Formalising Adaptation Patterns for Autonomic Ensembles*

Luca Cesari[1,4], Rocco De Nicola[2], Rosario Pugliese[1], Mariachiara Puviani[3], Francesco Tiezzi[2], and Franco Zambonelli[3]

[1] Università degli Studi di Firenze, Italy
[2] IMT Advanced Studies Lucca, Italy
[3] Università degli Studi di Modena e Reggio Emilia, Italy
[4] Università di Pisa, Italy

Abstract. Autonomic behavior and self-adaptation in software can be supported by several architectural design patterns. In this paper we illustrate how some of the component- and ensemble-level adaptation patterns proposed in the literature can be rendered in SCEL, a formalism devised for modeling autonomic systems. Specifically, we present a compositional approach: first we show how a single generic component is modelled in SCEL, then we show that each pattern is rendered as the (parallel) composition of the SCEL terms corresponding to the involved components (and, possibly, to their environment). Notably, the SCEL terms corresponding to the patterns only differ from each other for the definition of the predicates identifying the targets of attribute-based communication. This enables autonomic ensembles to dynamically change the pattern in use by simply updating components' predicate definitions, as illustrated by means of a case study from the robotics domain.

1 Introduction

In the era of *autonomic computing* [1], where computer and software systems must manage themselves and their components, (self-)adaptation is a key aspect of software design. Self-adaptation is defined as the ability of a system to autonomously adapt its behaviour and/or structure to dynamic operating conditions [2], so as to preserve its capability of delivering the necessary services with acceptable quality levels. It is a key feature for *ensembles* [3], namely open-ended, large-scale and highly-parallel distributed systems, exhibiting complex interactions and behaviours. In fact, research on self-adaptive systems is attracting more and more attention among those interested in complex distributed systems [4].

Developers of autonomic ensembles have to understand and model not only the functional needs of their systems but also their adaptation needs. In particular, they have to check whether the provided models do offer the expected behaviour or attentively whether they are correct with respect to given specifications. At the same time, they have to identify the appropriate architectural schemes

* This work has been partially sponsored by the EU project ASCENS (257414).

for modelling individual components and the whole system as an ensemble of components. The goal of such choice being the guarantee that the adopted architectural scheme is instrumental for attesting that systems do self-adapt without severely undermining their intended functional behaviours.

Building on the large body of work in the area and on our own experience in the engineering of self-adaptive systems [5,6], we have previously identified and framed a few *adaptation patterns*, i.e. key architectural patterns that could be adopted to enforce self-adaptation at the level of individual components and ensembles. Software adaptation can indeed benefit from reuse in a similar way that designing software architectures has benefited from the reuse of software design patterns [7]. We identified context-aware and controllable *service components* (SCs) as the primitive entities to specify self-adaptive systems. In our view, a SC is a well-delimited piece of software (component) that provides a well-defined set of functionalities (services). This approach fits properly with all the software engineering features, namely modularity and reusability, other than simplicity.

Relying on this primitive entities, we have framed the many schemes by which *feedback loops* can be closed around individual SCs or ensembles of SCs, in order to achieve autonomic self-adaptive behaviours [8]. It is, indeed, widely recognized [9,10,11], especially in the MAPE-K architecture, that the capability of self-adaptation in a system necessarily requires the existence of feedback loops. This implies that, somehow, there exist means to inspect and analyse what is happening in the system (at the level of SCs, SC ensembles or the environment in which they are situated) and have components of the systems react accordingly. Therefore, looking at how these feedback loops appear implicitly or explicitly into SCs or into their ensembles, some categories of patterns can be identified.

Such analysis (extensively described in [12,13]) is still affected by two key limitations. Firstly, the patterns are modelled only in a semi-formal way, via UML diagrams and via a general description of the classes of self-adaptive goals that each pattern can satisfy (as from the SOTA goal-oriented requirements engineering approach [6]). It is then difficult to reason about the exact behaviour and properties of such patterns [14]. Secondly, the issue of rendering the presented patterns in some programming language is simply not considered at the moment.

In this paper, we address the above limitations by using SCEL [15], a formalism devised for modelling autonomic systems, to formalise both SCs of an autonomic ensemble and the adaptation patterns they use. By exploiting attributes associated to a component's interface, we can build patterns of communication that allow SCs to dynamically organise themselves into ensembles and implement specific adaptation patterns. Predicates over such attributes are used to specify the targets of communication actions, thus enabling a sort of *attribute-based* communication. In this way, an ensemble is not a rigid fixed network but rather a highly flexible structure where components linkages are dynamically established according to the chosen adaptation pattern.

Our aim is thus twofold. On the one hand, we show how SCs can enact adaptation by exploiting interfaces and attributes associated to them. On the other hand, we formalise the adaptation patterns via a language with an operational semantics

that paves the way to reasoning about them. Our ultimate goal is to provide a sound and uniform set of conceptual and practical guidelines and tools to drive developers of SC ensembles in the engineered exploitation of such mechanisms at the level of abstract system modelling, verification, and implementation.

Moreover, in this work we focus on system components' *linkage*. These connections can change at run-time, thus e.g. enabling the dynamic transition from one adaptation pattern to another, and we take advantage from the SCEL language for modelling these modifications (as shown in Section 6). The components' internal logic, comprehensive of their behaviour and feedback loops, is not specified in this work because it plays no role in the modelling of adaptation patterns.

The rest of the paper is organised as follows. In Section 2, we introduce some basic notions about service component interfaces and adaptation patterns, while in Section 3 we review the main ingredients of SCEL. In Section 4, we show how SCs and their environment are rendered in SCEL. These are then exploited in Section 5 to express in SCEL the patterns introduced in Section 2, and in Section 6 to model a robotics case study. Finally, in Section 7 we review some strictly related work and in Section 8 we hint at directions for future work.

2 Service Components and Adaptation Patterns

We base our categorization of adaptation patterns on a very general model for the interface of the primitive Service Component (SC). Therefore, we begin by introducing some basic notions about SC interfaces and adaptation patterns.

SC interfaces help to better understand how SCs interact and propagate adaptation. A generic SC interface has six ports:

- I - Input: for receiving service requests and responses;
- O - Output: for invoking services or replying to service requests;
- S - Sensor: for sensing the status of other components and of the environment;
- F - Effector: for adapting the behaviour of other components, thus acting as an Autonomic Manager (AM), or for propagating adaptation in the environment;
- E - Emitter: for issuing status information to an AM;
- C - Control: for receiving adaptation orders from an AM.

Notably, the same port may be connected to more than one SC and some ports of a given SC can be omitted whenever they do not play any role. In this way, we can characterise families of typical components as exemplified in Figure 1.

Depending on the SC ports that are enabled and how they are interconnected, different kinds of adaptation patterns can be obtained.

At the level of individual SCs, the categories of adaptation patterns are:

- *Reactive SC*: components able to react to environment's changes and not coupled with an explicit feedback loop; instead, such feedback loops exist only implicitly in the interactions of the components with the environment (as in reactive agent and component systems [16]).

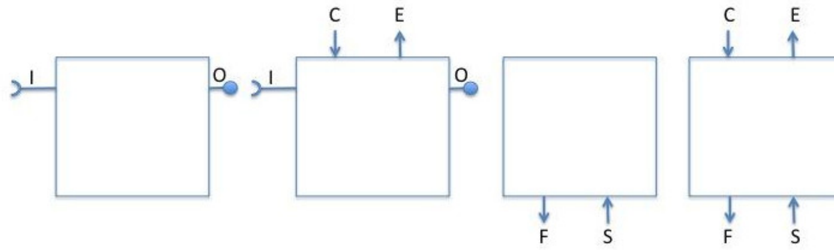

Fig. 1. Examples of SC interfaces of self-adaptive service components, adaptable service components, manager components, and adaptable manager components

- *Autonomic SC*: components explicitly coupled with an external feedback loop that monitors and directs their behaviour (as in most autonomic computing architectures [8,17]). This pattern is shown in Figure 2 (left), where the autonomic manager *AM* and the service component *SC* are such that:
  - *SC* has an interface appropriate for an adaptable service component;
  - *AM* has an interface appropriate for an adaptable manager component;
  - *AM* senses (along port S) whatever is emitted by *SC* (along port E);
  - *SC* obeys (along port C) *AM*'s control (along port F).
- *Proactive SC*: components that have an internal feedback loop to direct their goal/utility-oriented behaviour (as in intelligent and goal-oriented agents [16]). This pattern is shown in Figure 2 (right) and differs from the previous one for the following points:
  - the interfaces between *SC* and *AM* are encapsulated;
  - *AM* also monitors *SC*'s input (along port I).

Instead, at the level of SC ensembles, the categories of patterns are:

- *Centralised AM SCs Ensemble*: ensembles in which the overall adaptive behaviour is explicitly designed by means of specifically conceived interaction patterns between components (e.g., choreographies or negotiations [18,19]), and in which mutual interactions implies the existence of feedback loops. This pattern is shown in Figure 3;
- *P2P AMs SCs Ensemble*: ensembles in which there exists a set of components or "coded behaviours" that have the explicit goals of enforcing a global feedback loop over the ensembles, i.e., of controlling and directing their overall behavior (as in coordinated systems and electronic institutions [20]);
- *Reactive Stigmergy SCs Ensemble*: ensembles whose overall adaptive activities are not explicitly engineered by design, but for which adaptiveness (and feedback loops) emerges from the interaction of the components with a shared environment (as in pheromone-based [16,21] and field-based [22] approaches). This pattern is shown in Figure 4.

3 SCEL: Software Component Ensemble Language

SCEL (Software Component Ensemble Language) [15,23] is a language for programming service computing systems in terms of service components aggregated

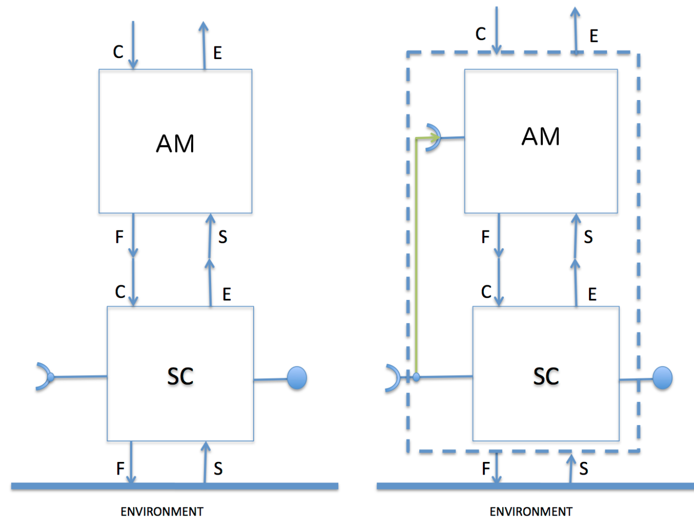

Fig. 2. Autonomic SC Pattern and Proactive SC Pattern

according to their knowledge and behavioural policies. The basic ingredient of SCEL is the notion of (*service*) *component* $\mathcal{I}[\mathcal{K}, \Pi, P]$ that consists of:

1. An *interface* $\mathcal{I}$ publishing and making available structural and behavioural information about the component itself in the form of *attributes*, i.e. names acting as references to information stored the component's repository. Among them, attribute *id* is mandatory and is bound to the name of the 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 information associated to its interface, which therefore can be dynamically manipulated by means of the operations provided by the knowledge repositories' handling mechanisms.
3. *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. Processes in P execute local computations, coordinate interaction with the knowledge repository or perform adaptation and reconfiguration.

The syntax of SCEL is presented in Table 1. 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 $(\nu n)_$.

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 *parametrized process invocation* ($A(\bar{p})$). The construct $P_1[\, P_2\,]$ abstracts the various forms of parallel composition commonly used in process calculi (see [15] for further details). Anyway, in this work, controlled composition will be interpreted as a standard interleaving, which means that in case of parallel processes only one process at a

Systems	$S ::= \mathcal{I}[\mathcal{K}, \Pi, P] \;\big|\; S_1 \parallel S_2 \;\big|\; (\nu n)S$
Processes	$P ::= \mathbf{nil} \;\big|\; a.P \;\big|\; P_1 + P_2 \;\big|\; P_1[P_2] \;\big|\; X \;\big|\; A(\bar{p})$
Actions	$a ::= \mathbf{get}(T)@c \;\big|\; \mathbf{qry}(T)@c \;\big|\; \mathbf{put}(t)@c \;\big|\; \mathbf{fresh}(n) \;\big|\; \mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$
Targets	$c ::= n \;\big|\; x \;\big|\; \mathsf{self} \;\big|\; \mathsf{P} \;\big|\; \mathsf{p}$

Table 1. SCEL syntax (Knowledge $\mathcal{K}$, Policies Π, Templates T, and Items t are parameters of the language)

time can perform an action (the others stay still). Process variables can support *higher-order* communication and enable a straightforward implementation of adaptive behaviors [24]. Indeed, they permit 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 let A to range over a set of parametrized *process identifiers* that are used in recursive process definitions. We assume that each process identifier A has a *single* definition of the form $A(\bar{f}) \triangleq P$, with $\bar{p}$ and $\bar{f}$ denoting 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 identified by c. These actions exploit templates T 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 $\mathcal{I}[\mathcal{K}, \Pi, P]$. Actions **get** and **qry** may cause the process executing them to wait for the wanted element if it is not (yet) available in the knowledge repository. The two actions differ for the fact that **get** removes the found item from the target repository while **qry** leaves the repository unchanged. Actions **put**, **fresh** and **new** are instead immediately executed.

Different entities may be used as the target c of an action. As a matter of notation, n ranges over component names, while x ranges over variables for names. 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* P or the name p, exposed as an attribute in the interface of the component, of a predicate that may dynamically change. A predicate is a standard boolean-valued expression obtained by applying to the results returned by the evaluation of relations between components' attributes and expressions. We adopt the following conventions about attribute names within predicates. If an attribute name occurs in a predicate without specifying (via prefix notation) the corresponding interface, it is assumed that this name refers to an attribute within the interface of the *object* component (i.e., a component that is a target of the communication action). Instead, if an attribute name occurring in a predicate is prefixed by the keyword this, then it is assumed that this name refers to an attribute within the interface of the *subject* component (i.e., the component hosting the process performing the

communication action). E.g., the predicate $\mathsf{this}.status = \text{“}sending\text{”} \wedge status = \text{“}receiving\text{”}$ is satisfied when the status of the subject component is *sending* and that of the object is *receiving*.

In actions using a predicate 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 P to be the target of the action. Thus, actions $\mathbf{put}(t)@n$ and $\mathbf{put}(t)@\mathsf{P}$ 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 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 $active = \text{“}yes\text{”} \wedge batteryLevel > low$.

4 Service Components and their Environment in SCEL

We now show how a generic SC is rendered in SCEL. Moreover, since most scenarios and patterns involve the *environment*, we also point out how it can be modelled in SCEL. Notably, the parallel composition between a generic SC and the environment gives rise to the *Reactive SC* pattern introduced in Section 2.

Service Components. A generic SC is rendered in SCEL as a component $\mathcal{I}_{SC}[\mathcal{K}_{SC}, \Pi_{SC}, SC]$ with

$$\begin{aligned}\mathcal{I}_{SC} \triangleq \{&(id, sc),\ (role, \text{“}component\text{”}/\text{“}manager\text{”}/\text{“}environment\text{”}),\\ &(controlFlag, \text{“}on\text{”}/\text{“}off\text{”}), (emitterFlag, \text{“}on\text{”}/\text{“}off\text{”}),\\ &(inputFlag, \text{“}on\text{”}/\text{“}off\text{”}), (outputFlag, \text{“}on\text{”}/\text{“}off\text{”}),\\ &(effectorFlag, \text{“}on\text{”}/\text{“}off\text{”}), (sensorFlag, \text{“}on\text{”}/\text{“}off\text{”}),\\ &(\mathsf{p}_{input}, \mathsf{P}_{input}),\ (\mathsf{p}_{output}, \mathsf{P}_{output}),\ (\mathsf{p}_{emitter}, \mathsf{P}_{emitter}),\\ &(\mathsf{p}_{effector}, \mathsf{P}_{effector}), \ldots\}\end{aligned}$$

$$SC \triangleq Control[Sensor[Input[Emitter[Effector[Output[InternalLogic]]]]]]$$

The component exposes in its interface at least twelve attributes. The attribute *id* indicates the name of the component, while *role* is used to define the role of the SC in a pattern (it can take one of the values *component*, *manager* or *environment*). Moreover, for each port, the interface contains a flag attribute used to enable (value *on*) or disable (value *off*) the port. Finally, four attributes, i.e. p_{input}, p_{output}, $\mathsf{p}_{emitter}$, and $\mathsf{p}_{effector}$, are used to refer the predicates P_{input}, P_{output}, $\mathsf{P}_{emitter}$ and $\mathsf{P}_{effector}$, respectively. These predicates can identify single components or ensembles. Specifically, P_{input} and $\mathsf{P}_{emitter}$ identify the component(s) managing the considered SC, P_{output} identifies the addressee(s) of the

output messages, and $\mathsf{P}_{\mathit{effector}}$ identifies the target of management actions (e.g., to enact adaptation), which can be either components or the environment.

The definition of action targets by means of attributes referring to predicates permits dynamically changing the predicates regulating the communication among SCEL components, which enables the dynamic transition from one adaptation pattern to another. We will come back to this point in Section 6.

Each port of the SC is then represented in SCEL as a process *PortName* that manages the data received or sent through the port and acts as a mediator between the external world and the knowledge repository of the component. These processes are executed in parallel with the process *InternalLogic* implementing the internal logic. This latter process, as well as the knowledge $\mathcal{K}_{SC}$ and the policy Π_{SC}, are left unspecified because they do not play any role in the modelling of adaptation patterns. The processes associated to the component's ports follow.

Input. The input data port can receive requests from other components. Its behaviour is expressed in SCEL as follows:

$$\begin{aligned} \mathit{Input} \triangleq\ & \mathbf{qry}(\mathit{inputFlag}, \text{``}\mathit{on}\text{''})@\mathsf{self}.\ \mathbf{get}(\text{``}\mathit{inputPort}\text{''}, ?\mathit{data}, ?\mathit{replyTo})@\mathsf{self}. \\ & \mathbf{put}(\text{``}\mathit{input}\text{''}, \mathit{data}, \mathit{replyTo})@\mathsf{self}. \\ & \mathbf{put}(\text{``}\mathit{inputPort}\text{''}, \mathit{data}, \mathit{replyTo})@\mathsf{p}_{\mathit{input}}.\ \mathit{Input} \end{aligned}$$

This process performs recursively the following behaviour. First, it checks the corresponding flag. If the port is enabled, it retrieves from the knowledge repository of the component an item (tagged with "*inputPort*") containing the input data and a predicate and sends one copy of such informations (tagged with "*input*") to the component's internal logic and one copy (tagged with "*inputPort*") to the input port of each component acting as a manager. Indeed, if the SC is self-adaptive (see Figure 2, right-hand side), its manager(s) must access the information received in input by the SC and, hence, the data received along the input port must be replicated to the manager(s) input port; otherwise, the forwarding of input messages is deactivated by simply setting the predicate referred by $\mathsf{p}_{\mathit{input}}$ to *false*. The provided predicate, bound to variable *replyTo*, will be used to respond to the requester(s).

Output. The output port is represented in SCEL as a process that fetches messages (e.g., responses to service requests) and a predicate (identifying, e.g., service requesters) generated by the internal logic, sets this predicate as $\mathsf{p}_{\mathit{output}}$ and sends the messages.

$$\begin{aligned} \mathit{Output} \triangleq\ & \mathbf{qry}(\mathit{outputFlag}, \text{``}\mathit{on}\text{''})@\mathsf{self}.\ \mathbf{get}(\text{``}\mathit{output}\text{''}, ?\mathit{data}, ?\mathit{recipients})@\mathsf{self}. \\ & \mathbf{get}(\mathsf{p}_{\mathit{output}}, ?\mathit{oldOut})@\mathsf{self}.\mathbf{put}(\mathsf{p}_{\mathit{output}}, \mathit{recipients})@\mathsf{self}. \\ & \mathbf{put}(\text{``}\mathit{inputPort}\text{''}, \mathit{data})@\mathsf{p}_{\mathit{output}}.\ \mathit{Output} \end{aligned}$$

To guarantee a correct identification of the addressee(s), *Output* processes an outgoing response message at a time and we assume that the predicate referred by $\mathsf{p}_{\mathit{output}}$ can be modified only by this process. Notably, such assumption only involves processes of the components' internal logic, because the processes

associated to the other ports do not modify the predicate, and no adaptation pattern prescribes a specific configuration for it. It is also worth noticing that, in case the same requester sends more than one request simultaneously to the component, the requester has to specify in the request data a correlation identifier that will be then inserted into the response data in order to allow the requester to properly correlate each response to the corresponding request.

Emitter. The emitter port is used to send awareness data to manager(s). The corresponding process is similar to the previous one, except for the item tags and the **put**'s predicate.

$$\begin{aligned} \mathit{Emitter} \triangleq\ & \mathbf{qry}(emitterFlag, \text{“}\mathit{on}\text{”})@\mathsf{self}.\ \mathbf{get}(\text{“}\mathit{emitter}\text{”}, ?data)@\mathsf{self}. \\ & \mathbf{put}(\text{“}\mathit{sensorPort}\text{”}, data)@\mathsf{p}_{emitter}.\ \mathit{Emitter} \end{aligned}$$

Effector. The effector port is used to enact adaptation on the managed element or to interact with the environment. The corresponding process is similar to the emitter one, except for the item tags and the **put**'s predicate.

$$\begin{aligned} \mathit{Effector} \triangleq\ & \mathbf{qry}(\mathit{effectorFlag}, \text{“}\mathit{on}\text{”})@\mathsf{self}.\ \mathbf{get}(\text{“}\mathit{effector}\text{”}, ?data)@\mathsf{self}. \\ & \mathbf{put}(\text{“}\mathit{controlPort}\text{”}, data)@\mathsf{p}_{\mathit{effector}}.\ \mathit{Effector} \end{aligned}$$

Sensor. The sensor port is used to sense the status of the component(s) managed by the considered SC or to retrieve information from the environment. The corresponding process gets the data coming from the sensor port and sends it to the component's internal logic:

$$\begin{aligned} \mathit{Sensor} \triangleq\ & \mathbf{qry}(sensorFlag, \text{“}\mathit{on}\text{”})@\mathsf{self}.\ \mathbf{get}(\text{“}\mathit{sensorPort}\text{”}, ?data)@\mathsf{self}. \\ & \mathbf{put}(\text{“}\mathit{sensor}\text{”}, data)@\mathsf{self}.\ \mathit{Sensor} \end{aligned}$$

Control. The control port is used to receive adaptation orders from manager(s). The corresponding process is similar to the sensor one, except for the item tags.

$$\begin{aligned} \mathit{Control} \triangleq\ & \mathbf{qry}(\mathit{controlFlag}, \text{“}\mathit{on}\text{”})@\mathsf{self}.\ \mathbf{get}(\text{“}\mathit{controlPort}\text{”}, ?data)@\mathsf{self}. \\ & \mathbf{put}(\text{“}\mathit{control}\text{”}, data)@\mathsf{self}.\ \mathit{Control} \end{aligned}$$

Environment. Since many adaptation patterns involve the environment where SCs are deployed, the environment must be modelled as well in SCEL in order to get a complete specification of patterns. It can be rendered as one or more components, whose precise definition may vary from one scenario to another. A generic environment could be expressed, e.g., as a component $\mathcal{I}_{Env}[\mathcal{K}_{Env}, \Pi_{Env}, Env]$ where its interface is defined as

$$\mathcal{I}_{Env} \triangleq \{(id, env), (role, \text{“}\mathit{environment}\text{”}), \ldots\}$$

and a possible sketch of the hosted process is

$$\begin{aligned} Env \triangleq \ldots\ & \mathbf{get}(\text{“}\mathit{controlPort}\text{”}, ?data)@\mathsf{self}\ \ldots \\ \ldots\ & \mathbf{put}(\text{“}\mathit{sensorPort}\text{”}, newData)@\mathsf{p}_{emitter}\ \ldots \end{aligned}$$

The environment component receives awareness data from components of the system. Such components should be connected to the environment via their effector and sensor ports, and have to use a predicate definition in their interface such as $(\mathsf{p}_{\mathit{effector}}, id = env)$. Moreover, the environment process provides data to components through their sensor ports by means of the predicate referred by $\mathsf{p}_{emitter}$, which dynamically selects the partner(s) of the communication. E.g., if the environment has to communicate with only one component sc, the predicate could be defined as $(\mathsf{p}_{emitter}, id = sc)$. Instead, if the environment needs to communicate data to all components of the considered system, it could be used the predicate $(\mathsf{p}_{emitter}, role = \text{``}component\text{''})$. As another example, if the environment must interact with a subset of the available components (e.g., those that are currently active), the predicate becomes:

$$(\mathsf{p}_{emitter}, role = \text{``}component\text{''} \;\wedge\; status = \text{``}active\text{''})$$

Finally, the environment could comprise multiple SCEL components, such as a room containing various devices (wifi access points, temperature sensors, motion sensors, etc.). In this scenario, each device is an environment component, thus an SC interacting with this 'smart ambient' accesses the environment components appropriate for each specific interaction. For example, the effector predicate is

$$(\mathsf{p}_{\mathit{effector}}, role = \text{``}environment\text{''} \;\wedge\; distance(\mathsf{this}.x, \mathsf{this}.y, x, y) <= range)$$

where $(\mathsf{this}.x, \mathsf{this}.y)$ identifies the coordinates of the emitting SC, while (x, y) identifies the coordinates of each environment component within a given range.

5 Adaptation Patterns in SCEL

We show now how the previous concepts can be used to express in SCEL some of the patterns introduced in Section 2, that will be exploited in the case study of Section 6. We refer to [25] for the SCEL models of the remaining patterns.

In SCEL every pattern results from the composition of the SCEL components corresponding to the involved SCs, AMs and environment[5], and by appropriately tuning the predicate definitions and the interface's attributes. We leave the predicate referred by p_{output} unspecified because it is context-dependent. Notably, for any pattern, processes SC and AM running in the SCEL components corresponding to SCs and AMs, respectively, have always the following form:

$$Control[Sensor[Input[Emitter[\mathit{Effector}[Output[InternalLogic]]]]]]$$

Centralized AM SCs Ensemble.

Intent. Any SC needs an external feedback loop to adapt. All SCs need to share knowledge and adaptation logic, so they are managed by the same AM.

Context. This pattern can be adopted when:

[5] For the sake of presentation, here we model the environment as a single SCEL component $\mathcal{I}_{Env}[\mathcal{K}_{Env}, \Pi_{Env}, Env]$ (see Section 4).

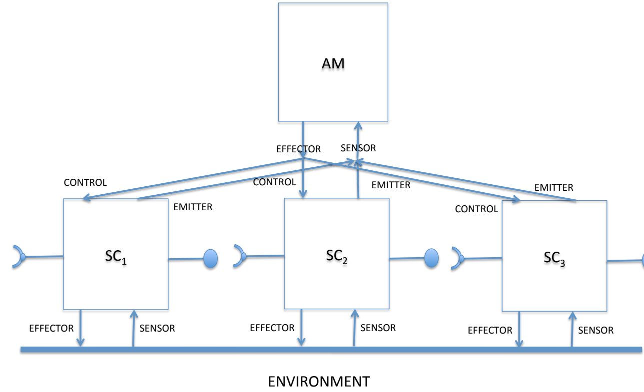

Fig. 3. Centralized AM SCs Ensemble

- an AM is necessary to manage adaptation;
- direct communication between SCs is allowed;
- a centralised feedback loop is more suitable because a single AM has a global vision on the system;
- the ensemble only includes a few, simple components.

Behaviour. This pattern, shown in Figure 3, is designed around one feedback loop. All components are managed by a single AM that "controls" their behaviour and, by sharing knowledge about them, is able to propagate adaptation.

Consequences. To manage adaptation over the entire system, a single AM is more efficient than multiple ones because it has a global view and knowledge of the system, but it can become a single point of failure.

SCEL description. The pattern is rendered in SCEL as the parallel composition of the components representing the centralized AM, the environment and the SCs:

$$\mathcal{I}_{AM}[\mathcal{K}_{AM}, \Pi_{AM}, AM] \parallel \mathcal{I}_{Env}[\mathcal{K}_{Env}, \Pi_{Env}, Env] \\ \parallel \mathcal{I}_{SC_1}[\mathcal{K}_{SC_1}, \Pi_{SC_1}, SC_1] \parallel \mathcal{I}_{SC_2}[\mathcal{K}_{SC_2}, \Pi_{SC_2}, SC_2] \parallel \mathcal{I}_{SC_3}[\mathcal{K}_{SC_3}, \Pi_{SC_3}, SC_3]$$

where the interfaces of manger and SCs (with $i \in \{1, 2, 3\}$) are as follows:

- $\mathcal{I}_{AM} \triangleq \{(id, am),\ (role, \text{``}manager\text{''}),$
  $(controlFlag, \text{``}off\text{''}), (emitterFlag, \text{``}off\text{''}),$
  $(inputFlag, \text{```}on\text{''}), (outputFlag, \text{``}off\text{''}),$
  $(effectorFlag, \text{``}on\text{''}), (sensorFlag, \text{``}on\text{''}),$
  $(\mathsf{p}_{input}, false),\ (\mathsf{p}_{output}, \mathsf{P}_{output}),\ (\mathsf{p}_{emitter}, \mathsf{P}_{emitter}),$
  $(\mathsf{p}_{effector}, id \in \{sc_1,\ sc_2,\ sc_3\})), \ldots\}$

  The AM description, in order to work as desired, needs to:
  - activate only the sensor, input and effector ports;
  - deactivate the forwarding of input messages to other components, by setting the predicate referred by p_{input} to *false*;
  - configure the predicate referred by $\mathsf{p}_{effector}$ accordingly to communicate only with the three managed SCs (i.e., only by components whose identifier belongs to the set $\{sc_1, sc_2, sc_3\}$).

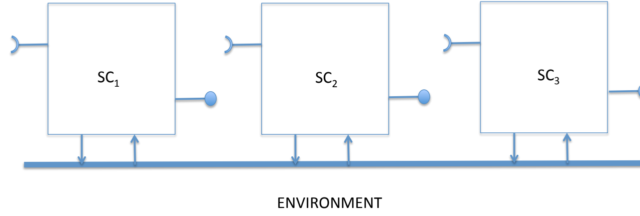

Fig. 4. Reactive Stigmergy SC

– $\mathcal{I}_{SC_i} \triangleq \{(id, sc_i),\ (role, \text{“}component\text{”}),$
$(controlFlag, \text{“}on\text{”}), (emitterFlag, \text{“}on\text{”}),$
$(inputFlag, \text{“‘}on\text{”}), (outputFlag, \text{“}on\text{”}),$
$(effectorFlag, \text{“}on\text{”}), (sensorFlag, \text{“}on\text{”}),$
$(\mathsf{p}_{input}, id = am),\ (\mathsf{p}_{output}, \mathsf{P}_{output}),\ (\mathsf{p}_{emitter}, id = am),$
$(\mathsf{p}_{effector}, role = \text{“}environment\text{”}), \ldots\}$

The SC description, to be properly controlled by the AM, needs to:

- activate all communication ports;
- configure the predicate referred by p_{input} in order to properly react to the received service requests by forwarding them to the manager *am*;
- configure the predicate referred by $\mathsf{p}_{emitter}$ suitably to send the control data to the manager *am*;
- configure the predicate referred by $\mathsf{p}_{effector}$ so to enable the interaction with the environment (i.e., all components playing the *environment* role).

Reactive Stigmergy SCs Ensemble.

Intent. There are several SCs that cannot directly interact with each other. The SCs simply react to the environment and sense the environment changes.

Context. This pattern has to be adopted when:

– the ensemble includes several components;
– the components are very simple, without having a lot of knowledge;
– the environment is frequently changing;
– direct communication between components is disallowed.

Behaviour. This pattern, shown in Figure 4, has not a direct feedback loop. Each single component acts like a bioinspired component. To satisfy its goal, the SC acts in the environment that senses with its “sensors” and reacts to the changes in it with its “effectors”. The different components are not able to communicate one with another, but are able to propagate information (and their actions) in the environment. Hence, they are able to sense the environment changes (e.g., other components reactions) and adapt their behaviour due to these changes.

Consequences. If the component is a proactive one, its behaviour is defined inside it with its internal goal. The behaviour of the whole system cannot be a priori defined. It emerges from the collective behaviour of the ensemble. The

components do not require a large amount of knowledge. The reaction of each component is quick and does not need managers since adaptation is propagated via the environment. The interaction model is an entirely indirect one.

SCEL description. The pattern is rendered in SCEL as the parallel composition of the components representing the SCs and their environment:

$$\mathcal{I}_{SC_1}[\mathcal{K}_{SC_1}, \Pi_{SC_1}, SC_1] \parallel \mathcal{I}_{SC_2}[\mathcal{K}_{SC_2}, \Pi_{SC_2}, SC_2] \parallel \mathcal{I}_{SC_3}[\mathcal{K}_{SC_3}, \Pi_{SC_3}, SC_3] \parallel \mathcal{I}_{Env}[\mathcal{K}_{Env}, \Pi_{Env}, Env]$$

where the SCs' interfaces, with $i \in \{1, 2, 3\}$, are as follows:

$$\begin{aligned}\mathcal{I}_{SC_i} \triangleq \{&(id, sc_i),\ (role, \text{“}component\text{”}),\\ &(controlFlag, \text{“}off\text{”}), (effectorFlag, \text{“}on\text{”}),\\ &(inputFlag, \text{“‘}on\text{”}), (outputFlag, \text{“}on\text{”}),\\ &(emitterFlag, \text{“}off\text{”}), (sensorFlag, \text{“}on\text{”}),\\ &(\mathsf{p}_{input}, false),\ (\mathsf{p}_{output}, \mathsf{P}_{output}),\ (\mathsf{p}_{emitter}, \mathsf{P}_{emitter}),\\ &(\mathsf{p}_{effector}, role = \text{“}environment\text{”}), \ldots\}\end{aligned}$$

The differences w.r.t. the SC description of the previous pattern are as follows:

- control and emitter ports are deactivated (hence, the predicate referred by $\mathsf{p}_{emitter}$ is left unspecified because it does not play any role);
- the forwarding of input messages is deactivated by setting the predicate referred by p_{input} to *false*.

6 Adaptation Patterns at Work

A key point about self-adaptation and self-adaptive patterns is the ability of dynamically changing the adaptation pattern in use if some circumstances occur during system lifetime. We illustrate this feature by means of a robotic case study concerning *object transportation*. For this task, robots need to find out objects (e.g., people to assist and rescue in case of disaster) and carry them back to a specific place (e.g., the external of a blazing building). A large number of robots can be used in the unknown environment in order to rapidly satisfy the system's goal. Thus, the most appropriate pattern to be used is the *Reactive Stigmergy SCs Ensemble* one red: the number of robots is large with respect to the size of the area to be explored; danger makes real the necessity to have simple and not too expensive components; the environment is unknown and frequently changing due to the disaster. The suitability of this pattern with respect to other ones, while considering different environment configurations, has been validated in [26] through some simulations carried out using a multi-robot simulator. It has been shown that a fully centralised approach (using the *Centralized AM SCs Ensemble* pattern) is not effective unless the position of all objects is known in advance.

Anyway, in realistic situations a single robot could not be able to carry a victim alone. So, since no pattern can be conveniently adopted for the whole lifetime of the system, in cases a robot collaboration is needed to manage a

specific task, a new pattern can be temporary applied for the necessary time. When the satisfaction of the object transportation task must be very short (e.g., in case of victims), the *Centralized AM SCs Ensemble* pattern is the best one. This is because the time for coordinating a single AM and all the other robots is shorter than the time for the coordination and negotiation among all robots (as, e.g., in the *P2P AMs SCs Ensemble* pattern). Thus, when a robot reaches an object that is too heavy, it changes its adaptation pattern becoming an AM. It also contacts other robots (the number that is needed to carry the object) that will change their pattern in order to behave as managed components. The AM then shares information about where the object is and how to carry it to the safe area. Finally, when the task is satisfied, all involved robots change again their pattern for coming back to the *Reactive Stigmergy SCs Ensemble* pattern.

This case study can be modelled in SCEL as follows. During the exploring phase all robots follow the *Reactive Stigmergy SCs Ensemble* pattern, thus each of them is rendered as a SCEL component with the following (excerpt of) interface:

$$\{(id, sc_i), (role, \text{“}component\text{”}), (\mathsf{p}_{input}, false), \\ (\mathsf{p}_{effector}, role = \text{“}environment\text{”}), \ldots\}$$

According to the *separation of concerns* design principle, the internal logic of components here is structured as follows:

$$InternalLogic \triangleq PatternHandler[ApplicationLogic]$$

where *PatternHandler*, that is in charge of changing the pattern when an object is found, is

$$\begin{aligned} &PatternHandler \triangleq \\ &\mathbf{get}(\text{“}sensor\text{”}, \text{“}objectFound\text{”}, ?objectData)@\mathsf{self}.BecomeManager(objectData) \\ &+ \mathbf{get}(\text{“}input\text{”}, \text{“}changePattern\text{”}, ?manager)@\mathsf{self}.BecomeManaged(manager) \end{aligned}$$

while *ApplicationLogic*, that implements the logic for the progress of the computation, is left unspecified as here we are not interested in this part of the internal behaviour of components. Intuitively, if the robot's sensors detect an object in the environment, the event is registered in the component's knowledge and, when the process above consumes it by means of the first **get** action, the execution of process *BecomeManager* is triggered. Similarly, the second **get** action is used to trigger the process *BecomeManaged* to react to a 'change pattern' request coming from another robot that has found an object.

The process $BecomeManager(data)$ is defined as follows:

$$\begin{aligned} &\mathbf{get}(role, \text{“}component\text{”})@\mathsf{self}.\ \mathbf{put}(role, \text{“}manager\text{”})@\mathsf{self}. \\ &\mathbf{get}(outputFlag, ?f)@\mathsf{self}.\ \mathbf{put}(outputFlag, \text{“}off\text{”})@\mathsf{self}. \\ &\mathbf{get}(\mathsf{p}_{effector}, ?oldEff)@\mathsf{self}.\ \mathbf{put}(\mathsf{p}_{effector}, id \in S_{data})@\mathsf{self}. \\ &\mathbf{put}(\text{“}inputPort\text{”}, \text{“}changePattern\text{”}, \mathsf{self})@\mathsf{p}_{effector}. \\ &\mathbf{get}(\text{“}sensor\text{”}, \text{“}taskCompleted\text{”})@\mathsf{self}.RestoreReactiveStigmergy \end{aligned}$$

where the set S_{data} of managed components, which are identified by $\mathsf{p}_{effector}$, depends on some elaborations on the object *data*. Thus, to become a manager,

first the component changes its role, the output port flag[6] and the effector predicate (as defined in Section 5). Then, it uses the new definition of this predicate to contact (via a **put** action) the appropriate number of robots that will be managed by it. When the object transportation task is completed, the process *RestoreReactiveStigmergy* is executed to reset the initial pattern.

The process $BecomeManaged(am)$, instead, is defined as follows:

$$
\begin{array}{l}
\mathbf{get}(controlFlag, ?cf)@\mathsf{self}.\ \mathbf{put}(controlFlag, \text{``}on\text{''})@\mathsf{self}.\\
\mathbf{get}(emitterFlag, ?ef)@\mathsf{self}.\ \mathbf{put}(emitterFlag, \text{``}on\text{''})@\mathsf{self}.\\
\mathbf{get}(\mathsf{p}_{input}, ?oldInp)@\mathsf{self}.\ \mathbf{put}(\mathsf{p}_{input}, id = am)@\mathsf{self}.\\
\mathbf{get}(\mathsf{p}_{emitter}, ?oldEmit)@\mathsf{self}.\ \mathbf{put}(\mathsf{p}_{emitter}, id = am)@\mathsf{self}.\\
\mathbf{get}(\text{``}sensor\text{''}, \text{``}taskCompleted\text{''})@\mathsf{self}.RestoreReactiveStigmergy
\end{array}
$$

This process enables the control and emitter ports, and modifies the predicates associated to the input and emitter ports as required by the *Centralized AM SCs Ensemble* pattern, by using the manger's identifier specified in the 'change pattern' request. Then, when the task is completed, it resets the initial pattern.

Finally, the process *RestoreReactiveStigmergy*, that restores the setting of the initial pattern and reinstalls the pattern handler process, is as follows:

$$
\begin{array}{l}
\mathbf{get}(role, ?oldRole)@\mathsf{self}.\ \mathbf{put}(role, \text{``}component\text{''})@\mathsf{self}.\\
\mathbf{get}(outputFlag, ?of)@\mathsf{self}.\ \mathbf{put}(outputFlag, \text{``}on\text{''})@\mathsf{self}.\\
\mathbf{get}(controlFlag, ?cf)@\mathsf{self}.\ \mathbf{put}(controlFlag, \text{``}off\text{''})@\mathsf{self}.\\
\mathbf{get}(emitterFlag, ?ef)@\mathsf{self}.\ \mathbf{put}(emitterFlag, \text{``}off\text{''})@\mathsf{self}.\\
\mathbf{get}(\mathsf{p}_{input}, ?oldInp)@\mathsf{self}.\ \mathbf{put}(\mathsf{p}_{input}, false)@\mathsf{self}.\\
\mathbf{get}(\mathsf{p}_{emitter}, ?oldEmit)@\mathsf{self}.\ \mathbf{put}(\mathsf{p}_{emitter}, false)@\mathsf{self}.\\
\mathbf{get}(\mathsf{p}_{effector}, ?oldEff)@\mathsf{self}.\ \mathbf{put}(\mathsf{p}_{effector}, role = \text{``}environment\text{''})@\mathsf{self}.\\
PatternHandler
\end{array}
$$

7 Related Works

The interest in engineering self-adaptive systems is growing, as shown by the number of recent surveys and overviews on the topic [10,4,27]. However, a comprehensive and rationally-organized analysis of architectural patterns for self-adaptation is still missing, despite the potential advantages of their use. For example, [28] proposes a classification of modelling dimensions for self-adaptive systems to provide the engineers with a common vocabulary for specifying the self-adaptation properties under consideration and select suitable solutions. However, although this work emphasizes the importance of feedback loops, it does not consider the patterns by which such feedback loops can be organized to promote self-adaptation. [29,7] focus on the mechanisms to perform adaptation actions, and on the various schemes that should be adopted to perform such adaptation actions at run-time and in a safe way. However they overlook the architectural patterns for the feedback loops that can identify and enact adaptation actions.

[6] Indeed, the only difference about ports in the two patterns concerns the output one.

Also [30] introduces the concept of patterns for self-adaptive systems based on control loops. It however focuses on how control loops can enforce adaptivity in a system and does not present a complete set of patterns.

Taking inspiration from control engineering, natural systems and software engineering, [9] presents some self-adaptive architectures that exhibit feedback loops. It also identifies the critical challenges that must be addressed to enable systematic and well-organized engineering of self-adaptive and self-managing software systems. In our work we aim at going further on and describing our patterns using a formalism, namely SCEL. Grounded on earlier works on architectural self-adaptation approaches [2], the FORMS model [31] enables engineers to describe, study and evaluate alternative design choices for self-adaptive systems. FORMS defines a shared vocabulary of adaptive primitives that – while simple and concise – can be used to precisely define arbitrary complex self-adaptive systems and can support engineers in expressing their design choices. This vocabulary is close to our choice of using SCEL to describe patterns, but it is not a formalism and rather has to be considered as a potentially useful complement to our work.

To the best of our knowledge, ours is the first work that addresses the formalisation of adaptation patterns. Rather, a bunch of works in the literature proposes formalisations of *design* patterns that, more in general, are devised to support component-based or object-oriented programming and are not specific for autonomic computing. We took inspiration from [32] and [33] to describe the patterns' template. Two main approaches have been considered: a group of works uses logics as target formalism (e.g., [34] relies on a temporal logic, while [35] on a predicate logic), whereas another group relies on new formalisms specifically devised for modelling design patterns (e.g., [36] uses the design model Abstract Data Views, while [37] proposes the use of Balanced Pattern Specification Language that, anyway, is still based on logics). Other works, as e.g. [38], formalise patterns in terms of graphs. Besides the fact that the above works do not deal with adaptation patterns, they differ from our work also because none of them uses a formalism based on process calculi, like SCEL. An approach using process calculi-like languages, namely CASPIS and COWS, is presented in [39], but it considers *methodological*, rather than *architectural* patterns.

8 Concluding Remarks

This paper reports on the way adaptation patterns for designing autonomic ensembles of SCs can be formalised by using the SCEL language. An application to a robotic case study is also presented, with the twofold aim of demonstrating the practical usage of the formalised adaptation patterns and of showing how dynamic change of adaptation patterns takes place.

From a technical point of view, the main challenge is in providing a compositional formalisation, where each pattern is rendered as the (parallel) composition of the models of the involved primitive components and where the dynamic change of pattern is still dealt with in a compositional way. Compositionality is also the key for allowing heterogeneous patterns to integrate well within the same

system. This motivates our choice of using SCEL for defining such formalisation, as it features a form of attribute-based communication that easily permits to express component linkages according to the chosen adaptation pattern and to dynamically adapt them according to a given pattern change.

The objective of the proposed formalisation is to provide an operational semantics for adaptation patterns that paves the way to reasoning about them. This can lead to verifiable development of autonomic SC ensembles from abstract architectural patterns.

In the near future, in order to provide a more concrete evidence of the benefits brought by the proposed formalisation, we plan to implement the formalised adaptation patterns considered in this work in jRESP [15], a Java runtime environment for developing autonomic and adaptive systems according to the SCEL paradigm. A long-term goal, instead, is to integrate in this pattern-based development approach the formal reasoning tools for SCEL programs that are currently under construction. Once also the internal logic of the components (e.g., behaviour, feedback loops) is modelled, this integration will permit to establish qualitative and quantitative properties of individual SCs and their ensembles.

References

1. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer **36** (2003) 41–50
2. Weyns, D., Holvoet, T.: An Architectural Strategy for Self-Adapting Systems. In: SEAMS, IEEE (2007) 3
3. Project InterLink: `http://interlink.ics.forth.gr` (2007)
4. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. on Autonomous and Adaptive Systems **4**(2) (2009) 14
5. Zambonelli, F., Viroli, M.: A survey on nature-inspired metaphors for pervasive service ecosystems. J. of Pervasive Comp. and Comm. **7** (2011) 186–204
6. Abeywickrama, D.B., Bicocchi, N., Zambonelli, F.: SOTA: Towards a General Model for Self-Adaptive Systems. In: WETICE, IEEE (2012) 48–53
7. Gomaa, H., Hashimoto, K.: Dynamic Self-Adaptation for Distributed Service-Oriented Transactions. In: SEAMS, IEEE (2012) 11–20
8. Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1) (2003) 41–50
9. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. In: Softw. Eng. for Self-Adaptive Systems. LNCS 5525. Springer (2009) 48–70
10. Cheng, B., et al.: Software engineering for self-adaptive systems: A research roadmap. In: Softw. Eng. for Self-Adaptive Systems. LNCS 5525. Springer (2009) 1–26
11. Pieter, V., et al.: On interacting control loops in self-adaptive systems. In: SEAMS, ACM (2011)
12. Cabri, G., Puviani, M., Zambonelli, F.: Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In: CTS, IEEE (2011) 508–515
13. Puviani, M., Cabri, G., Zambonelli, F.: A Taxonomy of Architectural Patterns for Self-Adaptive Systems. In: C3S2E, ACM (2013)
14. Clarke, E.M., Wing, J.M.: Formal Methods: State of the Art and Future Directions. ACM Comput. Surv. **28**(4) (1996) 626–643

15. De Nicola, R., et al.: SCEL: a language for autonomic computing. Technical Report (January 2013) `http://rap.dsi.unifi.it/scel/pdf/SCEL-TR.pdf`.
16. Bordini, R.H., et al.: A Survey of Programming Languages and Platforms for Multi-Agent Systems. Informatica (Slovenia) **30**(1) (2006) 33–44
17. Hariri, S., et al.: The autonomic computing paradigm. Cluster Computing **9**(1) (2006) 5–17
18. Beam, C., Segev, A.: Automated negotiations: A survey of the state of the art. Wirtschaftsinformatik **39**(3) (1997) 263–268
19. Jennings, N., et al.: Automated negotiation: prospects, methods and challenges. Group Decision and Negotiation **10**(2) (2001) 199–215
20. Esteva, M., et al.: On the Formal Specifications of Electronic Institutions. In: AgentLink. LNCS 1991, Springer (2001) 126–147
21. Kesäniemi, J., Terziyan, V.: Agent-environment interaction in mas-introduction and survey. In: Multi-Agent Systems: Modeling, Interactions, Simulations and Case Studies. InTech (2011) 203–226
22. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The TOTA approach. ACM Trans. Sw. Eng. and Meth. **18**(4) (2009)
23. De Nicola, R., et al.: A Language-based Approach to Autonomic Computing. In: FMCO 2011. LNCS 7542, Springer (2012) 25–48 `http://rap.dsi.unifi.it/scel/`.
24. Gjondrekaj, E., Loreti, M., Pugliese, R., Tiezzi, F.: Modeling adaptation with a tuple-based coordination language. In: SAC, ACM (2012) 1522–1527
25. Cesari, L., et al.: Formalising adaptation patterns for autonomic ensembles. Technical Report (2013) `http://rap.dsi.unifi.it/scel/pdf/patternsInSCEL-TR.pdf`.
26. Puviani, M., et al.: Is self-expression useful? evaluation by a case study. In: WETICE. (2013)
27. Weyns, D., et al.: Claims and supporting evidence for self-adaptive systems: A literature study. In: SEAMS, IEEE (2012) 89 –98
28. Andersson, J., et al.: Modeling dimensions of self-adaptive software systems. In: Softw. Eng. for Self-Adaptive Systems. LNCS 5525. Springer (2009) 27–47
29. Gomaa, H., et al.: Software adaptation patterns for service-oriented architectures. In: SAC, ACM (2010) 462–469
30. Weyns, D., et al.: On Patterns for Decentralized Control in Self-Adaptive Systems. In: Softw. Eng. for Self-Adaptive Systems. LNCS 7475. Springer (2013) 76–107
31. Weyns, D., Malek, S., Andersson, J.: Forms: Unifying reference model for formal specification of distributed self-adaptive systems. ACM TAAS **7**(1) (2012) 8
32. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley, Reading (MA) (1995)
33. Ramirez, A., Cheng, B.: Design patterns for developing dynamically adaptive systems. In: SEAMS, ACM (2010) 49–58
34. Mikkonen, T.: Formalizing design patterns. In: ICSE, IEEE (1998) 115–124
35. Bayley, I.: Formalising design patterns in predicate logic. In: SEFM, IEEE (2007) 25–36
36. Alencar, P.S.C., Cowan, D.D., de Lucena, C.J.P.: A Formal Approach to Architectural Design Patterns. In: FME. LNCS 1051, Springer (1996) 576–594
37. Taibi, T., Ling, D.N.C.: Formal Specification of Design Patterns - A Balanced Approach. Journal of Object Technology **2**(4) (2003) 127–140
38. Bottoni, P., Guerra, E., de Lara, J.: Formal foundation for pattern-based modelling. In: FASE. LNCS 5503, Springer (2009) 278–293
39. Wirsing et al., M.: SensoriaPatterns: Augmenting Service Engineering with Formal Analysis, Transformation and Dynamicity. In: ISoLA. CCIS 17, Springer (2008) 170–190