RESP:A runtime environment for SCEL programs

Michele Loreti

September 14, 2012

Abstract

SCEL is a new language specifically designed to model autonomic components and their interaction. It brings together various programming abstractions that permit to directly represent knowledge, behaviors and aggregations according to specific policies. It also supports naturally programming self-awareness, context-awareness, and adaptation. In this paper, we present RESP a runtime environment that permits executing SCEL programs.

Contents

1 Introduction

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

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

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

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

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

The challenge for language designers is to devise appropriate abstractions and linguistic primitives to deal with the large dimension of systems, the need to adapt to evolving requirements and to changes in the working

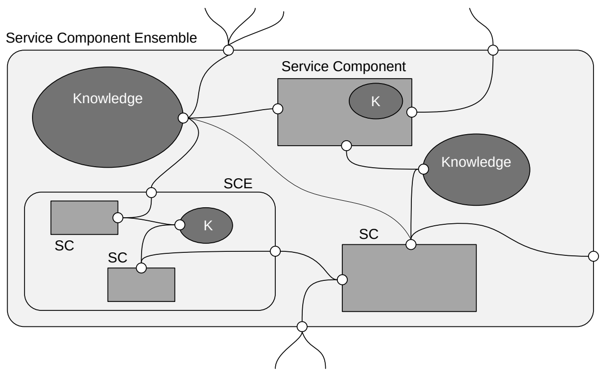


Figure 1: A Service Component Ensemble

environment, and the emergent behaviors resulting from complex interactions.

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

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

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

In [6] SCEL (Service Component Ensemble Language), a new language designed for autonomic computing has been presented. SCEL brings together different programming abstractions that permit to directly represent *knowledge*, *behaviors* and *aggregations* according to specific *policies*. It also supports naturally programming self-awareness, context-awareness and adaptation. SCEL's solid semantic grounds lay the basis for developing logics, tools and methodologies for formal reasoning about system behavior to establish qualitative and quantitative properties of both the individual components and the ensembles.

In this work we present RESP a Runtime Environment for SCEL Programs that permits executing SCEL programs. Our aim is to provide programmers with a framework that permits developing of autonomic and adaptive systems according to the SCEL paradigm.

2 SCEL: design principles

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

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

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

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

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

All these abstractions are aggregated by means of the notion of autonomic component. An *autonomic component* $\mathcal{I}[\mathcal{K}, \Pi, P]$ consists of:

1. an *interface* $\mathcal{I}$ publishing and making available structural and behavioral information about the component itself;

2. a *knowledge manager* $\mathcal{K}$, managing both application data and awareness data, together with the specific handling mechanism;

3. a set of *policies* Π regulating the interaction between the different internal parts of the component and the interaction of the component with the others;

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

A component's interface can be inquired to extract information about the component, its status or its execution environment, as well as the services offered by the component. In fact, the interface provides a set of *attributes* characterizing the component itself. Among these attributes, attribute *id* is mandatory and is bound to the name of the component. Additional attributes might, e.g., indicate the battery's charge level and the component's GPS position. Suitable attributes are also used to indicate the provided services and their signature. Notably, the whole information provided by the component interface is stored in the local knowledge of the component and therefore can be dynamically manipulated by means of the operations provided by the knowledge repositories' handling mechanisms.

3 A runtime environment for SCEL programs

In this section we present RESP. This is a runtime environment that permits executing SCEL programs. Our aim is to provide programmers with a framework that permits developing of autonomic and adaptive systems according to the SCEL paradigm.

Design principles SCEL aims at identifying linguistic constructs for uniformly modeling the control of computation, the interaction among possibly heterogeneous components, and the architecture of systems and ensembles. A SCEL *program* consists of a set of components, each of which is equipped with its own knowledge repository, that act and interact with each other. The number of components involved in the computation is not fixed but can change dynamically during the computation. SCEL *program* can be composed by a set of heterogeneous components that, relying on an highly dynamic communication infrastructure, concur and cooperate to achieve a set of *goals*. That being on, a first principle that has driven the design and implementation of RESP has been to avoid *centralised control*. In particular, components should be able to interact with each other by simply relying on the available communication media.

In the definition of SCEL, some categories, like *knowledge* and *policy*, are not fixed but these have to be fixed from time to time according to the specific application domain or to the taste of the language user. Other mechanisms, like for instance the underlying communication infrastructure, are not considered at all and *abstracted* in the operational semantics. For this reasons, all the framework is parametric with respect to specific implementations of above mentioned features. Design patterns have been largely used in RESP to simplify development of specific implementations of knowledge, policies and underlying communication infrastructure.

Finally, to simplify the integration with other tools/framework (like for instance Argos [?] and DEECo [?]), RESP relies on *open technologies* like, for instance, json [?]. Such tools, providing mechanisms for data-interchange format, simplify interactions between heterogeneous network components and provide the basis on which different runtime for SCEL programs can cooperate.

3.1 Components

The central element of RESP is the class Node. This class provides the implementation for a generic SCEL component[1]. The overall infrastructure of a generic node is reported in Figure 2.

We assume that each node is executed over a virtual machine or a physical device that provides the access to: input and output devices and network connections. Following the same structure considered in Section 2, each node contains:

- a *knowledge*, described in Section 3.2;
- a set of running processes/threads, described in Section 3.7;
- a *policy*, described in Section 3.8.

Like for a SCEL components, structural and behavioural information about a node can be collected into an *interface*. This is rendered in RESP via a set of *attribute collectors* (see Section 3.5) that, reading values from the knowledge, publish and and make available attribute values in the interface.

Nodes interact with each other via *ports* (Section 3.6). These provide mechanism for supporting both *one-to-one* and *gruop* communications.

3.2 Knowledge

The interface Knowledge[2], identifies a generic *knowledge repository* and provides the high level primitives to manage pieces of relevant information coming from different sources. This interface is contains the following methods:

- put(Tuple t): boolean

[1]From now on we will use *node* to refer to instances of class Node, while *component* will indicate a SCEL component.

[2]All the classes considered in this section are part of package org.cmg.ml.resp.knowledge.

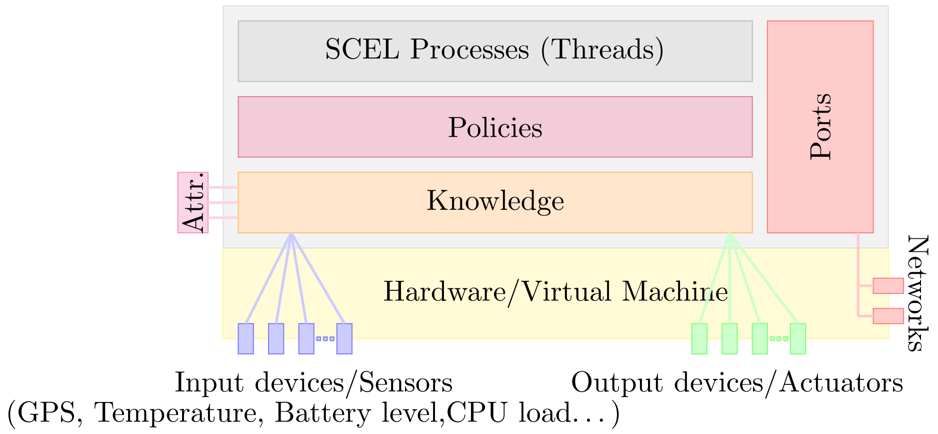


Figure 2: Node architecture

- get(Template t): Tuple
- getp(Template t): Tuple
- query(Template t): Tuple
- queryp(Template t): Tuple

Method put adds a tuple to the knowledge and returns a boolean value indicating if the operations has been successfully executed or not. Class Tuple, defined in the same package of Knowledge, identifies the basic information item. It consists of a sequence of values, i.e. Objects, that can be collected into a knowledge repository.

The other methods, that can be used to retrieve knowledge items from a knowledge repository, take as parameter an instance of Template. This is used to select elements in a knowledge repository and consists of is a sequence of TemplateField. A TemplateField is an interface containing the following single method:

- match(Object o): boolean

Class Template relies on this method to verify if a Tuple can be selected or not. Indeed, a tuple matches a template if both the object have the same length and the corresponding element matches.

Let k be a Knowledge. Method k.get(t) is used to remove a tuple matching template t from the knowledge k. This is a blocking operation.

Indeed, the thread invoking the method is blocked until a tuple matching t is not found. Method k.query(t) is similar. However, while with get the matching tuple is physically removed from the knowledge, with query this is only read. Moreover, implementation of interface Knowledge can provide mechanisms to *infer* tuples according to the actual value in knowledge.

Finally, methods getp and queryp are the predicative variants of get and query. Namely, getp and queryp are not blocking operations: if a k.getp(t) (respectively k.queryp(t)) is executed when a tuple matching t is not in the knowledge, value null is returned.

Currently, a single implementation of interface Knowledge is available in RESP. Class TupleSpace in package org.cmg.ml.resp.knowledge.ts provides an implementation for a Linda [7] tuple space.

3.3 Sensors

To retrieve data from external input devices, nodes can be equipped with *sensors*. These are instances of class Sensor that can be used to identify a generic source of information. Each sensor can be associated to a logical or physical device providing data that can be used by processes and that can be the subject of adaptation.

Each sensor exports data as a *tuple*. We will see below that the same actions used to retrieve values from knowledge, namely query, can be used to obtain data from a sensor. In fact, each node *hides* the real source of information to processes.

3.4 Actuators

Instances of class Actuator can be used to send data to external components or devices. Similarly to Sensor, an instance of class Actuator can be used to *control* an *external* component that identify a logical/physical actuator. Processes can pass values to actuators by relying on standard SCEL operations.

3.5 Attributes and Atribute collectors

An attribute is defined as a pair *(name,value)*. Attributes can be published in a node interface via *attribute collectors*. Attribute collectors can be implemented by extending abstract class AttributeCollector. This class has protected fields:

- name: String, containing the name associated to the published attribute;
- template: Template, identifying the template used to retrieve the knowledge element that will be used to compute actual attribute value.

When a node receives a request for an attribute *a*, a collector with the same name is selected. Hence, a tuple matching the template associated to the collector is retrieved from the knowledge. If an attribute with name *a* does not exists, or a tuple matching the collector template is not available in the knowledge, an attribute with value null is returned. Otherwise, the (abstract) method doEval is used to compute the actual attribute value (subclasses of AttributeCollector has to provide an implementation for this method).

3.6 Ports and network infrastructure

Each node is equipped with a set of *ports* that are able to handle both *point-to-point* interactions and *group* interactions (*ensemble* oriented). Indeed, a port provides a generic communication channel that follow a specific communication protocol.

Currently the following ports have been developed:

- InetPort, this kind of ports uses TCP to *point-to-point* interactions and UDP for the *group* ones;
- ServerPort, in this case a centralized server is used to collect and dispatch nodes' actions;
- VirtualPort, this is used to *simulate* nodes running on *virutal* devices.

To simplify interactions with other framework/tools, even developed in languages that are different from Java, messages are sent using a json format.

Each port is identified by a physical address. For instance, in the case of InetPort this is the *inet-address* associated to the socket where a thread is waiting for incoming connections.

3.7 Agents

Agents are the RESP active computational units and are threads used to program the behaviour of SCEL processes. Class Agent is an abstract class providing the following methods:

- abstract doRun(): void
- put(Tuple, Target): boolean
- get(Template, Target): Tuple
- getp(Template, Target): Tuple
- query(Template, Target): Tuple
- queryp(Template, Target): Tuple
- exec(Agent a): boolean

The first method is the one subclasses has to implement in order to define agent behaviour. This method is invoked when the agent starts is computation. Methods put, get, getp, query and query are used to manage shared knowledge repositories by withdrawing/retrieving/adding information items from/to a knowledge repository. These methods extend the one considered in Section 3.2 by with another parameter identifying the, possibly remote, node where the target knowledge repository is located.

A target can be either a *point-to-point* address, a *group* or *self*. A point-to-point address univocally identify the target of the considered action. A *group* identifies all the nodes that satisfy a given predicate on nodes attributes. Special target Self is used to refer to the node where an agent is running. The classes implementing the possible target, in conned in package org.cmg.ml.resp.topology, are PointToPoint, Group and Self, respectively.

3.8 Policies

Policies are attached to a node in order to control and adapt the performed actions to guarantee the achievement of specific goals or the satisfaction of specific properties. In particular, when an agent invokes a method, this is first delegated to the policy associated to the associated policy. This guarantee that SCEL action execution is appropriately regulated.

The interface Policy provides methods of the form:

- put(Agent a, Tuple t , Target l)
- acceptGet(Locality l , Template t)

The first one is used when agent a executes an action (in this case a put). While the second one is used to control the actions executed by remote components (in this case a get).

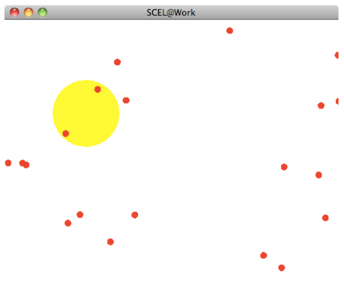
Policies are organized in a *stack*. The policy at one level relies on the one at the level below to actually execute SCEL actions. The policy at the lower level is the one that allows any operation.

4 RESP at work

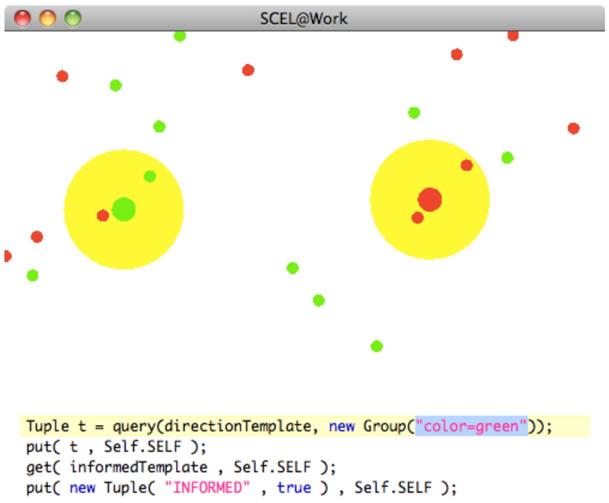
In this section we present a simple example that shows how RESP can be used to support development and execution of SCEL systems. We consider a set of robots have to reach a safe-area. To discover the location of safe-area, each robot follows a *random walk*. As soon as a robot reach the area, it publishes locally its location in the local knowledge repository. Thanks to By relying on *group* queries, other robots can get informed about the location of the safe-area and then can move directly to the target.

We assume that each robot is equipped with a GPS sensor, a sensor to verify if the target are has been reached or not. Moreover, to control movement, we assume that each robot is equipped with an actuator to change direction and with an actuator to stop movement.

To implement this scenario we let each robot be implemented as a SCEL component. To permits simulating the scenario we a virtual port to handle robot communication and an ad-hoc data structure to take care of robot position and movement.



```
while (!found&&!informed) {
    Tuple t = query( gpsTemplate , Self.SELF );
    x = t.getDoubleAt(1);
    y = t.getDoubleAt(2);
    if ((x==0)||(y==0)||(x==maxX)||(y==maxY)) {
    put( new Tuple( "DIR" , r.nextDouble()*2*Math.PI ) ,  Self.SELF );
    t = query( targetTemplate , Self.SELF );
    found =  t.getBooleanAt(1);
    t = query( informedTemplate , Self.SELF );
    informed = t.getBooleanAt(1);
}
```



```
Tuple t = query(directionTemplate, new Group("color=green"));
put( t , Self.SELF );
get( informedTemplate , Self.SELF );
put( new Tuple( "INFORMED" , true ) , Self.SELF );
```

5 Conclusions

We have presented (first version of) a framework that permits developing and executing SCEL oriented applications in Java.

Considered framework should be now *populated* with specific implementations for *policies*, *knowledge*.

Moreover, we plan to develop examples that, starting from Ascens case studies, can be used to *validate* the proposed solutions.

We are now working on a *top-level* programming language that, enriching SCEL with standard programming primitives, permits simplifying development of SCEL programs.

References

[1] Hölzl, M., Rauschmayer, A., Wirsing, M.: Software engineering for ensembles. In: Software-Intensive Systems and New Computing Paradigms, Springer (2008) 45–63

[2] IBM: An architectural blueprint for autonomic computing. Technical report (June 2005) Third edition.

[3] InterLink, P.: `http://interlink.ics.forth.gr/central.aspx` (2007) Last accessed: 2011-11-28.

[4] ASCENS, P.: `http://www.ascens-ist.eu/` (2010) Last accessed: 2011-11-28.

[5] Wirsing, M., Hölzl, M.M., Tribastone, M., Zambonelli, F.: ASCENS: Engineering autonomic service-component ensembles. In Beckert, B., Bonsangue, M., de Boer, F., Damiani, F., eds.: Proc. of the 10th Int. Symposium on Formal Methods for Components and Objects (FMCO11). LNCS, Springer (2012)

[6] De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: Languages primitives for coordination, resource negotiation, and task description. ASCENS Deliverable D1.1 (September 2011) `http://rap.dsi.unifi.it/scel/`.

[7] Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7** (1985) 80–112