A Language-Based Approach to Autonomic Computing[⋆]

Rocco De Nicola[1], Gianluigi Ferrari[2], Michele Loreti[3], and Rosario Pugliese[3]

[1] IMT, Institute for Advanced Studies Lucca, Italy
[2] Università degli Studi di Pisa, Italy
[3] Università degli Studi di Firenze, Italy

Abstract. SCEL is a new language specifically designed to model autonomic components and their interaction. It brings together various programming abstractions that permit to directly represent knowledge, behaviors and aggregations according to specific policies. It also supports naturally programming self-awareness, context-awareness, and adaptation. In this paper, we first present design principles, syntax and operational semantics of SCEL. Then, we show how a dialect can be defined by appropriately instantiating the features of the language we left open to deal with different application domains and use this dialect to model a simple, yet illustrative, example application. Finally, we demonstrate that adaptation can be naturally expressed in SCEL.

1 Introduction

The increasing complexity, heterogeneity and dynamism of current computational and information infrastructures is calling for new ways of designing and managing computer systems and applications. *Adaptation*, namely "the capability of a system to change its behavior according to new requirements or environment conditions" [1], has been largely proposed as a powerful means for taming the ever-increasing complexity of today's computer systems and applications. Besides, a new paradigm, named *autonomic computing* [2], has been advocated that aims at making modern distributed IT systems *self-manageable*, i.e. capable of continuously self-monitoring and selecting appropriate operations.

More recently, to capture the relevant features and challenges, the 'Interlink WG on software intensive systems and new computing paradigms' [3] has proposed to use the term *ensembles* to refer to:

> The future generation of software-intensive systems dealing with massive numbers of components, featuring complex interactions among components and with humans and other systems, operating in open and non-deterministic environments, and dynamically adapting to new requirements, technologies and environmental conditions.

[⋆] This work has been partially sponsored by the EU project ASCENS (257414).

Systems partially satisfying the above definition of ensemble have been already built, e.g. national infrastructures such as power grids, or large online cloud systems such as Amazon or Google. But significant human intervention is needed to dynamically adapt them. Instead, one crucial requirement is to ensure that an ensemble continues to function reliably in spite of unforeseen changes and that adaptation does not render systems inoperable, unsafe or insecure.

To move from the engineering of traditional systems to that of ensembles, an higher level of abstraction is needed. Many research efforts are currently devoted to the search of methodologies and tools to build ensembles by exploiting techniques developed in different research areas such as software engineering, artificial intelligence and formal methods. The aim is the definition of linguistic primitives and methodologies to program autonomic and adaptive systems while relying on rigorous foundations that support verification of their properties.

The challenge for language designers is to devise appropriate abstractions and linguistic primitives to deal with the large dimension of systems, the need to adapt to evolving requirements and to changes in the working environment, and the emergent behaviors resulting from complex interactions.

The notions of *service components* (SCs) and *service-component ensembles* (SCEs) have been put forward as a means to structure a system into well-understood, independent and distributed building blocks that interact in specified ways. SCs are autonomic entities that can cooperate, with different roles, in open and non-deterministic environments. SCEs are instead sets of SCs with dedicated knowledge units and resources, featuring goal-oriented execution. Most of the basic properties of SCs and SCEs are already guaranteed by current service-oriented architectures; the novelty lays in the notions of goal-oriented evolution and of self-awareness and context-awareness.

A possible way to achieve awareness is to equip SCs and SCEs with information about their own state and behavior, to enable them to collect and store information about their working environment and to use this information for redirecting and adapting their behavior. A typical SCE is reported in Figure 1, which evidences that ensembles are structured sets of components, with dedicated *knowledge units* to represent shared, local and global knowledge, that can be interconnected via highly dynamic *infrastructures*.

These notions of SCs and SCEs are the starting point of the EU project ASCENS [4,5] that aims at investigating different issues ranging from languages for modelling and programming SCEs to foundational models for adaptation, dynamic self-expression and reconfiguration, from formal methods for the design and verification of reliable SCEs to software infrastructures supporting deployment and execution of SCE-based applications. The aim is to develop formal tools and methodologies supporting the design of self-adaptive systems that can autonomously adapt to, also unexpected, changes in the operating environment, while keeping most of their complexity hidden from administrators and users.

In this paper we present some of the work done to develop linguistic supports for modelling and programming service components and their ensembles. More specifically, we introduce SCEL (Service Component Ensemble Language), a

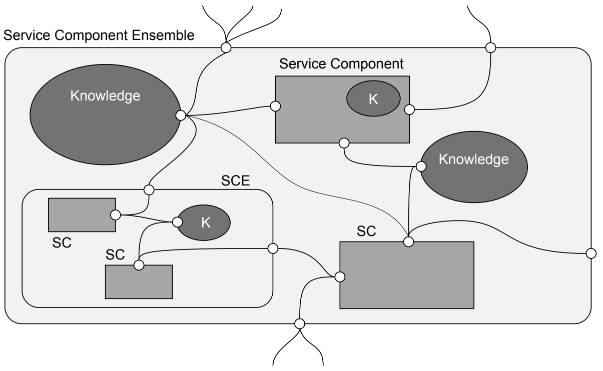

Fig. 1. A Service Component Ensemble

new language designed for autonomic computing. SCEL brings together different programming abstractions that permit to directly represent *knowledge*, *behaviors* and *aggregations* according to specific *policies*. It also supports naturally programming self-awareness, context-awareness and adaptation. SCEL's solid semantic grounds lay the basis for developing logics, tools and methodologies for formal reasoning about system behavior to establish qualitative and quantitative properties of both the individual components and the ensembles.

The rest of the paper is organized as follows. We present SCEL's design principles, syntax and operational semantics in Sections 2, 3 and 4, resp.. In Section 5, we show an example of how a dialect of SCEL can be defined by appropriately instantiating the features of the language we left open to deal with different application domains and in Section 6 we demonstrate how the proposed dialect can be used to model a simple yet illustrative example application. In Section 7 we argue that adaptation can be naturally expressed in SCEL. We review more strictly related work in Section 8 and conclude in Section 9 with some final remarks and hints for future work.

2 SCEL: Design Principles

SCEL provides abstractions explicitly supporting autonomic computing systems in terms of *Aggregations*, *Behaviors* and *Knowledge* according to specific *Policies*.

Aggregation Abstractions describe how different entities are brought together to form *components* and *systems* and to offer the possibility to construct the *software architecture* of autonomic systems. Composition of components and their interaction is implemented by exploiting the notion of *interface* that can be queried to determine the attributes and the functionalities provided and/or required by components. *Ensembles* are specific aggregations of components that represent *social or technical networks* of autonomic components. The key point is that the formation rule is endogenous to components: components of an ensemble are connected by the interdependency relations established in their interfaces. Therefore, an ensemble is not a rigid fixed network but rather a dynamic graph-like structure where component linkages are dynamically established.

Behavioral Abstractions describe how computations progress. These abstractions are modelled as processes in the style of standard process calculi. *Interaction* comes in when components access data in the knowledge repositories of other components. *Adaptation* emerges as the result of knowledge acquisition and manipulation.

Knowledge Abstractions provide the high level primitives to manage pieces of relevant information coming from different sources. Knowledge is represented through items stored in repositories. Knowledge items contain either *application data* or *awareness data*. The former are used for determining the progress of component computations, while the latter provide information about the environment in which the different components are running (e.g. monitored data from sensors) or about the actual status of an autonomic component (e.g. about its current position or the remaining battery's charge level). We assume that each knowledge repository provides three abstract operations that can be used by autonomic components for *adding* new knowledge to the repository, for *retrieving* knowledge from the repository and for *withdrawing* knowledge from it.

Policy Abstractions deal with the way behaviors are regulated. Since few assumptions can be made about the operational environment, that is frequently open, highly dynamic and possibly hostile, the ability of programming and enforcing a finer control on behavior is essential to assure that valuable information is not lost. Policies are the mean to guarantee such control. Interaction policies and Service Level Agreement (SLA) policies provide two standard examples of policy abstractions. Other examples are security properties maintaining the right linkage between data values and their associated usage policies (data-leakage policies) or limiting the flow of sensitive information to untrusted sources (access control and reputation policies).

The two central ingredients of SCEL are the notions of *autonomic component* and of *ensemble* that we shall additionally consider below.

2.1 Components

An *autonomic component* $\mathcal{I}[\mathcal{K}, \Pi, P]$ consists of:

1. an *interface* $\mathcal{I}$ publishing and making available structural and behavioral information about the component itself;
2. a *knowledge manager* $\mathcal{K}$, managing both application data and awareness data, together with the specific handling mechanism;
3. a set of *policies* Π regulating the interaction between the different internal parts of the component and the interaction of the component with the others;
4. a *process* P together with a set of process definitions that can be dynamically activated. Some of the processes in P perform local computation, while others may coordinate processes interaction with the knowledge repository and deal with the issues related to adaptation and reconfiguration.

Component interfaces can be inquired to extract components name, the interdependencies among components, and the services offered by components. Indeed, the interface of a component provides at least the following attributes:

- *id*: its name;
- *ensemble*: a predicate on interfaces used to determine the ensemble the component has created and currently coordinates;
- *membership*: a predicate on the interfaces used to determine the ensembles which the component is willing to be member of.

Additional attributes might, e.g., indicate the battery's charge level and the component's GPS position.

Notably, the whole information provided by the component interface is stored in the local knowledge of the component and therefore it can be dynamically changed by using the appropriate operators for knowledge handling.

2.2 Ensembles

Ensembles are aggregations of components characterized by means of suitable predicates associated to the attributes *ensemble* and *membership*. Surprisingly (it might be), no specific syntactic category or operator for forming ensembles is provided by SCEL. Rather, to better capture their dynamicity, ensembles are 'synthesized' dynamically by exploiting the values of the components attributes. This design choice guarantees high dynamicity and flexibility in forming, joining and leaving ensembles and does avoid resorting to structure ensembles through rigid syntactic constructs.

For example, the names of the components that can be members of an ensemble can be fixed via the predicate

$$P(\mathcal{I}) \stackrel{def}{=} \mathcal{I}.id \in \{n, m, p\}$$

If the attribute *ensemble* of a component C has assigned $P(\mathcal{I})$, then a component C' is part of the ensemble coordinated by C if its name is n, m or p. Of course, the predicate assigned to *ensemble* can be changed dynamically, thus permitting to modify at run-time the members of the ensemble coordinated by C.

As another example, to dynamically characterize the members of an ensemble that are active and have a battery charge level greater than 30%, the predicate

$$P(\mathcal{I}) \stackrel{def}{=} \mathcal{I}.active = yes \land \mathcal{I}.battery\_level > 30\%$$

could be used. Here, we are assuming that the interface of each component willing to be part of the ensemble contains the attributes *active* and *battery_level*.

Components, in turn, could be willing to be part of any ensemble. This is modelled by letting attribute *membership* be associated to the predicate *true*. On the contrary, components may not want to be part of any ensemble, in this case *membership* will be set to be $false$. More generally, components can place restrictions on the ensembles which they are willing to be member of by appropriately setting the attribute *membership*. For example, using the predicate

$$P(\mathcal{I}) \stackrel{def}{=} \mathcal{I}.trust\_level > medium$$

a component can express its willingness to be only part of those ensembles coordinated by components whose (certified) trust level is greater than medium.

Table 1. SCEL syntax ($\mathcal{K}$, Π, T, and t are parameters)

SYSTEMS:	COMPONENTS:
$S ::= C \mid S_1 \parallel S_2 \mid (\nu n)S$	$C ::= \mathcal{I}[\mathcal{K}, \Pi, P]$

PROCESSES:

$P ::= \mathbf{nil} \mid a.P \mid P_1 + P_2 \mid P_1[\,P_2\,] \mid X \mid A(\bar{p}) \quad (A(\bar{f}) \triangleq P)$

ACTIONS:

$a ::= \mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c \mid \mathbf{exec}(P) \mid \mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$

TARGETS:

$c ::= n \mid x \mid \mathsf{self}$

3 SCEL: Syntax

The syntax of SCEL is illustrated in Table 1. There, different syntactic categories are defined that constitute the main ingredients of our language. The basic category of the syntax is that relative to PROCESSES that are used to build up COMPONENTS that in turn are used to define SYSTEMS. PROCESSES model the flow of the ACTIONS that can be performed. Each ACTION has among its parameters a TARGET, that indicates the other component that is involved in that action, and either an ITEM or a TEMPLATE, that determines the part of KNOWLEDGE to be added, retrieved or removed. POLICIES are used to control and adapt the actions of the different components in order to guarantee the achievement of specific goals or the satisfaction of specific properties.

It has to be said that our aim is to identify linguistic constructs for uniformly modeling the control of computation, the interaction among possibly heterogeneous components, and the architecture of systems and ensembles. Therefore, we have left open some syntactic categories, namely KNOWLEDGE (ranged over by $\mathcal{K}$), POLICIES (Π), TEMPLATES (T), and ITEMS (t). These represent additional language features that need to be introduced, e.g. to represent and store knowledge of different forms (e.g. constraints, clauses, records, tuples) or to express a variety of policies (e.g. to regulate knowledge handling, resource usage, process execution, process interaction, actions priority, security, trust, reputation). We don't want to take a definite standing about these categories and prefer they be fixed from time to time according to the specific application domain or to the taste of the language user. In the rest of this section, we consider one by one the explicitly defined categories and describe them in detail.

PROCESSES are the SCEL active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* ($a.P$), *nondeterministic choice* ($P_1 + P_2$), *controlled composition* ($P_1[\,P_2\,]$), *process variable* (X), *parameterised process invocation* ($A(\bar{p})$), and *parameterised process definition* ($A(\bar{f}) \triangleq P$). The construct $P_1[\,P_2\,]$ abstracts the various forms of parallel composition commonly used in process calculi. Process variables are used to support higher-order communication, namely the capability to exchange (the code of) a process 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.

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 c. These operations exploit templates T as patterns to select knowledge items t in the repositories. They rely heavily on the used knowledge repository and are implemented by invoking the handling operations it provides. Action $\mathbf{exec}(P)$ triggers a controlled (parallel) execution of process P. Action $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ creates a new component $\mathcal{I}[\mathcal{K}, \Pi, P]$.

Action **get** is a *blocking* action, in the sense that the process executing it has to wait for the wanted element if it is not (yet) available in the knowledge repository. Action **qry**, exactly like **get**, suspends the process executing it if the knowledge repository does not (yet) contain or cannot 'produce the wanted element. The two blocking actions differ also for the fact that **get** removes the found item from the knowledge repository while **qry** leaves the target repository unchanged. Actions **put**, **exec** and **new** are instead non-blocking and are immediately executed.

Component names are denoted by n, n', ..., variables for names are denoted by x, x', ..., while c stands for a name or a variable. The distinguished variable self can be used by processes to refer to the name of their hosting component.

SYSTEMS aggregate COMPONENTS (see Section 2.1) through the *composition* operator $\_ \parallel \_$. It is also possible to restrict the scope of a name, say n, by using the *name restriction* operator $(\nu n)\_$. Thus, 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, it allows components to communicate restricted names thus enlarging their scope to encompass also the receiving components (like restriction in π-calculus [6]).

4 SCEL: Operational Semantics

The operational semantics of SCEL is given in the SOS style [7] by relying on the notion of Labelled Transition System (LTS), which is a triple $\langle \mathcal{S}, \mathcal{L}, \longrightarrow \rangle$ made of a set of states $\mathcal{S}$, a set of transition labels $\mathcal{L}$, and a labelled transition relation $\longrightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ accounting for the actions that can be performed from each state and the new state reached after each such transition. The semantics is defined in two steps: first, the semantics of processes specifies process commitments ignoring process allocation, available data, regulating policies, etc.; then, by taking process commitments and system configuration into account, the semantics of systems provides a full description of systems behavior.

To define the semantics, we use the sets of *bound* variables $bv(E)$ and *free* variables $fv(E)$, and the sets of names $n(E)$, *bound* names and *free* names occurring in a syntactic term E. These sets, as usual, can be defined inductively on the syntax of actions, processes, components, and systems by taking into account that the only binding constructs are actions **get** and **qry** as concerns variables and action **new** and the restriction operator as concerns names. More precisely,

Table 2. Operational semantics of processes

$$a.P \xmapsto{a} P \quad (a \neq \mathbf{exec}(Q)) \qquad \mathbf{exec}(Q).P \xmapsto{\mathbf{exec}(Q)} P[\,Q\,] \qquad P \xmapsto{\circ} P$$

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

$$\frac{P \xmapsto{\alpha} P' \quad Q \xmapsto{\beta} Q'}{P[\,Q\,] \xmapsto{\alpha[\,\beta\,]} P'[\,Q'\,]}\ bv(\alpha) \cap bv(\beta) = \emptyset \qquad \frac{P =_\alpha P' \quad P' \xmapsto{\beta} P''}{P \xmapsto{\beta} P''}$$

actions $\mathbf{get}(T)@c$ and $\mathbf{qry}(T)@c$ bind the variables occurring in the template T, while action $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ binds the name associated to attribute $\mathcal{I}.id$; the scope of these binders is the process P_1 syntactically following the action in a prefix form $a.P_1$. The restriction operator $(\nu n)\_$ binds n in the scope $\_$. A term without free variables is deemed *closed* (it may contain free names).

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

4.1 Operational Semantics of Processes

The semantics of processes specifies process commitments, i.e. the actions that processes can initially perform. That is, given a process P, its semantics points out all the actions that P can initially perform and the continuation process P' obtained after each such action. To simplify the rules, we do not restrict them (and the semantics) to the subset of closed processes, although when defining the semantics of systems we only consider the transitions from closed processes (see Section 4.2). Moreover, we only consider processes that are such that their bound names are pairwise distinct and different from their free names.

The LTS defining the semantics of processes is given as follows:

- the set of states coincides with the set of processes as defined in Table 1;
- the set of transition labels is generated by the following production rule

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

  meaning that a label is either an action as defined in Table 1, or the symbol $\circ$, denoting inaction, or the composition $\alpha[\,\beta\,]$ of two labels α and β;
- the labelled transition relation $\longmapsto$ is the least relation induced by the inference rules in Table 2. We will use P and Q, possibly indexed, to range over processes and write $P \xmapsto{\alpha} Q$ instead of $\langle P, \alpha, Q\rangle \in \longmapsto$.

The rules defining the labelled transition relation are straightforward. In particular, **exec** spawns a new concurrent process whose execution can be controlled by the continuation of the process performing the action. The rule defining the semantics of $P[\,Q\,]$ states that a transition labeled $\alpha[\,\beta\,]$ is performed when Q makes the action β while P makes the action α. However, P and Q are not forced to synchronise. Indeed, thanks to the third rule, that allows any process to perform a $\circ$-labelled transition, α and/or β may always be $\circ$. The semantics of $P[\,Q\,]$ at the level of processes is indeed absolutely permissive and generates all possible compositions of the commitments of P and Q. This semantics is then specialized at the level of systems by means of interaction predicates for taking policies into account. Condition $bv(\alpha) \cap bv(\beta) = \emptyset$ means that the variables freed by the action $\alpha[\,\beta\,]$ in the two processes P and Q must be different: this because they correspond to bound variables that were intended to be different (although they might have had the same identity) and, once they get free, could be subject to possibly different substitutions (substitutions are generated and applied by rule *(pr-sys)* in Table 3). Notably, also this condition is not strict: it can be always made true by application of the last rule stating that *α-equivalent processes*, i.e. processes only differing in the identity of bound variables (this equivalence relation is denoted by $=_\alpha$), perform the same transitions.

4.2 Operational Semantics of Systems

The operational semantics of systems is defined in two steps. First, we define an LTS to derive the transitions enabled from systems without occurrences of the name restriction operator.Then, by exploiting this LTS, we provide the semantics of generic systems by means of a (unlabelled) transition system (TS), that is a pair $\langle \mathcal{S}, \rightarrowtail \rangle$ made of a set of states $\mathcal{S}$ and a (unlabelled) transition relation $\rightarrowtail \subseteq \mathcal{S} \times \mathcal{S}$ accounting for the computation steps that can be performed from each state and the new state reached after each such transition. This approach allows us to avoid the intricacies, also from a notational point of view, arising when dealing with name mobility in computations (e.g. when opening and closing the scopes of name restrictions).

To simplify notation, we will use $\mathcal{I}$ and $\mathcal{J}$ to range over interfaces. Notation $\mathcal{I} \models \mathcal{J}.ensemble$ indicates that a component with interface $\mathcal{J}$ is willing to accept a component with interface $\mathcal{I}$ in the ensemble it coordinates. Similarly, $\mathcal{J} \models \mathcal{I}.membership$ indicates that $\mathcal{I}$ is willing to be one of the components of the ensemble coordinated by $\mathcal{J}$. We assume that it always implicitly holds that $\mathcal{I} \models \mathcal{I}.ensemble \;\wedge\; \mathcal{I} \models \mathcal{I}.membership$, i.e. that a component is always part of the ensemble it coordinates. Moreover, we assume that the names of the attributes of a component are just pointers to the actual values contained in the knowledge repository associated to the component. This amounts to saying that in terms of the form $\mathcal{I}[\mathcal{K}, \Pi, P]$, $\mathcal{I}$ only includes the names of the attributes, as their corresponding values can be easily retrieved from $\mathcal{K}$. However, when $\mathcal{I}$ is used in isolation, it also includes the attributes' values.

The LTS defining the semantics of systems without restricted names is $\langle \mathcal{S}, \mathcal{L}, \longrightarrow \rangle$ where

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

$$\begin{aligned}\lambda ::= \tau \;\big|\; \mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P) \;\big|\; \mathcal{I} \diamond \mathcal{J} \;\big|\; \mathcal{I} : t \triangleleft c \;\big|\; \mathcal{I} : t \blacktriangleleft c \;\big|\; \mathcal{I} : t \triangleright c \\ \;\big|\; \mathcal{I} : t \bar{\triangleleft} \mathcal{J} \;\big|\; \mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J} \;\big|\; \mathcal{I} : t \bar{\triangleright} \mathcal{J}\end{aligned}$$

  where τ denotes an internal computation step, $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$ denotes the willingness of component $\mathcal{I}$ to create the new component $\mathcal{J}[\mathcal{K}, \Pi, P]$, $\mathcal{I} \diamond \mathcal{J}$ denotes the willingness of two components with interfaces $\mathcal{I}$ and $\mathcal{J}$ to interact, $\mathcal{I} : t \triangleleft c$ ($\mathcal{I} : t \blacktriangleleft c$) denotes the intention of component $\mathcal{I}$ to withdraw (retrieve) item t from the repository at c, $\mathcal{I} : t \triangleright c$ denotes the intention of component $\mathcal{I}$ to add item t to the repository at c, $\mathcal{I} : t \bar{\triangleleft} \mathcal{J}$ ($\mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J}$) denotes that component $\mathcal{I}$ is allowed to withdraw (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 $\mathcal{J}$;
- $\longrightarrow$ is the labelled transition relation induced by the inference rules in Table 3. We will write $S \xrightarrow{\lambda} S'$ instead of $\langle S, \lambda, S' \rangle \in \longrightarrow$.

The labelled transition relation relies on the following two predicates:

- the *interaction predicate*, $\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma$, means that under policy Π and interface $\mathcal{I}$, process label α yields system label λ and substitution σ;
- the *authorization predicate*, $\Pi, \mathcal{I} \vdash \lambda$, means that under policy Π and interface $\mathcal{I}$, system label λ is allowed.

The interaction predicate establishes a relation between process labels and system labels and thus determines the system label λ to exhibit and the substitution σ to apply when a process performs a transition labeled α. It is called interaction predicate because its main role is determining the effect of the concurrent execution of different actions by different processes that, e.g., exhibit labels of the form $\alpha_1[\,\alpha_2\,]$. Many different interaction predicates can thus be defined to capture well-known process computation and interaction patterns such as interleaving, asynchronous communication, synchronous communication, full synchrony, broadcasting, etc. Despite the several interaction predicates that can be defined, we expect anyway that a well-defined interaction predicate satisfies some obvious criteria. For example, a process label of the form $\mathbf{get}(T)@c$ should be related to system labels of the form $\mathcal{I} : t \triangleleft c$, where t is any item 'matching' the template T, while a process label of the form $\mathbf{put}(t)@c$ should be related to system labels of the form $\mathcal{I} : t' \triangleright c$, where t' is any item resulting from the evaluation of t. We refer the reader to [8] for some notable examples.

The authorization predicate is used to determine the actions that can be performed according to specific policies. Likewise the interaction predicate, many different reasonable authorization predicates can be defined depending on Π.

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

Table 3. Semantics of systems: labelled transition relation (symmetric of rules *(syncget)*, *(syncqry)*, *(syncput)*, *(enscomm)* and *(async)* omitted)

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

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:\mathbf{new}(\mathcal{J},\mathcal{K}',\Pi',P')} C \qquad n = \mathcal{J}.id \qquad n \notin n(\mathcal{I}[\mathcal{K}, \Pi, \mathbf{nil}])}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} (\nu n)(C \parallel \mathcal{J}[\mathcal{K}', \Pi', P'])} \ \textit{(newc)}$$

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

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

$$\frac{S_1 \xrightarrow{\mathcal{I}:t\triangleleft n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t\,\bar{\triangleleft}\,\mathcal{J}} S_2' \qquad \mathcal{J}.id = n \qquad ens(\mathcal{I}, \mathcal{J}) \Rightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2'} \ \textit{(syncget)}$$

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

$$\frac{\mathcal{K} \vdash t \qquad \Pi, \mathcal{J} \vdash \mathcal{I} : t \,\bar{\blacktriangleleft}\, \mathcal{J}}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t\,\bar{\blacktriangleleft}\,\mathcal{J}} \mathcal{J}[\mathcal{K}, \Pi, P]} \ \textit{(accqry)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t\blacktriangleleft n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t\,\bar{\blacktriangleleft}\,\mathcal{J}} S_2' \qquad \mathcal{J}.id = n \qquad ens(\mathcal{I}, \mathcal{J}) \Rightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2'} \ \textit{(syncqry)}$$

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

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

$$\frac{S_1 \xrightarrow{\mathcal{I}:t\triangleright n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t\,\bar{\triangleright}\,\mathcal{J}} S_2' \qquad \mathcal{J}.id = n \qquad ens(\mathcal{I}, \mathcal{J}) \Rightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2'} \ \textit{(syncput)}$$

$$\frac{S \xrightarrow{\mathcal{I}\diamond\mathcal{J}} S' \qquad \mathcal{I} \in \mathcal{I}' \wedge \mathcal{J} \in \mathcal{I}' \qquad \Pi, \mathcal{I}' \vdash \mathcal{I} \diamond \mathcal{J}}{\mathcal{I}'[\mathcal{K}, \Pi, P] \parallel S \xrightarrow{\tau} \mathcal{I}'[\mathcal{K}, \Pi, P] \parallel S'} \ \textit{(enscomm)}$$

$$\frac{S_1 \xrightarrow{\lambda} S_1'}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2} \ \textit{(async)}$$

- $\mathcal{K} \ominus t = \mathcal{K}'$: 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 $\mathcal{K}'$.

Rule *(pr-sys)* transforms process labels into system labels by exploiting the interaction predicate $\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma$. In particular, it generates the following system labels: τ, $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$, $\mathcal{I} : t \lhd c$, $\mathcal{I} : t \blacktriangleleft c$ and $\mathcal{I} : t \rhd c$. As a consequence of this transformation, a substitution σ (i.e. a function from variables to values) is generated and applied to the continuation of the process that has exhibited label α. This is necessary when α contains a **get** or a **qry**, because, due to the way the semantics of processes is defined, the continuation P' may contain free variables even if P is closed. It is worth noting that the domain of σ is the set of variables that are bound in α, thus, since $fv(P') \subseteq bv(\alpha)$, the process $P'\{\sigma\}$ is closed. The application of the rule also replaces self with the corresponding name.

No specific system label is used for indicating execution of action **exec**. Indeed, this action is always local to the component executing it, and no other component is involved in that action. Hence, when applying rule *(pr-sys)*, all the information (i.e. Π) needed to decide if the action can be allowed or not is present. When **exec** is allowed, the interaction predicate in the premise of the rule is of the form $\Pi, \mathcal{I} : \mathbf{exec}(Q) \succ \tau, []$, where $[]$ denotes the empty substitution, and the transition corresponds to an internal computation step.

Like the **exec**, action **new** is decided by using the information within a single component. However, since it affects the whole system as it creates a new component, its execution is indicated by a specific system label $\mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$ (generated by rule *(pr-sys)*) carrying enough information for the creation of the new component to take place. When the new component is actually created *(newc)*, it is checked that its name n is not already used in the creating component possibly except for the process part (this condition can be always made true by exploiting α-equivalence among processes) and, if so, a restriction is put in the system obtained after the computation step to delimit the scope of n.

The successful execution of the remaining three actions requires, at system level, appropriate synchronizations. For this reason, for each action we have a pair of complementary labels. Action **get** withdraws an item either from the local repository, rule *(lget)*, or from a specific repository, rule *(syncget)*. In both cases, this transition corresponds to an internal computation step. However, in case of remote withdrawal, it is also needed to make sure that the interacting components belong to the same ensemble. We have two cases to consider, depending on predicate $ens(\mathcal{I}, \mathcal{J})$ defined as $(\mathcal{I} \models \mathcal{J}.ensemble \ \wedge \ \mathcal{J} \models \mathcal{I}.membership) \vee (\mathcal{J} \models \mathcal{I}.ensemble \ \wedge \ \mathcal{I} \models \mathcal{J}.membership)$:

- Predicate $ens(\mathcal{I}, \mathcal{J})$ holds true, i.e. the component with interface $\mathcal{I}$ is part of the ensemble defined by the component with interface $\mathcal{J}$, or viceversa. Then, the (conditional) premise $ens(\mathcal{I}, \mathcal{J}) \Rightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}$ of rule *(syncget)* sets λ to τ and the inference of the computation step terminates.
- Predicate $ens(\mathcal{I}, \mathcal{J})$ holds false and the two components with interface $\mathcal{I}$ and $\mathcal{J}$ are both part of the ensemble coordinated by another component,

say $\mathcal{I}'[\mathcal{K},\Pi,P]$. Indeed, we write $\mathcal{I} \in \mathcal{I}' \wedge \mathcal{J} \in \mathcal{I}'$ as a shorthand for condition $(\mathcal{I} \models \mathcal{I}'.\mathit{ensemble} \wedge \mathcal{I}' \models \mathcal{I}.\mathit{membership}) \wedge (\mathcal{J} \models \mathcal{I}'.\mathit{ensemble} \wedge \mathcal{I}' \models \mathcal{J}.\mathit{membership})$. We now take advantage of the 'else' case of the premise $\mathit{ens}(\mathcal{I},\mathcal{J}) \Rightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}$ of rule *(syncget)* that sets λ to $\mathcal{I} \diamond \mathcal{J}$. Consequently, rule *(enscomm)* exploits the authorization predicate $\Pi, \mathcal{I}' \vdash \mathcal{I} \diamond \mathcal{J}$ to check whether the policy Π in force at $\mathcal{I}'$ authorizes interaction between $\mathcal{I}$ and $\mathcal{J}$ and, if so, infers the computation step.

The label $\mathcal{I} : t \,\bar{\lhd}\, \mathcal{J}$, generated by rule *(accget)*, denotes the willingness of a component $\mathcal{J}$ to provide t to a component $\mathcal{I}$. When $\mathcal{J}.id = n$, its complementary label is $\mathcal{I} : t \lhd n$ generated by rule *(pr-sys)* when a component $\mathcal{I}$ wants to withdraw t from the repository at n. When the target of the action denotes a remote repository, rule *(syncget)*, the action is only allowed if $\mathcal{J}.id = n$, namely if n is the name of the component with interface $\mathcal{J}$. The semantics of action **qry** is modelled by rules *(lqry)*, *(accqry)* and *(syncqry)*. This action behaves similarly to **get**, the only difference being that it invokes the retrieval operation of the repository's handling mechanism, rather than the withdrawal operation. Thus, if the action succeeds, the repository after the computation step remains unchanged. Action **put** adds item t to a repository. Its behavior is modelled by rules (namely *(lput)*, *(accput)* and *(syncput)*) similar to those of actions **get** and **qry**, the major difference being now that the addition operation of the repository's handling mechanism is invoked. In any case, for remote synchronisation to take place, it could require authorisation through the application of rule *(enscomm)*.

Finally, rule *(async)* allows a whole system to asynchronously evolve when only some of its components evolve.

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

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

– the set of states $\mathcal{S}$ includes all the systems defined in Table 1;
– the transition relation $\succ\!\!\longrightarrow$ is the least relation induced by the inference rules in Table 4. As a matter of notation, we will write $S \succ\!\!\longrightarrow S'$ instead of $\langle S, S' \rangle \in \succ\!\!\longrightarrow$. Moreover, $\bar{n}$ denotes a (possibly empty) sequence of names and $\bar{n}, n'$ is the sequence obtained by composing $\bar{n}$ and n'. $(\nu\bar{n})S$ abbreviates $(\nu n_1)((\nu n_2)(\cdots(\nu n_m)S\cdots))$, if $\bar{n} = n_1, n_2, \cdots, n_m$, and S, otherwise. $S\{n'/n\}$ denotes the system obtained by replacing any free occurrence in S of n with n'. When considering a system S, a name is deemed *fresh* if it is different from any name occurring in S.

The rules defining the transition relation are straightforward. The first rule accounts for the computation steps of a system where all (possible) name restrictions are at top level, while the last two rules permit to manipulate the syntax of

Table 4. Semantics of systems: transition relation

$$\frac{S \xrightarrow{\tau} S'}{(\nu\bar{n})S \succ\!\!\longrightarrow (\nu\bar{n})S'}\ \textit{(res-tau)}$$

$$\frac{(\nu\bar{n}, n'')(S_1 \parallel S_2\{n''/n'\}) \succ\!\!\longrightarrow S' \quad n''\ \textit{fresh}}{(\nu\bar{n})(S_1 \parallel (\nu n')S_2) \succ\!\!\longrightarrow S'}\ \textit{(res-top-r)}$$

$$\frac{(\nu\bar{n}, n'')(S_1\{n''/n'\} \parallel S_2) \succ\!\!\longrightarrow S' \quad n''\ \textit{fresh}}{(\nu\bar{n})((\nu n')S_1 \parallel S_2) \succ\!\!\longrightarrow S'}\ \textit{(res-top-l)}$$

Table 5. Semantics of systems: inter-ensemble communication

$$\frac{S_1 \xrightarrow{\lambda_1} S_1' \qquad S_2 \xrightarrow{\lambda_2} S_2'}{S_1 \parallel S_2 \xrightarrow{\lambda_1 \diamond \lambda_2} S_1' \parallel S_2'}\ \textit{(ens1)}$$

$$\frac{S \xrightarrow{\lambda'} S' \qquad \Pi, \mathcal{I} \vdash \lambda' \succ \lambda}{\mathcal{I}[\mathcal{K}, \Pi, P] \parallel S \xrightarrow{\lambda} \mathcal{I}[\mathcal{K}, \Pi, P] \parallel S'}\ \textit{(ens2)}$$

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\lambda_1} C \qquad S \xrightarrow{\lambda_2} S' \qquad \Pi, \mathcal{I} \vdash \lambda_1 \diamond \lambda_2 \succ \lambda}{\mathcal{I}[\mathcal{K}, \Pi, P] \parallel S \xrightarrow{\lambda} C \parallel S'}\ \textit{(ens3)}$$

a system, by moving all name restrictions at top level, thus putting it into a form to which the first rule 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).

On inter-ensemble communication. According to the semantics, two components can interact only if they are part of the same ensemble. Here, we tune the semantics for permitting more complex interaction patterns among two or more components, possibly belonging to different ensembles, by exploiting interaction predicates to regulate them. Therefore, first we extend system labels as follows

$$\lambda \ ::= \ \ldots \ \big| \ \lambda_1 \diamond \lambda_2$$

where label $\lambda_1 \diamond \lambda_2$ denotes the concurrent execution of those transitions corresponding to labels λ_1 and λ_2. Then, we add the rules in Table 5 to the operational rules for systems in Table 3.

Basically, the idea is to generalize the mechanism already present in the operational semantics of systems (see e.g. rule *(enscomm)* regulating intra-ensemble communications), by replacing the authorization predicate $\Pi, \mathcal{I} \vdash \lambda$ with predicate $\Pi, \mathcal{I} \vdash \lambda' \succ \lambda$. The latter, while checking whether a transition can be allowed according to the policy Π in force at $\mathcal{I}$, also translates label λ' into λ.

On ensemble-wide broadcast communication. In [8], we also present an extension of SCEL enabling a sort of multicast communication where the potential recipients are all the members of the ensembles of which the sender is part of. Communication is anonymous and takes place through the coordinators of these ensembles. Due to lack of space, we refer the reader to [8] for a complete account.

5 How to 'Cook' Your Own SCEL Dialect

In this section, we show how dialects of SCEL can be easily defined by appropriately specifying the parameters of the language. As a concrete example, we demonstrate how KLAIM [9] can be obtained.

In order to define a dialect with specific features, one has to fix the parameters SCEL depends on, that is

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

Now, to get KLAIM as a dialect of SCEL, we can make the following choices.

Policies must express KLAIM *allocation environments*. These are functions associating logical names to physical names (i.e. addresses) of the different components, thus regulating components visibility and establishing systems architecture. Therefore, policies are rendered as functions from variables to names. As interaction predicate, we take the *interleaving* one (see [8] for details), which is obtained by interpreting controlled composition as the interleaved parallel composition of the two involved processes, while, as authorisation predicate, we take a predicate that does not block any action.

Knowledge items are sequences of values, i.e. *tuples*, while templates are sequences of values and variables. More generally, a value can result from the evaluation of some given expression e belonging to an appropriate language of EXPRESSIONS. KLAIM in turn is parametric with respect to the language of expressions (we assume that it contains *strings* and *integers*).

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

In practice, we can complete the syntax of KNOWLEDGE, ITEMS and TEMPLATES as shown in Table 6, where e denotes an EXPRESSION producing values.

If we were interested in capturing alternative versions of KLAIM that, e.g., use types to enforce access control (see, e.g. [10]), we can simply add these types, i.e. functions from names to sets of *capabilities*, to the language of policies.

Table 6. Tuple-based SCEL (e is an EXPRESSION)

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

As for component interfaces, the only meaningful attribute is *id* which is set to the name of the component, while attributes *ensemble* and *membership* are set to true and no other attribute is used. At this point, **get**/**qry**/**put**/**new** correspond to KLAIM's **in**/**read**/**out**/**newloc**, while KLAIM's **eval**, that permits to spawn a new process for execution possibly on a remote component, can be rendered in SCEL by means of an appropriate protocol relying on higher-order communication and on action **exec** (see, e.g. [11]).

6 SCEL at Work

In this section we show how the SCEL dialect defined in Section 5 can be used to model a simple yet illustrative example. In particular, we will mainly focus on the *goal-oriented* interaction among SCs and SCEs that are the novelty of our proposal.

In our application scenario, we consider a collection of service components, *all* offering the same services. Each component manages and elaborates service requests with different policies, roughly summarized by the following three quality levels: *gold*, *silver* and *base*. These policies are defined via a combination of predicates on the hardware configuration and the runtime state. For example, the runtime state can give a measure of the number of service requests currently handled locally. The parameters of the different policies are identified by suitable attributes of the component interfaces. In particular, we assume that the tuples $\langle$"*attr*", "*hw*", $i\rangle$ and $\langle$"*attr*", "*load*", $p\rangle$ are stored in the local tuple space of each component. For example, value i (an integer in $[0, 10]$) in the tuple $\langle$"*attr*", "*hw*", $i\rangle$ gives an indication of the capacity of the hardware configuration of the component; while value p (an integer in $[0, 100]$) in the tuple $\langle$"*attr*", "*load*", $p\rangle$ estimates the actual computational load of the component. Notice that the hardware measure is static while the load estimate is updated whenever a component receives or completes a service request.

Each service component also publishes in its interface the signature of the available services through suitable attributes. Here we assume that *aService* is the name of the available service and requires a *string* as input parameter and yields a *string* value as a result. Furthermore, additional information about the client and the session has to be provided when the service is invoked.

Service components constitute three ensembles depending on the *quality of service* they can provide. In particular, we consider three components, named c_g, c_s and c_b, each of which coordinates the service components operating at *gold*, *silver* and *base* level, respectively. Each of these components acts as a *proxy* for the replicated services. The ensemble coordinator accepts client invocations and allows service components to retrieve them. Then, the invoked service component sends the obtained results back to the client component.

Since ensemble aggregation is goal-oriented, the following predicates

- $S_g(\mathcal{I}) = \mathcal{I}.hw \geq 7$
- $S_s(\mathcal{I}) = (\mathcal{I}.hw \geq 4) \wedge (S_g(\mathcal{I}) \rightarrow \mathcal{I}.load < 40)$
- $S_b(\mathcal{I}) = (S_s(\mathcal{I}) \rightarrow \mathcal{I}.load < 40) \wedge (S_g(\mathcal{I}) \rightarrow \mathcal{I}.load < 20)$.

are assigned to attribute *ensemble* of the three coordinating components c_g, c_s and c_b, respectively. Thus, the *gold* ensemble identifies a gold component by the high measure of its hardware configuration (value greater or equal to 7). The *silver* ensemble is less demanding: a component has to provide an hardware configuration with a level that is at least 4 and, whenever a component provides a hardware configuration that is valued more than 7, the computational load must be less than 40% ($\rightarrow$ stands for logical implication). This last condition guarantees that gold components can handle requests at silver level only when their computational load is under 40%. The same schema is used to define the *base* ensemble. Of course, all the components, independently of their hardware level, can be part of this ensemble. However, gold and silver components are involved only when their computational load is under 20% and 40% respectively.

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

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

$$\mathbf{put}(\text{“}invoke\text{”}, \text{“}aService\text{”}, v, c, s)@u.\mathbf{get}(\text{“}result\text{”}, \text{“}aService\text{”}, !x, c, s)@\mathsf{self}.P$$

The client posts the invocation in the tuple space of the coordinator of the ensemble (u is a variable assuming value among c_b, c_s or c_g). Value v is the required input string, while the pair c, s provides the bookkeeping information: c is the client name and s is a value representing the working session. After issuing the invocation, the client waits for the result (recall that action **get** is blocking). Whenever the result of the service invocation is made available, the client can withdraw it from the local tuple space and continue as process P.

Processes running at service components execute the following code fragment:

$$\begin{array}{l}\mathbf{get}(\text{“}invoke\text{”}, \text{“}aService\text{”}, !Param, !Client, !Session)@u.\\ \mathbf{get}(\text{“}attr\text{”}, \text{“}load\text{”}, !x)@\mathsf{self}.\\ \mathbf{put}(\text{“}attr\text{”}, \text{“}load\text{”}, (x+5))@\mathsf{self}.\\ \mathbf{exec}(Q)\end{array}$$

The process is triggered by a client request retrieved from the coordinator's repository. Whenever this happens, the computational load is updated[1] and the

[1] Here we assume each service instance uses 5% of the component computational resources.

process Q, which actually computes the result of the invoked service "*aService*", is executed. We assume that, before its termination, process Q updates the value of attribute *load* and puts the result of the computation into the tuple space of the requesting client.

The application scenario discussed above exploits different forms of communication. First, the invoking client uses inter-ensemble communication for putting its request in the coordinator's repository. Then, the service component uses standard (intra-ensemble) communication to retrieve the request from the coordinator's repository. The processing of the request increases the computational load of the component, which may cause the service component to leave the ensemble where the service request has been retrieved. Therefore, when the service completes, the result is sent back to the client's repository by using inter-ensemble communication. Afterwards, the result can be retrieved by the client through local communication.

7 Adaptation in SCEL

In this section we argue that adaptation can be naturally expressed in SCEL. As we have seen in Section 3, the knowledge repository of components can contain both application data and awareness data. At this level of abstraction, we are not concerned with the way data are actually represented, we only assume that they can be appropriately tagged to distinguish awareness data from application data. This distinction is indeed crucial, as it is at the basis of a tangible notion of *adaptation* [12], which is defined as the run-time modification of awareness data. A component is then deemed *adaptive* if it has a precisely identified collection of awareness data that are modified at run-time, at least in some of its computations. Besides, it is *self-adaptive* if it is able to modify its own awareness data at run-time.

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

The one outlined above is perhaps the simplest form of adaptation, but we can envisage more sophisticated forms by taking the nature of the awareness data into account. Suppose, for example, that the process part of a component is split into an *autonomic manager* controlling execution of a *managed element*. The autonomic manager monitors the state of the component, as well as the execution context, and identifies relevant changes that may affect the achievement of its goals or the fulfillment of its requirements. It also plans adaptations in order to meet the new functional or non-functional requirements, executes them, and monitors that its goals are achieved, possibly without any interruption[2].

[2] The whole body of activities mentioned above has been named MAPE-K loop (Monitoring, Analyzing, Planning, and Executing, through the use of Knowledge) [2].

In practice, the autonomic manager implements the rules for adaptation. Now, by exploiting SCEL higher-order features, namely the capability to store/retrieve (the code of) processes in/from the knowledge repositories and to dynamically trigger execution of new processes (by means of action **exec**), it is e.g. possible to dynamically replace (part of) the managed element process or even the autonomic manager process. In this case, we are also changing the rules, i.e. processes, with which the awareness data are manipulated, since these rules are represented as awareness data themselves.

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

$$\begin{aligned} ME \triangleq\ & \mathbf{qry}(\text{``}\mathit{required\_functionality\_id}\text{''}, !X)@\mathsf{self}.\\ & \mathbf{get}(\text{``}\mathit{required\_functionality\_id\_args}\text{''}, !y, !z)@\mathsf{self}.\\ & \mathbf{exec}(X(y,z)).\\ & \mathbf{get}(\text{``}\mathit{wait\_termination\_id}''')@\mathsf{self}.ME \end{aligned}$$

Items containing processes or parameters can be thought of as awareness data. Autonomic managers can add/remove/replace these data from the knowledge repositories thus implementing the adaptation logic and therefore changing the managed element behavior. For example, different versions of the process providing a requested service may exist. While managed elements could only read these data, the autonomic manager could dynamically change the association between the service request and the service process by simply performing:

$$\begin{aligned} & \mathbf{get}(\text{``}\mathit{required\_functionality\_id}\text{''}, !X)@\mathsf{self}.\\ & \mathbf{put}(\text{``}\mathit{required\_functionality\_id}\text{''}, Q)@\mathsf{self}. \end{aligned}$$

which has the effect of replacing the 'old' service implementing the functionality *id* with a possibly new one Q.

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

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

$$\begin{aligned} & \mathbf{qry}(\mathit{pid}, \text{``}\mathit{ko}\text{''})@\mathsf{self}.\mathbf{put}(\mathit{pid}, \text{``}\mathit{dead}\text{''})@\mathsf{self}.\mathbf{nil}\\ + & \ \mathbf{qry}(\mathit{pid}, \text{``}\mathit{ok}\text{''})@\mathsf{self}.P \end{aligned}$$

This would allow an autonomic manager to send a termination request to the process with identifier *pid* and wait for its termination, assuming that both items (*pid, "ok"*) and (*pid, "ko"*) are used for coordination purposes.

$$\mathbf{get}(pid, \text{``}ok\text{''})@\mathsf{self}.\mathbf{get}(pid, \text{``}dead\text{''})@\mathsf{self}$$

As we have seen, it is the autonomic manager to choose *which* adaptation to use. The decision about *when* to perform adaptation is jointly taken by the autonomic manager and the application designer. This is reminiscent of another approach, named *context-oriented programming* (COP) [13]. COP is a novel programming paradigm introduced to manage and control adaptivity of programs. It allows developers to define *behavioral variations*, chunks of code that can be activated depending on the current working environment (the context), to dynamically modify program execution and thus adapt to its environment. In this approach, the application designer has to insert *adaptation hooks* in the application code and is thus able to control when adaptation can take place. Leaving the designer to specify where and when to adapt has its advantages, because adaptations would be explicit in the code and thus more visible, and the application designer could better plan some adaptations. However, not being transparent to the application designer has significant disadvantages, because only adaptations planned at design phase could be exploited. When the autonomic computing approach is used, the autonomic manager, which continuously monitors awareness data or event occurrences, reacts to changes of contexts or of goals.

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

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

8 Related Work

The term "ensemble" has been recently introduced in the literature (see, e.g., [3,1,14]) to denote a category of systems characterized by heterogeneous collections of computing resources, huge number of potential interactions, context-awareness, dynamically changing network topologies, and unreliable communications. A mathematical model of ensembles and their composition has been introduced in [15]. Ensembles and their constituent parts are abstractly described as relations on sets of inputs and outputs. The "black-box" view of adaptivity is then formally defined. This leads to a preorder relation on ensembles which captures the the ability of ensembles to satisfy goals or maximize a performance measure in different environments. Differently from this denotational model, we introduce an operational model of ensembles and a formal language that allows the description of ensembles in a compact and formal way. A language for programming ensembles, named Meld, has been proposed in [16,17]. Meld is a declarative language originally designed for programming overlay networks. It allows ensembles to be programmed as a unified whole from a global perspective and then compiled automatically into fully distributed local behaviors. This approach is somehow reminiscent of Declarative Networking [18], a programming methodology that supports the high level specification of network protocols and services, that are then compiled into a dataflow framework and executed. SCEL, instead, is a formal language that could be used as the core of a programming language for ensembles.

The way in which ensembles are characterized in SCEL resembles the way software elements are dynamically grouped into homogenous clusters in [19], where an architecture for the design of component-based distributed self-adaptive systems is outlined. Indeed, both approaches adopt application-specific metrics. Each cluster is headed by a distinguished component, in charge of supervising it and of gathering information from the rest of the system. The supervision mechanism also identifies situations that trigger adaptations.

Context-Oriented Programming (COP) [13] can also be used to write ensemble applications [20]. It exploits *ad hoc* explicit language-level abstractions to express context-dependent behavioral variations and, notably, their run-time activation. So far, most of the efforts in the field of COP have been directed towards the design and implementation of concrete languages. Only a few papers in the literature provide a foundational account of programming languages extended with COP facilities, as e.g. the object-oriented ones of [21,22,23] and the functional one of [24]. All these approaches are however quite different from ours, that instead focusses on distribution and goal-oriented aggregations and supports a highly dynamic notion of adaptation.

Several works have been proposed that use formal techniques to model autonomic computing systems. For example, [25] presents an approach to develop an autonomic service-oriented architecture. This and other examples (e.g., [26,27]), however, focus on the use of formal techniques for specific target applications.

Our work, instead, aims at introducing general techniques to achieve autonomicity rather than at modeling specific autonomic systems. SCEL formal semantics permits to better understand how autonomicity is obtained.

Core languages designed in the area of concurrency theory are natural candidates for the specification of autonomic systems. Many such formalisms aim at modelling dynamically changing network topologies, a feature common to many types of distributed systems and to ensembles. For example, CWS [28] deals with communication aspects that are specific of wireless communications, while the ω-calculus [29] addresses the modeling problems of mobile ad-hoc networks. We want also to mention [30], that uses the Gamma formalism, a computing model inspired by the chemical reaction metaphor, to develop a higher-order language for specifying autonomic systems, and [31], that presents a biochemical calculus expressive enough to represent adaptive systems, together with a formal framework for property checking.

9 Concluding Remarks and Future Directions

We have introduced SCEL, a new language that brings together various programming abstractions that permit directly representing *knowledge*, *behaviors* and *aggregations* according to specific *policies*, and naturally programming interaction, adaptation and self- and context-awareness. Our *language-based* approach enables us to govern the complexity of the issues under consideration by imposing a structure over the variety of computational entities involved. A further advantage is that all programming abstractions are based on solid semantic grounds. This lays the basis for developing logics, tools and methodologies for formal reasoning about system behavior in order to establish qualitative and quantitative properties of both the individual components and the ensembles.

We are currently assessing to which extent SCEL achieves its goals, i.e. modeling the behavior of service components and their ensembles, their interactions, their sensitivity and adaptivity to the environment. As testbeds we will use three case studies from three different application domains: Robotics (collective transport), Cloud-computing (transiently available computers), and e-Mobility (cooperative e-vehicles). This process might require tuning the language features. After this, we plan to implement SCEL, possibly by exploiting the distributed software framework IMC [32].

We also want to develop a methodology that enables components to take decisions about possible alternative behaviors by choosing among the best possibilities while being aware of the consequences. By relying on an abstract description of the evolving environment, each component will be able to verify locally the possibility (or the probability) of guaranteeing the wanted properties or of achieving the wanted goals by analyzing the possible outcome of its interactions with the abstract model. This kind of information will then be used to take decisions about the choices that a component has to face. This abstract description is not fixed but may change according to the interactions of the component with the rest of the system or as a consequence of the changes in the, possibly imprecise, contextual information in which the entity is currently running.

Our proposal combines notions from different research fields. This will permit the cross fertilization of concepts and techniques. For instance, in the long run, we expect that analytical methods typical of the so called *big data* science can be fruitfully adopted to discover aggregation patterns and, consequently, predict behavior of highly complex SCEs. Understanding how aggregations of SCs may evolve is a key issue for developing optimization techniques.

References

1. Hölzl, M., Rauschmayer, A., Wirsing, M.: Software Engineering for Ensembles. In: Wirsing, M., Banâtre, J.-P., Hölzl, M., Rauschmayer, A. (eds.) Software-Intensive Systems. LNCS, vol. 5380, pp. 45–63. Springer, Heidelberg (2008)
2. IBM: An architectural blueprint for autonomic computing. Technical report, 3rd edn. (June 2005)
3. Project InterLink (2007), `http://interlink.ics.forth.gr/central.aspx`
4. Project ASCENS (2010), `http://www.ascens-ist.eu/`
5. Wirsing, M., Hölzl, M., Tribastone, M., Zambonelli, F.: ASCENS: Engineering Autonomic Service-Component Ensembles. In: Beckert, B., de Boer, F., Bonsangue, M., Damiani, F. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 1–24. Springer, Heidelberg (2012)
6. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I & II. Inf. Comput. 100(1), 1–77 (1992)
7. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
8. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: Languages primitives for coordination, resource negotiation, and task description. ASCENS Deliverable D1.1 (September 2011), `http://rap.dsi.unifi.it/scel/`
9. 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)
10. Gorla, D., Pugliese, R.: Dynamic management of capabilities in a network aware coordination language. J. Log. Algebr. Program. 78(8), 665–689 (2009)
11. De Nicola, R., Gorla, D., Pugliese, R.: On the expressive power of klaim-based calculi. Theor. Comput. Sci. 356(3), 387–421 (2006)
12. Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., Vandin, A.: A Conceptual Framework for Adaptation. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 240–254. Springer, Heidelberg (2012)
13. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3), 125–151 (2008)
14. Want, R., Schooler, E., Jelinek, L., Jung, J., Dahle, D., Sengupta, U.: Ensemble computing: Opportunities and challenges. Intel Technology Journal 14(1), 118–141 (2010)
15. Hölzl, M., Wirsing, M.: Towards a System Model for Ensembles. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 241–261. Springer, Heidelberg (2011)
16. Ashley-Rollman, M.P., Goldstein, S.C., Lee, P., Mowry, T.C., Pillai, P.: Meld: A declarative approach to programming ensembles. In: IROS, pp. 2794–2800. IEEE (2007)

17. Ashley-Rollman, M.P., Lee, P., Goldstein, S.C., Pillai, P., Campbell, J.D.: A Language for Large Ensembles of Independently Executing Nodes. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 265–280. Springer, Heidelberg (2009)
18. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. Commun. ACM 52(11), 87–95 (2009)
19. Baresi, L., Guinea, S., Tamburrelli, G.: Towards decentralized self-adaptive component-based systems. In: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2008, pp. 57–64. ACM, New York (2008)
20. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A programming paradigm for autonomic systems. CoRR abs/1105.0069 (2011)
21. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (2001)
22. Hirschfeld, R., Igarashi, A., Masuhara, H.: ContextFJ: a minimal core calculus for context-oriented programming. In: Proceedings of the 10th International Workshop on Foundations of Aspect-Oriented Languages, FOAL 2011, pp. 19–23. ACM, New York (2011)
23. Clarke, D., Costanza, P., Tanter, E.: How should context-escaping closures proceed? In: Proc. of COP 2009, pp. 1:1–1:6. ACM, New York (2009)
24. Degano, P., Ferrari, G.-L., Galletta, L., Mezzetti, G.: Types for Coordinating Secure Behavioural Variations. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 261–276. Springer, Heidelberg (2012)
25. Bhakti, M.A.C., Azween, A.: Formal modeling of an autonomic service oriented architecture. In: CSIT, vol. 5, pp. 23–29. IACSIT Press (2011)
26. Li, Z., Parashar, M.: Rudder: An agent-based infrastructure for autonomic composition of grid applications. Multiagent and Grid Systems 1(3), 183–195 (2005)
27. Dong, X., Hariri, S., Xue, L., Chen, H., Zhang, M., Pavuluri, S., Rao, S.: Autonomia: an autonomic computing environment. In: IPCCC, pp. 61–68. IEEE (2003)
28. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. Electr. Notes Theor. Comput. Sci. 158, 331–353 (2006)
29. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Sci. Comput. Program. 75(6), 440–469 (2010)
30. Banâtre, J.P., Radenac, Y., Fradet, P.: Chemical Specification of Autonomic Systems. In: IASSE, pp. 72–79. ISCA (2004)
31. Andrei, O., Kirchner, H.: A Higher-Order Graph Calculus for Autonomic Computing. In: Lipshteyn, M., Levit, V.E., McConnell, R.M. (eds.) Graph Theory, Computational Intelligence and Thought. LNCS, vol. 5420, pp. 15–26. Springer, Heidelberg (2009)
32. Bettini, L., De Nicola, R., Falassi, D., Lacoste, M., Loreti, M.: A Flexible and Modular Framework for Implementing Infrastructures for Global Computing. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 181–193. Springer, Heidelberg (2005)