www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D1.1: First Report on WP1

Language Primitives for Coordination, Resource Negotiation, and Task Description

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **7.6.2010**

Lead contractor for deliverable: **IMT**
Author(s): **Rocco De Nicola (IMT), Gianluigi Ferrari (UNIPI), Michele Loreti (UDF), Rosario Pugliese (UDF)**

Due date of deliverable: **September 30, 2011**
Actual submission date: **November 15, 2011**
Revision: **Final**
Classification: **PU**

Project coordinator: **Martin Wirsing (LMU)**
Tel: **+49 89 2180 9154**
Fax: **+49 89 2180 9175**
E-mail: **wirsing@lmu.de**

Partners: **LMU, UNIPI, UDF, Fraunhofer, UJF-Verimag, UNIMORE, ULB, EPFL, VW, Zimory, UL, IMT, Mobsya, CUNI**

SEVENTH FRAMEWORK PROGRAMME

Executive Summary

SCEL (Service Component Ensemble Language) is a new language, specifically designed to program autonomic components and their interaction while supporting formal reasoning on their behaviors. SCEL brings together various programming abstractions that permit directly representing *aggregations*, *behaviors* and *knowledge* according to specific *policies*. Moreover, it permits naturally programming interaction, self- and context-awareness, and adaptation. SCEL relies on solid semantics grounds that lay the basis for developing 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 overall systems.

In this deliverable, SCEL design principles are presented and after introducing the main constructs of the language it is discussed how component interfaces and attributes are modeled and it is explained how they can be exploited to model ensembles. The operational semantics of SCEL is then presented together with sophisticated semantic mechanisms to control components interaction and to offer the possibility of ensemble-wide broadcast interaction. It is also demonstrated that adaptation can be naturally modeled in SCEL and an example is given of how a dialect can be defined by appropriately instantiating some of the open features of the proposed syntax.

Contents

1 Introduction

The behaviors of autonomic components, their interactions, their sensitivity to the environment and their adaptivity could be programmed in any of the existing programming languages, even in Assembly.

However, given the intricacy of the issues under consideration and the need to foresee the emergent behavior of many interacting agents and to guarantee that specific functionalities are offered, it would be better to resort to a programming language such that notions like: Component, Interaction, Interface, Distribution, Mobility, Knowledge, Awareness, Adaptation are first class elements. One would then avoid resorting to elaborate constructions to model them.

Moreover, in many cases, a very few assumptions about the operating environment can be made. The environment is frequently open, in the sense that other components may join and, in some cases, might even be hostile. This might might lead to unintended behaviors of the autonomic components and to the loss of valuable information. It is thus essential to be able to program interaction and access policies directly in order to have a finer control on all components.

Finally, it is essential that functional and non-functional properties of the modeled systems be guaranteed; it is thus important that the used language be based on a solid semantic ground. Only this could permit the development of methodologies and software tools supporting formal reasoning and thus establishing qualitative (e.g., functional correctness) and quantitative (e.g., optimal use of resources) properties of individual components and ensembles.

To address the above mentioned issues, we are designing SCEL, a new language that brings together various linguistic abstractions permitting a direct representation of *Aggregations*, *Behaviors* and *Knowledge* and provides specific *Policies* for naturally programming interaction, self-awareness, context-awareness, and adaptation.

The abstractions relative to *Knowledge* describe how knowledge is represented and handled. The abstractions relative to *Behaviors* describe how components progress. The abstractions relative to *Aggregations* describe how different entities are brought together to form components, systems and, possibly, ensembles. The abstractions relative to *Policies* deal with the way properties of computations are represented and enforced. Moreover, sophisticated mechanisms are introduced to control components interaction and to offer the possibility of ensemble-wide broadcast interaction.

By building on the tradition of process algebras and coordination languages, all proposed abstractions are equipped with a structural operational semantics that helps in clarifying their meaning and lays the basis for building the appropriate structures on which to perform program analysis.

In this deliverable, we introduce SCEL and discuss possible alternative design choices. The rest of this document is organized as follows. In Section 2 we introduce SCEL design principles and in Section 3 we present its syntax. In Section 4 we illustrate component interfaces, while in Section 5 we explain how ensembles are rendered in SCEL. In Section 6 we define the operational semantics of SCEL and in Section 7 we show how it is affected by interaction predicates. In Section 8 we provide an example of how a dialect can be easily defined by appropriately specifying the parameters of the language, while in Section 9 we demonstrate how adaptation can be expressed in SCEL. In Section 10 we present a smooth extension of the language with a more powerful interaction mechanism. The document ends with Section 11 that contains a few concluding remarks and sketches the Work Plan for next years.

2 SCEL: design principles

SCEL brings together various programming abstractions that provide us with the linguistic constructs to define *software architectures of autonomic systems.* Indeed, we could say that any language for autonomic service-component ensembles would greatly benefit from having abstractions for representing *Knowledge*, *Behaviors* and *Aggregations*, according to specific *Policies*.

- The abstractions relative to *Knowledge* describe how knowledge is managed. We do distinguish between *knowledge representation* and *knowledge handling* mechanisms. We assume that knowledge is represented through items stored in repositories. Some of these items contain *application data*, namely data used for the progress of the computation carried out by the components, while other ones contain *control data*, namely data providing 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 the current position or about the remaining battery's charge level). We also assume that the handling mechanism of each knowledge repository provides then three abstract operations that can be used by autonomic components for
  - *adding* new knowledge to the repository,
  - *retrieving* information from the repository,
  - *withdrawing* information from the repository.
- The abstractions relative to *Behaviors* describe how the computation progresses and are modeled as processes in the style of process calculi. *Interaction* is modeled by allowing components to access to knowledge repositories. *Adaptation* is modeled by retrieving both information about the changing context and suggestions about the code to be executed as a reaction to these changes from the knowledge repository.
- The abstractions relative to *Aggregations* describe how different entities are brought together to form *components*, *systems* and, possibly, *ensembles* and permit modeling allocation and distribution of resources. Thus, each component might have a private knowledge repository that, depending on the chosen policies, might be accessible by others. In this way, the notion of *administrative domains* (sets of resources and computations of a given entity under the control of a specific authority) can be modeled. Compositionality and interoperability are supported by *interfaces*, that specify attributes and functionalities provided and/or required by components.
- The abstractions relative to *Policies* deal with the way properties of computations are represented and enforced. Interaction and Service Level Agreement (SLA) provide two standard examples of policy abstractions. Other examples are security properties maintaining the appropriate links between data values and their usage policies (*data-leakage policies*) or limiting the flow of sensitive information to untrusted sources (*access control* and *reputation policies*).

Figure 1 summarizes the main ingredients of SCEL. The knowledge manager $\mathcal{K}$ is further structured in two components, see Figure 2.

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 helps in determining the part of KNOWLEDGE to be added, retrieved

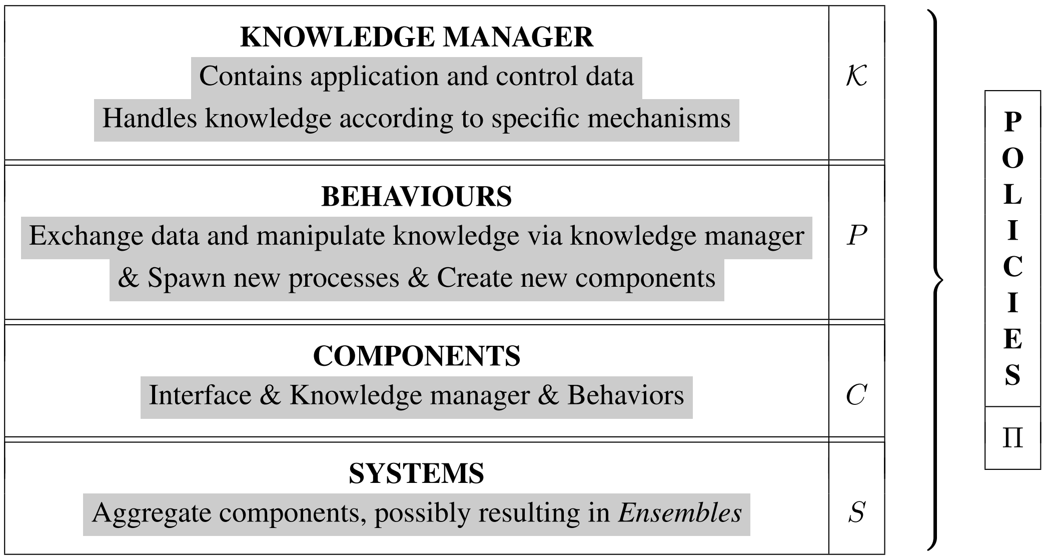
KNOWLEDGE MANAGER
Contains application and control data
Handles knowledge according to specific mechanisms
$\mathcal{K}$
BEHAVIOURS
Exchange data and manipulate knowledge via knowledge manager
& Spawn new processes & Create new components
P
COMPONENTS
Interface & Knowledge manager & Behaviors
C
SYSTEMS
Aggregate components, possibly resulting in *Ensembles*
S
POLICIES
Π

Figure 1: SCEL abstractions

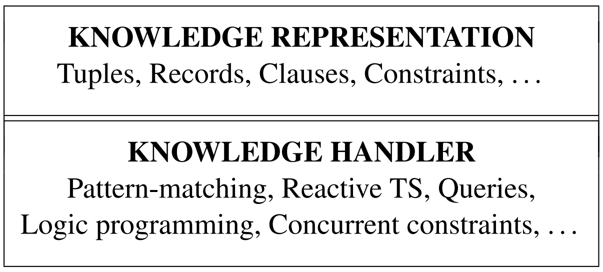
KNOWLEDGE REPRESENTATION
Tuples, Records, Clauses, Constraints, ...
KNOWLEDGE HANDLER
Pattern-matching, Reactive TS, Queries,
Logic programming, Concurrent constraints, ...

Figure 2: The Knowledge Manager

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 is worth remarking that, might be surprisingly, no syntactic category for ensembles is present. Indeed, to better support their dynamicity, ensembles are determined via the attributes of the different COMPONENTS.

Let us now consider one by one the different syntactic 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,
- *parameterized process invocation*: $A(\bar{p})$,
- *parameterized process definition*: $A(\bar{f}) \triangleq P$.

The construct $P_1[\,P_2\,]$ can be seen as an abstract way of modeling the various forms of parallel composition of P_1 and P_2 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

SYSTEMS:
$S ::= C \mid S_1 \parallel S_2 \mid (\nu n)S$

COMPONENTS:
$C ::= \mathcal{I}[\mathcal{K}, \Pi, P]$

KNOWLEDGE:
$\mathcal{K} ::= \ldots$

POLICIES:
$\Pi ::= \ldots$

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

ITEMS:
$t ::= \ldots$

TEMPLATES:
$T ::= \ldots$

Table 1: SCEL syntax

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 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 processes executing it if the knowledge repository does not (yet) contain, cannot 'produce' or cannot infer 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 and are used to insert new items in the knowledge repository, to spawn new processes, and to create new names and/or limit their scope.

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

Every *component* C includes

1. an *interface* $\mathcal{I}$ containing information about the component itself. The interface is represented

KNOWLEDGE:
$\mathcal{K} \ ::= \ \langle t \rangle \ \big| \ \mathcal{K}_1 \parallel \mathcal{K}_2$

ITEMS:
$t \ ::= \ e \ \big| \ c \ \big| \ P \ \big| \ t_1, t_2$

TEMPLATES:
$T \ ::= \ e \ \big| \ c \ \big| \ P \ \big| \ !\,x \ \big| \ !\,X \ \big| \ T_1, T_2$

Table 2: Tuple-based SCEL

by a set of names of attributes and provided functionalities. It is required that at least the attributes *id*, *ensemble* and *membership* are present in any component interface;

2. a *knowledge manager* $\mathcal{K}$ providing local, and possibly part of the global, knowledge (i.e. control data) in addition to the application data, together with a 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* term P together with a set of process definitions that can be dynamically activated. Some of the processes composing P perform the local computation, while others may coordinate processes interaction with the knowledge repository and/or deal with the issues related to adaptation and reconfiguration.

Systems aggregate components by means of 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 *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 name restriction in π-calculus [MPW92a, MPW92b]).

It has to be said that the syntax of SCEL leaves some ingredients unspecified. They can be chosen according to the specific application domain or to the taste of the language designer. For instance, if we borrow the KLAIM [DFP98] idea of using tuples for knowledge representation and tuple spaces as knowledge repositories, we can complete the syntax of Knowledge, Items and Templates as shown in Table 2. Other syntactical ingredients are, however, still underspecified. These represent additional language features that need to be introduced to express policies of various kinds (e.g. to regulate knowledge handling, resource usage, process execution, process interaction, actions priority, security, trust, reputation), and to define the expressions producing values (and the corresponding evaluation mechanisms). In the sequel we will use notation from Table 2 to make concrete examples of knowledge representation and selection.

4 Interfaces

The *interface* of a component contains attributes and functionalities provided by the component.

Attributes are used to provide names for specific features and are represented as pairs of the form (*name*,*value*). If we borrow the notation of Table 2 and if attribute x of component n is currently associated to value v, then the knowledge repository at n contains an item of the form $\langle$"$attr$", "x"$, v\rangle$.

Attribute values can thus be dynamically changed through the specific knowledge handling mechanism. The interface of a component C must contain at least the following three attributes:

- id: the name of the component C;
- $ensemble$: a formula or a predicate on interfaces used to determine the actual components of the ensemble created and coordinated by C;
- $membership$: a formula or a predicate on the interfaces used to determine the ensembles to which C is willing to be member of.

Additional attributes might, e.g., indicate the battery's charge level, the component's GPS position, the operating environment's control data. Attribute selection will be modeled by means of structured names, thus, if $\mathcal{I}$ is the interface of component C, $\mathcal{I}.id$ indicates its name.

A functionality is a behavior (defined through a *process definition*) that is made available by a component for external invocation. As an example let us consider a component named $stack$, implementing a stack data structure that provides two functionalities, one named $push$ and the other named pop. These names would then be included in the interface of $stack$ while two items $\langle$"$pfun$", "$push$", "in"$\rangle$ and $\langle$"$pfun$", "pop", "out"$\rangle$, indicating that functionality $push$ has an input parameter while functionality pop has an output parameter, are inserted in the knowledge repository associated to component $stack$. Items of this form represent the signature of a functionality (they could also include information on the type of each parameter) and can be retrieved by potential clients. Once a client knows how to invoke a specific functionality, we expect that it adds a suitably tagged item containing the actual parameters to the repository of the component providing the functionality. The body of the functionality is such that its first action withdraws this item to appropriately initialize its formal parameters, then starts computing. Results are transmitted by following to a similar protocol. In our example, assuming that pattern matching is used as a retrieval mechanism, the protocol for the actual invocation of $push$ to add value v to $stack$, would be:

- the invoker has to perform action: $\mathbf{put}($"$invoke$", "$push$"$, v)@stack$;
- the definition of functionality $push$ at $stack$, if P is the actual implementation of this functionality, has to be of the form: $push(x) \triangleq \mathbf{get}($"$invoke$", "$push$"$, !x)@\mathsf{self}.P$.

5 Ensembles

SCEL has no specific syntactic construct for defining ensembles. Instead, ensembles are dynamically formed by exploiting the fact that an interface specifies not only what the component provides but also, via, e.g., attributes *membership* and *ensemble*, what it requires to the other partners. This design choice of having 'synthesized' ensembles dynamically determined supports high dynamicity and flexibility in forming, joining and disjoining ensembles, permits to avoid structuring ensembles through rigid syntactic constructs, and provides additional control on the communication capabilities of components (and of the processes therein).

In the sequel, we shall use notation $\mathcal{I} \models \mathcal{J}.ensemble$ to indicate that $\mathcal{J}$ is willing to accept component $\mathcal{I}$ in the ensemble it coordinates and $\mathcal{J} \models \mathcal{I}.membership$ to indicate that $\mathcal{I}$ is willing to be one of the components of the ensemble coordinated by $\mathcal{J}$. We shall 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.

For example, the names of the components that can be members of an ensemble can be explicitly mentioned, as in the predicate

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

Now, if the interface $\mathcal{J}$ of a component C has attribute $(ensemble, P(\mathcal{I}))$, then a component C' with interface $\mathcal{I}'$ is part of the ensemble coordinated by C if $P(\mathcal{I}')$ holds, namely if the name of C' is n, m or p.

Predicate P can be instantiated so to characterize the members of an ensemble by requiring that they are active and have a battery charge level not less than 30%, as in the predicate

$$P(\mathcal{I}) \overset{def}{=} \mathcal{I}.active = yes \wedge \mathcal{I}.battery\_level \geq 30\%$$

We are assuming here that in the interface of each component willing to be part of the ensemble there are the attributes $active$ and $battery\_level$ indicating if the component is active and storing its battery charge level.

An ensemble can be also determined by such a predicate as

$$P(\mathcal{I}) \overset{def}{=} range_{max} \geq \sqrt{(\mathsf{self}.x - \mathcal{I}.x)^2 + (\mathsf{self}.y - \mathcal{I}.y)^2}$$

stating that the ensemble is composed by those components within a given range from the component coordinating the ensemble (that is the one setting up the attribute $ensemble$). Here we are assuming that in the interface of each component there are the attributes x and y storing the numerical values of the coordinates of a Cartesian system specifying the position in a plane of the corresponding component.

Components, in turn, could be willing to be part of any ensemble, which is expressed by letting attribute $membership$ be associated to predicate $true$, or, on the contrary, they could not want to be part of any ensemble, which is expressed by letting $membership$ be associated to $false$. More generally, components can put restrictions on the ensembles which they are willing to be member of by appropriately setting the attribute $membership$. For example, using the following predicate

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

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

6 SCEL: operational semantics

The operational semantics is given in the SOS style [Plo04] by relying on the notion of Labeled Transition System (LTS), that 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 labeled 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 will make use of the sets $bv(E)$ and $fv(E)$ of *bound* and *free* variables and the sets $n(E)$, $bn(E)$ and $fn(E)$ of names, *bound* names and *free* names, respectively, 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, 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 $(\nu n)\_$

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

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

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

Table 3: Operational semantics of processes

binds the name n in the scope _. A term without free variables is deemed *closed* (notice that it may contain free names).

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

6.1 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 (as we will see in Section 6.2). 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

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

  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 labeled transition relation $\longmapsto$ is the least relation induced by the inference rules in Table 3. To simplify notation, 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 labeled 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 synchronize. Indeed, thanks to the third rule, that allows any process to perform a $\circ$-labeled 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 will be then specialized at the level of systems by means of interaction predicates in order to also take polices into account (see Section 6.2). 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 4). Notably, also this condition is not strict: it can be always made true by application of the last rule saying 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.

6.2 Operational semantics of systems

The operational semantics of systems is defined in two steps. First, we define an LTS to derive the transitions of systems without restricted names. Notice that, although name restrictions do not appear in the system before a transition, they can appear in the system obtained after a transition, as in rule *(newc)*. Then, by exploiting this LTS, we provide the semantics of generic systems by means of a (unlabeled) transition system (TS), that is a pair $\langle \mathcal{S}, \rightarrowtail \rangle$ made of a set of states $\mathcal{S}$ and a (unlabeled) 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 permits 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[1]). It also permits a smooth extension of the syntax and of the operational semantics of the language to consider more powerful interaction mechanisms (see Section 10). To simplify notation, we will use $\mathcal{I}$ and $\mathcal{J}$ to range over interfaces. 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

- $\mathcal{S}$ is the set of states containing all and only the systems defined in Table 1.
- $\mathcal{L}$ is the set of transition labels generated by the following production rule[2]

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

  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

[1] If we would like to modify the above defined LTS to capture the semantics of systems with restricted names, its definition would become quite tricky because of the handling of name mobility. Indeed, if we open and close the scope of restricted names when they are subject to communication by using standard techniques (as in π-calculus [MPW92a, MPW92b]), we should tackle the problem of determining when we can safely close the scope of a (restricted) name exported by a component through a communication: this may not be trivial as the component may want the name to be received by all the ensembles which it is part of and we must guarantee that all of them can actually receive the same restricted name. The two step approach we are presenting in this section is a way to get around this problem.

[2] We would like to remark that although c stands for a variable, a name or self, in the production rules for labels definition it can be only a name, n, or self.

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 labeled transition relation induced by the inference rules in Table 4. We will write $S \xrightarrow{\lambda} S'$ instead of $\langle S, \lambda, S' \rangle \in \longrightarrow$.

The labeled transition relation relies on the following two predicates:

- *interaction predicate* $\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma$: under policy Π and interface $\mathcal{I}$, process label α yields system label λ and substitution σ.

- *authorization predicate* $\Pi, \mathcal{I} \vdash \lambda$: 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 rôle 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[3] to capture well-known process computation and interaction patterns such as interleaving, asynchronous communication, synchronous communication, full synchrony, broadcasting, etc. We refer the reader to Section 7 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:

- $\mathcal{K} \ominus t = \mathcal{K}'$: 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 four system labels: $\tau, \mathcal{I} : \mathbf{new}(\mathcal{J}, \mathcal{K}, \Pi, P)$, $\mathcal{I} : t \triangleleft c, \mathcal{I} : t \blacktriangleleft c$ and $\mathcal{I} : t \triangleright c$. 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 (Π) needed to decide if the action can be allowed or

[3]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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 4: Semantics of systems: labeled transition relation (symmetric of rules *(syncget)*, *(syncqry)*, *(syncput)*, *(enscomm)* and *(async)* omitted)

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 visibility 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 vice versa. Then, the (conditional) premise $ens(\mathcal{I}, \mathcal{J}) \Rrightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}$ 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 $(\mathcal{I} \models \mathcal{I}'.ensemble \ \wedge \ \mathcal{I}' \models \mathcal{I}.membership) \wedge (\mathcal{J} \models \mathcal{I}'.ensemble \ \wedge \ \mathcal{I}' \models \mathcal{J}.membership)$. We now take advantage of the 'else' case of the premise $ens(\mathcal{I}, \mathcal{J}) \Rrightarrow \lambda = \tau, \lambda = \mathcal{I} \diamond \mathcal{J}$ 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 modeled 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 modeled 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 synchronization to take place, it could require authorization through the application of rule *(enscomm)*.

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

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

- the set of states includes all the systems defined as in Table 1;

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

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

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

Table 5: Semantics of systems: transition relation

- the transition relation $\rightarrowtail$ is the least relation induced by the inference rules in Table 5. As a matter of notation, we will write $S \rightarrowtail S'$ instead of $\langle S, S' \rangle \in \rightarrowtail$. 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. Basically, 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 a system, by moving all name restrictions at top level, in order to put 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. We can also further modify the semantics to permit more complex interaction patterns among two or more components, possibly, belonging to different ensembles, in the style of those expressed trough interaction predicates. This can be done simply by

- extending 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;

- adding the following rules to the set of operational rules for systems

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

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

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

Basically, the idea is to generalize the mechanism already present in the operational semantics of systems, 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. This mechanism enables more complex interaction patterns, e.g., it could consider some details as in rule *(enscomm)* regulating intra-ensemble communications.

7 Interaction Predicates

The operational semantics introduced in Section 6 (and later revised in Section 10) relies on the interaction predicate[4]

$$\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma$$

This predicate establishes a relation between a given triple, consisting of a policy Π, an interface $\mathcal{I}$ and a process label α, and a pair, consisting of a system label λ and a substitution σ. 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. An interaction predicate then permits defining sophisticated policies for regulating the interaction among processes of a component, while possibly taking other policies (e.g. for access control) into account. However, for the sake of simplicity, we will be only concerned with policies Π controlling process interaction.

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

The following notations will be used:

- $\mathcal{E}[\![\, t \,]\!]_{\mathcal{I}}$ (resp. $\mathcal{E}[\![\, T \,]\!]_{\mathcal{I}}$) denotes the evaluation of item t (resp. template T) with respect to interface $\mathcal{I}$: attributes occurring in t (resp. T) are replaced by the corresponding value in $\mathcal{I}$;
- $\mathcal{N}[\![\, c \,]\!]_{\mathcal{I}}$ denotes the evaluation of target c according to interface $\mathcal{I}$;
- $\mathcal{P}[\![\, P \,]\!]_{\mathcal{I}}$ denotes the evaluation of P according to interface $\mathcal{I}$: functionalities in P are *replaced* by the corresponding code in $\mathcal{I}$.

[4]Interaction predicates are reminiscent of *synchronization algebras* introduced by Glynn Winskel in a seminal paper on Event Structures [Win86] as a device to specify how events from parallel processes do synchronize, thus associating with any synchronization algebra a particular parallel composition.

$$\Pi_\oplus, \mathcal{I} : \mathbf{exec}(P) \succ \tau, [] \qquad \frac{\mathcal{E}[\![\, T \,]\!]_\mathcal{I} = T' \quad \mathcal{N}[\![\, c \,]\!]_\mathcal{I} = n \quad match(T', t) = \sigma}{\Pi_\oplus, \mathcal{I} : \mathbf{get}(T)@c \succ \mathcal{I} : t \lhd n, \sigma}$$

$$\frac{\mathcal{E}[\![\, T \,]\!]_\mathcal{I} = T' \quad \mathcal{N}[\![\, c \,]\!]_\mathcal{I} = n \quad match(T', t) = \sigma}{\Pi_\oplus, \mathcal{I} : \mathbf{qry}(T)@c \succ \mathcal{I} : t \blacktriangleleft n, \sigma} \qquad \frac{\mathcal{E}[\![\, t \,]\!]_\mathcal{I} = t' \quad \mathcal{N}[\![\, c \,]\!]_\mathcal{I} = n}{\Pi_\oplus, \mathcal{I} : \mathbf{put}(t)@c \succ \mathcal{I} : t' \rhd n, []}$$

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

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

Table 6: Interleaving interaction predicate $\Pi_\oplus, \mathcal{I} : \alpha \succ \lambda, \sigma$

Interleaving. The interaction predicate *interleaving*, denoted by $\Pi_\oplus$, is obtained by interpreting controlled composition as the *interleaved* parallel composition of the two involved processes. The inference rules defining predicate $\Pi_\oplus, \mathcal{I} : \alpha \succ \lambda, \sigma$ are reported in Table 6. We have a rule for each different kind of process action, plus two additional rules (the last ones) ensuring that in case of controlled composition of multiple processes only one process can perform an action (the other stays still). The (five) rules for process actions basically state that, at the level of the operational semantics of systems, process action **exec** corresponds to a computation step τ, while the other actions correspond to properly labeled transitions.

Monitoring. The *monitoring* interaction predicate, denoted by $\Pi_\otimes[\Pi_1, \Pi_2]$, can be used to ensure that in a controlled composition $P[\, Q \,]$, P can actually control the actions performed by Q. This makes controlled composition a non-commutative operator, differently from the interleaving interaction predicate $\Pi_\oplus$ described before.

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

Predicate $\Pi_\otimes[\Pi_1, \Pi_2]$ is actually a predicate 'schema' as it is parametric with respect to predicates Π_1 and Π_2 defining the interaction policies of processes P and Q. The monitoring predicate could be combined with the interleaving predicate to obtain more refined interaction patterns for processes.

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

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

$$\frac{\Pi_1, \mathcal{I} : \alpha \succ \mathcal{I} : t \triangleleft n, \sigma_1 \quad \Pi_2, \mathcal{I} : \beta \succ \mathcal{I} : t \triangleright n, \sigma_2}{\Pi_\otimes[\Pi_1, \Pi_2], \mathcal{I} : \alpha[\,\beta\,] \succ \tau, \sigma_1 \cdot \sigma_2}$$

$$\frac{\Pi_1, \mathcal{I} : \alpha \succ \mathcal{I} : t \blacktriangleleft n, \sigma_1 \quad \Pi_2, \mathcal{I} : \beta \succ \mathcal{I} : t \triangleright n, \sigma_2}{\Pi_\otimes[\Pi_1, \Pi_2], \mathcal{I} : \alpha[\,\beta\,] \succ \mathcal{I} : t \triangleright n, \sigma_1 \cdot \sigma_2}$$

$$\frac{\Pi_2, \mathcal{I} : \alpha \succ \lambda, \sigma}{\Pi_\otimes[\Pi_1, \Pi_2], \mathcal{I} : \circ[\,\alpha\,] \succ \lambda, \sigma} \; \lambda \neq \mathcal{I} : t \triangleright n$$

$$\frac{\Pi_1, \mathcal{I} : \alpha \succ \lambda, \sigma}{\Pi_\otimes[\Pi_1, \Pi_2], \mathcal{I} : \alpha[\,\circ\,] \succ \lambda, \sigma} \; \lambda \neq \mathcal{I} : t \triangleleft n \wedge \lambda \neq \mathcal{I} : t \blacktriangleleft n$$

Table 7: Monitoring interaction predicate $\Pi_\otimes[\Pi_1, \Pi_2], \mathcal{I} : \alpha \succ \lambda, \sigma$

$$\frac{\Pi_1, \mathcal{I} : \alpha \succ \mathcal{I} : t \triangleleft n, \sigma_1 \quad \Pi_2, \mathcal{I} : \beta \succ \mathcal{I} : t \triangleright n, \sigma_2}{\Pi_N[\Pi_1, \Pi_2], \mathcal{I} : \alpha[\,\beta\,] \succ \tau, \sigma_1 \cdot \sigma_2} \; n \in N$$

$$\frac{\Pi_1, \mathcal{I} : \alpha \succ \mathcal{I} : t \blacktriangleleft n, \sigma_1 \quad \Pi_2, \mathcal{I} : \beta \succ \mathcal{I} : t \triangleright n, \sigma_2}{\Pi_N[\Pi_1, \Pi_2], \mathcal{I} : \alpha[\,\beta\,] \succ \mathcal{I} : t \triangleright n, \sigma_1 \cdot \sigma_2} \; n \in N$$

$$\frac{\Pi_2, \mathcal{I} : \alpha \succ \lambda, \sigma}{\Pi_N[\Pi_1, \Pi_2], \mathcal{I} : \circ[\,\alpha\,] \succ \lambda, \sigma} \; \lambda \notin \{\mathcal{I} : t \triangleright n | n \in N\}$$

$$\frac{\Pi_1, \mathcal{I} : \alpha \succ \lambda, \sigma}{\Pi_N[\Pi_1, \Pi_2], \mathcal{I} : \alpha[\,\circ\,] \succ \lambda, \sigma} \; \lambda \notin \{\mathcal{I} : t \triangleleft n, \mathcal{I} : t \blacktriangleleft n | n \in N\}$$

Table 8: Limited monitoring interaction predicate $\Pi_N[\Pi_1, \Pi_2], \mathcal{I} : \alpha \succ \lambda, \sigma$

The inference rules defining the predicate $\Pi_N[\Pi_1, \Pi_2], \mathcal{I} : \alpha \succ \lambda, \sigma$, are reported in Table 8. The rules extend those of $\Pi_\otimes[\Pi_1, \Pi_2]$ by adding new *side conditions*. The first two rules guarantee that synchronization only occur on labels involving names in N, while the last two rules model the case processes Q and P, in a process of the form $P[\,Q\,]$, can evolve independently. Again, the basic labels are dealt with exactly as in the case of the interleaving interaction predicate, that is by the first five rules of Table 6.

Remarks. The monitoring and the limited monitoring interaction predicates provide just a few examples of the expressive power of interaction predicates. It is not difficult to envisage more general situations where, e.g., actions performed by Q in a controlled composition of the form $P[\,Q\,]$ are intercepted by suitable actions by P and appropriately transformed into labels at the level of systems. This allows P to act as a sort of 'execution monitoring' for Q and is somehow reminiscent of the approach for enforcing security policies that relies on the so called *security automata* [Sch00].

All the interaction predicates we have considered, and the interaction policies they define, are static: they do never change for taking into account the progress of a system. More sophisticated interaction policies could be defined by allowing to change the predicate after any application, for example by defining judgments of the form $\Pi, \mathcal{I} : \alpha \succ \Pi', \lambda, \sigma$ where Π' is the predicate to be applied next time. We could also allow action **exec** to install new interaction policies: it could additionally have an argument specifying the interaction predicate in charge of regulating interactions within the controlled process and another argument specifying the interaction predicate regulating the overall controlled composition. Likewise, action **new** could have an additional argument specifying the policies regulating the interaction with the new component.

8 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 [DFP98] 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 languages for policies Π, together with an *interaction predicate* $\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma$ and an *authorization predicate* $\Pi, \mathcal{I} \vdash \lambda$;
2. the language for representing knowledge items and repositories, together with the three operations, i.e. withdrawal, query and addition, that we assumed each knowledge repository's handling mechanism must provide;
3. the languages for specifying the expressions e producing values and the corresponding evaluation mechanisms.

Now, to get KLAIM as a dialect of SCEL, we can take respectively

1. the languages for policies has to include functions from (name) variables to names for expressing KLAIM *allocation environments*, namely sort of architectural policies regulating visibility of components within a system, and while, as interaction predicate, we can take the interleaving one (introduced in Section 7) and, as authorization predicate, we can take the one that does not block any action;
2. tuples, multisets, templates and *pattern-matching* as mechanisms for representing, storing and selecting information (see also Table 2);
3. any language for specifying value expressions (also KLAIM leaves this unspecified).

If we were interested in capturing alternative version of KLAIM that use types to enforce access control (see, e.g. [GP09]), we can take functions from names to *capabilities* to express the constraints enforced by types.

As regards component interfaces, in any of them we can set attributes $ensemble$ and $membership$ to true and use no other attribute; in other words, the only meaningful attribute is id which is set to the name of the corresponding component. At this point, **get**/**qry**/**put**/**new** correspond to KLAIM's **in**/**read**/**out**/**newloc**, while KLAIM's remote **eval** can be rendered in SCEL by means of an appropriate protocol exploiting higher-order communication and the **exec** (see, e.g. [DGP06]).

9 Adaptation in SCEL

As we have seen in the previous sections, SCEL is parametric with respect to the knowledge manager. Indeed, knowledge is *abstractly* represented through items stored in repositories which are then handled by suitable mechanisms. We only require that the handling mechanism of knowledge repositories provides the processes with three operations for managing knowledge, namely for adding knowledge items and for retrieving/withdrawing knowledge items.

That knowledge items can contain both application data, namely data used by the processes for the progress of the computation, and control data, namely data providing information about the environment in which the components are running (e.g. monitored data from sensors) and the current status of a component (e.g. its position or its battery charge level). Both kinds of data are part of the knowledge of components. 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 control data from application data.

This distinction is crucial. Indeed, it provides the basis of a tangible notion of *adaptation* [BCG+11], which is defined as the run-time modification of control data. A component is then said to be *adaptive* if it has a precisely identified collection of control data that are modified at run-time, at least in some of its computations. A component is *self-adaptive* if it is able to modify its own control data at run-time. These definitions fit a more general vision of adaptation [HRW08] defined as "the capability of a system to change its behavior according to new requirements or environment conditions" and proposed as the key for *autonomic computing* (i.e. computer and software systems that can manage themselves in accordance with high-level guidance from humans by relying on strategies inspired by biological systems).

In general, a component in SCEL is adaptive (and, hence, autonomic) because its control data can be dynamically modified by means the actions **put**/**get**/**qry**. Moreover, a component is self-adaptive as the hosted process can trigger modifications of its control 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 control 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[5]. 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 control data are manipulated, since these rules are represented as control data themselves.

A managed element can be seen as an empty "executor" which retrieves from the knowledge repository the process P to execute and bounds it to a variable X, performs an **exec** to send P for execution and waits until it terminates (this coordination can be worked out by exchanging appropriate synchronization items). Also actual parameters for processes can be stored as knowledge items and retrieved by the executor (or by the process itself) when needed (see below)

$$\begin{array}{rcl} ME & \triangleq & \mathbf{qry}(``required\_functionality\_id", !P)@\mathsf{self}. \\ & & \mathbf{get}(``required\_functionality\_id", !args)@\mathsf{self}. \\ & & \mathbf{exec}(P(args)). \\ & & \mathbf{get}(``wait\_P\_termination'')@\mathsf{self}.ME \end{array}$$

[5]The whole body of activities mentioned above has been named MAPE-K loop (Monitoring, Analyzing, Planning, and Executing, through the use of Knowledge) by IBM [IBM05].

Items containing processes or parameters can be thought of as control 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:

$\mathbf{get}(\text{“}required\_functionality\_id\text{”}, !P1)@\mathsf{self};$
$\mathbf{put}(\text{“}required\_functionality\_id\text{”}, P2)@\mathsf{self};$

Of course, the autonomic manager can also add a new service or even remove an existing one. Notably, the autonomic manager 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 and, possibly, signal its termination.

$\mathbf{qry}(pid, \text{“}ko\text{”})@\mathsf{self}.\mathbf{put}(pid, \text{“}dead\text{”})@\mathsf{self}.\mathbf{nil}$
$+$
$\mathbf{qry}(pid, \text{“}ok\text{”})@\mathsf{self}.P$

where P is the continuation of the proceed identified by *pid*. 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 tags *pid, “ok”* and *pid, “ko”* are present.

$\mathbf{get}(pid, \text{“}ok\text{”})@\mathsf{self};$ *//remove the life item*
$\mathbf{get}(pid, \text{“}dead\text{”})@\mathsf{self};$ *//wait for termination*

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. It is useful to relate our approach with, *context-oriented programming* (COP) [SGP11] that exploits *ad hoc* explicit language-level abstractions to express context-dependent behavioral variations and, notably, their run-time activation. 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 adaptation planned at design phase could be exploited. When the autonomic computing approach is used, the autonomic manager, which continuously monitors control data or event occurrences, reacts to changes of contexts or of goals.

In addition to the language-level adaptation used in COP, an architectural approach 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 this coarse-grained approach since component's membership of 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.

In case of distributed applications one can plan to have (i) control data residing at autonomic element 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 control data

$$\text{TARGETS:}\quad c \;::=\; n \;\big|\; x \;\big|\; \mathsf{self} \;\big|\; \mathsf{super}$$

Table 9: The syntax of SCEL with ensemble-based communication

and global autonomic processes. The distributed approach may give raise to 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.

10 Extending SCEL with ensemble-wide broadcast communication

We now consider an extension of the language where a process can indicate as a target of actions **put** and **qry** the ensembles of which its hosting component is part of. This extension enables a component to insert an item within the repository of all the ensembles that contain it (action **put**). It also allows to nondeterministically retrieve an item from any of such repositories (action **qry**). Since a component might not know the name of all the ensembles which it is a member of (membership is dynamic and determined by attributes), it uses the reserved keyword super to refer to them.

The extension of SCEL syntax, that leads to what we call SCEL$_e$, is obtained by enriching the TARGETS syntactic category with the keyword super as illustrated in Table 9. However, we only allow the new target to be used for actions **put** and **qry**, not for action **get**[6].

The operational semantics of the new language follows the same pattern of the one for original SCEL and like before it is defined in two steps by defining first process commitments and then the semantics of systems. This latter description is again done in two steps by first introducing a labeled transition relation and then an unlabeled one that is built from the former by fully taking into account name restrictions. As before, the semantics is only defined for closed systems.

10.1 Operational semantics of processes

The semantics of the richer processes specifies 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. The actions that SCEL$_e$ processes can perform are those reported in Table 1 extended with

$$\mathbf{put}(t)@\mathsf{super} \quad \text{and} \quad \mathbf{qry}(T)@\mathsf{super}$$

and transition labels α and β are changed accordingly. After these changes for actions and transition labels, the semantics of the new processes remains exactly as in Table 3.

10.2 Operational semantics of systems

The LTS defining the semantics of systems without restricted names is defined as

[6] This is a design decision that we might reconsider. However, while we consider natural information broadcasting ($\mathbf{put}(t)@\mathsf{super}$), we have difficulties in conceiving a simultaneous withdrawal of information from distributed repositories ($\mathbf{get}(T)@\mathsf{super}$).

- The set of states as defined in Table 1 possibly using also super as target of **put** and **qry**.
- The same set of transition labels of SCEL semantics plus:

  $$\mathcal{I} : t \triangleright \mathsf{super} \text{ and } \mathcal{I} : t \blacktriangleleft \mathsf{super} \text{ and } \mathcal{I} : t \,\bar{\triangleright}\, \star$$

  where the first two have the interpretation as in Section 6 while $\mathcal{I} : t \,\bar{\triangleright}\, \star$ indicates that component $\mathcal{I}$ is allowed to add item t to the repository of each coordinator of ensembles which it is part of.
- The labeled transition relation $\longrightarrow$ is the least relation induced by the inference rules in Tables 10 and 11.

The actual semantic rules for systems have been split in two tables for the sake of presentation by dividing the rules for **put** from the others. This is because action **put** has the greatest impact on the semantics and requires adding a number of new rules to properly deal with broadcasting of information.

Like before, the labeled transition relation relies on the predicates $\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma$ and $\Pi, \mathcal{I} \vdash \lambda$ and on the three operations on knowledge repositories $\mathcal{K} \ominus t = \mathcal{K}'$, $\mathcal{K} \vdash t$ and $\mathcal{K} \oplus t = \mathcal{K}'$.

Now, let us comment on the new rules (within gray boxes) in the two tables. Rule *(superqry)* describes the possibility of the acting process to enlarge any of its local searches for matching items to the ensemble containing it. This is the natural complement of the broadcasting **put** that inserts its argument into the repository of all the ensembles enclosing it[7]. Rule *(synchqry-g)* exploits the possibility offered by *(superqry)* and permits synchronization between a component, with interface $\mathcal{I}$, looking for information and a component, with interface $\mathcal{J}$, offering the item only if the former is a member of the ensemble defined by the latter.

Let us now consider the new rules of Table 11. Action **put**, whenever its target is super, can add item t to the repository of all the components defining an ensemble of which the component performing the action is part *(syncputens)*. A component $\mathcal{J}$ can perform a transition labeled $\mathcal{I} : t \,\bar{\triangleright}\, \star$ when it accepts the item t from a component $\mathcal{I}$ which is part of the ensemble defined by $\mathcal{J}$ *(accputenssucc)*. A transition with the same label can also be performed (rule *(accputensfail)*) by those components $\mathcal{J}$ that either do not allow the action (i.e. $\Pi, \mathcal{J} \not\vdash \mathcal{I} : t \,\bar{\triangleright}\, \star$) or do not include $\mathcal{I}$ in the ensemble they define: in this case, however, the component is not affected by the transition. The complementary label now is $\mathcal{I} : t \triangleright \mathsf{super}$, generated by rule *(pr-sys)* when a component $\mathcal{I}$ wants to add t to the repository of all the ensembles which it is part of. Hence, the transition resulting from the synchronization triggered by an action **put** is either a computation step $\mathcal{I} : t \triangleright \mathsf{super}$, when many components are possibly allowed to receive the item t produced by component $\mathcal{I}$, or a computation step τ.

Finally, rule *(async)* allows a whole system to asynchronously evolve when only some of its components do evolve, provided that the transition label is not of the form $\mathcal{I} : t \,\bar{\triangleright}\, \star$ or $\mathcal{I} : t \triangleright \mathsf{super}$. Rule *(sync)*, instead, forces all the components of a system to make a transition with the same label $\mathcal{I} : t \,\bar{\triangleright}\, \star$: notably, any component can always perform such a transition since this label is generated by rules

[7] *A possible alternative*: In the rules in Table 10, a local **qry** is automatically transformed into a **qry** at super apart from the fact that the required item is found or not in the local repository. This means that the search for the item is nondeterministically done in the local repository and in the repositories of all the ensembles of which the component performing the action is part of. If we want to guarantee that the search is done first in the local repository and then, only if no matching item is locally found, the search is repeated in the repositories of the including ensembles, then we can replace rule *(superqry)* with the following one:

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

where $\neg(\mathcal{K} \vdash t)$ is the negation of $\mathcal{K} \vdash t$ and means that t cannot be retrieved from $\mathcal{K}$.

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

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

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

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

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

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

$$\frac{S \xrightarrow{\mathcal{I}:t\blacktriangleleft n} S' \qquad n = \mathcal{I}.id}{S \xrightarrow{\mathcal{I}:t\blacktriangleleft \mathsf{super}} S'}\ \textit{(superqry)}$$

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

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

$$\frac{S_1 \xrightarrow{\mathcal{I}:t\blacktriangleleft \mathsf{super}} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \bar{\blacktriangleleft} \mathcal{J}} S_2' \qquad \mathcal{I} \models \mathcal{J}.ensemble \qquad \mathcal{J} \models \mathcal{I}.membership}{S_1 \parallel S_2 \xrightarrow{\tau} S_1' \parallel S_2'}\ \textit{(syncqry-g)}$$

Table 10: Semantics of systems: labeled transition relation, part I (symmetric of rules *(syncget)*, *(syncqry)* and *(syncqry-g)* omitted)

(accputenssucc) and *(accputensfail)* that have complementary premises. This guarantees that whenever a component with interface $\mathcal{I}$ wants to add an item t to the repositories of all the ensembles which it is part of, all (and only) such ensembles effectively add it to their repository.

Now, the transition system defining the semantics of generic systems is essentially the same as the one for the simpler calculus. Only we have to consider that computation steps may additionally be of

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

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

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

$$\frac{\mathcal{K} \oplus t = \mathcal{K}' \qquad \Pi, \mathcal{J} \vdash \mathcal{I} : t \bar{\triangleright} \star \qquad \mathcal{I} \models \mathcal{J}.ensemble \qquad \mathcal{J} \models \mathcal{I}.membership}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t\bar{\triangleright}\star} \mathcal{J}[\mathcal{K}', \Pi, P]}\;\textit{(accputenssucc)}$$

$$\frac{\Pi, \mathcal{J} \not\vdash \mathcal{I} : t \bar{\triangleright} \star \quad \vee \quad \mathcal{I} \not\models \mathcal{J}.ensemble \quad \vee \quad \mathcal{J} \not\models \mathcal{I}.membership}{\mathcal{J}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t\bar{\triangleright}\star} \mathcal{J}[\mathcal{K}, \Pi, P]}\;\textit{(accputensfail)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t\triangleright\mathsf{super}} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t\bar{\triangleright}\star} S_2'}{S_1 \parallel S_2 \xrightarrow{\mathcal{I}:t\triangleright\mathsf{super}} S_1' \parallel S_2'}\;\textit{(syncputens)}$$

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

$$\frac{S_1 \xrightarrow{\lambda} S_1' \qquad \lambda \neq \mathcal{I} : t \bar{\triangleright} \star \qquad \lambda \neq \mathcal{I} : t \triangleright \mathsf{super}}{S_1 \parallel S_2 \xrightarrow{\lambda} S_1' \parallel S_2}\;\textit{(async)}$$

$$\frac{S_1 \xrightarrow{\mathcal{I}:t\bar{\triangleright}\star} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t\bar{\triangleright}\star} S_2'}{S_1 \parallel S_2 \xrightarrow{\mathcal{I}:t\bar{\triangleright}\star} S_1' \parallel S_2'}\;\textit{(sync)}$$

Table 11: Semantics of systems: labeled transition relation, part II (symmetric of rules *(syncput)*, *(syncputens)*, *(enscomm)*, *(async)* and *(sync)* omitted)

the form $S \xrightarrow{\mathcal{I}:t\triangleright\mathsf{super}} S'$ and thus we need to transform them into transitions of the form $S \rightarrowtail S'$.

$$\frac{S \xrightarrow{\tau} S'}{(\nu\bar{n})S \rightarrowtail (\nu\bar{n})S'}\ \textit{(res-tau)} \qquad \frac{S \xrightarrow{\mathcal{I}:t\triangleright\mathsf{super}} S'}{(\nu\bar{n})S \rightarrowtail (\nu\bar{n})S'}\ \textit{(res-super)}$$

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

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

Table 12: Semantics of systems: transition relation

11 Work Plan for Year Two

The work description of the technical Annex relative to WP1, mainly concerned with the development of SCEL, mentions four main lines of research that concern the modeling of:

1. The behaviors of components and their ports and their interactions;

2. The topology of the network needed for interaction, taking into account resources, locations and visibility/reachability issues;

3. The environment where components operate and resource-negotiation takes place, taking into account open ended-ness and the need of adaptation;

4. The global knowledge of the systems together with the description of the tasks to be accomplished by the different ensembles, the properties to be guaranteed and the constraints to be respected.

During the first year, we have concentrated on the first two research items. In the second year we will, on the one hand, assess and improve the work relative to them and, on the other hand, start tackling the other two research items.

In particular, we will continue with the work on different interaction policies and we will consider the outcome of WP5 and study the possibility of taking an approach similar to the one followed in BIP for modeling synchronization and communication.

To experiment with different models of knowledge and with different mechanisms for knowledge handling, we will consider different kinds of knowledge repositories based, e.g., on concurrent constraints or on unification. While, to model components and ensembles goals, we will study the impact of the knowledge representation languages and of the handling mechanisms developed in WP3.

To improve components sensitivity and adaptivity to the environment, we will experiment with the adaptation patterns designed in WP4. While, to deal with resources negotiations and uncertain behaviors and to guide components strategies, we will consider stochastic variants of the language and will explore the possibility of using probabilistic model checking to help support components in taking decisions.

All this blending will be supported and validated by the throughout use of the foundational models developed in WP2, that will be used prescriptively to assess our design choices. Moreover, our linguistic choices will be validated by using as testbeds the three case studies considered in WP7, dealing with different application domains: Robotics (collective transport), Cloud-computing (transiently available computers) and e-Mobility (cooperative e-vehicles).

Of course, this process might require tuning the language features and its enrichment with constructs to support architectural descriptions. Once the language will be considered sufficiently stable, we shall start with its implementation, possibly by exploiting the distributed software framework IMC developed in previous EU projects. The actual implementation, possibly together with programming supporting tools, will be developed in collaboration with WP6. In parallel with the implementation of the language, we will proceed with the development of software tools, or with the tuning of existing ones, in order to provide the necessary software support to formal reasoning on systems behavior, and for establishing qualitative and quantitative properties of both the individual components and the ensembles.

References

[BCG$^+$11] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. A conceptual framework for adaptation. Manuscript, 2011.

[DFP98] Rocco De Nicola, GianLuigi Ferrari, and Rosario Pugliese. Klaim: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.

[DGP06] Rocco De Nicola, Daniele Gorla, and Rosario Pugliese. On the expressive power of klaim-based calculi. *Theor. Comput. Sci.*, 356(3):387–421, 2006.

[GP09] Daniele Gorla and Rosario Pugliese. Dynamic management of capabilities in a network aware coordination language. *J. Log. Algebr. Program.*, 78(8):665–689, 2009.

[HRW08] Matthias Hölzl, Axel Rauschmayer, and Martin Wirsing. Software engineering for ensembles. In *Software-Intensive Systems and New Computing Paradigms*, pages 45–63. Springer, 2008.

[IBM05] IBM. An architectural blueprint for autonomic computing. Technical report, June 2005. Third edition.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[SGP11] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A programming paradigm for autonomic systems. *CoRR*, abs/1105.0069, 2011.

[Win86] Glynn Winskel. Event structures. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1986.