www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D1.3: Third Report on WP1

Task Descriptions, Dynamic Reconfiguration, Context-Aware Resource Negotiation and Performance Metrics

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

Lead contractor for deliverable: **UDF**
Author(s): **Rocco De Nicola (IMT), Michele Loreti (UDF), Rosario Pugliese (UDF), Francesco Tiezzi (IMT), Andrea Vandin (IMT)**

Reporting Period: **3**
Period covered: **October 1, 2012 to September 30, 2013**
Submission date: **November 8, 2013**
Revision: **Final**
Classification: **PU**

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

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

Executive Summary

SCEL (Service Component Ensemble Language) is a new language specifically designed to program autonomic components and their interaction, while supporting formal reasoning on their behaviors. SCEL brings together various programming abstractions that allow one to directly represent behaviors, knowledge and aggregations according to specific policies. Given that the syntax of the behavioural language is by now stable, during the third year we have concentrated our efforts on other aspects. We have thus defined a language for specifying policies used to control actions and interactions. We have also developed an integration of the behavioural language with external reasoners that are used to support processes in taking their decisions to adapt to changing environments. In addition to this, we have developed MISSCEL, a prototype implementation of SCEL in MAUDE that permits to exploit the rich MAUDE framework to reason on SCEL specifications, and that can be easily integrated with reasoners written by using the same formalism. Moreover, we have continued with the implementation of the language by enriching jRESP, the Runtime Environment for SCEL programs, with new interaction patterns; and have defined new interfaces to permit the integration of jRESP with different reasoners. All activities of WP1 have used as testbeds many scenarios taken from the three case studies investigated in WP7 and in the other WPs.

Contents

1 Introduction

In this deliverable we present some linguistic supports for modeling and programming service components and their ensembles, their interactions, their sensitivity and adaptivity to the environment. More specifically, we introduce a dialect of SCEL [DFLP12, DLPT13], named SCEL$_{TS}$, defined by specifying knowledge repositories as multiple distributed tuple-spaces (à la KLAIM [DFP98]). We will illustrate the main features of SCEL$_{TS}$ in a step-by-step fashion using the running example from the automotive domain. We refer the interested reader to the technical report [DLPT13] for a full account of the language's semantics; this report is a working document that describes in full detail the updated version of the language we are considering.

Apart for providing the specific dialect of SCEL based on tuple spaces we also present SACPL (SCEL Access Control Policy Language), a simple, yet expressive, language for defining access control policies and access requests, and its integration with SCEL. SACPL is inspired by, but simpler and less expressive than, the OASIS standard for policy-based access control XACML [OAS12]. In this report we only sketch the main ingredients, and refer the interested reader to [DLPT13] for a full account of the language.

The solid semantics foundations of SCEL has laid the basis for formal reasoning. MISSCEL, a MAUDE Interpreter and Simulator for SCEL, is an executable operational semantics of SCEL that can be fed to the rich MAUDE toolset [CDE^{+}07] to analyze SCEL programs by performing: state-space generation, qualitative analysis via MAUDE's invariant and LTL model checkers, debugging via probabilistic simulations and animations generation, statistical quantitative analysis.

Together with the MAUDE interpreter for SCEL$_{TS}$, that would be useful for rapid prototyping, we have also continued working on the actual implementation of our language. It is based on jRESP, the Java runtime environment providing an API that permits using in Java programs the SCEL's linguistic constructs for controlling the computation and interaction of autonomic components, and for defining the architecture of systems and ensembles. To adapt to the alternative instantiations of key notions such as knowledge representation or policies, that SCEL permits, jRESP relies extensively on the use of design patterns.

SCEL is sufficiently powerful for dealing with coordination and interaction issues. However it does not provide explicit machineries for specifying components that take decisions about the action to perform basing on their context, or of a partial perception of it. Obviously, the language could be extended in order to encompass such possibilities, but we preferred to have separate reasoning components that SCEL programs can invoke when decisions have to be taken. We have recently started investigation in this direction by studying how MISSCEL could be integrated with reasoners specifically built for the systems under consideration. Moreover, we have developed a methodology that is not restricted to a particular reasoner but would allow us to take advantage of many reasoners at the same time, each performing particular reasoning tasks for which it is best suited.

In the next subsection we shall present the scenario that will be used to illustrate the main features of SCEL and of the policy language and to describe how the two implementations of SCEL$_{TS}$ based on MAUDE and Java can be exploited to execute program and to interact with external reasoners.

An Automotive Scenario

Throughout this deliverable we shall use a running example, drawn from the Automotive Case Study described in Deliverable D7.2, to illustrate the main achievements of WP1 during the third year.

We consider a scenario where e-vehicle users have to perform multiple activities throughout a day, which take place at different physical locations (called *Points Of Interest*, POIs). In particular, the scenario features a connected mobility environment of smart car parks (possibly equipped with charging stations) and intelligent e-vehicles, which can monitor their states, reason and adapt in order to reach their goals. By exploiting the connected mobility environment, planning battery recharging and locating available parking lots close to the POIs to visit is transparent to drivers that only take care of the sequences of activities they want to perform. It is assumed that POI lists are computed before starting the journey, by aiming at optimizing time and cost of the travel. Different policies can be applied by drivers during the journey for regulating parking lots selection. Similarly, car parks rely on specific policies for regulating cars admission.

Relations with other WPs

SCEL$_{TS}$ is the result of discussion and interaction carried out last year with many other researchers involved in the project. Several collaborations have started regarding issues considered in other work packages. They can be summarized as follows:

- One of the collaborations with WP2 is mainly focused on the investigation of adaptation mechanisms and their realization in SCEL. First, the *control-data* [BCG^{+}12a] approach is being developed in parallel with the development of SCEL. Both lines of research are enriching each other, in particular in what regards the identification of suitable adaptation mechanisms for ensembles (e.g. based on policies or reflection). Second, the proof-of-concept carried out in [BCG^{+}12b] where Maude was used to validate the suitability of certain architectural patterns and linguistic mechanisms for ensembles, has been fundamental in the development of MISSCEL [BDVW]. Another collaboration strand (see Deliverable D2.3) is focussed on the use of soft constraint paradigm as a specification technique that encompasses flexible knowledge representation and reasoning (declarative aspects) with knowledge manipulation primitive (procedural aspects). The integration of soft-constraint techniques in SCEL is still under development.

- The collaborations with WP3 has intensified with the aim of investigating the possibility of integrating `KnowLang` with SCEL. We have actually worked on the integration of simpler reasoners with SCEL$_{TS}$ [BDVW]. As soon as the prototype implementation of `KnowLang` is available we shall proceed with its integration that will have to rely on an intelligent cooperation between procedural and declarative aspects of system behavior. SCEL provides the procedural components and is instead parametric wrt the declarative ones; the integration will have to exploit the correspondence between the KnowLang operators ASK and TELL and the SCEL$_{TS}$ actions for retrieving information from shared knowledge repositories (**qry**) and for adding information to them (**put**).

- The cooperation with WP4 has continued and we have shown how some of the component- and ensemble-level adaptation patterns proposed in the literature and/or developed within WP4 can be rendered in SCEL (see Deliverable D4.3, Section 4, and [CNP^{+}13]). Specifically, we have defined a compositional approach: each adaptation pattern is rendered as the (parallel) composition of the SCEL terms corresponding to the involved primitive components (and, possibly, to their environment). 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.

- In the context of WP5, we moved our first steps in providing SCEL with formal reasoning techniques and tools by implementing MISSCEL. In fact, the implementation of a SCEL interpreter in MAUDE allowed us to exploit its reach toolset to perform qualitative and quantitative analysis of SCEL specifications. For example, in [BDVW] and in the Section 2 of Deliverable JD3.1 we discuss the statistical model checking of a SCEL specification regarding robots moving in an arena. Another line of research that we are pursuing in the context of WP5 consists in defining a translation from SCEL to Promela, which enables the analysis of systems specified in SCEL by means of the SPIN model checker[Hol04]. This should also serve as a guideline for a SCEL to BIP translation, which could be investigated in collaboration with WP5.

- The cooperation with WP6 has continued with the development of a new version of jRESP: a runtime environment providing an API for programming in Java autonomic and adaptive applications based on the SCEL paradigm. The new version of jRESP provides mechanisms for handling group oriented communications via P2P protocols; for supporting the integration of external reasoners; and for supporting *monitoring* of SCEL component. Other features, such as a library supporting specification and evaluation of policies and authorization requests, will be integrated next year.
- All activities of WP1 have paid full attention to the case studies investigated in WP7. In particular, in this deliverable we use a scenario from the automotive case study as a running example for illustrating the WP1 achievements of the third year. Concerning the cloud case study, we have used SCEL and SACPL for modeling the high-load scenario (see Deliverable JD3.2, Section 3.2.3, and [MKH+13]), and FACPL (an extension of SACPL) and its related tools for implementing a policy-based manager of a cloud IaaS system (see Deliverable JD3.1, Section 7, and [MMPT13b]). Concerning the robotics case study, we have used SCEL and MISSCEL to model and analyze a collision avoidance scenario (see Deliverable JD3.1, Section 2, and [BDVW]). SCEL and jRESP have been also used to model a rescue scenario (see Deliverable JD3.2). Moreover, we have applied the SCEL models of adaptation patterns defined in cooperation with WP4 to a robotic case study concerning object transportation (see Deliverable D4.3, Section 4.2, and [CNP+13]).

Structure of the Document

The rest of this document is organized as follows. In Section 2 we introduce SCEL$_{TS}$, i.e. the tuple-based incarnation of SCEL. In Section 3 we present the policy description language SACPL. In Section 4 we introduce MISSCEL, the SCEL$_{TS}$ interpreter exploiting MAUDE and its rich tool set. In Section 5 we present jRESP, the Java framework for providing implementations of the various SCEL variants. In Section 6 we present the methodology we have developed to integrate procedural descriptions of behavior with declarative reasoning. Finally, in Section 7 we draw some conclusions and sketch the work plan for the final year of the project.

2 SCEL$_{TS}$: a SCEL dialect based on tuple-spaces

In this section, we present a dialect of SCEL [DFLP12, DLPT13], named SCEL$_{TS}$, defined by specifying knowledge repositories as multiple distributed tuple-spaces (à la KLAIM [DFP98]). We will illustrate the main features of SCEL$_{TS}$ in a step-by-step fashion using the running example from the automotive domain described in the Introduction. We refer the interested reader to [DLPT13] for a full account of the language's semantics.

SCEL$_{TS}$ syntax is presented in Table 1. Its 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. KNOWLEDGE repositories are multisets of ITEMS, which are sequences of values (i.e., tuples). TEMPLATES, instead, are sequences of values and variables.

SYSTEMS:	$S ::=$	$C \mid S_1 \parallel S_2 \mid (\nu n)S$
COMPONENTS:	$C ::=$	$\mathcal{I}[\mathcal{K}, \Pi, P]$
PROCESSES:	$P ::=$	$\textbf{nil} \mid a.P \mid P_1 + P_2 \mid P_1[P_2] \mid X \mid A(\bar{p})$
ACTIONS:	$a ::=$	$\textbf{get}(T)@c \mid \textbf{qry}(T)@c \mid \textbf{put}(t)@c \mid \textbf{fresh}(n) \mid \textbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$
TARGETS:	$c ::=$	$n \mid x \mid \mathsf{self} \mid \mathsf{P} \mid \mathsf{p}$
KNOWLEDGE:	$\mathcal{K} ::=$	$\langle t \rangle \mid \mathcal{K}_1 \parallel \mathcal{K}_2$
ITEMS:	$t ::=$	$e \mid c \mid P \mid t_1, t_2$
TEMPLATES:	$T ::=$	$e \mid c \mid ?x \mid ?X \mid T_1, T_2$

Table 1: SCEL$_{TS}$ syntax (POLICIES Π are a parameter of the language, see Table 2 for an example)

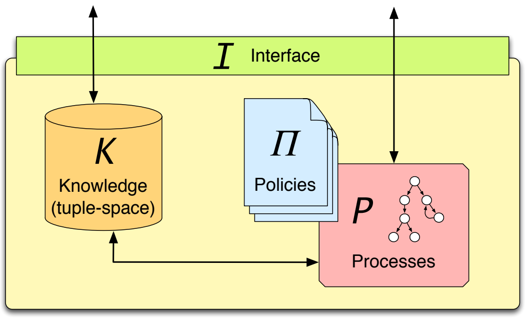

Figure 1: SCEL$_{TS}$ component

SCEL$_{TS}$ is parametric with respect to POLICIES; a concrete example of a language for expressing POLICIES is shown in Section 3.

Systems and components. 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 $(\nu n)\_$. 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. A component $\mathcal{I}[\mathcal{K}, \Pi, P]$, graphically illustrated in Figure 1, consists of:

- An *interface* $\mathcal{I}$ publishing and making available structural and behavioral information about the component itself in the form of *attributes*, i.e. names acting as references to information stored in the component's knowledge repository.

- A *knowledge repository* $\mathcal{K}$ managing application data, internal status data (supporting *self-awareness*) and environmental data (supporting *context-awareness*). 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.

- 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. Access control policies provide a standard example of policy abstractions; by exploiting them, components can protect themselves against unauthorised access, hence behaving in a *self-protecting* way.

- A *process* P, together with a set of process definitions that can be dynamically activated. Some of the processes in P execute local computations, while others may coordinate interaction with the knowledge repository or perform adaptation and reconfiguration. *Interaction* is obtained by allowing components to access knowledge in the repositories of other components.

Running example (step 1/5). The automotive scenario can be expressed in SCEL$_{TS}$ as a system S defined as follows

$$\begin{aligned} S \triangleq\ & \mathcal{I}_{ev1}[\mathcal{K}_{ev1}, \Pi_{ev1}, P_{ev1}] \parallel \ldots \parallel \mathcal{I}_{evn}[\mathcal{K}_{evn}, \Pi_{evn}, P_{evn}] \\ & \parallel \mathcal{I}_{park1}[\mathcal{K}_{park1}, \Pi_{park1}, P_{park1}] \parallel \ldots \parallel \mathcal{I}_{parkm}[\mathcal{K}_{parkm}, \Pi_{parkm}, P_{parkm}] \end{aligned}$$

E-vehicles and parks are rendered as components that concurrently interact. The interfaces of such components have the following forms:

$$\begin{aligned} \mathcal{I}_{evi} \triangleq \{ & (id, ev_i), (type, \text{“}\mathit{e\text{-}vehicle}\text{”}), (supply, \text{“}\mathit{electrical}\text{”}), (batteryLevel, l_i), \\ & (walkingDist, d_i), (maxCostPerHour, c_i), (riskClass, r_i), \ldots \} \end{aligned}$$

$$\begin{aligned} \mathcal{I}_{parkj} \triangleq \{ & (id, park_j), (type, \text{“}\mathit{park}\text{”}), (x_{park}, x_j), (y_{park}, y_j), (chargingStation, b_j), \\ & (costPerHour, c_j), \ldots \} \end{aligned}$$

Although attribute names specified in the interfaces are just pointers to the actual values contained in the knowledge repository associated to the component, for the sake of presentation we denote interfaces as collections of pairs $(attributeName, attributeValue)$. An e-vehicle interface exposes attributes concerning the vehicle (e.g., its kind of supply and charge level of the battery) and driver-specified parameters (e.g., the maximum distance the driver is willing to walk between the POI to be visited and a close parking lot, the maximum cost per hour that he/she is willing to pay for parking, and the driver's risk class of the insurance associated to the car). Similarly, a park interface specifies features of the park (e.g., its position, the availability of charging station for e-vehicles, and the cost per hour for parking). Notably, both interfaces expose the mandatory attribute id, which is bound to the name of the component, and the attribute $type$, which identifies the component type (i. e., $e\text{-}vehicle$ or $park$). □

Knowledge. A KNOWLEDGE repository $\mathcal{K}$ is a tuple-space, i.e. a (possibly empty) multiset of stored tuples $\langle t \rangle$, composed by the operator $\_ \parallel \_$. Values within tuples can either be targets c, or processes P or, more generally, can result from the evaluation of some given *expression* e. We assume that expressions may contain attribute names, *boolean*, *integer*, *float* and *string* values and variables, together with the corresponding standard operators. To pick a tuple out from a tuple-space by means of a given template T, the *pattern-matching* mechanism is used: a tuple matches a template if they have the same number of elements and corresponding elements have matching values or variables; variables match any value of the same type ($?\,x$ and $?\,X$ are used to bind variables to values and processes, respectively), and two values match only if they are identical. If more tuples match a given template, one of them is arbitrarily chosen.

Running example (step 2/5). Besides the data associated to the attributes exposed in the interface, the knowledge repository $\mathcal{K}_{ev\,i}$ of an e-vehicle contains information about the list of POIs to visit and some coordination data. Specifically, the POIs list is rendered as a collection of tuples of the form $\langle \text{“}\mathit{poi}\text{”}, j, x_j, y_j \rangle$, indicating that the j-th POI to visit has coordinates (x_j, y_j). Notably, at runtime, this list could be extended to include further POIs indicated by the driver, or automatically modified by a re-planning process (e.g., to deal with varying traffic conditions). Moreover, to coordinate the vehicle movements and the interactions with the driver, the following tuples are used:

$$\langle \text{“}\mathit{goTo}\text{”}, x, y \rangle \qquad \langle \text{“}\mathit{arrivedAt}\text{”}, x, y \rangle \qquad \langle \text{“}\mathit{searchNextPoi}\text{”} \rangle$$

The first tuple indicates the coordinates of the current destination, which is a reserved parking lot close to the POI that is going to be visited. The second tuple notifies that the vehicle is arrived at destination, while the third one triggers the search of a parking lot close to the next POI.

The knowledge repository $\mathcal{K}_{park\,j}$ stores information about the availability of parking lots. In particular, the tuple $\langle\text{“}plot\text{”}, x, y, park_j\rangle$ denotes an available parking lot with coordinates (x, y) and also provides the identifier $park_j$ of the park component. When the parking lot is booked, such tuple is withdrawn from the repository, which corresponds to the acquisition of the lock associated to this resource; then, when the parking lot is freed, the tuple is reinserted in the repository. □

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* ($A(\bar{p})$). In SCEL, the construct $P_1[\,P_2\,]$ abstracts the various forms of parallel composition commonly used in process calculi; in SCEL$_{TS}$ this construct is instantiated with a standard *interleaving* semantics, i.e. it is interpreted as the interleaved parallel composition of the two involved processes. Process variables can support *higher-order* communication, namely the capability to exchange (the code of) a process, and possibly execute it, by first adding an item containing the process to a knowledge repository and then retrieving/withdrawing this item while binding the process to a process variable. As shown in [GLPT12], this form of higher-order communication enables a straightforward implementation of adaptive behaviors. 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 $A(\bar{f}) \triangleq P$. In *process invocation* $A(\bar{p})$ and definition $A(\bar{f}) \triangleq P$, $\bar{p}$ and $\bar{f}$ denote lists of actual and formal parameters, respectively.

Running example (step 3/5). The process $P_{ev\,i}$ running on the i-th e-vehicle, i.e. on the component $\mathcal{I}_{ev\,i}[\mathcal{K}_{ev\,i}, \Pi_{ev\,i}, P_{ev\,i}]$, has the form $ParkSearch(1)[\,P\,]$, meaning that the process identified by $ParkSearch$ is invoked (with actual parameter 1) and executed in parallel with process P (left unspecified), denoting other processes run by the vehicle on-board system (e.g., processes dealing with driver instructions). Process $ParkSearch$ is defined as follows:

$$ParkSearch(j) \;\triangleq\; a_1 \,.\, a_2 \,.\, \cdots \,.\, a_n \,.\, ParkSearch(j+1)$$

where actions a_k, with $k \in \{1..n\}$, are sequentially executed to search a park close to the j-th POI. Once all such actions are completed, the process restarts (with an increased actual parameter). Notably, for the sake of simplicity, this process does not permit to execute alternative behaviours in case of such events as lack of available parking lots or requests for change of destination. Anyway, this kind of expected events could be easily dealt with by introducing alternative actions by means of the choice operator $\_ + \_$. Adaptation to unexpected events, instead, could be instrumented by means of the higher-order communication capability of the language, as already shown in Deliverable D1.1, Section 9. □

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 (i.e., tuples) from/to the knowledge repository identified by c. 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 tuple if it is not (yet) available in the repository. The two actions differ for the fact that **get** removes the found tuple 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', …, while variables for names are denoted by x, x', …. The distinguished variable self can be used by processes to refer to the name of the component hosting them. The possible targets could, however, be also singled out via predicates expressed as boolean-valued expression obtained by logically combining the evaluation of relations between attributes and expressions. Thus targets could also be an explicit *predicate* P or the name p of a predicate that is exposed as an attribute of a component interface whose value may dynamically change. 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 that performs the communication action).

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 are considered as the *ensemble* with which the process performing the action intends to interact. Indeed, our language does not have any specific syntactic category or operator for forming them. 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 $active = \text{``}yes\text{''} \wedge batteryLevel > \text{``}low\text{''}$.

Running example (step 4/5). By specifying actions a_k, with $k \in \{1..n\}$, the definition of process $ParkSearch$ becomes

$$\begin{array}{rcl} ParkSearch(j) & \triangleq & \mathbf{get}(\text{``}poi\text{''}, j, ?x_{poi}, ?y_{poi})@\mathsf{self}\,. \\ & & \mathbf{get}(\text{``}plot\text{''}, ?x_{plot}, ?y_{plot}, ?park)@\mathsf{P}_{parks}\,. \\ & & \mathbf{put}(\text{``}goTo\text{''}, x_{plot}, y_{plot})@\mathsf{self}\,. \\ & & \mathbf{get}(\text{``}searchNextPoi\text{''})@\mathsf{self}\,. \\ & & \mathbf{put}(\text{``}plot\text{''}, x_{plot}, y_{plot}, park)@park\,. \\ & & ParkSearch(j+1) \end{array}$$

where P_{parks} stands for predicate

$$type = \text{``}park\text{''} \wedge \sqrt{(x_{poi} - x_{park})^2 + (y_{poi} - y_{park})^2} \leq \mathsf{this}.walkingDist$$

This process first retrieves (and consumes) an element of the POI list and binds the coordinates of the current POI to variables x_{poi} and y_{poi}. Then, it interacts with the ensemble, identified by predicate P_{parks}, of parks (attribute $type$) whose location (attributes (x_{park}, y_{park})) is within a specified walking distance (attribute $walkingDist$) from the position of the considered POI (variables (x_{poi}, y_{poi})). Keyword this indicates that name $walkingDist$ refers to an attribute specified in the interface of the subject component, i.e. the e-vehicle; all other attributes refer to the object components, i.e. the parks. In particular, the process performs a group-oriented **get** action that books a parking lot in one of the parks belonging to the ensemble (by consuming the corresponding tuple and reading its content). If there is no matching tuple, i.e. no parking lot is available within the specified walking distance from the POI, the process is blocked; when such a tuple become available, the process resumes and locally produces a $goTo$ tuple with the parking lot's coordinates and waits for the next search request (tuple

POLICIES:	Π	$::=$	$\langle \mathit{Decision}\,;\text{target:}\{\ \mathit{Targets}\ \}\rangle \mid \Pi\ \text{p-o}\ \Pi \mid \Pi\ \text{d-o}\ \Pi$
DECISIONS:	$\mathit{Decision}$	$::=$	$\text{permit} \mid \text{deny}$
TARGETS:	$\mathit{Targets}$	$::=$	$\mathit{MatchF}(\mathit{Designator},\mathit{Expr})$
			$\mid \mathit{Targets}\ \text{or}\ \mathit{Targets} \mid \mathit{Targets}\ \text{and}\ \mathit{Targets}$
MATCHING FUNC.:	MatchF	$::=$	$\text{equal} \mid \text{pattern-match} \mid \text{less-than} \mid \ldots$
DESIGNATORS:	$\mathit{Designator}$	$::=$	$\text{action} \mid \text{item} \mid \text{subject}.\mathit{attr} \mid \text{object}.\mathit{attr}$
EXPRESSIONS:	Expr	$::=$	$\mathbf{get} \mid \mathbf{qry} \mid \mathbf{put} \mid \mathbf{fresh} \mid \mathbf{new} \mid T \mid \mathit{value}$
			$\mid \text{subject}.\mathit{attr} \mid \text{object}.\mathit{attr} \mid \text{not}\ \mathit{Expr} \mid \mathit{Expr}\ \text{or}\ \mathit{Expr}$
			$\mid \mathit{Expr}\ \text{and}\ \mathit{Expr} \mid \mathit{Expr} + \mathit{Expr} \mid \mathit{Expr} \times \mathit{Expr}$
			$\mid \mathit{Expr} < \mathit{Expr} \mid \mathit{Expr} = \mathit{Expr} \mid \ldots$

Table 2: SACPL policy syntax

$\mathit{searchNextPoi}$). Once the next search is triggered, the process reinserts the plot tuple in the park's repository (via a point-to-point **put** action), thus freeing the parking lot, and restarts. □

3 SACPL

In the previous section we have seen that SCEL$_{TS}$ is parametric with respect to the language for expressing policies. Such policies refine components behaviour for guaranteeing accomplishment of specific tasks or satisfaction of specific properties (e.g., protection of private information, careful management of resource usage, activation of adaptation procedures, etc.). As an example of policy language for SCEL$_{TS}$, we consider here SACPL (SCEL Access Control Policy Language), a simple, yet expressive, language for defining access control policies. In this section, we briefly present SACPL and focus on its integration with SCEL. We only sketch here the main ingredients and refer the interested reader to [DLPT13] for a full account of the language.

Access control is a fundamental mechanism for restricting the operations users can perform on protected resources. Many models of access control have been defined in the literature. SACPL focuses on the Policy Based Access Control (PBAC) model [NIS09], that is by now the de-facto standard model for enforcing access control policies in service-oriented architectures. In this model, a request to access a protected resource is evaluated with respect to one or more policies that define which requests are authorized. An authorization decision is based on attribute values required to allow access to a resource according to policies stored in system's components. Component attributes are used to describe the entities that must be considered for authorization purposes; they might concern: the *subject* who is demanding access, the *action* that it is requested to be performed, the *object* impacted by the action, and the *environment* identifying the context in which access is requested.

According to the PBAC model, SACPL policies are evaluated to decide if authorization requests are granted or forbidden. A *request* can be thought of as a function mapping (attribute) names to elements, and is generated from a label produced by the SCEL$_{TS}$ operational semantics in correspondence of a given action. For example, the request corresponding to the tentative of executing action $\mathbf{put}(t)@n$ provides information about the attributes of the request's subject, i.e. the component performing the action, the attributes of the request's object, i.e. the component identified by n, the exchanged item t and, of course, the type of the action, i.e. **put**. Each SCEL$_{TS}$ action is executed only if it is authorized by the policies in force at the component willing to perform the action and at the target component(s). In particular, when the target of an action **put** denotes a set of repositories satisfying a given target predicate, each insertion of the item in these repositories must be authorized separately by the policy in force at the corresponding component; such policy evaluation, however, does not affect the authorization of the insertions in the other target repositories. Instead, in case of a group-oriented action **get** or **qry**, only one authorization is required from the target side, since only one repository is selected for the interaction. Thus, SACPL policies regulate (intra- or inter-components) interactions by simply enabling or disabling behaviours; as discussed in Section 7, the possibility of adding new actions to components' behaviours as result of policies evaluation is left for future investigation.

SACPL syntax is presented in Table 2. Policies are hierarchically structured as trees. Indeed, a *policy* is either an atomic policy or a pair of simpler policies combined through one of the decision-combining operators p-o (*permit override*) and d-o (*deny override*). To match a composed policy $(\Pi_1 \text{ p-o } \Pi_2)$, an authorization request is only required to match one of Π_1 and Π_2, while it must match both Π_1 and Π_2, in order to match the policy $(\Pi_1 \text{ d-o } \Pi_2)$. An *atomic policy* is a pair consisting of a decision and a target. The target defines the set of requests to which the policy applies. If the target is empty, any request matches the policy. The *decision* — permit or deny — is the effect returned when the policy is 'applicable', i.e. the request belongs to the target. Otherwise, i.e. when a request does not belong to the policy's target, then the policy is not-applicable (this is a third kind of decision that can be returned by the semantics).

A *target* is either an atomic target or a pair of simpler targets combined using the standard logic operators or and and. To match a composed target $(\mathit{Targets}_1 \text{ or } \mathit{Targets}_2)$, a request is only required to match one of $\mathit{Targets}_1$ and $\mathit{Targets}_2$, while it must match both $\mathit{Targets}_1$ and $\mathit{Targets}_2$, in order to match the target $(\mathit{Targets}_1 \text{ and } \mathit{Targets}_2)$. An *atomic target* is a triple denoting the application of a matching function to values from the request and the policy, like e.g. less-than(subject.$\mathit{batteryLevel}$, 20%). To base an authorization decision on some characteristics of the request, e.g. subjects' or objects' identity, atomic targets use *designators* (i.e. *attribute names*) to point to specific values contained in the request. Specifically, the designator action refers to the action to be performed (such as **get**, **qry**, **put**, etc.). E.g., a request matches an atomic target of the form equal(action,**put**) if the request's action corresponds to the action **put** identified by the target. Similarly, item permits referring to the item exchanged in the considered interaction and, hence, an atomic target of the form pattern-match(item,T) is matched by all requests whose item matches the template T. Designators subject.attr and object.attr refer to the specific attribute attr provided, respectively, by the request's subject or object (like, e.g., subject.$\mathit{batteryLevel}$ or object.$\mathit{costPerHour}$). Finally, Expressions are built from values and attributes through various operators. The evaluation of an atomic target involving a subject (resp. object) designator consists in obtaining the subject (resp. object) interface from the request, retrieving the value of the attribute from the interface, evaluating the expression by possibly retrieving other attribute values from the request elements and, finally, calling the corresponding match function.

Running example (step 5/5). In the automotive scenario, an e-vehicle component (with identifier ev_i) could refine its behaviour by specifying Π_{ev_i} as the SACPL policy resulting from the composition, by means of the d-o (deny override) operator, of the following policies:

⟨permit ; target: { } ⟩

⟨deny ; target: { equal(action,**get**) and
equal(subject.id,ev_i) and
pattern-match(item,("$plot$", _, _, _)) and
less-than(subject.$\mathit{batteryLevel}$,20%) and
equal(object.$\mathit{chargingStation}$,false) } ⟩

$$\langle \mathsf{deny}\,;\mathsf{target}\!:\{\ \begin{array}{l}\mathsf{equal}(\mathsf{action},\mathbf{get})\quad \mathsf{and}\\ \mathsf{equal}(\mathsf{subject}.id,ev_i)\quad \mathsf{and}\\ \mathsf{pattern\text{-}match}(\mathsf{item},(\text{“}plot\text{”},\_,\_,\_))\quad \mathsf{and}\\ \mathsf{greater\text{-}than}(\mathsf{object}.costPerHour,\mathsf{subject}.maxCostPerHour)\ \}\ \rangle\end{array}$$

The composed policy says that all actions are permitted apart for the withdrawal of $plot$ tuples by ev_i from an object park component that either has no charging station while the e-vehicle has a low battery level or has a cost per hour greater than the maximum cost that the driver is willing to pay. For the sake of readability, we use "_" to denote a *don't care* formal field in a template.

Similarly, a park component (with identifier $park_i$) could regulate the access to its parking lot tuples by specifying Π_{park_i} as the SACPL policy resulting from the composition, by means of the d-o (deny override) operator, of the following policies:

$$\langle \mathsf{permit}\,;\mathsf{target}\!:\{\ \ \}\ \rangle$$

$$\langle \mathsf{deny}\,;\mathsf{target}\!:\{\ \begin{array}{l}\mathsf{equal}(\mathsf{action},\mathbf{get})\quad \mathsf{and}\\ \mathsf{equal}(\mathsf{object}.id,park_i)\quad \mathsf{and}\\ \mathsf{pattern\text{-}match}(\mathsf{item},(\text{“}plot\text{”},\_,\_,\_))\quad \mathsf{and}\\ \mathsf{equal}(\mathsf{subject}.supply,\text{“}lpg\text{”})\ \}\ \rangle\end{array}$$

$$\langle \mathsf{deny}\,;\mathsf{target}\!:\{\ \begin{array}{l}\mathsf{equal}(\mathsf{action},\mathbf{get})\quad \mathsf{and}\\ \mathsf{equal}(\mathsf{object}.id,park_i)\quad \mathsf{and}\\ \mathsf{pattern\text{-}match}(\mathsf{item},(\text{“}plot\text{”},\_,\_,\_))\quad \mathsf{and}\\ \mathsf{equal}(\mathsf{subject}.riskClass,\text{“}high\text{”})\ \}\ \rangle\end{array}$$

This composed policy allows all actions except for the **get** actions on $plot$ tuples performed by e-vehicles whose power supply is LPG (such rule is typically applied by underground parks) or driver's class risk is high. □

4 MISSCEL

SCEL comes with solid semantics foundations laying the basis for formal reasoning. MISSCEL[1], a MAUDE Interpreter and Simulator for SCEL, is a first step in this direction. MISSCEL is an implementation of SCEL's operational semantics based on Rewriting Logic [Mes12], and it is written in MAUDE [CDE+07], an instantiation of Rewriting Logic which permits to execute rewrite theories. What we obtain is then an executable operational semantics for SCEL, that is an interpreter.

MISSCEL currently focuses on SCEL_{TS}, i.e., as described in Section 2, repositories are implemented as multisets of tuples, while the processes of a SCEL component evolve in a pure interleaving fashion. Access control policies are supported, even if no policy language has been integrated yet: as default, every request is currently authorized.

Given a SCEL specification, thanks to MISSCEL it is possible to exploit the rich MAUDE toolset [CDE+07] to perform:

- automatic state-space generation;
- qualitative analysis via MAUDE's invariant and LTL model checkers;
- debugging via probabilistic simulations and animations generation;
- statistical quantitative analysis via the recently proposed MULTIVESTA [SV], a distributed statistical analyser extending VESTA and PVESTA [AM11, SVA05b].

[1] `http://sysma.lab.imtlucca.it/tools/misscel/`

```
 1  SC( I(tId('SCId), tId('batteryLevel), tId('maxCostPerHour), tId('riskClass), tId('supply),
 2        tId('type), tId('walkingDist)),
 3      K(< tId('SCId) ; av(id('ev-1))        >, < tId('batteryLevel) ; av("high")  >,
 4        < tId('maxCostPerHour) ; av(2.0)    >, < tId('riskClass) ; av(10)         >,
 5        < tId('supply) ; av("electrical")   >, < tId('type) ; av("e-vehicle")     >,
 6        < tId('walkingDist) ; av(50.0)      >, < av("pos") av(41.0) av(3.0)       >,
 7        < av("poi") av(1) av(2.0) av(3.0)   >, < av("poi") av(2) av(8.0) av(10.0) >,
 8        < av("poi") av(3) av(20.0) av(20.0) >),
 9      Pi(INTERLEAVING-PROCESSES_AUTHORIZE-ALL),
10      P(get(< av("poi") av(1) ?x('xpoi) ?x('ypoi) >)@ self .
11        get(< av("plot") ?x('xplot) ?x('yplot) ?x('park) >)@ Pparks(x('xpoi), x('ypoi)) .
12        put(< av("goto") x('xplot) x('yplot) >)@ self .
13        get(< av("searchNextPoi") >)@ self .
14        put(< av("plot") x('xplot) x('yplot) x('park) >)@ x('park) .
15        pDef('ParkSearch, av(2)))
16    )
```

Listing 1: A MISSCEL component representing an e-vehicle

```
op Pparks : FormalOrActualValue FormalOrActualValue -> Predicate .
vars xpoi ypoi : ActualValue .
eq Pparks(xpoi,ypoi)
 = remote. tId('type) = av("park") AND
   dist(xpoi,ypoi, remote. tId('xpark), remote. tId('ypark)) <= this. tId('walkingDist) .
```

Listing 2: The P_{parks} predicate in MISSCEL

A further advantage of MISSCEL is that SCEL specifications can now be intertwined with raw MAUDE code, exploiting its great expressiveness. This permits to obtain cleaner specifications in which SCEL is used to model behaviours, aggregations, and knowledge manipulation, leaving scenario-specific details like, e.g., e-vehicles movements or computation of distances to MAUDE. As discussed, each e-vehicle and park of our scenario is modelled as a SCEL component. Listing 1 provides the MISSCEL representation of an e-vehicle, as described in the *steps 1-4/5* of our running example.

In MISSCEL, a SCEL component is defined as a MAUDE term with sort `ServiceComponent` built with the operation `op SC : Interface Knowledge Policies Processes -> ServiceComponent`. By implementation choice, in MISSCEL tuples may have an identifier (e.g., `< tId('type) ; av("e-vehicle") >` is a tuple with identifier `type`), but it is not mandatory (e.g., `< av("pos") av(41.0) av(3.0) >` has no identifier). Note that, for implementation reasons, actual values (e.g. strings and integers) are enclosed in the operation `av`. Note moreover that `'type` is a MAUDE term with sort quoted identifier (similar to strings) built by prefixing alphanumeric words with the operator "`'`". However, only tuples with identifier can be exposed by the interface, as identifiers are used as pointers to the actual values of the tuples stored in the knowledge. Then, as depicted in lines 1-2 of Listing 1, an interface is just a set of tuple identifiers enclosed in the MAUDE operation `I`, while, as depicted in lines 3-8, the knowledge is a multiset of tuples enclosed in the operation `K`. For example, the described e-vehicle has id `id('ev-1)`, a battery level `high` and type `e-vehicle`. Line 9 specifies that the default policy is enforced.

Lines 10-15 contain the behaviour specification of the e-vehicle, i.e., the process *ParkSearch* presented in the *step 4/5* of our running example. Note the almost one-to-one correspondence between the process specification given in our running example, and the one given in MISSCEL. SCEL variables with type value are built with the MAUDE operations `?x` (when binding) or `x`, having as parameter the name of the variable (we also have the corresponding process variables `?X` and `X`).

Predicates are defined as MAUDE operations with sort `Predicate`. As depicted in line 11 of

```
eq [movementActuator] :
   SC(I,K(< av("goto") av(xplot) av(yplot) >, < av("pos") av(xve) av(yve) >    , k),Pi,P)
 = SC(I,K(< av("searchNextPoi") >          , < av("pos") av(xplot) av(yplot) >, k),Pi,P) .
```

Listing 3: The MAUDE equation to actuate e-vehicles movements

```
eq invoke(pDef('ParkSearch, av(j)))
 = get(< av("poi") av(j) ?x('xpoi) ?x('ypoi) >)@ self .
   get(< av("plot") ?x('xplot) ?x('yplot) ?x('park) >)@ Pparks(x('xpoi), x('ypoi)) .
   put(< av("goto") x('xplot) x('yplot) >)@ self .
   get(< av("searchNextPoi") >)@ self .
   put(< av("plot") x('xplot) x('yplot) x('park) >)@ x('park) .
   pDef('ParkSearch, av(j + 1)) .
```

Listing 4: The definition of process *ParkSearch* in MISSCEL

Listing 1, we defined the predicate Pparks, corresponding to P_{parks} of the *step 4/5* of our running example. Clearly, we have to provide the predicate with two parameters, i.e., the coordinates of the POI obtained in line 10. Listing 2 provides the specification of Pparks. In line 1 we define the MAUDE operation Pparks with sort Predicate having as parameters two FormalOrActualValue (i.e., either SCEL variables or actual values). Then, in lines 3-5 we provide the body of the predicate in the form of a MAUDE equation. MAUDE equations are *executed* by the MAUDE engine to rewrite occurrences of terms (in this case Pparks) matching the left-hand side (LHS) of the equation (i.e., before the =) in the term specified in the right-hand side (RHS) of the equation (i.e., after the =), in this case the body of the predicate. Given that at line 2 we specify the MAUDE variables (i.e., place-holders for any term with the same sort) xpoi and ypoi with sort ActualValue, we have that only instantiated occurrences of Pparks (i.e., where all the SCEL variables have been replaced by actual values) match with the LHS of the equation. Note that in predicates we follow the convention of prefixing local tuple identifiers with the keyword this, while we use remote for those referring to the target of the communication.

Line 5 of listing 2 provides an interesting example demonstrating the usefulness of mixing SCEL and MAUDE specifications: dist is a MAUDE operation which computes the distance between a POI and a park. In our simple case study, this actually correspond to the Euclidean distance. Noteworthy, in case we would consider *distances* with different assumptions, e.g., considering public transportations, it would be sufficient to change the MAUDE operations leaving unchanged the SCEL specification.

The MAUDE equation of Listing 3 provides another example in which we intertwine SCEL and MAUDE specifications: the equation actuates the movement of an e-vehicle towards the booked parking lot, making it instantaneous.[2] Again, in case we would consider movements with different assumptions (e.g., time spent in the trip, possibly keeping into account traffic jams or spatial information), we would just need to modify the equation.

Finally, it may be worth to note that line 15 of Listing 1 provides an hint on how MISSCEL deals with process definitions. Intuitively, once all the preceding actions have been executed, the process definition pDef('ParkSearch, av(2)) is invoked, i.e., it is replaced with its body, defined using the MAUDE equation of Listing 4 (where j is a MAUDE variable with the sort of natural numbers), similarly to what described for predicates.

Coming to semantics-related aspects, the operational semantics of SCEL is defined in two steps:

[2]Note that this is a model-specific equation used to abstract from the movements of e-vehicles in SCEL. The equational theory remains confluent, as in our example scenario there will never be more than a "goto" tuple per component. In the general case one should add further equations that solve inconsistencies (e.g. apply the equation only when one "goto" is present, and delete multiple "goto" according to a suitable resolution strategy).

```
op commit : Process -> Commitment .
rl commit(P) => commitment(inaction,P) .
rl commit(a . P) => commitment(a,P) .
crl commit(P + Q) => commitment(a, P1) if commit(P) => commitment(a, P1) .
```

Listing 5: The rules of Figure 2 implemented in MISSCEL

the semantics of processes, and the semantics of systems. First, the semantics of processes specifies their commitments, ignoring the structure of SCEL components. Namely, issues like allocation of processes to a component, available data in the knowledge, and regulating policies are ignored at this level. Then, by taking process commitments and system configuration into account, the semantics of systems provides a full description of systems behavior. The same happens in MISSCEL. For easiness of presentation, we now exemplify the correspondence of SCEL semantics and its implementation in MISSCEL for the semantics of processes only.

Figure 2 depicts four of the rules defining SCEL's semantics of processes, specifying, respectively from left to right, that: any process can commit in itself executing an inaction, a process composed by P prefixed by an action a can commit in P by executing a, a process $P + Q$, in which P can commit in P' executing an action, or Q can commit in Q' executing another action, can commit either in P' or in Q' executing the corresponding action.

Listing 5 depicts (omitting unnecessary details) how we implemented the rules of Figure 2 in MISSCEL. Where `P`, `Q` and `P1` are MAUDE variables with sort `Process` (i.e. place-holders for any term with the specified sort), while `a` is an `Action` variable. The correspondence is straightforward. Note that we need only one rule for the + operator, as we defined it with the `comm` axiom, meaning that it has the commutative property, meaning the when applying a rule to `P + Q`, MAUDE will try to match the rule also with `Q + P`.

Interestingly, in [BDVW] we exploited MISSCEL and the recently proposed MULTIVESTA [SV] to perform a statistical quantitative analysis of a robotic collision avoidance scenario modelled in SCEL. Note however that MISSCEL is an executable operational semantics for SCEL, and as such, given a SCEL specification representing a system's state (i.e. a set of SCEL components), MISSCEL executes it by applying a rule of SCEL's semantics to (part of) the state. According to such semantics, a system evolves non-deterministically by executing the process of one of its components, and in particular by consuming one of its actions. As usual (especially in the MAUDE context, e.g., [BÖ12, BCG$^+$12b, AMS06, EMA$^+$12]), in order to perform statistical analysis, it is necessary to obtain probabilistic behaviours out of non-deterministic ones by resolving non-determinism in probabilistic choices. For this reason, we defined a Java wrapper for MISSCEL, together with a set of external schedulers which allow to obtain probabilistic simulations of SCEL specifications, which can then be exploited by MULTIVESTA to perform statistical analysis, like statistical model checking [SVA05a]. More details are provided in [BDVW] and in the Section 2 of Deliverable JD3.1.

$$P \downarrow_a P \qquad a.P \downarrow_a P \qquad \frac{P \downarrow_\alpha P'}{P\, +\, Q \downarrow_\alpha P'} \qquad \frac{Q \downarrow_\alpha Q'}{P\, +\, Q \downarrow_\alpha Q'}$$

Figure 2: Four of the rules of SCEL's semantics of processes.

5 jRESP: A Java runtime environment for SCEL programs

In Section 2 and Section 3 we have shown how SCEL can be used to specify a significative case study. This specification can then be analyzed via MISSCEL. This tool, presented in Section 4, implements SCEL semantics in MAUDE and can be used to support qualitative and quantitative analysis of SCEL programs.

The next step in the development process is the deployment. To perform this operation we need to use a runtime environment supporting the execution SCEL programs. To execute SCEL programs, in the ASCENS project two frameworks – jRESP and jDEECo – have been developed. Both frameworks implements the concepts of SCEL in Java and have been presented in Deliverable D1.5.

In this section we recall some basic features of jRESP and we show how this framework can be used to execute SCEL programs via Java programs. A detailed description of jRESP can be found in [DLPT13] .

jRESP[3] is a Java runtime environment providing a framework for developing autonomic and adaptive systems according to the SCEL paradigm. Specifically, jRESP provides an API that permits using in Java programs the SCEL's linguistic constructs for controlling the computation and interaction of autonomic components, and for defining the architecture of systems and ensembles.

The implementation of jRESP fully relies on the SCEL's formal semantics. This close correspondence enhances confidence on the behaviour of the jRESP implementation of SCEL programs, once the latter have been analysed through formal methods made possible by the formal operational semantics.

We have already explained in the previous sections that SCEL is parametric with respect to some aspects, e.g. knowledge representation, that may change to tailor to different application domains. For this reason, also jRESP is designed to accommodate alternative instantiations of the previously mentioned features. Indeed, thanks to the large use of design patterns, the integration of new features in jRESP is greatly simplified.

SCEL's operational semantics abstracts from a specific communication infrastructure. A SCEL *program* typically consists of a set of (possibly heterogeneous) components, each of which is equipped with its own knowledge repository. These components concur and cooperate in a highly dynamic environment to achieve a set of *goals*. In this kind of systems the underlying communication infrastructure can change dynamically as the result of local component interactions. To cope with this dynamicity, jRESP communication infrastructure has been designed to avoid *centralized control*. Moreover, to facilitate interoperability with other tools and programming frameworks, jRESP relies on JSON[4]. This is an open data interchange technology that permits simplifying the interactions between heterogeneous network components and provides the basis on which SCEL programs can cooperate with external services or devices.

5.1 Automotive scenario in jRESP

We report here the code of the jRESP implementation of the SCEL specification, presented in Section 2, of the automotive scenario. The Java classes reported in this section permit appreciating how close the SCEL processes are to their implementation in jRESP. The complete source code for the scenario, together with a simulation environment, can be downloaded from `http://jresp.sourceforge.net/`.

Process $ParkSearch(j)$ is rendered as the agent ParkSearch defined next:

[3]`http://jresp.sourceforge.net/`.
[4]`http://www.json.org/`.

```
public class ParkSearch extends Agent {

  private int j;
  public ParkSearch( int j ) {
    super("ParkSearch");
    this.j = j;
  }
  protected void doRun() throws Exception {
      Tuple t = query(new Template(new ActualTemplateField("poi"),
                                              new ActualTemplateField(j),
                                              new FormalTemplateField(Double.class),
                                              new FormalTemplateField(Double.class)),
                           Self.SELF);
      double xPoi = t.getElementAt(Double.class,2);
      double yPoi = t.getElementAt(Double.class,3);
      Tuple t2 = get(new Template(new ActualTemplateField("plot"),
                                              new FormalTemplateField(Double.class),
                                              new FormalTemplateField(Double.class),
                                              new FormalTemplateField(Locality.class),
                           getPredicate( xPoi , yPoi ) );
      double xplot = t2.getElementAt(Double.class,1);
      double yplot = t2.getElementAt(Double.class,2);
      Locality park = t2.getElementAt(Locality.class,3);
      put( new Tuple(
           "goTo" ,
           xplot ,
           yplot ,
        ) , Self.SELF );
      get( new Template( new ActualTemplateField("seachNextPoi") ) , Self.SELF );
      put(new Tuple( "plot" , xplot , yplot , park ) , park );
      ParkSearch next = new ParkSearch(j+1);
      next.call();
  }

  public Group getPredicate( double x , double y ) {
      return new Group(
        new AndPredicate(
                new HasValue( "type" , "park" ) ,
                new LessThen(
                        new Distance(
                                new Value(x) ,
                                new Value(y) ,
                                new AttributeName("xpark") ,
                                new AttributeName("ypark")
                        ) ,
                        new Value( getValue(Double.class , "wakingDistance") )
                )
        )
      );
  }

}
```

When an instance of class Agent is executed, the method doRun() is invoked. This method defines the agent behaviour. In the case of ParkSearch, it consists of sequence of actions implementing the protocol for discovering the next available parking lot. The method query(), used to retrieve data from a knowledge repository, is defined in the base class Agent and implements the SCEL's action **qry**. This method takes as parameters an instance of class Template and a target, and returns a matching tuple. In the previous case, the target is the local component (referred by Self.SELF) while the retrieved tuple is one consisting of four fields. The first two fields are the constant "poi" and an integer j identifying the index of the searched point-of-interest. The last two fields are formal fields and are used to retrieve the coordinates of the j-th point-of-interest in the route. The retrieved values, both doubles, are stored in variables $xpoi$ and $ypoi$. After that, method get is invoked. This method implements the SCEL's action **get** and is used to book a parking lot from a parking located at a *walking distance* from the position $(xpoi, ypoi)$. This is a *group oriented* interaction. Indeed, to book a parking lot a knowledge element is retrieved from one component among those satisfying the requested predicate. jRESP relies on specific protocols guaranteeing the appropriate implementation of this complex distributed interaction. A description of these protocols can be found in [DLPT13] and in Deliverable D1.5.

To move towards the position of the reserved parking lots (stored in $xplot$ and $yplot$) the method put() is invoked. This implements action **put** and it is used to indicate that the car should move towards location $xpoi$ and $ypoi$. Finally, the agent waits until a tuple containing value seachNextPoi is available in the local knowledge. When this tuple is retrieved, the parking lot is released and the agent responsible of moving to the next *poi* (the one with index $j + 1$) is executed (via method call()).

6 Enriching SCEL components with reasoning capabilities

SCEL is sufficiently powerful for dealing with coordination and interaction issues. However it does not provide explicit machineries for specifying components that take decisions about the action to perform basing on their context, or on a partial perception of it. Obviously, the language could be extended in order to encompass such possibilities, and one could have specific reasoning phases, or dedicated SCEL components, triggered by the perception of changes in the context.

In our view, it is however preferable to have separate reasoning components specified in another language, that SCEL programs can invoke when they need to take decisions. Having two different languages for computation and coordination, and for *reasoning*, does guarantee *separation of concerns*, a fundamental property to obtain reliable and maintainable specifications. As we will see, for this reason we discarded the first obvious choice of using dedicated ordinary SCEL components that take care of reasoning.

Also, it may be beneficial to have a methodology for integrating with a given programming language different reasoners designed and optimised for specific purposes. What we envisage is having SCEL programs that whenever have to take decisions have the possibility of invoking an external reasoner by providing to it information about the relevant knowledge they have access to, and receiving in exchange informed suggestions about how to proceed. In a scenario like the e-vehicles one, reasoners could be exploited by e-vehicles in order to react to unexpected events, like e.g., the failure of the booking of a parking lot, traffic jams, or the unavailability of booked parking lots. Consider for example the case of a failure of a booking of a parking lot, e.g., when no parking lots at walking distance from the next POI are currently available. As discussed in the Introduction, we assume that the lists of POIs are computed before starting the journey, by possibly optimizing time and cost of the travel. However, in this case a dynamic replanning of such list may be useful rather than waiting for the availability of a parking lot near to the considered POI. Intuitively, the list of remaining POIs should be provided to a reasoner, which would shuffle it following some criteria, and would return the obtained list to the SCEL component that required it.

In [BDVW] we started our investigation towards the actual integration of SCEL components and reasoners. In particular, we provided a general methodology to enrich SCEL components with reasoning capabilities by resorting to explicit *reasoner integrators*, we instantiated the methodology for MISSCEL, and we discussed the integration of MISSCEL with the PIRLO reasoner [Bel13]. This permits to specify *reasoning service component ensembles*. The concrete integration has been indeed simplified by the common underlying logical tools, as PIRLO is based on rewriting logic and MAUDE as it is the case of MISSCEL. Nevertheless, the work in [BDVW] paves the way to the design of interfaces and methodologies to be used for building up systems composed of separated components concerned with computational aspects and decision making.

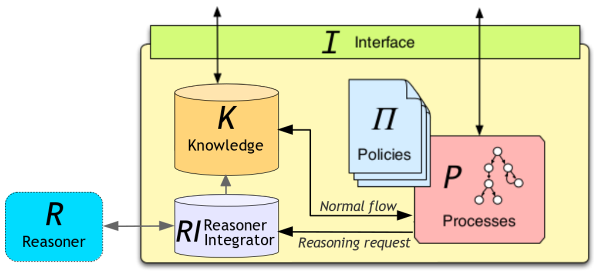

Figure 3: Enriched SCEL component.

Noteworthy, the use of the MAUDE framework for the implementation of MISSCEL and PIRLO paves the way towards the exploitation of tools and techniques for analysing the behaviour of reasoning service component ensembles, allowing thus to *reason on* reasoning service component ensembles. As an example, in [BDVW] we have shown how MULTIVESTA [SV], a recently proposed statistical analyzer for probabilistic systems, can be used to evaluate the implementation of a simple collision avoidance scenario consisting of a group of robots moving in an arena, where some of the robots exploit PIRLO to choose the movements to be performed in order to reduce the number of collisions.

In the following we present our approach to enrich SCEL components with external reasoning capabilities (Section 6.1), and we show how it has been instantiated for MISSCEL (Section 6.2).

6.1 Methodology

We aim at enriching SCEL components with an external reasoner to be *invoked* when necessary (e.g., if a replanning of the list of POIs to be visited is necessary). Ideally, this should be done by minimally extending SCEL. In Figure 1 we depicted the constituents of a SCEL component: interfaces, policies, processes and repositories. Interfaces will not be involved in the extension, as they only expose the local knowledge to other components. Moreover, we currently restrict ourselves to not explicitly consider policies in the extension. Since processes store and retrieve tuples in repositories, the interaction between a process and its local repository is a natural choice where to plug-in a reasoner: we can use special data (*reasoning request tuples*) whose addition to the local knowledge (i.e., via a **put** at `self`) triggers the reasoner. For example, assuming to have a reasoner offering the capability of replanning the list of POIs, an e-vehicle may require the help of the reasoner in case no parking lots are currently available for the next POI by resorting to an action like $\mathbf{put}(\text{“}\mathit{reasoningRequest}\text{”}, \text{“}\mathit{replan}\text{”}, \mathit{remainingPOIs})@\mathsf{self}$, where $\mathit{remainingPOIs}$ is the list of POIs to be visited. Reasoning results can then be stored in the knowledge as *reasoning result tuples*, allowing local processes to access them as any other data (e.g., via a **get** from `self`). For example, the list of POIs generated by the reasoner can be accessed by resorting to an action like $\mathbf{get}(\text{“}\mathit{reasoningResult}\text{”}, \mathit{shuffledPOIs})@\mathsf{self}$, where $\mathit{shuffledPOIs}$ is the list of POIs generated by the reasoner.

We could have either passive reasoners invoked when necessary, or active ones that continuously monitor the repositories, and act when necessary. We currently focus on the first type.

Figure 3 depicts such an *enriched* SCEL *component*, together with a generic external reasoner *R*. With respect to Figure 1, now local communications are filtered by *RI*, a *reasoner integrator*.

As depicted by the grey arrow between *RI* and the external reasoner *R*, in case of reasoning requests, *RI* invokes *R*, which evaluates the request and returns back the result of the reasoning phase. *RI* then stores the obtained result in the knowledge, allowing the local processes to access it via common **get** or **qry** actions. In case of normal data, the flow goes instead directly to the knowledge. Note that in our methodology only local **get** of reasoning request tuples trigger a reasoner.

Actually, *RI* has the further fundamental role of translating data among the internal representations used by SCEL and by the reasoner, acting hence as an adapter between them. For example, the reasoner may use a different representation for the lists of POIs with respect to SCEL. To sum up, *RI* performs three tasks: it first translates the parameters of the reasoning requests from SCEL's representation to the reasoner's one (*scel2reasoner*), then it invokes the reasoner (*invokeReasoner*), and finally translates back the results (*reasoner2scel*). Clearly, each reasoner requires its own implementation of the three operations. Hence, as depicted in Figure 4, we separate the *RI* component into an *Abstract Reasoning Interface* and a *Concrete Adapter*. The former is given just once and contains the definition of the three operations, while the latter is reasoner- and domain-specific, and provides the actual implementation of the three operations. In Section 6.2 we discuss the instantiation for MISSCEL of the Abstract Reasoning Interface. The three operations implemented by a Concrete Adapter provide a connection from SCEL to a particular reasoner taking care of the translation of syntactical representations and of the actual execution of the reasoning operation. An example of a concrete adapter is presented in [BDVW] for the reasoner PIRLO in the context of a collision avoidance robotic scenario, where each robot provides its perception of the surrounding environment to the reasoner, which then computes the movement with minimal probability of colliding with other robots.

Note that the presented methodology is not restricted to a particular reasoner. Moreover, many reasoners could be used at the same time, each performing particular reasoning tasks for which they are best suited. To this end, particular *reasoning services* (like e.g., the *replan* one) can be requested by a SCEL process according to the task at hand.

Finally, it may be worth to discuss the fact that, as mentioned before, we did not investigated yet the role of policies in extending SCEL's components with reasoning capabilities. However, they already play an important role in our methodology, as they can manipulate the flow of data among processes and local repositories, and thus can intercept, modify or generate reasoning requests and reasoning results. Moreover, we can easily foresee a scenario in which complicated policies, possibly involving reasoning tasks, resort to a reasoner as well following the proposed methodology. For example, in case of a group **get** like, e.g., the second action of the process *ParkSearch* presented in the *step 4/5* of our running example, which is used to select the next park, it may be useful to allow policies to use reasoners in order to select the *best* tuple among the many matching ones present in a distributed repository (e.g., the parks at walking distance from the current POI) according to some specific criteria (e.g., the park reachable following the cheapest path).

6.2 Providing the Abstract Reasoning Interface in MISSCEL

We now discuss how we enriched MISSCEL to provide components with Abstract Reasoning Interfaces.

Listing 6 depicts the MISSCEL's Abstract Reasoning Interface (omitting unnecessary details). Lines 3-4 defines the reasoner-side sorts and variables for reasoning requests and results. Spe-

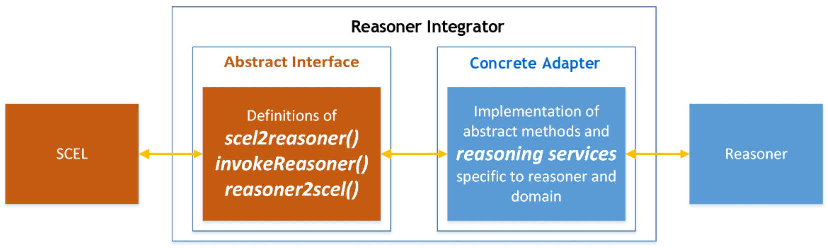

Figure 4: An architectural perspective of the reasoner integrator.

```
 1  mod ABSTRACT-REASONING-INTERFACE is
 2   --- importings of modules are omitted
 3   sorts RRequest RRequestParameters RResult RResultParameters .
 4   var rReq : RRequest . var rRes : RResult .
 5   var requestParameter resultParameter : List{ActualValue} . var t : Tuple .
 6
 7   op scel2reasoner  : List{ActualValue} -> RRequestParameters .
 8   op invokeReasoner : RRequest -> RResultParameters .
 9   op reasoner2scel  : RResult -> List{ActualValue} .
10
11   op invokeReasonerIfNecessary : Tuple -> Tuple .
12   ceq invokeReasonerIfNecessary(< tId('reasoningRequest) ; requestParameter >)
13     =                           < tId('reasoningResult)  ; resultParameter  >
14    if rReq := scel2reasoner(requestParameter)
15    /\ rRes := invokeReasoner(rReq)
16    /\ resultParameter := reasoner2scel(rRes) .
17   eq invokeReasonerIfNecessary(t) = t [ owise ] .
18  endm
```

Listing 6: The MISSCEL's Abstract Reasoning Interface.

cific constructors to build terms with these sorts have to be provided by the Concrete Adapters. Line 5 defines the variables used to match the SCEL-side parameters of the reasoning results and requests (lists of actual values like e.g., integers, strings or more complex ones like data structures containing POIs information). Lines 7-9 define the three operations discussed in Section 6.1. Note how `scel2reasoner` goes from SCEL-side to reasoner-side values, `reasoner2scel` does the opposite, and `invokeReasoner` deals with reasoner-side values only. Lines 11-17 show how our methodology is actuated: in case of a local **put** of a tuple `t`, we actually store the result of `invokeReasonerIfNecessary(t)`. If `t` is a reasoning request tuple (i.e., has id `reasoningRequest`, line 12), then its parameters are translated by `scel2reasoner` (line 14), the reasoner is invoked (line 15), and the obtained result is translated back by `reasoner2scel` (line 16). Note that the result is enclosed in a reasoning result tuple (line 13). Finally, if `t` is not a reasoning request tuple, the equation of line 17 is applied (due to the `owise` clause standing for *otherwise*), leaving it unchanged.

Intuitively, in order to integrate a reasoner in MISSCEL, it is necessary to provide a MAUDE module specifying the body of the three operations `scel2reasoner`, `reasoner2scel` and `invokeReasoner` via MAUDE equations. For example, in the example discussed in Section 6.1, it would be necessary to translate lists of POIs between SCEL's and reasoner's representations (via, respectively, `scel2reasoner` and `reasoner2scel`). And then `invokeReasoner` should invoke the replanning capability of the reasoner.

7 Concluding Remarks and Work Plan for Year Four

We have introduced a dialect of SCEL, a formalism that brings together various programming abstractions that permit directly representing *knowledge*, *behaviors* and *aggregations* according to specific *policies*, and naturally programming interaction, adaptation and self- and context-awareness. The language is parametric with respect to the Knowledge representation and handling mechanisms and with respect to the policy language, used for controlling both actions execution and process interaction. The dialect, that we call SCEL$_{TS}$, takes a precise standing with respect to knowledge representation that is defined by specifying knowledge repositories as multiple distributed tuple spaces. We have then introduced a specific language for specifying policies for access control and described how the two formalisms can co-exist.

Moreover, we have described MISSCEL, an interpreter for SCEL based on MAUDE that exploits the solid semantics foundations of SCEL to lay the basis for formal reasoning on SCEL specifications by resorting to the MAUDE toolset. After this we have described jRESP, the Java runtime environment providing an API that permits using the SCEL's linguistic constructs in Java programs. We have concluded by discussing a possible strategy for integrating SCEL programs with external reasoners to be invoked when specific decisions have to be taken.

All formalisms and methodologies have been presented by resorting to a simple scenario from the automotive case study. During the fourth year we plan to do further work along the lines described above.

SCEL at Work. We will assess the extent to which SCEL achieves its goals. As testbeds we will continue using the different case studies considered in the project. This process might require further tuning the language features and, hence, the related jRESP and MISSCEL implementations. Specific attention will be dedicated to the integration of reasoners and SCEL programs.

New Policy Languages. We plan to integrate with SCEL a more flexible and expressive policy language, which is also closer to real-world policy languages. We are currently working on FACPL [MMPT13a], a policy language featuring more structured policy specifications, additional policy combining operators, and the possibility to associate actions that should be performed in conjunction with the enforcement of an authorisation decision. FACPL can express access control policies as well as policies dealing with other systems' aspects, as e.g. resource usage and adaptation. Therefore, once integrated with SCEL, FACPL could be used to regulate interaction and adaptation of components.

Programming support for policies. The development and the enforcement of FACPL policies will be supported by practical software tools: an Integrated Development Environment (IDE), in the form of an Eclipse plugin, and a Java implementation library. The policy designer can use the IDE for writing the desired policies in FACPL syntax, by taking advantage of the supporting features provided, e.g. code-completion and syntax checks. Then, the tool automatically produces a set of Java classes implementing the FACPL code by using the specification classes defined in the FACPL library. The library, given as input a set of Java-translated policies and the request to evaluate, also supplies the request evaluation process according to the rules defining the language's semantics.

Extensions of jRESP with FACPL. We plan to continue the validation of FACPL and its tools. Moreover, we intend to integrate the FACPL evaluation environment within the jRESP runtime environment, thus enabling a full-evaluation of the policy layer when programming ensembles using SCEL. Another research line we intend to pursue is the development of methods and techniques for analysing FACPL policies. In particular, they will be first theoretically defined and, then, integrated in our software tools in order to achieve a complete framework for developing trustworthy policies.

SCEL SDK. We intend to implement an integrated environment for supporting the development of adaptive systems. For this reason we plan to define a *high-level* programming language (HL-SCEL) that, by enriching SCEL with standard programming constructs (e.g. control flow constructs, such as **while** or **if-then-else**, structured data types,...), simplifies the programming task. The SCEL SDK will provide a *compiler* that starting from a HL-SCEL program generates jRESP code. (Semi-)Automatic analysis tools, based on the SCEL's formal semantics, will be also integrated in the environment.

Stochastic extensions of SCEL. We plan to define a timed/stochastic extension of SCEL where *time* is explicitly considered and described by means of *random variables*. In particular, we plan to study possible alternative stochastic semantics for *group oriented operations*. Indeed, different level of abstractions can be considered: from concrete ones, where the underlying protocol governing the component interactions is take into account, to abstract ones, where the underlying communication infrastructure is abstracted. We also plan to study the right modal logic to express *quantitative properties* of SCEL systems.

Adding reasoning capabilities to SCEL. We want to continue developing the methodology that enables SCEL components to take decisions about possible alternative behaviors by choosing among the best possibilities while being aware of the consequences. We shall continue investigating the integration of SCEL with “reasoners” to be invoked by processes when needing to face choices. Also, it may be beneficial to have a methodology for using different reasoners or meta-reasoners designed and optimised for specific purposes. Further, we plan to better investigate the integration between reasoners and policies.

References

[AM11] Musab AlTurki and José Meseguer. Pvesta: A parallel statistical model checking and quantitative analysis tool. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 386–392. Springer, 2011.

[AMS06] Gul A. Agha, José Meseguer, and Koushik Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In Antonio Cerone and Herbert Wiklicky, editors, *QAPL 2005*, volume 153(2) of *ENTCS*, pages 213–239. Elsevier, 2006.

[BCG$^+$12a] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. A conceptual framework for adaptation. In de Lara and Zisman [dLZ12], pages 240–254.

[BCG$^+$12b] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with Maude. In Francisco Durán, editor, *WRLA 2012*, volume 7571 of *LNCS*, pages 118–138. Springer, 2012.

[BDVW] Lenz Belzner, Rocco De Nicola, Andea Vandin, and Martin Wirsing. Reasoning (on) Service Component Ensembles in Rewriting Logic. To appear in the proceedings of SAS 2014, Springer LNCS Festschrift.

[Bel13] Lenz Belzner. Action programming in rewriting logic (technical communication). *Theory and Practice of Logic Programming, On-line Supplement*, 2013.

[BÖ12] Lucian Bentea and Peter Csaba Ölveczky. A probabilistic strategy language for probabilistic rewrite theories and its application to cloud computing. In Narciso Martí-Oliet and Miguel Palomino, editors, *WADT*, volume 7841 of *Lecture Notes in Computer Science*, pages 77–94. Springer, 2012.

[CDE$^+$07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[CNP$^+$13] Luca Cesari, Rocco De Nicola, Rosario Pugliese, Mariachiara Puviani, Francesco Tiezzi, and Franco Zambonelli. Formalising Adaptation Patterns for Autonomic Ensembles. In *FACS*, LNCS. Springer, 2013. To appear.

[DFLP12] Rocco De Nicola, GianLuigi Ferrari, Michele Loreti, and Rosario Pugliese. A Language-based Approach to Autonomic Computing. In *FMCO 2011*, LNCS 7542, pages 25–48. Springer, 2012.

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

[DLPT13] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. SCEL: a language for autonomic computing. Technical report, Univ. Firenze, 2013. `http://rap.dsi.unifi.it/scel/pdf/SCEL-TR.pdf`.

[dLZ12] Juan de Lara and Andrea Zisman, editors. *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7212 of *Lecture Notes in Computer Science*. Springer, 2012.

[EMA$^+$12] Jonas Eckhardt, Tobias Mühlbauer, Musab AlTurki, José Meseguer, and Martin Wirsing. Stable availability under denial of service attacks through formal patterns. In de Lara and Zisman [dLZ12], pages 78–93.

[GLPT12] Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. Modeling adaptation with a tuple-based coordination language. In *SAC*, pages 1522–1527. ACM, 2012.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[Mes12] José Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*, 81(7-8):721–781, 2012.

[MKH$^+$13] Philip Mayer, Annabelle Klarl, Rolf Hennicker, Mariachiara Puviani, Francesco Tiezzi, Rosario Pugliese, Jaroslav Keznikl, and Tomáš Bureš. The Autonomic Cloud: A Vision of Voluntary, Peer-2-Peer Cloud Computing. In *SASO*, pages 1–6. IEEE, 2013. To appear.

[MMPT13a] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. A Formal Software Engineering Approach to Policy-based Access Control. Technical report, DiSIA, Univ. Firenze, 2013. Available at `http://rap.dsi.unifi.it/facpl/research/Facpl-TR.pdf`.

[MMPT13b] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Developing and Enforcing Policies for Access Control, Resource Usage, and Adaptation – A Practical Approach. In *WSFM*, LNCS. Springer, 2013. To appear.

[NIS09] NIST. A survey of access control models, August 2009.

[OAS12] OASIS XACML TC. eXtensible Access Control Markup Language (XACML) version 3.0, September 2012.

[SV] Stefano Sebastio and Andea Vandin. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. Submitted `eprints.imtlucca.it/1798`.

[SVA05a] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2005.

[SVA05b] Koushik Sen, Mahesh Viswanathan, and Gul A. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In Christel Baier, Giovanni Chiola, and Evgenia Smirni, editors, *QEST 2005*, pages 251–252. IEEE Computer Society, 2005.