Self-expression and Dynamic Attribute-Based Ensembles in SCEL

Giacomo Cabri[1], Nicola Capodieci[1], Luca Cesari[2], Rocco De Nicola[3], Rosario Pugliese[2], Francesco Tiezzi[3], and Franco Zambonelli[1]

[1] Università degli Studi di Modena e Reggio Emilia, Via Amendola, 2 - 42122 Reggio Emilia, Italy
{giacomo.cabri,nicola.capodieci,franco.zambonelli}@unimore.it

[2] Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy
{luca.cesari,rosario.pugliese}@unifi.it

[3] IMT Institute for Advanced Studies Lucca, Piazza S. Francesco, 19 - 55100, Lucca, Italy
{rocco.denicola,francesco.tiezzi}@imtlucca.it

Abstract. In the field of distributed autonomous computing the current trend is to develop cooperating computational entities enabled with enhanced self-* properties. The expression self-* indicates the possibility of a component inside an ensemble, i.e. a set of collaborative autonomic components, to self organize, heal (repair), optimize and configure with little or no human interaction. We focus on a self-* property called *self-expression*, defined as the ability to deploy run-time changes of the coordination pattern of the observed ensemble; the goal of the ensemble is to achieve adaptivity by meeting functional and non-functional requirements when specific tasks have to be completed. The purpose of this paper is to rigorously present the mechanisms involved whenever a change in the coordination pattern is needed, and the interactions that take place. To this aim, we use SCEL (Software Component Ensemble Language), a formal language for describing autonomic components and their interactions, featuring a highly dynamic and flexible way to form ensembles based on components' attributes.

Keywords: Self-expression, coordination patterns, ensemble computing.

1 Introduction

The current trend in designing distributed systems is to conceive them as ensembles of several, possibly heterogeneous, components. Ensembles are often required to solve complex problems of real life, even situations in which the level of interaction between humans and components of the ensemble is strongly limited or even absent. Therefore, their components usually collaborate with each other in order to achieve a common goal. This calls for further features that increase the self-management capability of the systems, such as self-configuration, self-healing, self-optimization, and self-protection, leading to what is known in literature as self-* properties in autonomic computing [16,31]. The goal is to

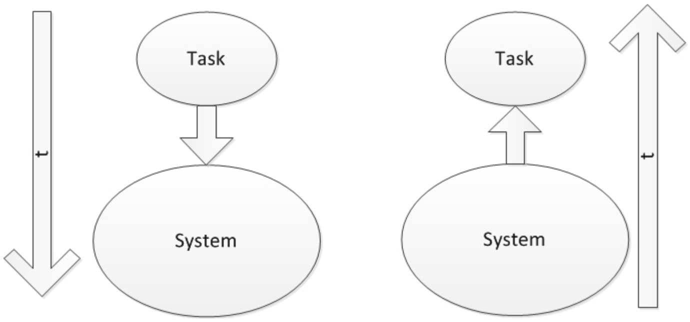

Fig. 1. A two-perspective relationship between task to solve and surrounding system

have self-adaptive ensembles able to promptly react to dynamic changes of the surrounding environment and to optimize their performance when addressing tasks of variable complexity in the presence of dynamic environments. In addition to the previously listed properties, there can be situations in which changes in the components' coordination pattern (as a refactoring of behaviors, roles and interactions among components) can be useful at runtime, especially in open and non-deterministic scenarios. This change should be autonomously made by the ensemble itself by relying on adaptive collaboration of its components. This ability could enable the ensemble to face unexpected and unpredictable situations, often modelled as changes in the environment or fault tolerance issues [26]. Moreover, it could increase performance and robustness of the designed ensemble, because different coordination patterns could require different utilities or qualities for solving a specific task. The dynamic modification of the coordination pattern according to the changes in the external conditions is called *self-expression* [30], meaning that the autonomic system *expresses* itself (i.e., the system still does what is supposed to do) independently of unexpected situations and, to accomplish this, it is capable of modifying its original internal organization.

This paper aims at showing how to *enable* self-expression in a concrete way by exploiting a formal language for defining ensembles. We refer the interested reader to [30,5] for further motivations and details about self-expression.

1.1 More Details on Self-expression

Enhancing adaptivity of an ensemble through self-expression is a problem that can be seen from two different perspectives, as shown in Fig. 1.

A first perspective (left part of Fig. 1) is to think about engineering an ensemble able to solve a specific problem starting from an initial task that can be then subdivided into several sub-tasks to be assigned to ensemble components. For instance, the task "explore a given area" could be split in the sub-tasks "act as a master that proposes sub-areas to explore", "act as a slave for executing the received orders from a master", "act as a peer to negotiate sub-areas to explore", etc. Self-expression can be then represented by a Business Process Logic (BPL) specification that regulates relationships among sub-tasks and how these are assigned to each component. Since each of these sub-tasks is related to how the ensemble will assign roles, behaviors and interaction rules, the BPL specification provides information on how the ensemble will coordinate its components for solving the specified task. BPL specifications are subject to change over time according to the dynamics of the surrounding environment, so to have ensembles whose components will undergo modifications of the sub-tasks assignment and of the way they will coordinate according to the changes in the external conditions.

A second perspective (right part of Fig. 1) is to think about an already existing system that potentially is able to solve a multitude of tasks and for each of these tasks, each component of the system knows different *ways* to collaboratively complete it. For instance, some available tasks could be "area exploration" and "dragging/pushing objects outside the area", while the different ways to accomplish them could be "master-slave", "peer-to-peer" and "swarm". Later on, a request from outside, or a specific contingency, could prepare the system for solving a specific task. Self-expression here is seen as the capability to collaboratively select the fittest way, according to the currently perceived environmental conditions, for solving the selected task. The fittest collaborative effort can be thought of as a coordination pattern that results in an appropriate Quality of Service (QoS). Its selection is a decision that is ideally shared throughout the whole ensemble. As external conditions change over time, the ensemble has to adapt itself by choosing a different way to coordinate.

The modelling and description of the mechanisms for deploying self-expression according to this latter perspective present interesting challenges and are investigated in the rest of the paper by using a formal language specifically designed for defining ensembles.

1.2 Self-expression in a Formal Language for Defining Ensembles

The language SCEL (Software Components Ensemble Language) [10,11] has been introduced to deal with the challenges posed by the design of ensembles of autonomic components. In SCEL, autonomic components are entities with dedicated knowledge repositories and resources that can cooperate while playing different roles. Knowledge repositories also enable components to store and retrieve information about their working environment, and to use it for redirecting and adapting their behavior. Each component is equipped with an *interface*, consisting of a collection of *attributes*, such as provided functionalities, spatial coordinates, group memberships, trust level, response time, etc. Attributes are used by the components to dynamically organize themselves into ensembles.

The way sets of partners are selected for interaction, and thus how ensembles are formed, is one of the main novelties of SCEL. In fact, individual components not only can single out communication partners by using their identities, but they can also select partners by exploiting the attributes in the interfaces of the individual components. Predicates over such attributes are used to specify the targets of communication actions, thus providing a sort of *attribute-based* communication. In this way, the formation rule of ensembles is endogenous to components: members of an ensemble are connected by the interdependency relations defined through predicates. An ensemble is therefore not a rigid fixed network, but rather a highly dynamic structure where components' linkages are dynamically established.

The purpose of this work is to show that SCEL can be conveniently exploited to naturally model ensembles able to deploy self-expression. Indeed, ensembles can be addressed as single entities (by exploiting predicates) and, at the same time, are composed of sub-entities (the ensemble components, which are the actual recipients of ensemble invocations). Our characterization has the additional benefit of fostering dynamic identification of sub-sets of ensembles, since ensembles are highly dynamic structures where components linkages are dynamically established.

The rest of the paper is organized as follows. In Section 2, we recap the SCEL language, while in Section 3 we show how to implement self-expression with it. In Section 4, we present an application of our approach to a case study from the robotics domain. Section 5 discusses more strictly related work. Finally, in Section 6, we draw some conclusions and sketch how our approach can be further extended.

2 SCEL: Software Component Ensemble Language

SCEL is a kernel language for programming autonomic computing systems in terms of Behaviors, Knowledge and Aggregations, according to specific Policies. *Behaviors* describe how computations progress and are modeled as processes executing actions. *Knowledge* is represented through items containing either application data enabling the progress of components' computations, or awareness data providing information about the environment in which the components are running (e.g. monitored data from sensors) or about the status of a component (e.g. its current location). *Aggregations* describe how different entities are brought together to form components and ensembles. In particular, components result from a form of syntax-based aggregation that puts together a knowledge repository, a set of policies and a set of behaviors, by wrapping them in an interface providing a set of *attributes*, i.e. names referring to information stored in the knowledge repository. Components' composition and interaction are implemented by exploiting the attributes exposed in components' interfaces. This form of semantics-based aggregation of components permits defining ensembles, representing social or technical networks of components, and configuring them to dynamically adapt to changes in the environment. Finally, *policies* control and adapt the actions of the different components for guaranteeing accomplishment of specific tasks or satisfaction of specific properties.

The syntax of SCEL is presented in Table 1. There, different syntactic categories are defined that constitute the main ingredients of the language. The basic category is the one defining PROCESSES that are used to build up COMPONENTS that in turn are used to define SYSTEMS. PROCESSES specify the flow of the ACTIONS that can be performed. ACTIONS can have a TARGET to determine the other components that are involved in that action. The rest of this section is devoted to the description of the SCEL's syntactic categories.

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

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

Systems and components. The key notion is that of *component* $\mathcal{I}[\mathcal{K},\Pi,P]$, that is graphically depicted in Figure 2 and 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 (not necessarily unique) name of the component.
2. A *knowledge repository* $\mathcal{K}$ managing both application data and awareness data, together with a specific handling mechanism providing operations for *adding*, *retrieving*, and *withdrawing* knowledge items. The knowledge repository of a component stores also the information associated to its interface, which therefore can be dynamically manipulated by means of the operations provided by the knowledge repositories' handling mechanisms.
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.

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, restricted names can be exchanged in communications thus enabling the receiving components to use those "private" names.

Processes. PROCESSES are the active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* ($a.P$), *nondeterministic choice* ($P_1 + P_2$), *controlled composition* ($P_1[\,P_2\,]$), *process variable* (X), and *parameterized process invocation* ($A(\bar{p})$). We will omit trailing occurrences of **nil**, writing e.g. a instead of $a.\mathbf{nil}$. The construct $P_1[\,P_2\,]$ abstracts the various forms of parallel composition commonly used in process calculi. Process variables can support *higher-order* communication, namely the capability to exchange (the code of) a process, and possibly execute it, by first adding an item containing the process to a knowledge repository and then retrieving/withdrawing this item

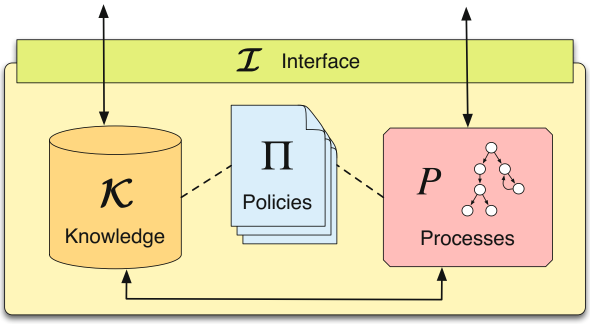

Fig. 2. A SCEL component

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.

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

Action **get** may cause the process executing it to wait for the wanted element if it is not (yet) available in the knowledge repository. Action **qry**, exactly like **get**, may suspend the process executing it if the knowledge repository does not (yet) contain or cannot 'produce' the wanted element. The two actions differ for the fact that **get** removes the found item from the knowledge repository while **qry** leaves the target repository unchanged. Actions **put**, **fresh** and **new** are instead immediately executed, provided that their execution is allowed by the policies in force.

Different entities may be used as the target c of an action. Component names are denoted by $n, n', \ldots$, while variables for names are denoted by $x, x', \ldots$. The distinguished variable self can be used by processes to refer to the name of the component hosting them. The target can also be a *predicate* $\mathcal{P}$ or the name p of a predicate, exposed as an attribute in the interface of the component, that may dynamically change. A predicate could be a boolean-valued expression obtained by applying standard boolean operators to the results returned by the evaluation of relations between attributes and expressions. Attribute names occurring in a predicate refer to attributes within the interface of the object components, i.e. components that are target of the communication action.

In actions using a predicate $\mathcal{P}$ to indicate the target (directly or via p), predicates act as 'guards' specifying *all* components that may be affected by the execution of the action, i.e. a component must satisfy $\mathcal{P}$ to be the target of the action. Thus, actions $\mathbf{put}(t)@n$ and $\mathbf{put}(t)@\mathcal{P}$ give rise to two different primitive forms of communication: the former is a *point-to-point* communication, while the latter is a sort of *group-oriented* communication.

The set of components satisfying a given predicate $\mathcal{P}$ used as the target of a communication action can be considered as the *ensemble* with which the process performing the action intends to interact. For example, the names of the components that can be members of an ensemble can be fixed via the predicate $id \in \{n, m, o\}$. When an action has this predicate as target, it will act on all components named n, m or o, if any. Instead, to dynamically characterize the members of an ensemble that are active and have a battery whose level is higher than *low*, by assuming that attributes *active* and *batteryLevel* belong to the interface of any component willing to be part of the ensemble, one can write $active = yes \land batteryLevel > low$.

3 Self-expression in SCEL

In this section, we provide a step-by-step explanation of how a change in the coordination pattern can be obtained in an ensemble of autonomic components described using SCEL. A visual representation of the pattern workflow is shown in Fig. 3, where requests are represented by red arrows (i.e., darker arrows in b/w) and responses by the green arrows (i.e., lighter arrows in b/w). We first present the workflow execution steps performed by the requester, that is a component that requests the execution of a task, and, then, the steps performed by each involved responder.

Changes in the external conditions should trigger a change in the coordination pattern, since once a task has been selected, each different known implementation is likely to result in a different QoS. The QoS may depend on the current conditions of the surrounding environment, therefore for each observable change in the external conditions, a different coordination pattern could have to be selected in order to obtain the desired QoS.

Suppose that one or more components can rely on a table like the one shown in Table 2 in order to select the fittest implementation, once the whole ensemble agreed on the task to solve. Each row of the table can be represented as an item stored in the knowledge repository of the component. Conditions represent all the important features regarding the surrounding environment in which the ensemble is located; for instance, in case of a robot ensemble, everything that can be perceived through sensors. Implementation, identified by an id, is the actual coordination pattern chosen by the whole ensemble among the different patterns of the specified task. The final column relates to the expected QoS.

To sum it up, each important change in the surrounding environment, i.e. each change causing a different set of conditions k_i to be satisfied, triggers a

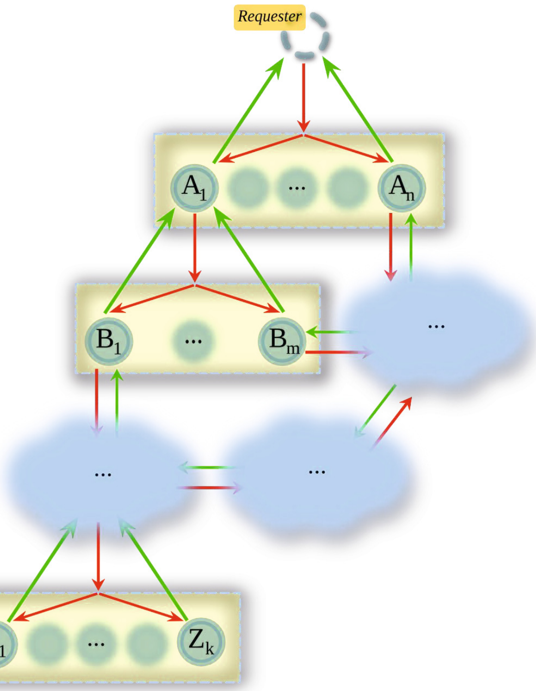

Fig. 3. Workflow of coordination patterns in SCEL

modification of the coordination pattern (i.e., implementation I_i for solving a previously agreed task) to be adopted, so to carry out the specific task with expected quality QoS_i. A single component that is aware that a specific coordination pattern is needed can trigger a dissemination request to all the other components of the ensemble, as we explain in the rest of the section.

3.1 Requester Workflow

We introduce here the steps of the task requester workflow.
Step 1: Task Request. The requester component needs a specific task to be carried out, so by using predicate $\mathcal{P}_r$ it contacts an ensemble of components that could fulfill the task. In order to receive a response, the requester adds its own identifier name (i.e., its component's address) to the request by means of the distinguished variable self, which allows a process to refer to the name of the component hosting it. The requesting action is rendered in SCEL as follows:

$$\mathbf{put}(\text{“}\mathit{taskRequest}\text{”},\ \text{“}\mathit{taskName}\text{”},\ \mathit{QoSconstraints},\ \mathsf{self})@\mathcal{P}_r$$

Notably, before sending the request item, variable self will be replaced by the component identifier running the process performing the above **put** action.

Table 2. Example table of Conditions-Implementation-Expected QoS related to a specified task

Conditions	*Implementation*	*Expected QoS*
k_1	I_1	QoS_t^1
k_2	I_2	QoS_t^2
k_3	I_3	QoS_t^3
$\vdots$	$\vdots$	$\vdots$

The predicate $\mathcal{P}_r$ can be declared, e.g., as:

$$\mathcal{P}_r \triangleq \text{“}taskName\text{”} \in providedTasks$$

where “*taskName*” is the name of the required task and *providedTasks* is an attribute, exposed in the interface of every component, indicating the set of tasks that the component can fulfil.

Step 2: Receipt of proposed implementations. The requester component receives the information about implementations from the contacted components and selects the one that best fits the wanted QoS. Before the selection phase, the component retrieves the proposed implementations from its local repository by means of actions of the form

$$\mathbf{get}(\text{“}implementation\text{”},\ ?implementationName,\ ?QoS,\ ?providers)@\mathsf{self}$$

where variable *implementationName* is bound to the name of a retrieved implementation, *QoS* to the effective QoS of the implementation, and *providers* to the data characterizing the providers of the implementation. The latter information is used to define the predicate $\mathcal{P}_{implementation}$ that will be used to contact the ensemble of components providing the selected implementation.

Step 3: Activation of the selected implementation. By exploiting a table like the one shown in Table 2, the requester selects the fittest implementation according to the currently perceived environmental conditions, the wanted QoS and the information retrieved and elaborated in Step 2. Then, it contacts the selected ensemble by exploiting predicate $\mathcal{P}_{implementation}$. In SCEL, this request can be represented as follows:

$$\begin{aligned}&\mathbf{put}(\text{“}executeImplementation\text{”},\\ &\qquad implementationName,\ arguments)@\mathcal{P}_{implementation}\end{aligned}$$

The *arguments* part can be empty if the selected implementation does not need contextual data.

3.2 Responder Workflow

We present now the steps performed by each responder component. The workflow of a responder component is presented in Fig. 4. Each number shows in which

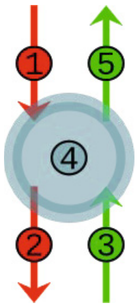

Fig. 4. Steps of a responder component

step the component takes a specific action. Again, the red arrows represent the requests and the green arrows the responses.

Step 1: Task request. Every component reached by a task request can get it by performing the following action:

$$\mathbf{get}(\text{“}\mathit{taskRequest}\text{”},\ ?\mathit{taskName},\ ?\mathit{QoSconstraints},\ ?\mathit{requester})@\mathsf{self}$$

Then, it checks if the requested task (stored in variable *taskName*) is provided by the component itself. If the component provides the task, the workflow execution can directly go to Step 5; anyhow, this depends on the component's selection criterion. If the component does not provide the task, the execution evolves to Step 2. In case of a 'smart' component, if the requested task is complex the responder component can decide to split it in simpler sub-tasks and handle the search of sub-task implementations.

Step 2: Requests dissemination. The responder component contacts an ensemble of components that, according to its knowledge, provides an implementation for the requested task. This operation is carried out similarly to a task request (see Step 1 in Section 3.1), but in this case it is used a different predicate ($\mathcal{P}_d$). The SCEL action used in this step is the following:

$$\mathbf{put}(\text{“}\mathit{taskRequest}\text{”},\ \mathit{taskName},\ \mathit{QoSconstraints},\ \mathsf{self})@\mathcal{P}_d$$

Step 3: Responses collection. Each component that has disseminated a request collects the responses from the contacted components. Notably, if the component itself provides a solution, this is added to the collected responses. The collection phase is driven by a criterion that depends on the application. For example, some criteria are:

- wait for the first response and go to the next step;
- bounce immediately all the received responses to the requester (that is, Step 4 is skipped);
- wait for k responses and go to the next step;
- wait for responses with a specific QoS value and go to the next step;
- wait for a specific amount of time and go to the next step.

Step 4: Implementation selection. The responder component now selects one or more implementations using some criterion. As in Step 3, the criterion depends on the application. Some examples of criteria are:

- select the first k responses;
- take all the received responses;
- select the best j responses according to a specific parameter;
- select the responses with a specific QoS value.

In order to be selected, an implementation must be accompanied by a set of additional information needed by the receiver to take its decision. Thus, an implementation consists of its name, the associated QoS and the data needed to define the predicate for contacting the partners that provide this particular implementation. This information can be expressed as an item with the following form:

$$(\text{“}\mathit{implementation}\text{”},\ \mathit{implementation\_name},\ \mathit{QoS\_data},\ \mathit{providers\_data})$$

Step 5: Response to the requester. After the selection phase, the component will send the results of the selection to the requester, whose identifier was bound to the variable *requester* at Step 1. Thus, for any implementation selected at the previous step, an action of the following form is performed:

$$\begin{array}{l}\mathbf{put}(\text{“}\mathit{implementation}\text{”},\\ \qquad \mathit{implementation\_name},\ \mathit{QoS\_data},\ \mathit{providers\_data})@\mathit{requester}\end{array}$$

4 An Illustrative Example: Multi-robot Exploration Task

In this section we briefly apply our approach to a specific example task. The task is represented by having an ensemble of robots initially randomly distributed in a confined space called *arena*. The robots have to distribute within the arena and start exploring it. The task can be represented as follows:

- id: exploreArena;
- Input: ensemble randomly distributed in an unexplored arena;
- Output: explored arena;
- QoS: minimize the Time-To-Complete (*minTTC*), equally distribute the workload among the robots (*eqDist*).

Regarding the implementations, we can identify three main coordination patterns for executing the task:

- master-slave (id: MS): a robot sends orders about areas to explore to a set of slaves;
- peer-to-peer (id: p2p): robots will ideally subdivide the arena into areas and then negotiate areas to explore;

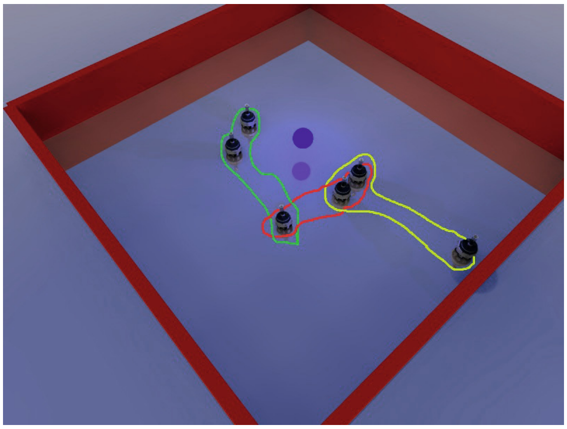

Fig. 5. A small ensemble is randomly distributed in the arena

– swarm (id: SW): all the robots randomly diffuse in the arena and mark areas with digital pheromones. If a robot detects a pheromone, a repulsion effect will take place, causing this latter robot to move to (and, therefore, explore) other areas.

The previous task and its respective different implementations are depicted in Fig. 5, where a small ensemble is randomly distributed in the arena. Different robots *own* different implementations. To own an implementation means that a component has all the processes, in the form of dynamically activable behaviors, that are needed in order to adopt a determined coordination pattern. In the figure, the robots are grouped according to the processes relative to the implementations they own: green for MS, yellow for p2p, and red for SW.

The requester actions are as follows.

Step 1: An external command or a contingency that reaches a single robot enforces the action:

$$\mathbf{put}(\text{“}\mathit{taskRequest}\text{”},\ \text{“}\mathit{exploreArea}\text{”},\ minTTC,\ \mathsf{self})@\mathcal{P}_r$$

with $\mathcal{P}_r \triangleq$ “*exploreArea*” $\in providedTasks$.

Step 2:

$$\mathbf{get}(\text{“}\mathit{implementation}\text{”},\ ?implementationName,\ ?QoS,\ ?providers)@\mathsf{self}$$

where variable *implementationName* gets one of the values MS, $p2p$, and SW, QoS indicates if that implementation aims at minimising TTC and/or to equally distribute the workload among the robots and *providers* is bound to the IDs of all the responding robots. These IDs are collected in order to define $\mathcal{P}_{implementation}$.

Step 3: At this point the requester robot knows which robot can execute the *exploreArea* task and according to which implementation; it can then choose the implementation that is likely to satisfy the desired QoS. To make this choice, the robot can ideally rely on a table like the one in Table 3, where

Table 3. *exploreArea* table of defined conditions

Conditions	*Implementation*	*Expected QoS*
k'	MS	QoS'
k''	$p2p$	QoS''
k'''	SW	QoS'''

- k': refers to the condition for optimally exploiting a master-slave approach, such as the presence of at least one robot with additional sensing capabilities;
- k'': refers to the conditions in which a peer-to-peer approach is possible, like communication capabilities and easiness to identify areas before negotiation;
- k''': refers to the conditions in which it would be suitable to use a swarm approach for the area exploration task, such as a sufficiently large number of available units.

Depending on how these three patterns are implemented, we can think that a swarm approach will perform better in terms of minimizing exploration time, while a peer-to-peer negotiation could more equally distribute energy consumption among the components. The master-slave approach could minimize Time-To-Complete and distribute the workload more equally, but it is less robust than the other ones because the master constitutes a single point of failure. Now, if we assume that the environmental conditions k' are sensed by the robots, the requester can perform the action

$$\mathbf{put}(\text{“}executeImplementation\text{”}, MS, \text{“}MasterID\text{”})@\mathcal{P}_{implementation}$$

where *MasterID* identifies the robot that will take the role of master (according to some internal logic of the component) and $\mathcal{P}_{implementation}$ potentially involves all the components whose identifiers have been collected in Step 2.

Once an implementation is selected by the requester, all the responding robots will start following that coordination pattern. If a robot does not have the necessary code embedded in its controller, we may think that a code migration process will be executed. Moreover, if every ensemble component is able to communicate with all other components, as in Fig. 5, the responder just executes Steps 1, 4 and 5 as described in Section 3.

5 Related Work

The first definitions of self-* properties can be traced back to the well-known manifestos by IBM in [14] and [16] about autonomic computing. From the point of view of the designer, a fairly complete survey on the efforts of designing autonomic systems with traditional methodologies, mainly coming from standard software engineering methodologies, can be found in [15]. In the design phase, more challenges arise when the observed systems are actually composed of large sets of potentially heterogeneous components. In this case indeed the blueprints for adaptive feedback loops (like IBM's MAPE-K [14]) have to be thought of as distributed, thus problems regarding inter-components coordination could become to light. More recently, [4] shows a complete life cycle for the design and development of ensembles of collaborative autonomic components.

Self-expression, as detailed in Section 1, is an additional instrument for the designer of autonomic distributed systems. However, in previous literature, the term self-expression is used for both describing reconfigurations at level of design patterns [6] and for describing reconfigurations at the level of roles, behaviors and interactions among components [7]. While a summary of possible real world applications for self-expression is presented in [5], a modelling choice that can help us understanding the concept of self-expression is the Holonic paradigm for Multi-Agent Systems (HMAS, [28]). Holons, i.e. self-repeating structures organized in hierarchies, present specific interfaces called capacities. A capacity is defined as a description of a know-how/service and can be associated to different implementations (representing different ways of providing that capacity). In our case, we can think that holons are single components, or subsets of the entire ensemble, and that a coordination pattern is the implementation of a capacity. To each implementation corresponds an organizational level in which behaviors, roles (i.e. specific states inside the same organization) and interactions (i.e. how parts in the same level influence each other) characterize a set of holons.

In the area of distributed artificial intelligence and multiagent systems [32], the idea of dynamically forming ensembles or coalitions of agents – getting together to cooperatively work towards some collective goals – has been extensively analyzed [13,17]. However, the accent of such researches has been mostly at analyzing the different strategies and algorithms for forming the ensembles and for controlling their cooperative behavior, rather than in the actual mechanisms to model and implement ensembles of agents capable of expressing the needed self-adaptive coordination scheme.

For what concerns SCEL, it combines the notion of ensemble with concepts that have emerged from different research fields of Computer Science and Engineering. Indeed, it borrows from software engineering the importance of component-based design and of separation of concerns [20], from multi-agent systems the relevance of knowledge handling and of spatial representation [27,3,29,2,9], from middleware and network architectures the importance of flexibility in communication [22,8,18,25,23], from distributed systems' security the role of policies [24], from actors and process algebras the importance of minimality and formality [1,21]. Summing it up, the main distinctive aspect of SCEL is the actual choice of the specific programming abstractions for autonomic computing and their reconciliation under a single roof with a uniform formal semantics. For a more complete account about SCEL and works related to it, we refer the interested reader to [12].

6 Concluding Remarks

In this paper we have illustrated how to foster the self-adaptive features of an ensemble of autonomic components by describing a previously introduced property called self-expression. More specifically, we have exploited SCEL as a language for properly describing and modeling the mechanisms involved in the run-time changes of coordination patterns. The rigorous grammar and the formal semantics that characterize SCEL provide a valuable instrument for understanding (1) *how* a change in the collaborative structure of the ensemble is performed and (2) *when* a change of the coordination pattern is needed. Regarding (1), we have presented a step-by-step description of inter-component interactions by means of workflows, in which we stressed how different requests that may lead to change of the coordination pattern can be disseminated among different parts of the observed ensemble. Regarding (2), we showed how the selection of the fittest pattern depends both on the current perceived environmental conditions and on the expected QoS: new patterns will have to be selected according to the dynamics of the variations in the external conditions and/or QoS. A simple, yet explicative case study in robotics is demonstrated to further clarify the presented concepts.

We are currently investigating how a component could autonomously extend and modify the table regarding Conditions/Implementations/Expected QoS so to provide more possibilities in terms of adaptivity. We will apply our approach to other case studies, not necessarily in the robotics domain.

We also plan to investigate the use of SCEL components policies to drive and regulate the selection of implementations and coordination patterns according to possibly locally different criteria. Specifically, according to the approach introduced in [19], we plan to use the FACPL language to express policies.

To show the effectiveness of the proposed SCEL-based solution to self-expression and provide a more concrete evidence of its benefits, we intend to implement the approach considered in this work in jRESP [12], a Java runtime environment for developing autonomic and adaptive systems according to the SCEL paradigm. In particular, jRESP provides a simulation environment that enables statistical model-checking, which will allow us to verify qualitative and quantitative properties of SCEL programs.

References

1. Agha, G.A.: ACTORS - a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence. MIT Press (1990)
2. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley Series in Agent Technology. John Wiley & Sons (2007)
3. Bordini, R.H., Hübner, J.F., Vieira, R.: Jason and the golden fleece of agent-oriented programming. In: Multi-Agent Programming. Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15, pp. 3–37. Springer (2005)
4. Bureš, T., De Nicola, R., Gerostathopoulos, I., Hoch, N., Kit, M., Koch, N., Monreale, G., Montanari, U., Pugliese, R., Serbedzija, N., Wirsing, M., Zambonelli, F.: A life cycle for the development of autonomic systems: The e-mobility showcase. In: 3rd Workshop on Challenges for Achieving Self-Awareness in Autonomic Systems. IEEE (2013)

5. Cabri, G., Capodieci, N.: Runtime change of collaboration patterns in autonomic systems: Motivations and perspectives. In: AINA Workshops, pp. 1038–1043. IEEE (2013)
6. Cabri, G., Puviani, M., Zambonelli, F.: Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In: CTS, pp. 508–515. IEEE (2011)
7. Capodieci, N., Hart, E., Cabri, G.: Designing self-aware adaptive systems: from autonomic computing to cognitive immune networks. In: 3rd Workshop on Challenges for Achieving Self-Awareness in Autonomic Systems. IEEE (2013)
8. Costa, P., Mottola, L., Murphy, A.L., Picco, G.: Tuple Space Middleware for Wireless Networks. In: Middleware for Network Eccentric and Mobile Applications, pp. 245–264. Springer (2009)
9. Dastani, M.: 2APL: A practical agent programming language. Autonomous Agents and Multi-Agent Systems 16(3), 214–248 (2008)
10. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: A language-based approach to autonomic computing. In: Beckert, B., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 25–48. Springer, Heidelberg (2012)
11. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: SCEL: A language for autonomic computing. TR (2013), `http://rap.dsi.unifi.it/scel/pdf/SCEL-TR.pdf`
12. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL Language. Transactions on Autonomous and Adaptive Systems (to appear, 2014)
13. Durfee, E., Lesser, V., Corkill, D.D.: Trends in cooperative distributed problem solving. IEEE Transactions on Knowledge and Data Engineering 1(1), 63–83 (1989)
14. Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical Report, October 15 (2001)
15. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing – degrees, models, and applications. ACM Comput. Surv. 40(3), 7:1–7:28 (2008)
16. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
17. Klusch, M., Gerber, A.: Dynamic coalition formation among rational agents. IEEE Intelligent Systems 17(3), 42–47 (2002)
18. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The TOTA approach. ACM Trans. Softw. Eng. Methodol. 18(4) (2009)
19. Margheri, A., Pugliese, R., Tiezzi, F.: Linguistic abstractions for programming and policing autonomic computing systems. In: UIC/ATC, pp. 404–409. IEEE (2013)
20. McKinley, P., Sadjadi, S., Kasten, E., Cheng, B.H.C.: Composing adaptive software. Computer 37(7), 56–64 (2004)
21. Milner, R.: Communication and concurrency. PHI Series in Computer Science. Prentice Hall (1989)
22. Mottola, L., Picco, G.P.: Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks. In: Gibbons, P.B., Abdelzaher, T., Aspnes, J., Rao, R. (eds.) DCOSS 2006. LNCS, vol. 4026, pp. 150–168. Springer, Heidelberg (2006)
23. Mottola, L., Picco, G.P.: Middleware for wireless sensor networks: an outlook. J. Internet Services and Applications 3(1), 31–39 (2012)
24. NIST. A survey of access control models (2009), `http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf`
25. Nordström, E., Gunningberg, P., Rohner, C.: A search-based network architecture for mobile devices. Uppsala University, TR 2009-003 (2009)

26. Puviani, M., Pinciroli, C., Cabri, G., Leonardi, L., Zambonelli, F.: Is Self-Expression Useful? Evaluation by a Case Study. In: Reddy, S., Jmaiel, M. (eds.) WETICE (AROSA Track), pp. 62–67. IEEE (2013)
27. Rao, A.S.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: Van de Velde, W., Perram, J.W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
28. Rodriguez, S., Gaud, N., Hilaire, V., Galland, S., Koukam, A.: An analysis and design concept for self-organization in holonic multi-agent systems. In: Brueckner, S.A., Hassas, S., Jelasity, M., Yamins, D. (eds.) ESOA 2006. LNCS (LNAI), vol. 4335, pp. 15–27. Springer, Heidelberg (2007)
29. Winikoff, M.: JACK^{TM} Intelligent Agents: An Industrial Strength Platform. In: Multi-Agent Programming. Multiagent Systems, Artificial Societies, and Simulated Organizations, pp. 175–193. Springer (2005)
30. Zambonelli, F., Bicocchi, N., Cabri, G., Leonardi, L., Puviani, M.: On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles. In: SASO Workshops, pp. 108–113. IEEE (2011)
31. Zambonelli, F., Castelli, G., Ferrari, L., Mamei, M., Rosi, A., Serugendo, G.D.M., Risoldi, M., Tchao, A.-E., Dobson, S., Stevenson, G., Ye, J., Nardini, E., Omicini, A., Montagna, S., Viroli, M., Ferscha, A., Maschek, S., Wally, B.: Self-aware pervasive service ecosystems. Procedia CS 7, 197–199 (2011)
32. Zambonelli, F., Omicini, A.: Challenges and research directions in agent-oriented software engineering. Autonomous Agents and Multi-Agent Systems 9(3), 253–283 (2004)