ascens

www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D1.2: Second Report on WP1

Languages for Coordinating Ensemble Components

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **Version 2.2 (30.7.2011)**

Lead contractor for deliverable: **UDF**
Author(s): **Rosario Pugliese (UDF) - editor, Tomáš Bureš (CUNI), Rocco De Nicola (IMT), Jaroslav Keznikl (CUNI), Michele Loreti (UDF), František Plášil (CUNI), Francesco Tiezzi (IMT)**

Reporting Period: **2**
Period covered: **October 1, 2011 to September 30, 2012**
Submission date: **November 12, 2012**
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 allow one to directly represent behaviors, knowledge and aggregations according to specific policies. It also supports naturally programming self- and context-awareness, and adaptation of components often operating in open, highly dynamic, distributed environments. The solid semantic grounds of the language 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.

This deliverable contains a short summary of our work on:

- SCEL: a refined version of the language;
- jRESP: a Runtime Environment for SCEL Programs;
- SACPL: a SCEL Access Control Policy Language;
- ccSCEL: a SCEL Dialect for Concurrent Constraint Programming;
- Towards High-level Design of SCEL-based Applications.

Contents

1 Introduction

Ensembles represent the future generation of software-intensive systems dealing with massive numbers of components, featuring complex interactions among components and with humans and other systems, operating in open and non-deterministic environments, and dynamically adapting to new requirements, technologies and environmental conditions. This definition has been proposed by the Interlink WG on software intensive systems and new computing paradigms [Int07].

The challenge for language designers is to devise appropriate abstractions and linguistic primitives to deal with:

- the large dimension of the systems,
- the need to adapt to (possibly unpredicted) changes of the working environment and to evolving requirements,
- the emergent behaviors resulting from complex interactions.

We are thus looking for methodologies and linguistic constructs that can be used to build ensembles while combining traditional software engineering approaches, techniques from autonomic, adaptive, knowledge-based and self-aware systems, and formal methods, in order to guarantee compositionality, expressiveness and verifiability.

To tame the complexity of ensemble-based computer systems, the notions of *service components* (SCs) and *service-component ensembles* (SCEs) have been put forward as a means to structure a system into well-understood, independent and distributed building blocks that interact in specified ways. SCs are autonomic entities with dedicated knowledge units and resources that can cooperate, with different roles, in open and non-deterministic environments. SCEs are instead sets of SCs featuring goal-oriented execution.

Most of the basic properties of SCs and SCEs are already featured by current service-oriented architectures; the novelty consists in the notions of goal-oriented evolution and of self-awareness and context-awareness. Indeed, self-management is a key challenge of modern distributed IT infrastructures spanning almost all levels of computing. Self-managing systems are designed to continuously monitor their behaviors and working environment in order to select the best meaningful operations to match the current status of affairs. After [IBM05], the term *autonomic computing* has been used to identify the self-managing features of computing systems. A variety of inter-disciplinary proposals has been launched to deal with autonomic computing. We refer to [ST09] for a detailed survey.

A possible way to achieve awareness is to equip SCs with information about their own state and behavior, to enable them to collect and store information about their working environment, and to use it for redirecting and adapting their behavior. SCs can then dynamically organize themselves through SCEs by conveniently exploiting the information provided by the *attributes* that they expose in their interfaces. This fosters a notion of ensembles that are not curbed by rigid structures, but rather are highly dynamic and flexible. A typical scenario involving SCEs is reported in Figure 1. It evidences that ensembles can be thought of as logical layers, superimposed on top of the physical component networks, that identify dynamic (overlay) subnetworks of components.

Similar issues to those outlined above do arise also for so called systems of systems or systems coalitions. The key characteristic of these systems is that they are assembled from other systems that are independently controlled and managed. Their interfaces are not always well defined and structured, and might be changing, while their interaction "mood" might be cooperative or competitive. Due to their inherent complexity, today's engineering methods and tools do not scale well with such systems. Therefore, new engineering techniques are needed to address the challenges of developing, integrating, and deploying these large-scale complex IT systems [SCC+12].

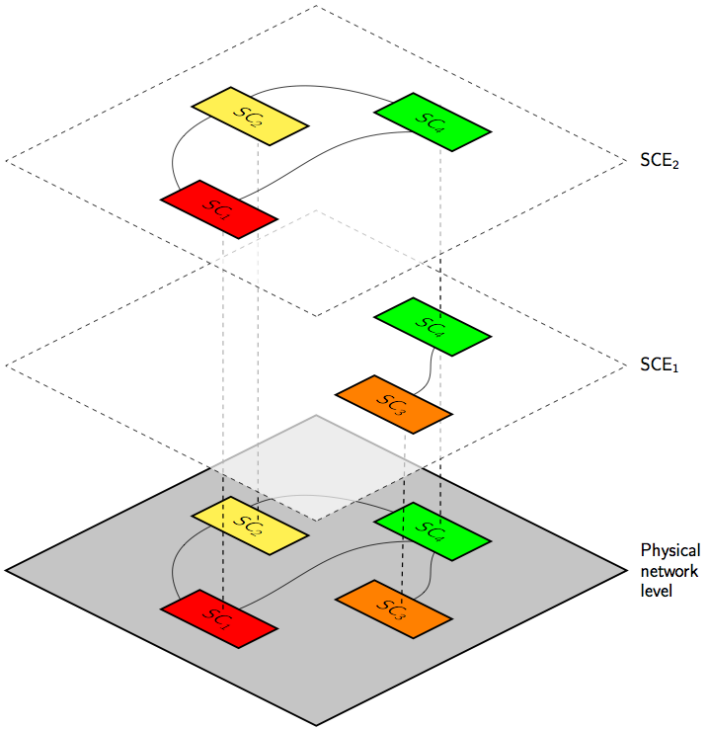

Figure 1: Service Component Ensembles

In this deliverable we present some of the results of the work done to develop linguistic supports for modelling and programming service components and their ensembles, their interactions, their sensitivity and adaptivity to the environment. More specifically, we introduce a revised version of SCEL (Service Component Ensemble Language), a language with *programming abstractions* for directly representing Knowledge, Behaviors and Aggregations according to Policies in order to naturally program SCEs, while dealing with interaction, self-awareness, context-awareness and adaptation. SCEL is equipped with a small set of basic constructs with *solid semantic grounds* so that logics, tools and methodologies can be developed for formal reasoning on systems behavior in order to establish qualitative and quantitative properties of both the individual components and the ensembles.

The main difference with the previous version of the language presented in [DFLP11, DFLP12] consists in the way sets of partners are selected for interaction. In the original version of SCEL, an ensemble is explicitly defined by a single component, acting as the coordinator of the ensemble, through an interface attribute bound to a predicate over components' attributes. In this way, the coordinating component could be a single point of centralization and, potentially, of failure. Instead, in the version of the language considered here (we refer the interested reader to [DLPT12] for a full account of the language), ensembles can be modeled by exploiting the notion of *attribute-based* communication. Ensemble members are still selected according to predicates over interfaces' attributes, representing specific properties, like spatial coordinates or group memberships, and properties that they can guarantee like security, trust level or response time. However, such predicates can now be used as targets of communication actions. In this way, ensembles are dynamically characterized by relying on interfaces' attributes to select the (set of components which are the) target of an action. The language thus provides primitives both for point-to-point and group-oriented communication.

Another difference between the two versions concerns the management of the policies in force at the system components. Indeed, the new version offers also the possibility to dynamically modify these policies during system evolution in order to properly fit new requirements and adapt the system's behavior to changing environmental conditions.

1.1 Relations with other WPs

The SCEL's refined version presented in Section 2 is the result of discussion and interaction carried out last year with many other researchers involved in the project. Several collaborations have started regarding issues considered in other work packages. They can be summarized as follows:

- The collaboration with WP2 is mainly focussed on the integration in SCEL of soft constraints as a form of knowledge. Soft constraints are particularly useful to represent partial knowledge, to deal with multi-criteria optimization, to express preferences, fuzziness, and uncertainty. We are thus defining a SCEL dialect (ccSCEL, Section 5) for concurrent constraint programming (CCP) where the SCEL primitives can be thought as a meta-programming layer on top of CCP.

- Preliminary collaborations with WP3 have started and will be intensified during the next year with the aim of investigating the possibility of integrating KnowLang with SCEL. Indeed, languages for self-aware, self-adaptive and self-expressive autonomic components and ensembles need to include: (i) procedural components; (ii) declarative knowledge representation components and their bookkeeping primitives; and (iii) primitives for the interaction of the two kinds of components. The latter part requires innovative ideas, since adaptivity and autonomicity rely mostly on an intelligent cooperation between procedural and declarative aspects of system behavior. SCEL provides the procedural components and is instead parametric wrt the declarative ones; there is however an obvious correspondence between the KnowLang operators ASK and TELL and the SCEL actions for retrieving information from shared knowledge repositories (**qry**) and for adding information to them (**put**). We have first defined a dialect of the SCEL language that relies on a simple notion of knowledge structured as a set of data tuples, then, with ccSCEL, we have performed an initial step in the direction of a "more active" knowledge handler. The next step will be the investigation of the possibility of delegating all decisions to a knowledge handler, e.g. modelled in the style of KnowLang.

- The cooperation with WP4 that started in the first year to study the possibility of expressing self-adaptation patterns in SCEL has continued over the second year. It is now focussing on how to take advantage of the novel features of the language, in particular attribute-based communication and dynamic changing policies, as a mean to facilitate self-expression (see also D4.2).

- One collaboration with WP5 is aimed at investigating how properties of SCEL systems can be guaranteed by exploiting the several BIP-based verification tools that are heavily used in WP5. To this aim we will rely on the SCEL's operational semantics and on maps from the transition system associated to a SCEL term into the internal representation of verification tools. Another collaboration aims at the design of a simple, yet expressive, language (SACPL Section 4) for defining access control policies and access requests, and its integration with SCEL.

- The cooperation with WP6 has started and will be intensified next year with the aim of developing a runtime environment providing an API for programming in Java autonomic and adaptive applications based on the SCEL paradigm. Other features, such as a library supporting specification and evaluation of SACPL policies and authorization requests, will be integrated next year. We intend also to experiment with the integration of existing constraint solvers to provide an implementation of the ccSCEL primitives.

- All activities of WP1 have paid and will pay special attention to the case studies investigated in WP7. For instance, the revised version of SCEL has been validated over the robotics case study, while the development of the SACPL policy language has been validated over the cloud case study.

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

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

1.2 Structure of the Document

The rest of this document is organized as follows. In Section 2 we introduce syntax and semantics of the new version of the language. In Section 3 we outline jRESP, a runtime environment for developing autonomic and adaptive systems according to the SCEL paradigm. In Section 4 we introduce SACPL, a language for defining access control policies, and show its integration with SCEL. In Section 5 we describe different knowledge management primitives at the basis of ccSCEL, a SCEL dialect for concurrent constraint programming. In Section 6 we illustrate the high-level design of SCEL-based applications using the DEECo component model. We conclude by comparing more strictly related work in Section 7 and by sketching the work plan for the next years in Section 8.

2 SCEL: a refined version of the language

The syntax of SCEL is presented in Table 1. The basic category of the syntax is the one relative to PROCESSES that are used to build up COMPONENTS that in turn are used to define SYSTEMS. PROCESSES specify the flow of the ACTIONS that can be performed. ACTIONS can have a TARGET to characterize the other components that are involved in that action.

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

PROCESSES are the active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* ($a.P$), *nondeterministic choice* ($P_1 + P_2$), *controlled composition* ($P_1[\,P_2\,]$), *process variable* (X), and *parameterized process invocation* ($A(\bar{p})$). The construct $P_1[\,P_2\,]$ abstracts the various forms of parallel composition commonly used in process calculi. Process variables can support higher-order communication, namely the capability to exchange (the code of) a process, and possibly execute it, by first adding an item containing the process to a knowledge repository and then retrieving/withdrawing this item while binding the process to a process variable. We assume that A ranges over a set of parameterized *process identifiers* that are used in recursive process definitions. We also assume that each process identifier A has a *single* definition of the form $A(\bar{f}) \triangleq P$ where all free

variables in P are contained in $\bar{f}$ and all occurrences of process identifiers in P are within the scope of an action prefixing. $\bar{p}$ and $\bar{f}$ denote lists of actual and formal parameters, respectively.

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 actions 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 $\mathbf{fresh}(n)$ introduces a scope restriction for the name n so that this name is ensured to be different from any other name previously used. Action $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ creates a new component $\mathcal{I}[\mathcal{K}, \Pi, P]$.

Different entities may be used as the target c of an action. Component names are denoted by n, n', ..., while variables for names are denoted by x, x', ... The distinguished variable self can be used by processes to refer to the name of their hosting component. The target can also be a *predicate* P or an attribute p associated to a predicate in the interface $\mathcal{I}$ of the component (thus the association may dynamically change). A predicate could be, for example, a boolean-valued expression obtained by applying standard boolean operators to the results returned by the evaluation of relations between attributes and expressions.

In actions using a predicate P to indicate the target, this predicate P acts as a 'guard' specifying *all* the components that may be affected by the execution of the action, i.e. a component must satisfy P in order for it to be the target of the action. Thus, e.g., actions $\mathbf{put}(t)@n$ and $\mathbf{put}(t)@\mathsf{P}$ give rise to two different primitive communication forms: the former is a *point-to-point* communication, while the latter is a sort of *group-oriented* communication (see also Remark 2.2).

The set of components satisfying a given predicate P used as the target of a communication action can be considered as the *ensemble* which the process performing the action is willing to interact with. For example, the names of the components that can be members of an ensemble can be fixed via the predicate $\mathsf{P}(\mathcal{I}) \stackrel{def}{=} \mathcal{I}.id \in \{n, m, o\}$. If this is the target, then the action will act on those components whose names are n, m and o, if any. As another example, to dynamically characterize the members of an ensemble that are active and have a battery charge level greater than 30%, by assuming that attributes $active$ and $battery\_level$ belong to the interface of each component willing to be part of the ensemble, the predicate $\mathsf{P}(\mathcal{I}) \stackrel{def}{=} \mathcal{I}.active = yes \wedge \mathcal{I}.battery\_level > 30\%$ could be used.

Remark 2.1 (On dynamically determined communication partners) *Differently from the original version of* SCEL *[DFLP11, DFLP12], attribute* ensemble *is no longer part of components' interface. The choice of dynamically determining an ensemble as a target of an action, rather than having it necessarily characterized by an interface attribute, has many important consequences. Firstly, it avoids to have a single component acting as the coordinator of the ensemble (the coordinating component would be a single point of centralization and, potentially, of failure). Secondly, it permits a more dynamic characterization of ensembles, since the target ensemble can potentially differ from one action to the next one. Finally, it simplifies the operational semantics, since an interaction between two components does not require a third party, i.e. the ensemble coordinator. Also attribute* membership *is no longer part of components' interface; its role can indeed be held by the authorization predicate (see the operational semantics rules for systems).*

An autonomic *component* $\mathcal{I}[\mathcal{K}, \Pi, P]$, graphically depicted in Figure 2, consists of:

1. An *interface* $\mathcal{I}$ publishing and making available structural and behavioral information about the component itself in the form of attributes. Among them, attribute id is mandatory and is bound to the name of the component. Notably, component names are not required to be unique; this would allow us to easily model replicated service components.

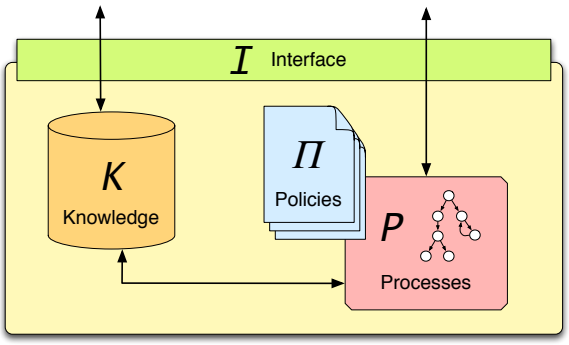

Figure 2: SCEL component

2. A *knowledge repository* $\mathcal{K}$ managing both application data and awareness data, together with the specific handling mechanism. The knowledge repository of a component stores also the whole information provided by its interface, which therefore can be dynamically manipulated by means of the operations provided by the knowledge repositories' handling mechanisms.

3. A tuple of *policies* Π regulating the interaction between the different internal parts of the component and the interaction of the component with the others.

4. A *process* P together with a set of process definitions that can be dynamically activated. Some of the processes in P perform local computation, while others may coordinate processes interaction with the knowledge repository and deal with the issues related to adaptation.

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

The operational semantics of SCEL is defined in two steps. First, the semantics of processes specifies process *commitments*, namely the actions that processes can initially perform and the continuation process obtained after each such action, while 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. For space limitation, here we only sketch the main ingredients and refer the reader to [DLPT12] for a full account of the semantics.

Process commitments are generated by the following production rule

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

meaning that a commitment is either an action as defined in Table 1, or the symbol $\circ$, denoting *inaction*, or the composition $\alpha[\,\beta\,]$ of two commitments α and β. We use P and Q, possibly indexed, to range over processes and write $P \downarrow_\alpha Q$ to mean that "P can immediately perform the commitment α and became Q in doing so". The relation $\downarrow$ is induced by a set of inference rules like those shown below:

$$\frac{-}{a.P \downarrow_a P} \qquad \frac{-}{P \downarrow_\circ P} \qquad \frac{P \downarrow_\alpha P' \qquad Q \downarrow_\beta Q'}{P[\,Q\,] \downarrow_{\alpha[\,\beta\,]} P'[\,Q'\,]}$$

The meaning of the rules is straightforward. The first one says that a process of the form $a.P$ first executes the commitment a, then continues as process P. The second rule allows any process

to perform a commitment $\circ$ while remaining unchanged. The third rule[1], defining the semantics of $P[\,Q\,]$, states that a commitment $\alpha[\,\beta\,]$ is performed when Q makes the commitment β and P makes the commitment α. However, P and Q are not forced to actually perform a meaningful action: thanks to the second rule, α and/or β may always be $\circ$.

The operational semantics of systems is defined in two steps. First, by means of a labeled transition relation indicating the actions performed by system's components, we derive the possible behaviors of systems without occurrences of the name restriction operator. Then, by exploiting this relation, we provide the semantics of generic systems by means of a (unlabelled) transition relation only accounting for systems' computation steps.

We write $S \xrightarrow{\lambda} S'$ to mean that "S can perform a transition labeled λ and became S' in doing so". The labeled transition relation is induced by a set of inference rules, an excerpt of which is reported in Table 3. The relation is parameterised with respect to the following two predicates:

- The *interaction predicate*, $\Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi'$, means that under policy Π and interface $\mathcal{I}$, process commitment α yields system label λ, substitution σ and policy Π'. Intuitively, λ identifies the effect of α at the level of components, while σ associates values to the variables occurring in α and is used to capture the changes induced by communication. Π' is the policy in force after the transition; in principle it may differ from that in force before the transition. This predicate is used to determine the effect of the simultaneous execution of actions by processes concurrently running within a component that, e.g., exhibit commitments of the form $\alpha[\,\beta\,]$.

- The *authorization predicate*, $\Pi \vdash \lambda, \Pi'$, means that under policy Π, (the action generating) the *authorization request* λ is allowed and the policy Π' is produced. This predicate is used to determine the actions allowed by specific policies, and the (possibly new) policy to be enforced. The authorization to perform an action is checked when a computation step can potentially take place, i.e. when it becomes known which is the component target of the action. By resorting to different policies, components can protect themselves against different threats, such as unauthorised access or denial-of-service attacks, hence behaving in a *self-protecting* way.

Many different interaction predicates can be defined to capture well-known process computation and interaction patterns such as interleaving, asynchronous communication, synchronous communication, full synchrony, broadcasting, etc. We present here a possible instance, that we call *interleaving*, and refer the interested reader to [DLPT12] for other two instances of interaction predicate.

The interaction predicate *interleaving* is obtained by interpreting controlled composition as the *interleaved* parallel composition of the two involved processes. The predicate is defined by the inference rules reported in Table 2, where the following notations are 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 $\mathcal{I}$, thus variables (resp. predicate names) are replaced by the corresponding component names (resp. predicates);

- $\mathcal{P}[\![\, P \,]\!]_{\mathcal{I}}$ denotes the evaluation of P according to interface $\mathcal{I}$: functionalities in P are *replaced* by the corresponding code in $\mathcal{I}$;

- $match(T', t) = \sigma$ means that the evaluated template T' and the item t do match and yield substitution σ for associating values to the variables occurring in T'.

[1] In the actual rule (see [DLPT12]), there is also a side condition ensuring that the variables used by the two processes P and Q are different in order to avoid improper captures.

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

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

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

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

Table 2: The *interleaving* interaction predicate

Basically, 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 state that, at the level of the operational semantics of systems, all process actions correspond to properly labeled transitions.

Likewise the interaction predicate, many different reasonable authorization predicates can be defined (some examples are shown in Section4, a few more are presented in [PT12]).

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 $\mathcal{K}'$.

Now, some comments about the rules in Table 3 follow. Rule *(pr-sys)* transforms process commitments into system labels by exploiting the interaction predicate. As a consequence of this transformation, a substitution σ is generated and applied to the continuation of the process that has exhibited the commitment 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.

Action **qry** can retrieve an item either from the local repository *(lqry)* or from a specific repository *(ptpqry)*, or from one of a set of repositories satisfying a given target predicate *(grqry)*. The label $\mathcal{I} : t \bar{\blacktriangleleft} \mathcal{J}$, generated by rule *(accqry)*, denotes the willingness of component $\mathcal{J}$ to provide the item t to component $\mathcal{I}$. Notably, the label is generated only if such willingness is authorized by the policy in force at the component $\mathcal{J}$. Thus, when the target of the action denotes a specific remote repository *(ptpqry)*, the action is only allowed if n is the name of the component $\mathcal{J}$ simultaneously willing to provide the wanted item and if the request to perform the action at $\mathcal{J}$ is authorized by the local policy. As a matter of notation, we use $\mathcal{I}.\pi$ to denote the policy in force at the component $\mathcal{I}$ and $S[\mathcal{I}.\pi := \Pi']$ to denote the replacement of the policy in force at the component $\mathcal{I}$ with policy Π'. When the target of the action denotes a set of repositories satisfying a given target predicate *(grqry)*, the action is only allowed if one of these repositories, say that of component $\mathcal{J}$, is willing to provide the wanted item and if the request to perform the action at $\mathcal{J}$ is authorized by the policy in force at the component performing the action. Relation $\mathcal{J} \models \mathsf{P}$ states that (the attributes of) the component $\mathcal{J}$ satisfy the predicate P; the definition of such relation depends on which kind of predicates is used. In any case, if the action succeeds, this transition corresponds to an internal computation step that leave all repositories unchanged.

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

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

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

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

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

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

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

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

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

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

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

Table 3: Labeled transition relation (Excerpt of rules)

Action **put** adds item t to one or more repositories. Rules *(lput)*, *(accput)* and *(ptpput)* are similar to the corresponding ones for action **qry**, the major difference being that the addition operation of the repository's handling mechanism is invoked. Differently from action **qry** that only interacts with one target component arbitrarily chosen among those satisfying the target predicate P and willing to provide the wanted item, $\mathbf{put}(t)@\mathsf{P}$ can interact with all components satisfying P and willing to accept the item t. In fact, rule *(grput)* permits the execution of a **put** for group-oriented communication when there is a parallel component, say $\mathcal{J}$, satisfying the target of the action and whose policy authorizes this remote access. Of course, the action must be authorized to use $\mathcal{J}$ as a target also by the policy in force at the component performing the action. Notably, the resulting transition maintains the same

label, thus further authorization actions performed by other parallel components satisfying the target of the action can be simultaneously executed. Instead, rule *(engrput)* enables a component to perform a $\mathbf{put}(t)@\mathsf{P}$ also when there is a parallel component which is not affected by execution of the action, because either it does not satisfy the target predicate or it does not authorize the action.

Finally, rule *(async)* requires that, while a **put** for group-oriented communication or an authorization for a **put** can only be performed by involving all the components of a system, all the other actions can be performed by only involving some of the system's components.

Remark 2.2 (On different forms of put) *The two actions* $\mathbf{put}(t)@n$ *and* $\mathbf{put}(t)@(\mathcal{I}.id \in \{n\})$ *have not the same meaning. Indeed, the former is a point-to-point communication and succeeds only whenever there is a component named* n *willing to receive the item* t*. The latter is a sort of group-oriented communication over a channel without message loss and can also succeed whenever a component named* n *does not exist or exists but does not authorise the action (i.e. is not willing to receive* t*). In the second case above,* $\mathbf{put}(t)@n$ *would get stuck, while* $\mathbf{put}(t)@(\mathcal{I}.id \in \{n\})$ *would terminate successfully (but* t *would not be added to the repository at* n*). Another way of writing the above group-oriented communication action is* $\mathbf{put}(t)@target$*, where* $target$ *is an attribute associated to the predicate* $\mathsf{P}(\mathcal{I}) \stackrel{def}{=} \mathcal{I}.id \in \{n\}$*. While the former two* ***put*** *actions will try to interact always with the component named* n*, the latter action* $\mathbf{put}(t)@target$ *may also interact with other components, because the association for the attribute* $target$ *may dynamically change and refer to a different predicate (e.g.* $\mathsf{Q}(\mathcal{I}) \stackrel{def}{=} \mathcal{I}.id \in \{m, o\}$*).*

The (unlabeled) transition relation providing the semantics of generic systems is defined on top of the labeled one by the following inference rules[2].

$$\frac{S \xrightarrow{\tau} S'}{(\nu\bar{n})S \rightarrowtail (\nu\bar{n})S'} \qquad \frac{S \xrightarrow{\mathcal{I}:t\triangleright\mathsf{P}} S'}{(\nu\bar{n})S \rightarrowtail (\nu\bar{n})S'} \qquad \frac{(\nu\bar{n}, n'')(S_1 \parallel S_2\{n''/n'\}) \rightarrowtail S' \qquad n'' \textit{ fresh}}{(\nu\bar{n})(S_1 \parallel (\nu n')S_2) \rightarrowtail S'}$$

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

Basically the rules state that computation steps correspond to transitions labeled either by τ or by $\mathcal{I} : t \triangleright \mathsf{P}$, that name restrictions can be moved at top level by possibly renaming restricted names with freshly chosen ones for avoiding improper name captures, and that systems' composition is a commutative and associative operator.

3 jRESP: a Runtime Environment for SCEL Programs

In this section we briefly present jRESP[3], a runtime environment, developed in Java, providing an API that permits developing autonomic and adaptive applications in Java based on the SCEL paradigm. A more detailed description of jRESP can be found in [Lor12, BGH+12].

In the definition of SCEL, some aspects, such as *knowledge representation*, are not fixed but can be identified from time to time according to the specific application domain or to the taste of the language user. Other aspects, like for instance the underlying communication infrastructure, are instead abstracted away from the SCEL's operational semantics. For this reason, jRESP is parameterized

[2] As a matter of notation, $\bar{n}$ denotes a (possibly empty) sequence of names, $\bar{n}, n''$ is the sequence obtained by composing $\bar{n}$ and n'' and $S_2\{n''/n'\}$ is the substitution of n' with n'' in S_2.

[3] `http://code.google.com/p/jresp/`

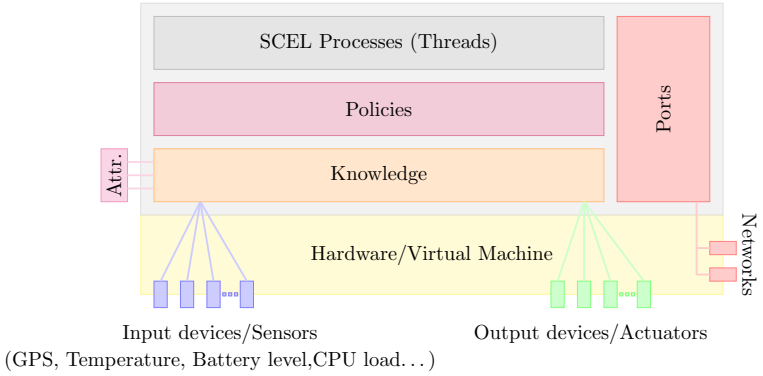

Figure 3: Node architecture

with respect to specific implementations of the above mentioned features. Besides, to simplify the integration of new features, it largely uses recurrent patterns.

jRESP communication infrastructure has been designed so to avoid any *centralized control*. Indeed, a SCEL *program* typically consists of a set of (possibly heterogeneous) components, each of which is equipped with its own knowledge repository. These components concur and cooperate in an highly dynamic environment to achieve a set of *goals*. The underlying communication infrastructure is not fixed, but can change dynamically during the computation. Hence, components can interact with each other by simply relying on the available communication media.

Aggregations. SCEL components are implemented via the class Node. The architecture of a generic node is shown in Figure 3. We assume that each node is executed over a virtual machine or a physical device that provides the access to input/output devices and to network connections. Each node aggregates a knowledge repository, a set of running processes/threads, and a set of policies. Structural and behavioral information about a node can be collected into an *interface* via a set of *attribute collectors*. Nodes interact through *ports* supporting both *point-to-point* and *group-oriented* communications.

Knowledge. The interface Knowledge identifies a generic knowledge repository and indicates the high-level primitives to manage pieces of relevant information coming from different sources. This interface contains the methods for withdrawing/retrieving/adding piece of knowledge from/to a repository. External data can be collected into the knowledge via *sensors*. Each sensor can be associated to a logical or physical device providing data that can be used by processes and that can be the subject of adaptation. Similarly, *actuators* can be used to collect data from knowledge repositories and forward them to external components. This approach allows SCEL processes to control exogenous devices that identify logical/physical actuators. Attribute values are published on component interfaces via *attribute collectors*. When a request for an attribute is received, the corresponding collector is selected. The latter interacts with the node's knowledge to compute the actual attribute value.

Behaviors. SCEL processes are implemented as threads via the abstract class Agent which provides the methods for withdrawing/retrieving/adding information items from/to shared knowledge repositories. These methods extend the ones considered in Knowledge with another parameter identifying the, possibly remote, node where the target knowledge repository is located.

Policies. Policies are organized in a *stack*. The policy at one level relies on the one at the level below. The policy at the lowest level allows any operation. When an agent invokes a method, its execution is delegated to the policy associated to the node where the agent is running.

Π	::=	$\langle Decision\,;\mathsf{target:}\{\ Targets\ \}\rangle$	(ATOMIC POLICIES)
	|	Π p-o Π | Π d-o Π	(COMPOSED POLICIES)
$Decision$	::=	permit | deny	(DECISIONS)
$Targets$	::=	$MatchF(Designator,Expr)$	(ATOMIC TARGETS)
	|	$Targets$ or $Targets$ | $Targets$ and $Targets$	(COMPOSED TARGETS)
$MatchF$	::=	equal | pattern-match | greater-than | ...	(MATCHING FUNCTIONS)
$Designator$	::=	action | pattern | subject.$attr$ | object.$attr$	(DESIGNATORS)
$Expr$	::=	**get** | **qry** | **put** | **fresh** | **new**	(EXPRESSIONS)
	|	T | $value$ | subject.$attr$ | object.$attr$	
	|	not $Expr$ | $Expr$ or $Expr$ | $Expr$ and $Expr$	
	|	$Expr + Expr$ | $Expr \times Expr$ | $Expr < Expr$ | $Expr = Expr$ | ...	

Table 4: SACPL policy syntax

4 SACPL: a SCEL Access Control Policy Language

In this section we present SACPL (SCEL Access Control Policy Language), a simple, yet expressive, language for defining access control policies and access requests, and its integration with SCEL. SACPL is inspired to, but much simpler than, the OASIS standard for policy-based access control XACML [OAS05]. Due to space limitation, here we only sketch the main ingredients and refer the interested reader to [PT12] for a full account of the language.

Access control is a fundamental mechanism for restricting what operations users can perform on protected resources. Many *models* of access control have been defined in the literature. Here, we focus on the Policy Based Access Control model [NIS09], that is by now the de-facto standard model for enforcing access control policies in service-oriented architectures. In this model, a request to access a protected resource is evaluated with respect to one or more policies that define which requests are authorized. An authorization decision is based on attribute values required to allow access to a resource according to policies stored in system's components. Component attributes are here used to describe the entities that must be considered for authorization purposes; they might concern:

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

SACPL syntax is presented in Table 4. Policies are hierarchically structured as trees. Indeed, a *policy* is either an atomic policy or a pair of simpler policies combined through one of the decision-combining operators p-o (*permit override*) and d-o (*deny override*). An *atomic policy* is a pair made of a decision and a target. The target defines the set of *access requests* to which the policy applies. The *decision*, i.e. permit or deny, is the effect returned when the policy is 'applicable', namely the access request belongs to the target. Otherwise, i.e. when a request does not belong to the policy's target, the policy is 'not-applicable', which in our simplified setting has the same effect as deny.

$$\frac{\Pi_1 \vdash \rho \quad \vee \quad \Pi_2 \vdash \rho}{(\Pi_1 \text{ p-o } \Pi_2) \vdash \rho} \qquad \frac{\Pi_1 \vdash \rho \quad \Pi_2 \vdash \rho}{(\Pi_1 \text{ d-o } \Pi_2) \vdash \rho}$$

$$\langle \mathsf{permit}\,;\mathsf{target}\colon\{\ \}\rangle \vdash \rho \qquad \frac{Targets \vdash \rho}{\langle \mathsf{permit}\,;\mathsf{target}\colon\{\,Targets\,\}\rangle \vdash \rho}$$

$$\frac{Targets_1 \vdash \rho \quad \vee \quad Targets_2 \vdash \rho}{(Targets_1 \text{ or } Targets_2) \vdash \rho} \qquad \frac{Targets_1 \vdash \rho \quad Targets_2 \vdash \rho}{(Targets_1 \text{ and } Targets_2) \vdash \rho}$$

$$\frac{\rho(\mathsf{action}) = Act}{\mathsf{equal}(\mathsf{action}, Act) \vdash \rho} \qquad \frac{match(T, \rho(\mathsf{item}))}{\mathsf{pattern\text{-}match}(\mathsf{pattern}, T) \vdash \rho}$$

$$\frac{MatchF(\rho(\mathsf{subject}).attr, \mathcal{E}[\![\, Expr \,]\!]_\rho)}{MatchF(\mathsf{subject}.attr, Expr) \vdash \rho} \qquad \frac{MatchF(\rho(\mathsf{object}).attr, \mathcal{E}[\![\, Expr \,]\!]_\rho)}{MatchF(\mathsf{object}.attr, Expr) \vdash \rho}$$

Table 5: SACPL semantics (where Act is any of **get**, **qry**, **put**, **fresh**, and **new**)

A *target* is either an atomic target or a pair of simpler targets combined using the standard logic operators and and or. An *atomic target* is a triple denoting the application of a matching function to values from the request and the policy, like e.g. greater-than(subject.skill, $threshold -$ object.dependability). To base an authorization decision on some characteristics of the request, e.g. subjects' or objects' identity, atomic targets use *designators* (i.e. *attribute names*) to point to specific values contained in the request. Specifically, the designator action refers to the action to be performed (such as **get**, **qry**, **put**, etc.), pattern permits referring to the item exchanged in the considered interaction via function pattern-match and template T, while subject.$attr$ and object.$attr$ refer to the specific attribute $attr$ provided, respectively, by the request's subject or object (like, e.g., subject.id, subject.skill, object.trust_level).

Finally, $Expr$essions are built from $values$ and $attr$ibutes through various operators.

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

$$\rho \;=\; \{(\mathsf{subject}, \mathcal{I}), (\mathsf{item}, t), (\mathsf{action}, \textbf{get}), (\mathsf{object}, \mathcal{J})\}$$

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

The semantics of SACPL is given in terms of a judgement $\Pi \vdash \rho$ meaning that the authorization decision returned by a policy Π in response to a request ρ is permit, i.e. access to the resource requested in ρ is granted by Π. In practice, the judgement $\Pi \vdash \rho$, inferred through the rules in Table 5, means that the request is allowed.

The meaning of the rules is straightforward. To match a composed policy $(\Pi_1$ p-o $\Pi_2)$, a request is only required to match one of Π_1 and Π_2, while it must match both Π_1 and Π_2, for it to match the policy $(\Pi_1$ d-o $\Pi_2)$. As for atomic policies, if the target is empty, the request matches the policy, otherwise the request is required to match the target. To match the composed target ($Targets_1$ or $Targets_2$), a request is only required to match one of $Targets_1$ and $Targets_2$, while it must match both $Targets_1$ and $Targets_2$, for it to match the target ($Targets_1$ and $Targets_2$). A request matches an atomic target of the form $\mathsf{equal}(\mathsf{action}, Act)$ if the request's action corresponds to the action Act identified by

the target. An atomic target of the form pattern-match(pattern,T) is matched by all requests whose item matches the template T; this is checked by means of a pattern-matching function $match$ whose definition is left unspecified, because it depends on the considered notion of items and templates. Finally, when an atomic target contains a subject's (resp. object's) attribute as designator, the evaluation consists in obtaining the subject (resp. object) interface from the request, retrieving the value of the attribute from the interface, evaluating the expression by possibly retrieving other attribute values from the request elements and, finally, calling the corresponding match function. This evaluation relies on a few auxiliary functions. First, we use the function $\mathcal{E}[\![\ Expr\ ]\!]_\rho$ to evaluate the expression $Expr$ after replacing the attributes occurring in $Expr$ by the corresponding values in the subject/object interfaces in ρ. Then, we exploit the definition of the matching functions specified in the target; e.g. equal(n,m) is true if $n = m$. Thus, for example, the atomic target equal(subject.$status$,"on") matches all authorization requests issued by a component whose $status$ attribute is set to on.

We conclude this section by discussing how SACPL policies and requests, as well as the related evaluation mechanism, integrate with SCEL. SCEL is indeed parametric with respect to the language used to specify the policies regulating the behavior of system components, as it is shown by the definition of its operational semantics. Orthogonal aspects of components' behavior can be regulated by means of different kinds of policies, which should be *enforced together* but *evaluated separately*. Hence, the policy Π specified within a component $\mathcal{I}[\mathcal{K}, \Pi, P]$ can be better thought of as a tuple of policies. For example, Π can be of the form (Π_i, Π_{ac}), where Π_i is one of the policies shown in [DLPT12] for regulating the interaction among processes inside a component, while Π_{ac} is a SACPL policy for regulating the access to the knowledge and resources of a component.

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

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

which also implies that the policy in force does never change owing to evaluation of a request. The authorization predicate definition relies on the function $\lambda 2\rho(\cdot)$ that maps (a subset of) the SCEL labels to SACPL requests. For example, the label $\mathcal{I} : t \triangleleft \mathcal{J}$ is converted into the authorization request $\{(\mathsf{subject}, \mathcal{I}), (\mathsf{item}, t), (\mathsf{action}, \mathbf{get}), (\mathsf{object}, \mathcal{J})\}$. We refer to [PT12] for the complete definition of function $\lambda 2\rho(\cdot)$. Hence, the authorization of a SCEL request λ over the policy Π_{ac} corresponds to establishing the authorization decision returned by the policy Π_{ac} in response to the corresponding SACPL request $\rho = \lambda 2\rho(\lambda)$, which is exactly the judgement $\Pi_{ac} \vdash \rho$ defined by the rules in Table 5.

We refer the interested reader to [PT12] for an account of dynamically changing policies, and for an application to the ASCENS Cloud Case Study [SMB+12] showing that SACPL permits to control not only access to resources, but also adaptation of components and systems.

5 ccSCEL: a SCEL Dialect for Concurrent Constraint Programming

As shown in Section 2, to fit different paradigms and application domains, some ingredients of the SCEL language have been intentionally left unspecified. By instantiating such parameters, different SCEL dialects can be derived. We present in this section a dialect of SCEL, called ccSCEL, specifically devised for concurrent constraint programming [SR90]. The motivations underlying the addition

$$
\begin{array}{rcll}
\mathcal{K} & ::= & \emptyset \mid t \parallel \mathcal{K} & \text{(KNOWLEDGE)} \\
t & ::= & \chi \mid \langle f \rangle & \text{(ITEMS)} \\
f & ::= & e \mid c \mid P \mid f_1, f_2 & \text{(TUPLE FIELDS)} \\
T & ::= & \chi \mid \langle F \rangle & \text{(TEMPLATES)} \\
F & ::= & e \mid c \mid ?x \mid ?X \mid F_1, F_2 & \text{(TEMPLATE FIELDS)} \\
\Pi & ::= & \Pi_{1store} \mid \Pi_{2store} & \text{(POLICIES)}
\end{array}
$$

Table 6: ccSCEL syntax (SYSTEMS S, COMPONENTS C, PROCESSES P, ACTIONS a and TARGETS c are defined in Table 1)

of constraints to SCEL are twofold. From the ASCENS's perspective, this work permitted experimenting with the definition of a SCEL dialect through the specification of (some of) the language parameters, i.e. knowledge items and templates, knowledge repository's operations, policies, and the interaction and authorization predicates. This activity also permitted gaining a deeper understanding of the issues underlying knowledge representation and manipulation. From a more general perspective, using constraints as a form of knowledge can bring benefits to address issues related to service component ensembles. Constraints are indeed suitable to represent partial knowledge, to deal with multi-criteria optimization, to express preferences, fuzziness, and uncertainty.

The syntax of ccSCEL is illustrated in Table 6, which only reports the new grammar definitions specifying the considered notion of knowledge, since the other definitions are directly borrowed by SCEL without any change. In particular, in this dialect, a knowledge repository $\mathcal{K}$ can contain two kinds of items: *constraints* χ and *data tuples* $\langle f \rangle$. Thus, a repository can play the role of both a constraint store and a tuple space. Hence, when the argument of actions **put**, **qry** and **get** is a constraint, they play the role of actions **tell**, **ask** and **retract**, respectively, commonly used in languages for concurrent constraint programming to add a constraint to a store, to check entailment of a constraint by a store and to remove a constraint from a store. Instead, when the argument of actions **put**, **qry** and **get** is a tuple, they play the role of actions **out**, **read** and **in**, respectively, commonly used in tuple-based languages to insert, read and withdraw tuples to/from a tuple space. ccSCEL relies on this specific model of knowledge, based on both constraints and data tuples, in order to deal at once with such issues as SLA achievement, constraint-driven decision making, coordinated asynchronous interactions, concurrent activities, resource usage, self-awareness and adaptation, in a distributed, open-ended setting.

It is worth noticing that we do not take a definite standing on which of the many notions of constraints to use. We just assume to rely on (soft) constraints based on *c-semiring* [BMR97]. Thus, constraints are functions of the form $\chi : (N \to D) \to S$, where N is a set of names, D is the domain of values that the names may assume, and $\langle S, +, \times, 0, 1 \rangle$ is a c-semiring, i.e. a partially ordered set of 'preference' values equipped with two suitable operations for combination ($\times$) and comparison ($+$) of values and constraints. Operation $+$ induces a partial order $\leq$ on S defined by $a \leq b$ iff $a + b = b$. From time to time, the appropriate kind of constraints to work with, and the corresponding constraint system, should be chosen depending on what one intends to model. Similarly, we intentionally leave unspecified the syntax of expressions e used in tuples and templates. Notably, a tuple is a sequence of actual fields, while a template is a sequence of actual and formal fields; the latter, written as $?x$ or $?X$, are used to bind variables to values or to processes, respectively.

The proposed dialect relies on two policies regulating the combination of constraints in case of remote insertion: when a constraint is required to be added to a remote store, Π_{1store} prescribes to

$$\mathcal{K} \ominus \langle f \rangle = \mathcal{K}' \;\; \textit{if } \mathcal{K} \equiv \mathcal{K}' \parallel \langle f \rangle \qquad \mathcal{K} \vdash \langle f \rangle \;\; \textit{if } \mathcal{K} \equiv \mathcal{K}' \parallel \langle f \rangle \qquad \mathcal{K} \oplus \langle f \rangle = \mathcal{K} \parallel \langle f \rangle$$

$$\mathcal{K} \ominus \chi = \begin{cases} \mathcal{K}' & \textit{if } \mathcal{K} \equiv \mathcal{K}' \parallel \chi \\ \mathcal{K} & \textit{otherwise} \end{cases}$$

$$\mathcal{K} \vdash \chi \qquad \textit{if } \; \mathcal{K} \equiv (\mathcal{K}_{tuples} \parallel \chi_1 \parallel \ldots \parallel \chi_m) \;\; \textit{and} \;\; (\chi_1 \times \ldots \times \chi_m) \leq \chi$$

$$\mathcal{K} \oplus \chi = \mathcal{K} \parallel \chi \;\; \textit{if } \; \mathcal{K} \equiv (\mathcal{K}_{tuples} \parallel \chi_1 \parallel \ldots \parallel \chi_m) \;\; \textit{and} \;\; (\chi_1 \times \ldots \times \chi_m \times \chi) \neq 0$$

Table 7: Knowledge repository operations

combine (by means of operation $\times$) only the constraints within the remote store, while Π_{2store} prescribes to combine the constraints within the remote store and the ones local to the process performing the action. Of course, when a constraint is added to the local store, no matter what is the local policy, only the local constraints are combined. Notably, when a remote **put** action is performed, if the policy is Π_{2store} and the store resulting from the combination of the constraint to be added and the two stores is consistent, the effect of the action is the same as that in case of policy Π_{1store}, i.e. the constraint argument of the **put** action is added to the remote store (and not to the local one). Notice also that, if Π_{2store} is used in case of group-oriented communication, the set of constraints within the store of the sender is *pairwise* combined with the sets of constraints stored in the receiving components.

Since ccSCEL is a dialect of SCEL, its semantics is defined by simply instantiating the SCEL's parameters without modifying or adding operational rules. We show below how such parameters have been instantiated to define ccSCEL.

Knowledge repositories' operations. In ccSCEL, the three operations provided by a knowledge repository are defined in two different ways, depending on whether the item t is a data tuple (rules in the upper part of Table 7) or a constraint (rules in the lower part of Table 7). Given a repository $\mathcal{K}$, we use $\mathcal{K}_{tuples}$ to denote the knowledge corresponding to all tuples within $\mathcal{K}$. We use $\mathcal{K}_1 \equiv \mathcal{K}_2$ to denote that $\mathcal{K}_1$ and $\mathcal{K}_2$ are equal up to commutation of items and addition/removal of the identity element $\emptyset$. Notably, in the definition of $\mathcal{K} \vdash \chi$ and $\mathcal{K} \oplus \chi$, if the constraint store is empty (i.e. $m = 0$), then it is sufficient to verify that χ is a tautology (i.e. it returns the c-semiring value 1 for any assignment) and that χ has at least a solution (i.e. it differs from value 0), respectively.

Interaction predicate. The interaction predicate used in ccSCEL is a variant of the *interleaving* interaction predicate defined in [DLPT12]. Indeed, a controlled composition $P[\,Q\,]$ is interpreted as the interleaved parallel composition of the two involved processes, i.e. only one process can perform an action (the other stays still). Moreover, as expected, this predicate associates to each process action α the corresponding system label λ. Depending on the argument of the action, different controls are performed: in case of a data template, the pattern-matching with a tuple is checked and a substitution is generated (by means of the function $match$), while, in case of a tuple or a constraint, no pattern-matching evaluation is performed and, hence, it is returned the empty substitution. As an example, we report below the rules dealing with the **get** action.

$$\frac{\mathcal{E}[\![\,\langle F \rangle\,]\!]_{\mathcal{I}} = \langle F' \rangle \quad \mathcal{N}[\![\,c\,]\!]_{\mathcal{I}} = \gamma \quad match(F', f) = \sigma}{\Pi, \mathcal{I} : \mathbf{get}(\langle F \rangle)@c \succ \mathcal{I} : \langle f \rangle \triangleleft \gamma, \sigma, \Pi} \qquad \frac{\mathcal{E}[\![\,\chi\,]\!]_{\mathcal{I}} = \chi' \quad \mathcal{N}[\![\,c\,]\!]_{\mathcal{I}} = \gamma}{\Pi, \mathcal{I} : \mathbf{get}(\chi)@c \succ \mathcal{I} : \chi' \triangleleft \gamma, \{\}, \Pi}$$

where $\mathcal{E}[\![\,\cdot\,]\!]_{\mathcal{I}}$ denotes the evaluation function for items and templates w.r.t. an interface $\mathcal{I}$, while $\mathcal{N}[\![\,\cdot\,]\!]_{\mathcal{I}}$ denotes the evaluation function for targets.

Authorization predicate. The ccSCEL's authorization predicate is defined by the following rules

enforcing policies Π_{1store} and Π_{2store}:

$$\Pi_{1store} \vdash \lambda \qquad \Pi_{2store} \vdash \mathcal{J} : \chi \bar{\triangleright} \mathcal{J} \qquad \frac{\lambda \neq \mathcal{I} : \chi \bar{\triangleright} \mathcal{J}}{\Pi_{2store} \vdash \lambda} \qquad \frac{\mathcal{K}_{\mathcal{I}} \oplus \mathcal{K}_{\mathcal{J}} \oplus \chi}{\Pi_{2store} \vdash \mathcal{I} : \chi \bar{\triangleright} \mathcal{J}}$$

where $\mathcal{K}_{\mathcal{I}}$ and $\mathcal{K}_{\mathcal{J}}$ are the knowledge repositories of the components $\mathcal{I}$ and $\mathcal{J}$, respectively. The first rule states that policy Π_{1store} allows to execute any kind of action, since this policy does not require combining remote constraints with local ones[4]. Similarly, the second rule always allows the local insertion of a constraint (indeed, the target component $\mathcal{J}$ coincides with the component executing the action), and the third rule authorizes any system label that does not correspond to the acceptance of a **put** action having a constraint as argument. The last rule deals with the remaining case: it always checks the consistency[5] of the combination of the constraint to be added with the remote and local stores when the policy is Π_{2store}.

We refer the interested reader to [BMPT12] for a full account of ccSCEL. Currently, in order to conveniently model case studies where other forms of knowledge or logical reasoning are required, we are considering to revise ccSCEL in order to deal with (*soft*) *constraint logic programming* [BMR01] (SCLP), an extension of constraint logic programming where logic predicates are extended to functions which, rather than returning booleans, yield more informative values such as preference values, fuzzy values, probabilities or costs, which form a c-semiring. One direction that we are exploring along this line of research focusses on the use of soft constraints for supporting Service-Oriented Computing. Service-oriented applications are indeed in the scope of ASCENS, but they should be studied with the particular focus of autonomicity and adaptivity. In particular, these requirements impose as little centralization as possible, open endedness and heterogeneity of components.

In this setting, each service component can represent knowledge as a set of constraints about itself and the surrounding world: (K_1, C_1), (K_2, C_2), ..., where C_i are the constraints, and K_i are the keys that explain the meaning of the constraints (e.g. distance, temperature, danger level). Constraints are a generic way to represent knowledge, and constraint operations correspond to knowledge operations.

According to some preliminary work [PM12], apparently it is feasible to encapsulate a variety of knowledge representation styles in different sites, and to carry on deduction steps and consistency checks in terms of distributed constraint handling, where the local, specific knowledge interacts with the constraint representation via suitable interfaces. The general approach is as follows. Only variables are distributed, while every SCLP subgoal and every constraint is considered a service call, and thus is under the responsibility of a particular site. When a subgoal is reduced, new subgoals are usually generated. Those owned by the present site are handled locally, while those, possibly sharing variables, owned by other sites, are shipped there. The third possibility is a subgoal to be a constraint. Then it is passed through the interface and it is handled by the "hardware" of the site. Global consistency of local constraints is checked via constraint propagation. In [PM12], a simple ontology representation and a relational database are introduced. Furthermore in the implementation it is shown how a mix of logic programming (employing the *PROVA* framework) and a general-purpose programming language such as Java can be used to integrate different knowledge representation formalisms. Constraints can be used as interfaces between heterogenous knowledge, using Java calls for accessing their implementation, but still keeping the code clean, and benefitting from the declarative aspects of the (soft) constraint logic programming paradigm.

[4]Notably, the premise $\mathcal{K} \oplus t = \mathcal{K}'$ (where $\mathcal{K}$ is the receiver's repository) of rule *(accput)* in Table 3 precisely enforces policy Π_{1store}, which hence does not need to be considered again by the authorization predicate.

[5]The premise in the fourth rule is satisfied if the store resulting from the composition $\mathcal{K}_{\mathcal{I}} \oplus \mathcal{K}_{\mathcal{J}} \oplus \chi$ is consistent, i.e. if there exists $\mathcal{K}$ such that $\mathcal{K}_{\mathcal{I}} \oplus \mathcal{K}_{\mathcal{J}} \oplus \chi = \mathcal{K}$.

6 Towards High-level Design of SCEL-based Applications

In the previous sections various low-level aspects of the SCEL language have been described. Nevertheless, in order to be able to model a complex system in SCEL, a design method elaborating system requirements down to the level of system architecture and reflecting this in SCEL implementation is needed. A key goal is to provide a comprehensive method that supports all the phases of software development process, from early requirements to an implementation based on the SCEL concepts (Knowledge, Behaviors, Aggregations, and Policies). Therefore, in this section we describe an initial proposal that gives directions for such a method for building systems of SCs and SCEs.

Specifically, the method operates with abstractions introduced by DEECo [KBPK12, BGH$^+$12], which is a reification of SCEL language for design of large-scale systems. To address the software engineering issues, DEECo features a more structured system design by making both components and ensembles explicit first-class architectural concepts. It also introduces the explicit concepts of members and coordinator of an ensemble, which serve for high-level modeling and simplify the communication and synchronization semantics for an application developer by assuming one-to-many interaction instead of general peer-to-peer interaction. It is worth noticing that the semantics and purpose of member and coordinator concepts is different from the original version of SCEL [DFLP11, DFLP12].

Following the idea of the top-down design paradigm, in this method design is based on a systematic decomposition and refinement of system requirements specification. It consists of three phases: *system level* design, *ensemble level* design, and *component level* design. Providing enough detail, the component level design can be directly followed by *implementation* either in SCEL, or a programming language for which a SCEL runtime environment is available (jRESP – Section 3, jDEECo [KBPK12, BGH$^+$12]). These three levels cover all the phases of software design, starting from early and late requirements phases addressed at the system level, followed by architecture design phase addressed at the system and ensemble levels, ending up with the detailed design phase addressed at the component level.

System level. A starting point of our method is obtaining the system's *stakeholders* and system *invariants*.

A stakeholder is a participant of the system that arises from the early phases of requirements analysis. In general, a stakeholder comprises *knowledge*, being essentially a (multi)set of *knowledge items*. Typically, knowledge items are attributes, obtained by domain analysis, that characterize the stakeholder. In general, in compliance with SCEL, we do not refer to any particular knowledge representation in our method. Formally, a stakeholder S is a tuple $\mathbb{S}\langle N, \mathcal{K}\rangle$ where N is the stakeholder's name and $\mathcal{K}$ its knowledge.

An invariant is a system property that does not vary over time. Specifically, an invariant is a predicate over the knowledge of a set of stakeholders. These stakeholders are associated with the invariant by taking a *role* in it. A stakeholder takes a role in an invariant when a subset of its knowledge items is involved in the associated predicate. More precisely, an invariant I is a tuple $\mathbb{I}\langle N, F, \mathcal{R}\rangle$, where N is its name, F the predicate (formula), and $\mathcal{R}$ is a set of stakeholder roles referenced in F. Each role R, i.e., an element of $\mathcal{R}$, is a tuple of the form $\mathbb{R}\langle N_R, N_S, \mathcal{K}_{S\to I}, A\rangle$, where N_R is the role name, N_S the name of the associated stakeholder S, $\mathcal{K}_{S\to I}$ the set of knowledge items of the stakeholder S that are involved in the associated invariant I, and A the cardinality of the role (with the domain $\{1, *\}$). Thus, a single stakeholder can take multiple roles in the same invariant. A role with cardinality 1 refers to exactly one stakeholder, while a role with cardinality $*$ refers to an unbounded set of stakeholders.

The system-level design process starts by identifying all top-level invariants, together with the stakeholders taking a role in them. Next, the process continues by iterative decomposition of the top-

level invariants into sets of (sub-)invariants[6]. The decomposition terminates once each leaf invariant in the decomposition tree is either of type *single-stakeholder* or *inter-stakeholder*. A single-stakeholder invariant is an invariant that references a single role only, i.e., its validity is determined by the knowledge of a single stakeholder. On the other hand, an inter-stakeholder invariant is an invariant that references more than one role, i.e., its validity is determined by the knowledge of several stakeholders, and it has the form of a conjunction of equalities among the involved knowledge items.

With such decomposition, we strive to reach the level of abstraction, at which the invariants can be easily represented in the SCEL component communication (for inter-stakeholder invariants) and computation semantics (for single-stakeholder invariants).

For the description of a graphical and textual representation of invariants, invariant decomposition, stakeholders and stakeholder roles, as well as for examples, we refer the interested reader to [BGH$^+$12].

Ensemble level. Upon the basic SCEL notions, we introduce *ensemble* as a first-class concept in order to explicitly capture the architecture of a system. An ensemble is a group of *components*, created dynamically, where both the *membership* in the group and the communication in the group in the form of *knowledge exchange* are expressed declaratively. Specifically, as an extension to SCEL, one component of the group is the *coordinator* of the group, while the other components are *members*; knowledge exchange takes place between the coordinator and members when triggered by a specific condition. An ensemble is derived by refinement of an inter-stakeholder invariant (multiple invariants can be refined by the same ensemble).

Note, that the use of coordinator and member concepts is not in contradiction with Remark 2.1. This is because their semantics is different from the original SCEL language specification. In particular, their use does not create a single point of failure, mainly because of the dynamic selection of the coordinator and replication of the knowledge via knowledge exchange.

In an ensemble, the coordinator, resp., the members are represented by a dedicated coordinator, resp., member *abstract interface*. An abstract interface is derived from a stakeholder in the following way: it entails the stakeholder's knowledge items that are involved in the inter-stakeholder invariant refined by the ensemble. Thus an abstract interface refines a stakeholder's role in this invariant.

In summary, an abstract interface is a refinement of a stakeholder, while an ensemble is a refinement of an inter-stakeholder invariant, and the relationship stakeholder–invariant is paralleled by the relationship interface–ensemble.

Formally, an abstract interface $\mathcal{AI}$ refining a role $\mathbb{R}\langle N_R, N_S, \mathcal{K}_{S\to I}, A\rangle$ is a tuple $\mathbb{AI}\langle N_{\mathcal{AI}}, \mathcal{K}\rangle$, where $N_{\mathcal{AI}}$ is the name of the interface and $\mathcal{K} = \mathcal{K}_{S\to I}$ the set of the entailed knowledge items. An ensemble E is a tuple $\mathbb{E}\langle N_E, \mathcal{AI}_C, \mathcal{AI}_M, M, X\rangle$, where N_E is the name of the ensemble, $\mathcal{AI}_C$, resp., $\mathcal{AI}_M$ the coordinator, resp., member abstract interface, M the membership predicate, and X the knowledge exchange. Here, M is a function $\mathcal{K}_{\mathcal{AI}_C} \times \mathcal{K}_{\mathcal{AI}_M} \to \{true, false\}$, where $\mathcal{K}_{\mathcal{AI}_C}$ and $\mathcal{K}_{\mathcal{AI}_M}$ are the sets of knowledge items of the coordinator and member abstract interface, respectively, while X is a function $\mathcal{K}_{\mathcal{AI}_C} \times \mathcal{K}^*_{\mathcal{AI}_M} \to \mathcal{K}_{\mathcal{AI}_C} \times \mathcal{K}^*_{\mathcal{AI}_M}$, where $*$ refers to an unbounded number of members of the ensemble.

The membership M and knowledge exchange X of an ensemble are to be inferred from the predicate of the invariant.

Component level. During the component level design, the goal is to refine a stakeholder by means of a (SCEL) *component*, the component knowledge in particular. In general, a stakeholder is to be

[6] Currently, such decomposition is always an AND-decomposition; i.e., for a parent invariant $\mathbb{I}\langle N_p, F_p, \mathcal{R}_p\rangle$ and its sub-invariants $\mathbb{I}\langle N_{s_i}, F_{s_i}, \mathcal{R}_{s_i}\rangle$, for $i = 1 \ldots n$, the following holds: $F_{s_1} \wedge \cdots \wedge F_{s_n} \implies F_p$.

refined by one or more components, while a single component can refine several stakeholders. The main goal of such stakeholder refinement into a component is to entail all the knowledge items relevant to all the roles the stakeholder takes in single-stakeholder invariants (knowledge relevant to the roles of inter-stakeholder invariants is discussed separately). Thus, for any stakeholder $S = \mathbb{S}\langle N_S, \mathcal{K}_S\rangle$ which is refined by the component $\mathcal{I}[\mathcal{K}, \Pi, P]$ and any single-stakeholder invariant the stakeholder S takes a role $\mathbb{R}\langle N_R, N_S, \mathcal{K}_{S\to I}, A\rangle$ in, the following holds: $\mathcal{K}_{S\to I} \subseteq \mathcal{K}$.

Further, any single-stakeholder invariant I is to be refined by means of a *process* $P_{loc(I)}$. In particular, the refinement is a *local process*, i.e., a process actions of which target only self (Table 1). This is because a single-stakeholder invariant refers only to the knowledge local to the component refining the stakeholder. In general, the goal of a local process is to maintain the validity of the refined invariant by manipulating the component's knowledge accordingly.

Finally, the abstract interfaces defined at the ensemble level are to be *reified* by components according to the anticipated participation of the components in the ensembles associated with the abstract interfaces. A reification of an abstract interface implies including all the knowledge items specified in the abstract interface into the (SCEL) interface of the component. Thus, for any abstract interface $\mathcal{AI} = \mathbb{AI}\langle N_{\mathcal{AI}}, \mathcal{K}_{\mathcal{AI}}\rangle$ which is reified by the component $\mathcal{I}[\mathcal{K}, \Pi, P]$ the following holds: $\mathcal{K}_{\mathcal{AI}} \subseteq \mathcal{I} \subseteq \mathcal{K}$.

Furthermore, it is necessary to explicitly represent ensembles by SCEL means, the knowledge exchange in particular, since SCEL does not provide any directly applicable concepts. In general, the knowledge exchange is to be refined by a process of the coordinator of the ensemble. Thus, any component $\mathcal{I}[\mathcal{K}, \Pi, P]$ that reifies the coordinator abstract interface $\mathcal{AI}_C = \mathbb{AI}\langle N_{\mathcal{AI}_C}, \mathcal{K}_{\mathcal{AI}_C}\rangle$ of an ensemble $E = \mathbb{E}\langle N_E, \mathcal{AI}_C, \mathcal{AI}_M, M, X\rangle$ has to include a dedicated process $P_{kex(E)}$ responsible for performing the knowledge exchange of the ensemble. In general, the goal of such a process is to maintain the validity of the invariant refined by the ensemble by exchanging the knowledge among the coordinator and members. For example, the process $P_{kex(E)}$ can have the following form:

$$
\begin{array}{lll}
P_{kex(E)} \triangleq & \mathbf{qry}(T_{\mathcal{AI}_C})@\mathsf{self}. & \textit{(load coordinator knowledge)} \\
& \mathbf{qry}(T_{\mathcal{AI}_M})@\mathsf{P}_{M(t_{\mathcal{AI}_C},\mathcal{I})}. & \textit{(load members' knowledge)} \\
& \mathbf{put}(X(t_{\mathcal{AI}_C}, t_{\mathcal{AI}_M}) \restriction_{\mathcal{AI}_C})@\mathsf{self}. & \textit{(store outcome of X for coordinator)} \\
& \mathbf{foreach}(t_{\mathcal{AI}_{M_i}} \mathbf{in}\ t_{\mathcal{AI}_M})\ \{ & \textit{(for each member)} \\
& \quad \mathbf{put}(X(t_{\mathcal{AI}_C}, t_{\mathcal{AI}_M}) \restriction_{\mathcal{AI}_{M_i}})@(\mathcal{I}.id = t_{\mathcal{AI}_{M_i}}.id). & \textit{(store outcome of X for the member)} \\
& \} & \\
& P_{kex(E)} & \textit{(repeat the whole process)}
\end{array}
$$

Here, $T_{\mathcal{AI}_C}$ and $T_{\mathcal{AI}_M}$ is a template corresponding to the coordinator and member abstract interface, respectively. Further, $M(t_{\mathcal{AI}_C}, \mathcal{I})$ is the membership predicate that for the coordinator-specific knowledge item variables contains the current values stored in the coordinator's knowledge repository ($t_{\mathcal{AI}_C}$). Further, $X(t_{\mathcal{AI}_C}, t_{\mathcal{AI}_M}) \restriction_{\mathcal{AI}_C}$ is the outcome of the knowledge exchange X applied to the values queried via $T_{\mathcal{AI}_C}$ and $T_{\mathcal{AI}_M}$, restricted to $\mathcal{AI}_C$ – the knowledge relevant to the coordinator. Similarly, $X(t_{\mathcal{AI}_C}, t_{\mathcal{AI}_M}) \restriction_{\mathcal{AI}_{M_i}}$ is the outcome of the knowledge exchange restricted to $\mathcal{AI}_{M_i}$ – the knowledge relevant to the member number i. In summary, $P_{kex(E)}$ recursively loads the current values of the knowledge items relevant to the ensemble from the coordinator and member knowledge repositories and stores back the outcome of the knowledge exchange function.

To summarize, a component $C = \mathcal{I}[\mathcal{K}, \Pi, P]$ will consist of (i) an interface $\mathcal{I}$ aggregating all the reified abstract interfaces

$$\mathcal{I} \stackrel{def}{=} \bigcup_{\substack{\mathcal{AI},\\ C \text{ reifies } \mathcal{AI}}} \mathcal{K}_{\mathcal{AI}}$$

(ii) knowledge $\mathcal{K}$ reflecting the component's interface (i.e., $\mathcal{I} \subseteq \mathcal{K}$), (iii) a policy Π including the in-

terleaving interaction predicate defined by the inference rules reported in Table 2, and (iv) a controlled composition P of the processes refining both relevant single-stakeholder invariants and knowledge exchange

$$P \triangleq \textbf{nil} \left[P_{loc(I)}\right]_{I \text{ is a single-stakeholder invariant relevant to } C} \left[P_{kex(E)}\right]_{C \text{ reifies the coordinator abstract interface of } E}$$

For details on how ensembles are reflected in DEECo, we refer the interested reader to [BGH+12].

7 Related work

The term "ensemble" has been recently introduced in the literature (see, e.g., [Int07, HRW08, WSJ+10]) to denote a category of systems characterized by heterogeneous collections of computing resources, huge number of potential interactions, context-awareness, dynamically changing network topologies, and unreliable communications. A mathematical model of ensembles and their composition has been introduced in [HW11]. Ensembles and their constituent parts are abstractly described as relations on sets of inputs and outputs. The "black-box" view of adaptivity is then formally defined. This leads to a preorder relation on ensembles which captures the ability of ensembles to satisfy goals or maximize a performance measure in different environments. Differently from this denotational model, we introduce an operational model of ensembles and a formal language that permits the description of ensembles in a compact and formal way.

Declarative programming has been proposed as an approach to program ensembles. For example, in [ARLG+09] the declarative language Meld [ARGL+07], originally designed for programming overlay networks, is used. Meld allows ensembles to be programmed as a unified whole from a global perspective and then to be compiled automatically into fully distributed local behaviors. This approach is somehow reminiscent of Declarative Networking [LCG+09], a programming methodology that supports the high level specification of network protocols and services, that are then compiled into a dataflow framework and executed. SCEL, instead, is a formal language that could be used as the core of a programming language for ensembles.

Context-Oriented Programming (COP) [HCN08] has also been advocated to program autonomic systems [SGP11]. It exploits ad-hoc explicit language-level abstractions to express context-dependent behavioral variations and their run-time activation. So far, most efforts have been directed towards the design and implementation of concrete languages. Only few works provide a foundational account of programming languages extended with COP facilities, as e.g. the object-oriented ones of [IPW01, CCT09, HIM11] and the functional one of [DFGM12]. All these approaches are however quite different from ours, that instead focusses on distribution and attribute-based aggregations and supports a highly dynamic notion of adaptation.

In the area of concurrency theory, calculi such as [BRF04, AK09], relying on the (bio)chemical programming paradigm, have been proposed for the specification of autonomic systems. Some other formalisms, like e.g. CWS [MS06] and ω-calculus [SRS10], aiming at modelling dynamically changing network topologies, a feature common to many types of distributed systems and to ensembles, can also be source of inspiration for linguistic primitives for specifying autonomic systems. Compared to these proposals, SCEL allows one to provide high-level abstract descriptions of systems that nevertheless have a direct correspondence with their implementation.

An extensive effort has been put into investigating software design methods based on requirements analysis, such as goal-oriented requirements engineering [vL01]. Although similarities to goal-oriented approaches like Tropos [BGG+03] can be found, the design method presented here focuses on an integrated view on the system's requirements and architecture.

8 Concluding Remarks and Work Plan for Year Three

We have introduced SCEL, a new language that brings together various programming abstractions that permit directly representing *knowledge*, *behaviors* and *aggregations* according to specific *policies*, and naturally programming interaction, adaptation and self- and context-awareness. We have then introduced a language for defining access control policies and shown its integration with SCEL, and described how different knowledge management primitives can live together well in a SCEL dialect for concurrent constraint programming. We have also outlined a runtime environment for developing autonomic and adaptive systems according to the SCEL paradigm, as well as the high-level design of SCEL-based applications using the DEECo component model.

Our language-based approach permits to govern the complexity of the systems under consideration by providing flexible abstractions, enabling transparent monitoring of the involved entities and supporting adaptation with different granularity. Besides, SCEL is based on solid semantic grounds which lay the basis for developing logics, tools and methodologies for formal reasoning about systems behavior in order to establish qualitative and quantitative properties of both the individual components and their ensembles.

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

During the third year we plan to do further work along the lines described above.

SCEL at Work. We will assess the extent to which SCEL achieves its goals. As testbeds we will use different scenarios defined in the ASCENS project for three case studies: Robotics (collective transport), Cloud-computing (transiently available computers), and e-Mobility (cooperative e-vehicles). This process might require further tuning the language features and, hence, the related jRESP implementation. Thus we will try to check whether it would be useful to have non blocking variants of get and query, or variants of group-oriented communication with a limited number of recipients, or a language for predicates, like, e.g., a decidable subset of first order logic.

Extensions of jRESP. We plan to integrate in the runtime environment jRESP a Java library supporting the specification and evaluation of SACPL policies and authorization requests. The implementation of such library would rely on the formal semantics of SACPL. We intend as well to experiment with the integration in jRESP of some constraint solvers in order to provide an implementation of the CCSCEL primitives.

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

Stochastic extensions of SCEL. We plan to develop formal tools that permit supporting quantitative analysis of self-adaptive systems specified in SCEL. To this purpose, we plan to define a timed/stochastic extension of SCEL where *time* is explicitly considered and described by means of *random*

variables. Moreover, we plan to define a logic to express *quantitative properties* of SCEL systems. Finally, we will consider model-checking algorithms that permit verifying whether a specification satisfies a given logical property.

From SCEL to BIP. We plan also to consider the possibility of using the operational semantics of SCEL as a starting point to generate systems' descriptions that can be provided as input to the BIP toolset. The challenge here is understanding how the dynamic part of SCEL specifications can be "constrained" to provide a full model to be analyzed in BIP.

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

Alternative Knowledge Managers. We would like to experiment with the management of different forms of knowledge representation, such as those based on concurrent constraints, logic programming, and KnowLang. This would require to integrate such alternative knowledge managers, at formal level, into SCEL and then, at implementation level, in jRESP. Moreover, to improve interoperability among heterogenous knowledge repositories, we plan to consider an ontology-based approach.

Adaptation Patterns. We intend to experiment with modelling the various adaptation patterns studied in WP4 using SCEL. Specifically, we would start by modelling the static architecture of different adaptation patterns, and then we would move on to investigate the use of the SCEL's attribute-based approach for expressing dynamic adaptation patterns, where not only the pattern participants but also the pattern itself may dynamically change (the latter could be considered as a sort of *meta-adaptation*).

References

[AK09] Oana Andrei and Hélène Kirchner. A Higher-Order Graph Calculus for Autonomic Computing. In *Graph Theory, Computational Intelligence and Thought*, pages 15–26. Springer, 2009.

[ARGL+07] Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, and Padmanabhan Pillai. Meld: A declarative approach to programming ensembles. In *IROS*, pages 2794–2800. IEEE, 2007.

[ARLG+09] Michael P. Ashley-Rollman, Peter Lee, Seth Copen Goldstein, Padmanabhan Pillai, and Jason Campbell. A language for large ensembles of independently executing nodes. In Patricia M. Hill and David Scott Warren, editors, *ICLP*, volume 5649 of *LNCS*, pages 265–280. Springer, 2009.

[BGG+03] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos, and Anna Perini. Tropos: An agent-oriented software development methodology, 2003.

[BGH+12] Tomas Bures, Ilias Gerostathopoulos, Vojtech Horky, Jaroslav Keznikl, Jan Kofron, Michele Loreti, and Frantisek Plasil. Language Extensions for Implementation-Level Conformance Checking. ASCENS Deliverable D1.5, 2012.

[BMPT12] Michele Boreale, Ugo Montanari, Rosario Pugliese, and Francesco Tiezzi. Constraint programming with SCEL. Technical Report, September 2012. `http://rap.dsi.unifi.it/scel/`.

[BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.

[BMR01] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based contstraint logic programming: syntax and semantics. *ACM Trans. Program. Lang. Syst.*, 23(1):1–29, 2001.

[BRF04] Jean-Pierre Banâtre, Yann Radenac, and Pascal Fradet. Chemical Specification of Autonomic Systems. In *IASSE*, pages 72–79. ISCA, 2004.

[CCT09] Dave Clarke, Pascal Costanza, and Éric Tanter. How should context-escaping closures proceed? In *Proc. of COP'09*, pages 1:1–1:6, New York, NY, USA, 2009. ACM.

[DFGM12] Pierpaolo Degano, Gian-Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. Typing for coordinating secure behavioural variations. In *Coordination Models and Languages*, volume 7274 of *LNCS*. Springer, 2012.

[DFLP11] Rocco De Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. Languages primitives for coordination, resource negotiation, and task description. ASCENS Deliverable D1.1, September 2011. `http://rap.dsi.unifi.it/scel/`.

[DFLP12] Rocco De Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. A language-based approach to autonomic computing. In *Proc. of the 10th International Symposium on Software Technologies Concertation on Formal Methods for Components and Objects (FMCO 2011)*, Lecture Notes in Computer Science. Springer, 2012. To appear.

[DLPT12] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. SCEL: a Language for Autonomic Computing. Technical Report, September 2012. http://rap.dsi.unifi.it/scel/.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[HIM11] Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. ContextFJ: a minimal core calculus for context-oriented programming. In *Proceedings of the 10th international workshop on Foundations of aspect-oriented languages*, FOAL '11, pages 19–23, New York, NY, USA, 2011. ACM.

[HRW08] Matthias M. Hölzl, Axel Rauschmayer, and Martin Wirsing. Engineering of software-intensive systems: State of the art and research challenges. In Martin Wirsing, Jean-Pierre Banâtre, Matthias M. Hölzl, and Axel Rauschmayer, editors, *Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, pages 1–44. Springer, 2008.

[HW11] Matthias M. Hölzl and Martin Wirsing. Towards a system model for ensembles. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *LNCS*, pages 241–261. Springer, 2011.

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

[Int07] Project InterLink. http://interlink.ics.forth.gr/central.aspx, 2007. Last accessed: 2011-11-28.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[KBPK12] Jaroslav Keznikl, Tomas Bures, Frantisek Plasil, and Michal Kit. Towards Dependable Emergent Ensembles of Components: The DEECo Component Model. In *Proceedings of WICSA/ECSA 2012*. IEEE, August 2012.

[LCG+09] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, November 2009.

[Lor12] Michele Loreti. jRESP: a Run-time Environment for SCEL Programs. Technical Report, September 2012. http://rap.dsi.unifi.it/scel/.

[MS06] Nicola Mezzetti and Davide Sangiorgi. Towards a calculus for wireless systems. *Electr. Notes Theor. Comput. Sci.*, 158:331–353, 2006.

[NIS09] NIST. A survey of access control models, 2009. http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf.

[OAS05] OASIS XACML TC. eXtensible Access Control Markup Language (XACML) version 2.0, 2005. http://docs.oasis-open.org/xacml/2.0/XACML-2.0-OS-NORMATIVE.zip.

[PM12] Olga Pustovalova and Ugo Montanari. Constraint Logic Programming for Service-Oriented Computing: A Case Study in Prova. Technical report, IMT Institute for Advanced Studies Lucca, 2012. Available online at `http://www.imtlucca.it/olga.pustovalova`.

[PT12] Rosario Pugliese and Francesco Tiezzi. SACPL: a Simple Access Control Policy Language. Technical Report, September 2012. `http://rap.dsi.unifi.it/scel/`.

[SCC+12] Ian Sommerville, Dave Cliff, Radu Calinescu, Justin Keen, Tim Kelly, Marta Z. Kwiatkowska, John A. McDermid, and Richard F. Paige. Large-scale complex it systems. *Commun. ACM*, 55(7):71–77, 2012.

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

[SMB+12] Nikola Serbedzija, Mieke Massink, Manuele Brambilla, Diego Latella, Marco Dorigo, and Mauro Birattari. Ensemble Model Syntheses with Robot, Cloud Computing and e-Mobility. ASCENS Deliverable D7.2, October 2012.

[SR90] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *POPL*, page 232245. ACM Press, 1990.

[SRS10] Anu Singh, C. R. Ramakrishnan, and Scott A. Smolka. A process calculus for mobile ad hoc networks. *Sci. Comput. Program.*, 75(6):440–469, 2010.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4(2), 2009.

[vL01] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour, 2001.

[WSJ+10] Roy Want, Eve Schooler, Lenka Jelinek, Jaeyeon Jung, Dan Dahle, and Uttam Sengupta. Ensemble computing: Opportunities and challenges. *Intel Technology Journal*, 14(1):118–141, 2010.