamics of the **put** action

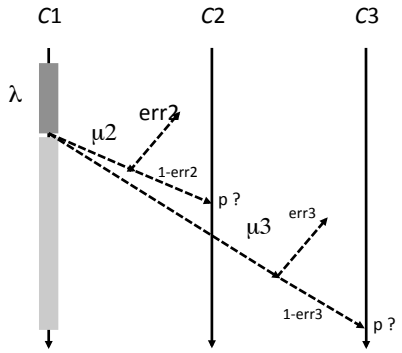

Fig. 7. Actual model of **put**

the figure illustrates the resumed execution of C_1). Each special message has to reach its destination component (in the figure this is illustrated by two dashed arrows), which checks whether its own interface satisfies p, and if so, it delivers v in its own knowledge repository. Observer delivery to C_2 (C_3 respectively) is performed with rate μ_2 (μ_3, respectively), which may depend on v and other parameters like the distance between C_1 where P_1 resides and the target component C_2 (C_3 respectively). Therefore, a *distinct* rate μ_j is associated to each target. In practice, one can be interested in modelling also the event of failed delivery of the observers. This is interesting both for producing more realistic models (with unreliable network communication), and for allowing the application of advanced analysis techniques based on fluid approximation [10], such as fluid model-checking [9]. Therefore, we add an error probability to the observers delivery, which we denoted p_{err} (or simply err, in Fig. 7). This more detailed semantics of the $\mathbf{put}(v)@p$ action is described below in more detail. The execution of **get**/**qry** actions is a little bit more complicated because the executor must remain *blocked* as long as a value matching the required pattern is sent back from *one* of the potential target components. This is realised by exploiting the *race condition* which arises from multiple competing potential target components and sophisticated use of interleaving semantics; the interested reader is referred to [33] for details.

7.2 Semantics of a StocS Fragment

In this section we present the fragment of the formal semantics definition for the **put** action in the *Network oriented* variant of StocS.

We recall that the interface $\mathcal{I}$ of a component makes information about the component available in the form of attributes, i.e. names acting as references to information stored in the components knowledge repository. It is convenient to make this dependency of the interface on the current knowledge $\mathcal{K}$ explicit. We do

this by using the notation $\mathcal{I}(\mathcal{K})$ for the *evaluation* of the interface $\mathcal{I}$ in the knowledge state $\mathcal{K}$. The set of possible interface evaluations is denoted by $\mathbb{E}$. Interface evaluations are used within the so-called *rate function* $\mathcal{R} : \mathbb{E} \times Act \times \mathbb{E} \to \mathbb{R}_{\geq 0}$, which defines the rates of actions depending on the interface evaluation of the *source* of the action, the action itself (where Act denotes the set of possible actions), and the interface evaluation of the *destination*. For this purpose, interface evaluations will be embedded within the transition labels to exchange information about source/destination components in a synchronisation action. We will conventionally use σ (δ, respectively) for the interface evaluation of a source (destination, respectively) component. The rate function is not fixed but instead is a parameter of the language. Considering interface evaluations in the rate functions, together with the executed action, allows us to take into account, in the computation of actions rates, various aspects depending on the component state such as the position/distance, as well as other time-dependent parameters. We also assume to have a *loss probability function* $f_{\mathsf{err}} : \mathbb{E} \times Act \times \mathbb{E} \to [0, 1]$ computing the probability of an error in message delivery.

As briefly sketched in Section 7.1, in the NET-OR semantics of StocS, several phases, or activities, are identified during the execution of an action. It is convenient to distinguish between output activities (those issued by a component) and input activities (those accepted by a component). To simplify the synchronisation of input and output activities , we assume input activities are "passive" and *probabilistic*, i.e. described by (discrete) probability distributions, and output activities have durations which are *stochastic*, therefore the composition of output and related input activities yields stochastic durations with parameters (i.e. rates) computed directly through multiplication.

The operational semantics rules of StocS are given in the FuTSs style [22] and, in particular, using its Rate Transition Systems (RTS) instantiation [21]. In RTSs a transition is a triple of the form $(P, \alpha, \mathscr{P})$, where the first and second components are the source state and the transition label, as usual, and the third component, $\mathscr{P}$, is the *continuation function* that associates a real non-negative value with each state P'. A non-zero value represents the rate of the exponential distribution characterising the time needed for the execution of the action represented by α, necessary to reach P' from P via the transition. Whenever $\mathscr{P}\, P' = 0$, this means that P' is not reachable from P via α[23]. RTS continuation functions are equipped with a rich set of operations that help to define these functions over sets of processes, components, and systems, as we will see.

Let $\mathbf{FTF}(S, \mathbb{R}_{\geq 0})$ denote the class of total functions from set S to $\mathbb{R}_{\geq 0}$ with finite support[24]. Given countable, non-empty sets S (of states) and A (of transition labels), an *A-labelled* RTS is a tuple $(S, A, \mathbb{R}_{\geq 0}, \rightarrowtail)$ where $\rightarrowtail \subseteq S \times A \times \mathbf{FTF}(S, \mathbb{R}_{\geq 0})$ is the A-labelled *transition relation*.

As for the standard SOS definition, also FuTS-based semantics are fully characterised by the smallest relation induced by a set of axioms and deduction

[23] We use Currying for continuation function application.

[24] A function $\mathscr{F}$ has *finite support* if and only if there exists finite $\{s_1, \ldots, s_m\} \subseteq S$, the *support* of $\mathscr{F}$, such that $\mathscr{F}\, s_i \neq 0$ for $i = 1 \ldots m$ and $\mathscr{F}\, s = 0$ otherwise.

rules [22]. The operational semantics of StocS is the FuTS induced by the rules for systems, which in turn are defined using those for processes and components. In Table 17 (18, 19, respectively) we show only the rules for **put** actions performed by processes (components and systems, respectively). It is worth noting that the rules in the tables use a structure for the transition labels which is simpler than that of the labels used in Section 2, due to the fact that we consider only a fragment of the language. Furthermore, for the sake of readability and given that our labels are simpler, we prefer to use explicit action names (e.g. **put**) instead of symbols (e.g. $\triangleright$). Finally, we typically use annotated transition labels $\overline{\alpha}$ for output activities α, while input activities are used as labels without annotations; labels for activities α which are not intended for synchronisation, i.e. *internal* activities, are denoted by $\overleftrightarrow{\alpha}$.

Rule *(env)* models the delivery of the envelope message, with delivery-time rate μ. The notation $[\mathbf{nil} \mapsto \mu]$ states that the only process which can be reached by $\{t@p\}_\mu$ via activity $\overline{\{t@p\}}$ is **nil** and the relevant rate is μ. This is a special case of the general notation $[d_1 \mapsto \gamma_1, \ldots, d_m \mapsto \gamma_m]$ for the function which associates γ_i with d_i and 0 with the other elements; $[]$ denotes the 0 constant function in $\mathbf{FTF}(X, \mathbb{R}_{\geq 0})$ and is used in Rule *(env$_\mathrm{B}$)*, which in fact states that $\overline{\{t@p\}}$ is the only activity $\{t@p\}_\mu$ can perform. A process of the form $\mathbf{put}(t)@c\,.\,P$ lounches the execution of an output activity $\overline{\mathbf{put}(t)@c}$, as postulated by Rule *(put)*. Note that the process transition is *parameterized* with respect to the evaluation σ of the interface of the specific component which the process will be running within, in the specific knowledge repository (state), as we will see when discussing Rule *(c-puto)*. Similarly, the rate λ for the execution time of the local activity of the **put** action is given by the (global) *rate function* $\mathcal{R}$ which takes σ as parameter, besides (a description of) the action itself. Note that λ does not depend on any specific destination.

$$\frac{}{\{t@p\}_\mu \xrightarrow{\overline{\{t@p\}}} [\mathbf{nil} \mapsto \mu]}\ \textit{(env)} \qquad \frac{\alpha \neq \overline{\{t@p\}}}{\{t@p\}_\mu \xrightarrow{\alpha} []}\ \textit{(env}_\mathrm{B}\textit{)}$$

$$\frac{\lambda = \mathcal{R}(\sigma, \mathbf{put}(t)@c, \_)}{\mathbf{put}(t)@c\,.\,P \xrightarrow{\overline{\mathbf{put}(t)@c}}_\sigma [P \mapsto \lambda]}\ \textit{(put)} \qquad \frac{\alpha \neq \overline{\mathbf{put}(t)@c}}{\mathbf{put}(t)@c.P \xrightarrow{\alpha} []}\ \textit{(put}_\mathrm{B}\textit{)}$$

Table 17. Operational semantics of **put** actions (processes, NET-OR)

Rule *(c-putl)* makes **put** actions *internal*, when they are targeted to self. The rule uses the notation $\mathcal{I}[\pi, \mathscr{P}]$; for interface $\mathcal{I}$ and continuation functions $\mathscr{F}_1$ and $\mathscr{F}_2$, function $\mathcal{I}[\mathscr{F}_1, \mathscr{F}_2]$ returns $(\mathscr{F}_1\, K) \cdot (\mathscr{F}_2\, P)$ when applied to component $\mathcal{I}\,[\mathcal{K},\ P]$, and 0 otherwise. Rule *(c-puto)* lifts the (start of the) execution of a non-local **put** at the source component level; the (Curried) characteristic function $\mathcal{X}$, with $\mathcal{X}\,\mathcal{K} = [\mathcal{K} \mapsto 1]$, is used in the obvious way. The *output* label $\overline{\sigma\ :\ \mathbf{put}(t)@p}$ is

$$\frac{\sigma = \mathcal{I}(\mathcal{K}) \quad P \xrightarrow{\overline{\mathbf{put}(t)@\mathsf{self}}}_{\sigma} \mathscr{P} \quad \mathcal{K} \oplus t = \pi}{\mathcal{I}\,[\,\mathcal{K},\,P\,] \xrightarrow{\overleftrightarrow{\sigma:\mathbf{put}(t)@\mathsf{self}}} \mathcal{I}[\pi, \mathscr{P}]} \textit{ (c-putl)}$$

$$\frac{\sigma = \mathcal{I}(\mathcal{K}) \quad P \xrightarrow{\overline{\mathbf{put}(t)@p}}_{\sigma} \mathscr{P}}{\mathcal{I}\,[\,\mathcal{K},\,P\,] \xrightarrow{\overline{\sigma:\mathbf{put}(t)@p}} \mathcal{I}[(\mathcal{X}\,\mathcal{K}), \mathscr{P}]} \textit{ (c-puto)}$$

$$\frac{\delta = \mathcal{I}(\mathcal{K}) \quad \mu = \mathcal{R}(\sigma, \{t@p\}, \delta) \quad p_{\mathsf{err}} = f_{\mathsf{err}}(\sigma, \{t@p\}, \delta)}{\mathcal{I}\,[\,\mathcal{K},\,P\,] \xrightarrow{\sigma:\mathbf{put}(t)@p} [\,\mathcal{I}\,[\,\mathcal{K},\,P\,] \mapsto p_{\mathsf{err}},\ \mathcal{I}[\mathcal{K}, P|\{t@p\}_{\mu}] \mapsto (1 - p_{\mathsf{err}})\,]} \textit{ (c-puti)}$$

$$\frac{P \xrightarrow{\overline{\{t@p\}}} \mathscr{P} \quad \mathcal{I}(\mathcal{K}) \models p \quad \mathcal{K} \oplus t = \pi}{\mathcal{I}\,[\,\mathcal{K},\,P\,] \xrightarrow{\overline{\{t@p\}}} \mathcal{I}[\pi, \mathscr{P}]} \textit{ (c-enva)} \qquad \frac{P \xrightarrow{\overline{\{t@p\}}} \mathscr{P} \quad \mathcal{I}(\mathcal{K}) \not\models p}{\mathcal{I}\,[\,\mathcal{K},\,P\,] \xrightarrow{\overline{\{t@p\}}} \mathcal{I}[(\mathcal{X}\mathcal{K}), \mathscr{P}]} \textit{ (c-envr)}$$

Table 18. Operational semantics of **put** actions (components, NET-OR)

$$\frac{S_1 \xrightarrow{\overline{\sigma:\mathbf{put}(t)@p}} \mathscr{S}_1^o \quad S_1 \xrightarrow{\sigma:\mathbf{put}(t)@p} \mathscr{S}_1^i \quad S_2 \xrightarrow{\overline{\sigma:\mathbf{put}(t)@p}} \mathscr{S}_2^o \quad S_2 \xrightarrow{\sigma:\mathbf{put}(t)@p} \mathscr{S}_2^i}{S_1 \parallel S_2 \xrightarrow{\overline{\sigma:\mathbf{put}(t)@p}} \mathscr{S}_1^o \parallel \mathscr{S}_2^i + \mathscr{S}_1^i \parallel \mathscr{S}_2^o} \textit{ (s-po)}$$

$$\frac{S_1 \xrightarrow{\sigma:\mathbf{put}(t)@p} \mathscr{S}_1 \quad S_2 \xrightarrow{\sigma:\mathbf{put}(t)@p} \mathscr{S}_2}{S_1 \parallel S_2 \xrightarrow{\sigma:\mathbf{put}(t)@p} \mathscr{S}_1 \parallel \mathscr{S}_2} \textit{ (s-pi)}$$

$$\frac{S_1 \xrightarrow{\overleftrightarrow{\sigma:\mathbf{put}(t)@\mathsf{self}}} \mathscr{S}_1 \quad S_2 \xrightarrow{\overleftrightarrow{\sigma:\mathbf{put}(t)@\mathsf{self}}} \mathscr{S}_2}{S_1 \parallel S_2 \xrightarrow{\overleftrightarrow{\sigma:\mathbf{put}(t)@\mathsf{self}}} \mathscr{S}_1 \parallel (\mathcal{X}\,S_2) + (\mathcal{X}\,S_1) \parallel \mathscr{S}_2} \textit{ (s-spl)}$$

Table 19. Operational semantics of **put** actions (systems, NET-OR)

used for launching a broadcast; note, in the transition label, the indication of the source σ which is the evaluated interface of the component at hand and which is required to be the same as the parameter of the process transition used in the premiss. As established by Rule *(c-puti)*, every (potentially target) component can perform an activity with the dual *input* label $\sigma : \mathbf{put}(t)@p$. The result is the instantiation of the envelope process $\{t@p\}_\mu$ in the component. In this way, a specific instance of the envelope is associated with the specific component. The transmission of the envelope will be modelled by the execution of $\{t@p\}_\mu$, with transmission time characterized by rate μ (see Rule *(env)* again). Note that rate μ may depend both on the source (σ) and on the specific destination components (δ); furthermore, the successful transmission of the envelope is subject to the absence of errors (with probability $1 - p_{\mathsf{err}}$). Rules *(c-enva)* and *(c-envr)* ensure that the repository is updated if the interface evaluation satisfies the predicate. Finally, Rules *(s-po)* and *(s-pi)* together realise the broadcast communication a component uses for sending the envelope to all the other components, while *(s-spl)* takes care of local, consequently internal, **put** actions. For continuation functions $\mathscr{F}_1$ and $\mathscr{F}_2$, function $\mathscr{F}_1 \parallel \mathscr{F}_2$ returns $(\mathscr{F}_1\, S_1) \cdot (\mathscr{F}_2\, S_2)$ when applied to a system $S_1 \parallel S_2$ and 0 otherwise, whereas function $\mathscr{F}_1 + \mathscr{F}_2$ is the point-wise extension of $+$, i.e. $(\mathscr{F}_1 + \mathscr{F}_2)S = (\mathscr{F}_1\, S) + (\mathscr{F}_2\, S)$.

8 Verification

In this section we present the verification approaches developed so far for guaranteeing properties of systems modelled in SCEL. Currently, rather than developing new ad-hoc verification tools for SCEL, we have exploited existent tools. In particular, for verifying *qualitative* properties we use the well-established model checker Spin, while for verifying *quantitative* ones we use a statistical model-checking approach relying on either the simulation environment provided by jRESP or the MAUDE-based interpreter of SCEL specifications MISSCEL.

8.1 Simulation and Analysis via jRESP

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

The simulation environment integrated in jRESP is based on a *discrete event simulator* and on a specialised variant of class Node, named SimulationNode, that allows the execution of SCEL programs in the simulated environment.

jRESP agents can be also directly executed on a SimulationNode (which shares the same interface of class Node). In this case, agents are rendered as specific simulation processes instead that Java threads. The discrete event simulator

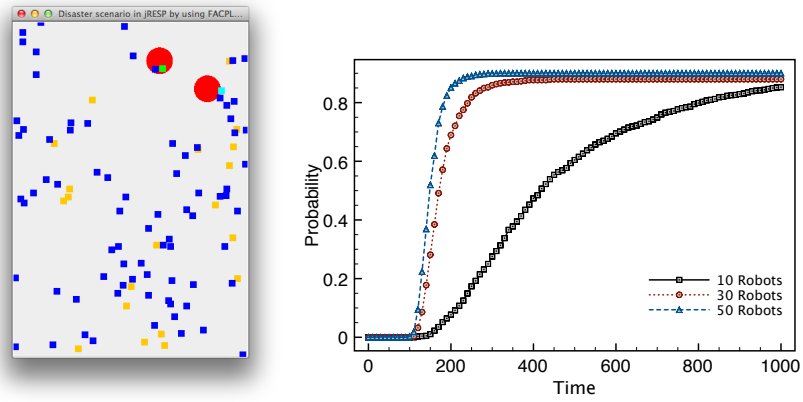

Fig. 8. Simulation and analysis of the robot swarm scenario in jRESP

is responsible for scheduling the execution of SCEL actions. Actions execution time is computed by an instance of class DelayFactory. This class, following the same approach considered in Section 7, computes the execution time of a SCEL action by considering the type of action performed, its arguments and the interfaces of the involved components. Notice that, StocS semantics can be easily obtained when DelayFactory computes the action execution time via the appropriate sampling of exponential distributed random variables.

To set-up the simulation environment in jRESP one has also to define a class that provides the machinery to manage the physical data of the scenario. This data includes, e.g., robots positions, direction and speed. Sensors and actuators *installed* in a SimulationNode are used to collect data from the scenario and to update the state of the simulation. This mechanism, for instance, can be used to *stop* the movement of a robot or to regulate its direction. In our case, we consider the class ScenarioArena that, in addition to the above mentioned data, also provides the methods for updating robots position and computing collisions. These methods are periodically executed by the jRESP simulation environment. For the sake of simplicity, in the simulation, only collisions with the borders of the arena are considered, while collisions among robots are ignored.

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

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

In Figure 8, we report a screenshot of the simulation simulation (left-hand side) and the results of the analysis (right-hand side) of the robot swarm scenario. In the screenshot, red semi-circles represent the locations of the victims, while squares represent robots, whose color is used to show their current state. Robots in the *explorer* state are *blue*, *rescuers* are *green*, *help rescuers* are *light blue* while the ones with *low battery* are *yellow*. The analysis results are represented as a chart showing the probability of rescuing the victims within a given time according to different numbers of robots (i.e., 10, 30 and 50). In the performed analyses we consider two victims each of which needs a swarm of three robots to be rescued. Notably, the victims can be rescued only after 100 time steps and, beyond a certain threshold, increasing the number of robots is not worthwhile (in fact, the difference in terms of rescuing time between 50 and 30 robots is marginal with respect to the cost of deploying a double number of robots).

8.2 Maude-based Verification

SCEL comes equipped with solid semantics foundations laying the basis for formal reasoning. This is exploited in MISSCEL (Maude Interpreter and Simulator for SCEL) which is an implementation of SCEL's operational semantics in the Maude framework [16]. MISSCEL currently focuses on a SCEL dialect where repositories are implemented as multisets of tuples (as in Section 3.1), while the processes of a SCEL component evolve in a pure interleaving fashion (i.e. the interaction predicate is the interleaving one defined in Table 3). Access control policies are supported, even if no policy language has been integrated yet: by default, every request is currently authorized.

Why Maude? MISSCEL exploits the rich Maude toolset to perform:

- automatic state-space generation;
- qualitative analysis via Maude's invariant and LTL model checkers;
- debugging via probabilistic simulations and animations generation;
- statistical model checking via the recently proposed MultiVeStA [51], a distributed statistical analyser extending VeStA and PVeStA [3, 52].

A further advantage of MISSCEL is that SCEL specifications can now be intertwined with raw Maude code, exploiting its expressiveness. This allows us to obtain sophisticated specifications in which SCEL is used to model behaviours, aggregations, and knowledge manipulation aspects, leaving scenario-specific details like, e.g. robots movements or computation of distances to Maude.

Reasoning in MISSCEL. In Section 3.3 we discussed about the enrichment of SCEL components with reasoning capabilities via external reasoners. As a matter of fact in [5] we have showed how to enrich MISSCEL components (and thus SCEL components) with reasoning capabilities exploiting the reasoner Pirlo [4], implemented in Maude as well, and we analyzed a collision-avoidance robotic scenario. Collision avoidance is a key feature of the robot navigation.

For example, in our robot disaster scenario, collision avoidance can be used to minimise collisions during the random walk phase, which is characterised by a high density of robots arbitrarily moving in unpredictable ways. Collision avoidance is also an archetypal example of how external reasoners can be applied to the scenario considered in this paper.

Using MISSCEL we can specify and evaluate two different random walks strategies: a normal one and an informed one. We considered two kinds of robots distinguished by the strategy they apply: *normal robots*, and *informed robots*. Normal robots choose randomly (with a uniform distribution) among five actions: to perform random walk in one of the four cardinal directions, or to stay idle. Informed robots monitor their surrounding environment by relying on proximity sensors, and exploit this information to choose actions aiming at reducing the number of collisions. The amount of environment perceived by an informed robot depends on its perception range. The positions up, right, down and left are reachable with a single move, while the diagonal ones are reachable with two moves. However, the perception of the diagonal positions is also useful for the computation of the next action, as a robot located there (e.g. one perceived in down-left) could move towards the same position chosen by the informed robot (e.g. up, if the informed robot moves left).

Statistical Analysis with MultiVeStA. In [5] we exploited MISSCEL and the recently proposed statistical model checker MultiVeStA to perform a statistical quantitative analysis of the robotic collision avoidance scenario.

MultiVeStA is a Java-based distributed statistical model checker which allows its users to enrich existing discrete event simulators with automated and statistical analysis capabilities. The analysis algorithms of MultiVeStA do not depend on the underlying simulation engine: MultiVeStA only makes the assumption that multiple discrete event simulations can be performed on the input model. The tool has been used to reason about public transportation systems [26], volunteer clouds [50], crowd-steering [41] and robotic collision avoidance [5] scenarios. Note however that MISSCEL is an executable operational semantics for SCEL, and as such, given a SCEL specification representing a system's state (i.e. a set of SCEL components), MISSCEL executes it by applying a rule of SCEL's semantics to (part of) the state. According to such semantics, a system evolves non-deterministically by executing the process of one of its components, and in particular by consuming one of its actions. As usual (especially in the Maude context, e.g. [6, 11, 2, 25]), in order to perform statistical analysis it is necessary to obtain probabilistic behaviours out of non-deterministic ones by resolving non-determinism in probabilistic choices. For this reason, we defined a Java wrapper for MISSCEL together with a set of external schedulers which permit to obtain probabilistic simulations of SCEL specifications, which can then be exploited by MultiVeStA to perform statistical model checking.

In our analysis in [5], we considered two scenarios with ten normal robots and an informed one, varying the size of the perception range of the informed robot. In the first scenario the informed robot perceives only the four surrounding positions (up, right, down, left). In the second scenario the informed robot has

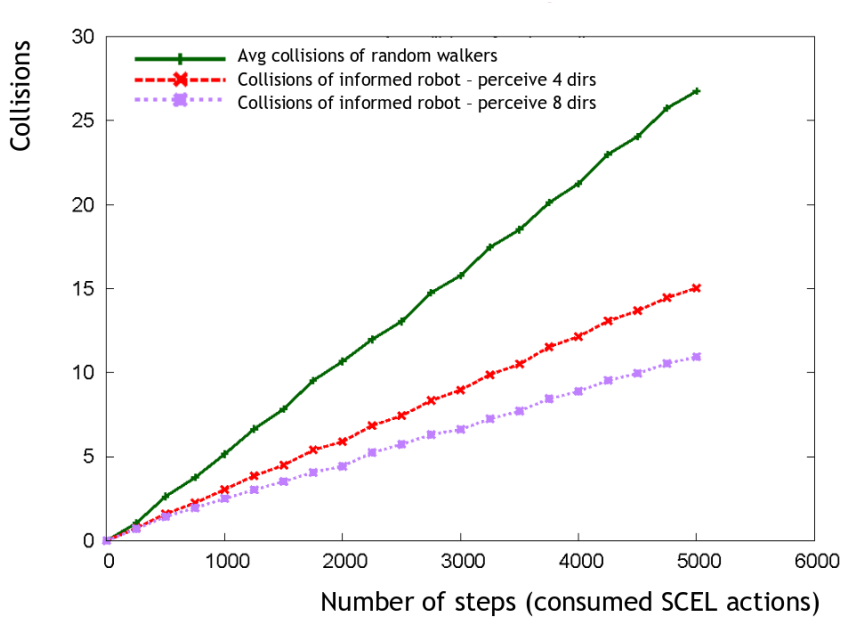

Fig. 9. Collisions of normal and informed robots at varying of number of steps.

```
SC(I(tId('SCId), tId('role), tId('x), tId('y), ...,
   K(< tId('SCId) ; av(id('robot-1)) >, < tId('role) ; av("rescuer") >,
     < av("pos") av(41) av(3)>, < tId('x) ; av(41)>, < tId('y) ; av(3)>  ...,
   Pi(INTERLEAVING-PROCESSES_AUTHORIZE-ALL),
   P(pDef('ProcessName))
  )
```

Listing 1.1. A MISSCEL component representing a robot

a wider perception range, allowing to perceive also the positions in the four diagonal directions (up-right, down-right, down-left, up-left). For both scenarios we first studied the expected value of the average number of collisions of the normal robots when varying of number of execution steps. Not surprisingly, we obtained very similar measures for both the scenarios, and hence we use only one plot in Figure 9 ("Avg collisions of random walkers"). More interesting is the case of informed robots. As depicted by the plots "Collisions of informed robot - perceive 4 dirs" and "Collisions of informed robot - perceive 8 dirs", informed robots do significantly less collisions than the normal ones, and wider perception ranges allow to further decrease the number of collisions.

Implementation details. Listing 1.1 provides an excerpt of a possible MISSCEL representation of a robot. As discussed, each robot of our scenario is modelled as a SCEL component. In MISSCEL, a SCEL component is defined as a MAUDE term with sort `ServiceComponent` built with the following operation

```
op SC : Interface Knowledge Policies Processes -> ServiceComponent
```

As an implementation choice, in MISSCEL tuples may have an identifier (e.g. `< tId('role) ; av("rescuer") >` is a tuple with identifier `role`), but it is not mandatory (e.g. `< av("pos") av(41.0) av(3.0) >` has no identifier). Note that, for implementation reasons, actual values (e.g. strings and integers) are enclosed in the constructor `av`. Note moreover that `'role` is a MAUDE term with sort quoted identifier (similar to strings) built by prefixing alphanumeric words with the operator "`'`". However, only tuples with identifiers can be exposed by the interface, as identifiers are used as pointers to the actual values of the tuples stored in the knowledge. Then, as depicted in line 1 of Listing 1.1, an interface is just a set of tuple identifiers enclosed in the MAUDE operation `I`, while, as depicted in lines 2-3, the knowledge is a multiset of tuples enclosed in the operation `K`. For example, the sketched robot has id `id('robot-1)`, role `rescuer` and position `(41,3)`. Line 4 specifies that the default policy is enforced.

Line 5 specifies that the behaviour of the robot is provided in the process definition `'ProcessName`. To provide an example of process definition, the MISSCEL representation of the process P_r of Section 5.2 is provided in Listing 1.2. Note the almost one-to-one correspondence between the process specification and its MISSCEL representation. SCEL variables with type value are built with the MAUDE operations `?x` (when act as binders, e.g. in a **get** or **qry** as in line 5) or `x` (when instantiated, e.g. as in line 7), having as parameter the name of the variable (we also have the corresponding process variables `?X` and `X`). Listing 1.2 also provides an hint on how MISSCEL deals with process definitions and their invocations. As depicted in lines 1-10, the body of a process is provided in the form of a MAUDE equation. MAUDE equations are *executed* by the MAUDE engine to rewrite occurrences of terms (in this case `invoke(pDef('Pr))`) matching the left-hand side (LHS) of the equation (i.e. before the `=`) in the term specified in the right-hand side (RHS) of the equation (i.e. after the `=`), in this case the body of the process. Intuitively, once all the preceding actions have been executed, a process definition (e.g. `pDef('Pr)`) is *invoked*. That is, it is encapsulated in the operation

```
op invoke : ProcessDefinition -> Process
```

(e.g. `invoke(pDef('Pr))`), which can then be matched with the LHS of the corresponding equation, causing the replacement of the process definition with its body.

Predicates are defined as MAUDE operations with sort `Predicate`. As depicted in line 5 of Listing 1.2, we exploited the predicate `Prescuers`, specified in lines 12-15. In line 12 we define the MAUDE operation `Prescuers` with sort `Predicate` having no parameters. Then, in lines 13-15 we provide the body of the predicate in the form of a MAUDE equation, similarly to what done for process definitions. Note that in predicates we follow the convention of prefixing tuple identifiers referring to the target of the communication with the keyword `remote` (while we use `this` for local ones).

```
1  eq invoke(pDef('Pr')) = (
2      (qry(< av("victimPerceived") av(true) >)@self.
3        put(< av("victim") x y av(3) >)@ self.
4        put(< av("rescue") >)@ self +
5       get(< av("victim") ?x('x) ?x('y) ?x('count) >)@ Prescuers .
6        pDef('HelpingRescuer)
7        put(< av("victim") x('x) x('y) 3 >)@ self.
8        put(< av("rescue") >)@ self
9       ) | pDef('RandomWalk) | pDef('IsMoving)
10 ) .
11
12 op Prescuers : -> Predicate .
13 eq Prescuers
14  = remote. tId('role) = av("rescuer") OR
15    remote. tId('role) = av("helpRescuer") .
16 op PrescuersWithDist : FormalOrActualValue FormalOrActualValue -> Predicate .
17 vars xvic yvic : ActualValue .
18 eq PrescuersWithDist(xvic,yvic)
19  = (remote. tId('role) = av("rescuer") OR
20     remote. tId('role) = av("helpRescuer")) AND
21    dist(xvic,yvic, remote. tId('x), remote. tId('y)) <= 10 .
```

Listing 1.2. The MISSCEL representation of process P_r of Section 5.2

Interestingly, predicates can also be defined with parameters. An example is `PrescuersWithDist` of lines 16-21, having as parameter the position of the victim, so to send the message only to rescuers *near* to the victim. In line 16 we define the Maude operation `PrescuersWithDist` with sort `Predicate` having as parameters two `FormalOrActualValue` (i.e. either SCEL variables or actual values). Then, similarly to `Prescuers`, in lines 18-21 we provide the body of the predicate in the form of a Maude equation. Given that at line 17 we specify the Maude variables (i.e. place-holders for any term with the same sort) `xvic` and `yvic` with sort `ActualValue`, we have that only instantiated occurrences of the predicate (i.e. where all the SCEL variables have been replaced by actual values) match with the LHS of the equation.

Line 21 of listing 1.2 provides an interesting example demonstrating the usefulness of mixing SCEL and Maude specifications: `dist` is a Maude operation which computes the distance between two points (e.g. the positions of two robots), which could for example correspond to the Euclidean distance. Noteworthy, in case we would consider *distances* with different assumptions, e.g. congested areas, it would be sufficient to change the Maude operations leaving unchanged the SCEL specification.

Coming to semantics-related aspects, we have seen in Section 2.3 that the operational semantics of SCEL is defined in two steps. The same happens in MISSCEL. For the sake of presentation, we now exemplify the correspondence of SCEL semantics and its implementation in MISSCEL for the semantics of processes only.

Let us consider the first four rules of the SCEL process semantics reported in Table 2. Listing 1.3 depicts (omitting unnecessary details) how we implemented these rules in MISSCEL, where `P`, `Q` and `P1` are Maude variables with sort `Process` (i.e. place-holders for any term with the specified sort), while `a` is an

```
1  op commit : Process -> Commitment .
2  rl commit(P) => commitment(inaction,P) .
3  rl commit(a . P) => commitment(a,P) .
4  crl commit(P + Q) => commitment(a, P1) if commit(P) => commitment(a, P1) .
```

Listing 1.3. The first four rules of SCEL process semantics implemented in MISSCEL

`Action` variable. The correspondence is straightforward. Note that we need only one rule for the `+` operator, as we defined it with the `comm` axiom, meaning that it has the commutative property, i.e. when applying a rule to `P + Q`, Maude will try to match the rule also with `Q + P`.

8.3 Spin-based Verification

We present here a verification approach for SCEL specifications based on the Spin model checker [28]. Specifically, we provide a translation of SCEL specifications into Promela, the input language of Spin, and show how to exploit it to verify some properties of interest of the swarm robotics scenario with Spin.

From SCEL to Promela. For the sake of presentation, we consider a simple instance of SCEL with no policies, standard interleaving interaction (as in Table 3), and knowledge repositories based on multiple distributed tuple spaces (as in Section 3.1). Moreover, we do not consider other sophisticated features of SCEL such as higher-order communication and dynamic creation of new names and components.

We present below the key points of the translation from SCEL to Promela by resorting to the robotics scenario[25]. The translation is defined by a family of functions $[\![\cdot]\!]$, whose formal definitions are given in [23].

Specifications. The Promela specification resulting from the translation (of a simplified variant) of the SCEL specification of the swarm robotics scenario is shown in Listing 1.4. It contains the declaration of the necessary data structures for representing interfaces, knowledge, components and processes. Data structures representing interfaces and knowledge repositories are declared with a global scope; in this way, attributes and knowledge items can be directly accessed by Promela processes.

Interfaces. The translation declares a structured type `interface` (lines 8-11) as a collection of variables, one for each attribute. In our scenario, all robots expose two attributes: `id`, ranging over integer values, and `role`, ranging over names in $\{$`rescuer`, `helpRescuer`, `explorer`$\}$. All interfaces are then recorded in the array `I` (line 14), whose size is given by the number of robots, defined by the constant `NROBOTS` (line 2).

[25] The complete specification of the scenario can be retrieved from `http://rap.dsi.unifi.it/scel/docs/SpinSpecificationSwarmRoboticsScenario.pml`

```
1  /* Constants declaration */
2  #define NROBOTS 10      /* Number of robots*/
3  #define CAPACITY 10     /* Maximum size of knowledge repositories */
4  ...
5
6  /* The type of the interface as a struct of attributes */
7  mtype={rescuer, helpRescuer, explorer}
8  typedef interface{
9          int id;
10         mtype role;
11 }
12
13 /* A component-indexed array of interfaces */
14 interface I[NROBOTS];
15
16 /* Component-indexed array of knowledge repositories */
17 mtype={victim, direction}
18 chan K[NROBOTS]=[CAPACITY] of {mtype, int, int, int};
19
20 /* Components specification */
21 active [NROBOTS] proctype Robot(){
22         /* Attribute initialization */
23         int id=_pid;
24         I[id].id=id;
25         I[id].role=explorer;
26
27         /* Component's process specification */
28         ...
29 }
```

Listing 1.4. Promela specification of the swarm robotics scenario

Repositories. All knowledge repositories are grouped together in the array `K` (line 18). Each repository is implemented as a channel of tuples, whose length is given by the maximum length of items used in the specification. In fact, to simplify message management in Promela, all tuples have the same length and are composed only of a value in {`victim`, `direction`} followed by integer values. To fulfil this assumption, messages representing shorter items are completed by using dummy values. The dimension of repositories is set by means of the constant `CAPACITY` (line 3).

Components. The translation of a component $\mathcal{I}_i[\mathcal{K}_i, \Pi_i, P_i]$ corresponds to a declaration of a Promela process, via the `proctype` construct (line 21), that initializes the data structure modelling the component attributes with values in $\mathcal{I}_i$ (lines 23-25). In our example, the data structure modelling the knowledge repository, i.e. channel `K`$[i]$, is not initialized because $\mathcal{K}_i$ is initially empty. Notably, component translations are automatically instantiated in the initial system state (by means of the keyword `active`).

Processes and actions. The composition of SCEL processes can be naturally translated into Promela process declarations and their composition. For example, multiple `run` statements can be used for the parallel execution of processes. Then, each SCEL action is translated in a small piece of Promela code that basically performs output or input operations on channels `K[i]`. For exam-

ple, the group-oriented action **get**("victim", $?x, ?y, ?count$)@($role$="*rescuer*" $\vee$ $role$="*helpRescuer*"), used by a robot to receive a request for help by other robots, is rendered in Promela as a non-deterministic choice among a set of input operations on each `K[i]`. In particular, for each robot component `i`, there is a choice branch

```
::atomic{(g_i -> K[i]??victim,x,y,count}
```

where the guard `g_i` is as follows:

```
(I[i].role==rescuer || I[i].role==helpRescuer) && K[i]??[victim,_,_,_]
```

This guard ensures the transition to fire only if the target predicate holds for component `i` (i.e., the component plays either role `rescuer` or `helpRescuer`) and `i` has a matching `victim` tuple in its repository. If that is the case, the tuple is indeed removed using the (consuming) input operation `K[i]??victim,x,y,count`. It is worth noticing that the `atomic` block is used to guarantee atomic execution of the operations.

Spin verification. We illustrate in this section some examples of how Spin can be used to check and verify properties of SCEL specifications, by resorting to the translation of the SCEL specification of our swarm robotics scenario.

Checking Deadlock Absence. A first property one would like to check is absence of deadlocks. Below, we report the result of invoking Spin for checking deadlock absence in our scenario with 10 robots:

```
State-vector 2188 byte, depth reached 415289, errors: 0
   415290 states, stored
```

The result is positive (no errors) and Spin explores more than 400.000 states.

Checking liveness. Another typical use of Spin that is very convenient for our purposes is to look for interesting executions, that is, we characterise them by means of an LTL formula. For example, in our scenario, we can specify a formula

```
[](I[i].role==helpRescuer ->
   <>(positionX[i]==VICTIMX && positionY[2]==VICTIMY))
```

which states that whenever the robot `i` becomes a HelpRescuer, it eventually reaches the victim. Spin provides a positive answer, since in our simplified scenario robots' batteries never discharge. We have also defined a variant of the scenario where each robot component has a battery level value that decreases after each movement. In this case, the formula is not satisfied and Spin returns a counterexample showing an execution of the system in which the robot becomes a HelpRescuer but then stops moving because the battery is completely discharged.

9 Concluding Remarks

We have presented the kernel language SCEL and its Java implementation together with alternative linguistic primitives and with automatic tools to support verification of qualitative and quantitative properties of its programs. To assess to which extent SCEL meets our expectations, we have used it to tackle a number of case studies from robotics [15, 24, 13, 34], service provision domains [19, 23], cloud-computing [36, 38] and e-Mobility domains [12]. Moreover, to verify the impact of SCEL on autonomic programming we have shown how it can be used to model a key aspect such as self-expression [13] and how it can flexibly model different adaptation patterns [15].

Our holistic approach to programming autonomic computing systems permits to govern systems complexity by providing flexible abstractions for modeling behaviors, knowledge and policies and for exploiting external reasoners whenever informed decisions need to be taken. We are now working on two different, almost opposite, directions.

On the one hand, we are developing 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. This would enable us to implement an integrated environment for supporting the development of autonomic 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. On the other hand, we are distilling from SCEL a minimal calculus where communication partners are selected according to predicates on attributes exposed by the different processes. Our aim is to understand the full impact on distributed programming of this novel paradigm that has proved very fruitful in modeling the interaction of large numbers of autonomic systems. For the new calculus we plan to develop behavioral relations, axiomatizations and logics that will help to devise new tools for supporting specification and verification of SCEL programs.

Other interesting topics that deserve further investigation are those connected to policies, reasoners and adaptation. We are currently working on defining different interaction policies to study the possibility of modeling different forms of synchronization and communication, and for example to guarantee local synchronous interaction and global asynchronous interaction between components. Moreover, we are studying the connections between knowledge handlers, reasoners and components goals described by means of appropriate knowledge representation languages. In this case the aim is the development of methodologies that enable components to take decisions, possibly after consulting external reasoners, about possible alternative behaviors by choosing among the best possibilities while being aware of the consequences.

Acknowledgements We would like to thank all friends of the ASCENS project, without their contributions and stimuli, SCEL would not have been conceived.

References

1. Abeywickrama, D., Combaz, J., Horky, V., Keznikl, J., Kofron, J., Lafuente, A.L., Loreti, M., Margheri, A., Mayer, P., Monreale, V., Montanari, U., Pinciroli, C., Tuma, P., Vandin, A., Vassev, E.: Tools for Ensemble Design and Runtime. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
2. Agha, G.A., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. In: Cerone, A., Wiklicky, H. (eds.) QAPL 2005. ENTCS, vol. 153(2), pp. 213–239. Elsevier (2006)
3. AlTurki, M., Meseguer, J.: Pvesta: A parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO. Lecture Notes in Computer Science, vol. 6859, pp. 386–392. Springer (2011)
4. Belzner, L.: Action programming in rewriting logic (technical communication). Theory and Practice of Logic Programming, On-line Supplement (2013)
5. Belzner, L., De Nicola, R., Vandin, A., Wirsing, M.: Reasoning (on) service component ensembles in rewriting logic. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi. Lecture Notes in Computer Science, vol. 8373, pp. 188–211. Springer (2014)
6. Bentea, L., Ölveczky, P.C.: A probabilistic strategy language for probabilistic rewrite theories and its application to cloud computing. In: Martí-Oliet, N., Palomino, M. (eds.) WADT. Lecture Notes in Computer Science, vol. 7841, pp. 77–94. Springer (2012)
7. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. J. ACM 44(2), 201–236 (1997)
8. Bistarelli, S., Montanari, U., Rossi, F.: Soft concurrent constraint programming. ACM Trans. Comput. Log. 7(3), 563–589 (2006)
9. Bortolussi, L., Hillston, J.: Fluid model checking. In: Koutny, M., Ulidowski, I. (eds.) CONCUR. Lecture Notes in Computer Science, vol. 7454, pp. 333–347. Springer (2012)
10. Bortolussi, L., Hillston, J., Latella, D., Massink, M.: Continuous approximation of collective system behaviour: A tutorial. Perform. Eval. 70(5), 317–349 (2013)
11. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: Modelling and analyzing adaptive self-assembly strategies with Maude. In: Durán, F. (ed.) WRLA 2012. LNCS, vol. 7571, pp. 118–138. Springer (2012)
12. Bures, T., De Nicola, R., Gerostathopoulos, I., Hoch, N., Kit, M., Koch, N., Monreale, G., Montanari, U., Pugliese, R., Serbedzija, N., Wirsing, M., Zambonelli, F.: A Life Cycle for the Development of Autonomic Systems: The e-mobility showcase. In: Proc. of SASOW. pp. 71–76. IEEE (2013)
13. Cabri, G., Capodieci, N., Cesari, L., De Nicola, R., Pugliese, R., Tiezzi, F., Zambonelli, F.: Self-expression and dynamic attribute-based ensembles in SCEL. In: ISoLA. LNCS, vol. 8802, pp. 147–163. Springer (2014)
14. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.I.T.: Scalable Application-Level Anycast for Highly Dynamic Groups. In: ICQT. pp. 47–57. LNCS 2816, Springer (2003)
15. Cesari, L., De Nicola, R., Pugliese, R., Puviani, M., Tiezzi, F., Zambonelli, F.: Formalising adaptation patterns for autonomic ensembles. In: FACS. LNCS, vol. 8348, pp. 100–118. Springer (2013)

16. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
17. Combaz, J., Bensalem, S., Kofron, J.: Correctness of Service Components and Service Component Ensembles. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
18. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. In: POLICY. pp. 18–38. LNCS 1995, Springer (2001)
19. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: A Language-based Approach to Autonomic Computing. In: FMCO 2011. pp. 25–48. LNCS 7542, Springer (2012), http://rap.dsi.unifi.it/scel/
20. De Nicola, R., Ferrari, G., Pugliese, R.: Klaim: A Kernel Language for Agents Interaction and Mobility. IEEE Trans. Software Eng. 24(5), 315–330 (1998)
21. De Nicola, R., Latella, D., Loreti, M., Massink, M.: Rate-based transition systems for stochastic process calculi. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S.E., Thomas, W. (eds.) ICALP (2). Lecture Notes in Computer Science, vol. 5556, pp. 435–446. Springer (2009)
22. De Nicola, R., Latella, D., Loreti, M., Massink, M.: A uniform definition of stochastic process calculi. ACM Comput. Surv. 46(1), 5:1–5:35 (Jul 2013)
23. De Nicola, R., Lluch-Lafuente, A., Loreti, M., Morichetta, A., Pugliese, R., Senni, V., Tiezzi, F.: Programming and verifying component ensembles. In: FPS. LNCS, vol. 8415, pp. 69–83. Springer (2014)
24. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A Formal Approach to Autonomic Systems Programming: The SCEL Language. TAAS 9(2), Article 7 (2014)
25. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable availability under denial of service attacks through formal patterns. In: de Lara, J., Zisman, A. (eds.) FASE 2012. Lecture Notes in Computer Science, vol. 7212, pp. 78–93. Springer (2012)
26. Gilmore, S., Tribastone, M., Vandin, A.: An analysis pathway for the quantitative evaluation of public transport systems. In: Albert, E., Sekerinski, E. (eds.) Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8739, pp. 71–86. Springer (2014)
27. Hölzl, M., Gabor, T.: Reasoning and Learning for Awareness and Adaptation. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
28. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. 23(5), 279–295 (1997)
29. IBM: Autonomic Computing Policy Language - ACPL, http://www.ibm.com/developerworks/tivoli/tutorials/ac-spl/
30. IBM: An architectural blueprint for autonomic computing. Tech. rep., IBM (June 2005), Third edition
31. Khakpour, N., Jalili, S., Talcott, C.L., Sirjani, M., Mousavi, M.R.: Formal modeling of evolving self-adaptive systems. Sci. Comput. Program. 78(1), 3–26 (2012)
32. Latella, D., Loreti, M., Massink, M., Senni, V.: Stochastically timed predicate-based communication primitives for autonomic computing. In: Bertrand, N., Bortolussi, L. (eds.) Proceedings of the Twelfth Workshop on Quantitative Aspects of Programming Languages (QAPL 2014). pp. 1–16. Electronic Proceed-

ings in Theoretical Computer Science, EPTCS (2014), iSSN: 2075-2180, DOI: 10.4204/EPTCS.154.1
33. Latella, D., Loreti, M., Massink, M., Senni, V.: On StocS: a Stochastic extension of SCEL. Tech. Rep. 11, ASCENS Project (February 2014), http://www.ascens-ist.eu/
34. Loreti, M., Margheri, A., Pugliese, R., Tiezzi, F.: On programming and policing autonomic computing systems. In: ISoLA. LNCS, vol. 8802, pp. 164–183. Springer (2014)
35. Margheri, A., Masi, M., Pugliese, R., Tiezzi, F.: A Formal Software Engineering Approach to Policy-based Access Control. Tech. rep., DiSIA, Univ. Firenze (2013), available at http://rap.dsi.unifi.it/facpl/research/Facpl-TR.pdf
36. Margheri, A., Pugliese, R., Tiezzi, F.: Linguistic Abstractions for Programming and Policing Autonomic Computing Systems. In: UIC/ATC. pp. 404–409. IEEE (2013)
37. Margheri, A., Pugliese, R., Tiezzi, F.: Linguistic abstractions for programming and policing autonomic computing systems. Tech. rep., Univ. Firenze (2013), http://rap.dsi.unifi.it/scel/pdf/PSCEL-TR.pdf
38. Mayer, P., Klarl, A., Hennicker, R., Puviani, M., Tiezzi, F., Pugliese, R., Keznikl, J., Bure, T.: The autonomic cloud: A vision of voluntary, peer-2-peer cloud computing. In: Proc. of SASOW. pp. 89–94. IEEE (2013)
39. OASIS XACML TC: eXtensible Access Control Markup Language (XACML) version 3.0 (January 2013), http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf
40. O'Grady, R., Groß, R., Christensen, A.L., Dorigo, M.: Self-assembly strategies in a group of autonomous mobile robots. Auton. Robots 28(4), 439–455 (2010), http://springerlink.metapress.com/content/qtm4736000150519/
41. Pianini, D., Sebastio, S., Vandin, A.: Distributed statistical analysis of complex systems modeled through a chemical metaphor. In: HPCS (MOSPAS workshop). pp. 416–423. IEEE (2014)
42. Pinciroli, C., Bonani, M., Mondada, F., Dorigo, M.: Adaptation and Awareness in Robot Ensembles: Scenarios and Algorithms. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project. Springer Verlag, Heidelberg (2015)
43. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
44. Project InterLink: http://interlink.ics.forth.gr (2007)
45. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier (2006)
46. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Middleware. pp. 329–350. LNCS 2218, Springer (2001)
47. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. TAAS 4(2) (2009)
48. Saraswat, V., Rinard, M.: Concurrent constraint programming. In: POPL. p. 232245. ACM Press (1990)
49. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000)
50. Sebastio, S., Amoretti, M., Lluch-Lafuente, A.: A computational field framework for collaborative task execution in volunteer clouds. In: SEAMS. pp. 105–114. ACM (2014)

51. Sebastio, S., Vandin, A.: MultiVeStA: statistical model checking for discrete event simulators. In: Horvath, A., Buchholz, P., Cortellessa, V., Muscariello, L., Squillante, M.S. (eds.) 7th International Conference on Performance Evaluation Methodologies and Tools, ValueTools '13, Torino, Italy, December 10-12, 2013. pp. 310–315. ACM (2013), http://dl.acm.org/citation.cfm?id=2631846
52. Sen, K., Viswanathan, M., Agha, G.A.: Vesta: A statistical model-checker and analyzer for probabilistic systems. In: Baier, C., Chiola, G., Smirni, E. (eds.) QEST 2005. pp. 251–252. IEEE Computer Society (2005)
53. Sommerville, I., Cliff, D., Calinescu, R., Keen, J., Kelly, T., Kwiatkowska, M.Z., McDermid, J.A., Paige, R.F.: Large-scale complex IT systems. Commun. ACM 55(7), 71–77 (2012)
54. Vassev, E., Hinchey, M.: Knowledge Representation for Adaptive and Self-Aware Systems. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag (2015)
55. FACPL website: http://rap.dsi.unifi.it/facpl