Programming and Verifying Component Ensembles *

Rocco De Nicola[1], Alberto Lluch Lafuente[1], Michele Loreti[2], Andrea Morichetta[1], Rosario Pugliese[1], Valerio Senni[1], and Francesco Tiezzi[1]

[1] IMT Institute for Advanced Studies Lucca, Italy
[2] Università degli Studi di Firenze, Italy

Abstract. A simplified version of the kernel language SCEL, that we call SCELight, is introduced as a formalism for programming and verifying properties of so-called cyber-physical systems consisting of software-intensive ensembles of components, featuring complex intercommunications and interactions with humans and other systems. In order to validate the amenability of the language for verification purposes, we provide a translation of SCELight specifications into Promela. We test the feasibility of the approach by formally specifying an application scenario, consisting of a collection of components offering a variety of services meeting different quality levels, and by using SPIN to verify that some desired behaviors are guaranteed.

Keywords: Cyber Physical Systems, Component-based Systems, Formal Methods, Process Calculi, Verification, Model Checking.

1 Introduction

Nowadays much attention is devoted to software-intensive cyber-physical systems. These are systems possibly made of massive numbers of components, featuring complex intercommunications and interactions with humans and other systems and operating in open and unpredictable environments thus needing to dynamically adapt to new requirements, technologies and contextual conditions. Such classes of systems include the so-called *ensembles* [1] and *systems of systems* [2], mainly characterized by the idea of assembling or aggregating groups of autonomous components, which may be independently controlled and managed, and whose interaction may be cooperative or competitive.

The design and the analysis that these classes of systems meet the expectations of their users pose big challenges to language designers and software engineers. The problem for language designers is to provide the right set of programming abstractions together with the formal machinery that permits guaranteeing that the expected behavior is exhibited. To deal with the above mentioned challenges, in [3] we have introduced the kernel language SCEL that permits governing the complexity of such systems by providing flexible abstractions, by

* Research supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA.

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 69–83, 2014.

enabling transparent monitoring of the involved entities and by supporting the implementation of self-* mechanisms such as self-adaptation. The key concepts of the language are those of *Behaviors*, *Knowledge*, *Aggregations* and *Policies* that have proved fruitful in modelling autonomic systems from different application domains such as, e.g., collective robotic systems [3,4], service provision and cloud-computing [5,6,7], and cooperative e-vehicles [8].

One of the distinguishing features of SCEL is the use of flexible, *group-oriented*, communication primitives that allows one to implicitly select the set of components to communicate with, by evaluating a given predicate $\mathcal{P}$ used as the target. When a communication action has predicate $\mathcal{P}$ as a target, it will involve all components that satisfy $\mathcal{P}$. For example, if a system contains elements that export attributes such as *serviceProvided* and *QoS* and one would like to program a component willing to interact with all the components that provide a service s and offer a QoS above q, (s)he can use the predicate $serviceProvided = s \wedge QoS > q$ to select the component's partners.

Contribution. This paper presents a first step towards using SCEL and the SPIN model checker [9] for guaranteeing systems properties. For ease of presentation we introduce a simple variant of SCEL that we call SCELight. We provide a translation of SCELight specifications into Promela, that is the input language of SPIN, and show how to exploit it to verify ensemble-based scenarios with SPIN. We test feasibility of the approach by considering an application scenario, borrowed from [5], consisting of a collection of components offering a variety of services meeting different quality levels.

Structure of the paper. The rest of the paper is organized as follows. In the next section, we introduce our application scenario that will be used also to describe the language constructs. In Section 3 we introduce syntax and informal semantics of SCELight, while in Section 4 we describe our translation and its intricacies demanded by the significantly different nature of SCELight and Promela. In Section 5 we show how SPIN can be used to check and verify properties of SCELight specifications, by relying on the translation into Promela of the SCELight specification of the scenario presented in Section 3. Finally, Section 6 concludes by also touching upon directions for future work.

2 A Service Provision Scenario

We consider an application scenario, borrowed from [5], consisting of a collection of components offering a variety of services. Each component manages and elaborates service requests with different requirements, roughly summarized by the following three service quality levels: *gold*, *silver* and *base*. These requirements are defined via a combination of predicates on the hardware configuration and the runtime state of the provider components. For example, the runtime state can give a measure of the number of service requests currently handled locally. Notice that the hardware measure is static while the load estimate is dynamically updated whenever a component receives or completes a service request.

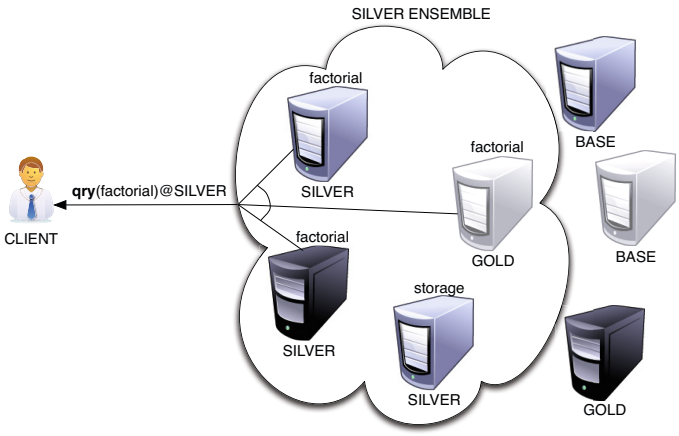

Fig. 1. Group-oriented communication in the service provision scenario

The quality of service, hence, implicitly defines three ensembles, which group together service provider components according to the quality requirements they are able to provide. Clearly, since the quality of service depends on the component state, ensembles are dynamic and components do not need to explicitly migrate from one ensemble to the other: their change of state will implicitly entail their membership to ensembles. The requirements characterizing the three ensembles of service providers are:

- *Gold*: components must have a high level of hardware configuration, i.e. a hardware level greater or equal to 7;
- *Silver*: components must provide a hardware configuration with a level that is at least 4 and, whenever a component provides a hardware level over 7, the computational load must be less than 40%; this latter condition guarantees that gold components can handle requests at silver level only when their computational load is under 40%;
- *Base*: components can have any hardware level, however if they are also gold or silver components then their computational load must be under 20% or 40%, respectively.

We remark that components dynamically and transparently leave or enter an ensemble when their computational load changes. For instance, a *gold* component leaves a *silver* ensemble when its computational load becomes higher than 40%.

Let us now consider a client component willing to submit a request for service *factorial*, which remotely computes the factorial of a natural number. Let us further assume that the client is interested in having the service from a *silver*-quality provider, to ensure the result to be provided within a reasonable amount of time (i.e., on a quite fast, light-loaded server). Before submitting its request, this component interacts with the ensemble of *silver* components searching a provider of the *factorial* service. This search is done by taking advantage of the group-oriented communication (Figure 1), which allows the client to dynamically

identify a component that exposes the service *factorial* at the wanted *silver* service level. If more than one provider component meets these requirements, one of them will be non-deterministically selected. Then, the client posts the actual request to the selected component and waits for the result.

Notice that the application scenario discussed above exploits different forms of communication. First, the invoking client uses group-oriented communication to identify the component that is able to handle specific service request. Then, point-to-point communication is used for client-server interaction.

3 The SCELight Language

SCEL (Software Component Ensemble Language) [3] is a language for programming service computing systems in terms of service components aggregated according to their knowledge and behavioural policies. To enhance flexibility with respect to different application domains, SCEL is parametric with respect to the language for expressing policies, the predicate regulating component interactions, and the notion of knowledge.

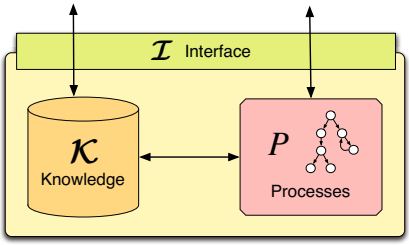

Fig. 2. A SCELight component

For ease of presentation we consider in this work an instantiation of SCEL named SCELight, where no policy language is provided, the interaction predicate interprets the composition of component's processes as a standard interleaving, and knowledge repositories are implemented as multiple distributed tuple-spaces à la Klaim [10]. Moreover, SCELight does not include other sophisticated features of SCEL such as higher-order communication and dynamic creation of new names and components. Last, SCELight includes a specific primitive for atomically updating attribute values, and replaces the non-deterministic choice of SCEL by an ordinary conditional choice. These two standard control flow constructs, that are part of the syntax of Promela, simplify the specification task and can be easily realized in SCEL.

The basic ingredient of SCELight is the notion of (*service*) *component* $\mathcal{I}[\mathcal{K}, P]$, graphically depicted in Figure 2, that consists of:

1. An *interface* $\mathcal{I}$ publishing and making available structural and behavioural information about the component itself in the form of *attributes*, i.e. names acting as references to information stored the component's repository.
2. A *knowledge repository* $\mathcal{K}$ managing both application and awareness data, together with specific handling mechanisms. It stores also the information associated to the interface.
3. A *process* P that can execute local computations, coordinate interaction with the knowledge repository or perform adaptation and reconfiguration.

Table 1. SCELight syntax

DEFINITIONS: $D ::= \emptyset \mid A(\bar{f}) \triangleq P \mid D_1, D_2$		SYSTEMS: $S ::= \mathcal{I}[\mathcal{K}, P] \mid S_1 \parallel S_2$
KNOWLEDGE: $\mathcal{K} ::= \emptyset \mid \langle t \rangle \mid \mathcal{K}_1 \parallel \mathcal{K}_2$	ITEMS: $t ::= e \mid t_1, t_2$	TEMPLATES: $T ::= v \mid x \mid ?\,x \mid T_1, T_2$
PROCESSES: $P ::= \mathbf{nil} \mid a.P \mid \mathbf{if}\ (e)\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \mid P_1 \mid P_2 \mid A(\bar{u})$		TARGETS: $c ::= n \mid \mathcal{P}$
ACTIONS: $a ::= \mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c \mid attr := e$		NAMES: $n ::= i \mid x$

A SCELight SPECIFICATION is a pair $\langle D, S \rangle$ grouping together a set of process DEFINITIONS D and a SYSTEM S. The syntax of definitions and systems is presented in Table 1. A (recursive) process definition has the form $A(\bar{f}) \triangleq P$, with A, $\bar{f}$ and P denoting a process identifier, a list of formal parameters, and a process, respectively. We will use $\bar{u}$ to denote a list of actual parameters. Definitions can be dynamically activated by processes running in system components. We assume that each process identifier has a single definition. Systems aggregate COMPONENTS through the *composition* operator $\_ \parallel \_$.

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 $\_ \parallel \_$. Tuples are knowledge ITEMS consisting of sequences of values. Such values can result from the evaluation of some given *expression* e. We assume that expressions may contain attribute names $attr$, values v (i.e., component identifiers i, strings and integers), and variables x, together with the corresponding standard operators. To pick a tuple out from a tuple-space by means of a given TEMPLATE T (i.e., a sequence of values and variables), 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 ($?\,x$ is used to bind variables to values) and two values match only if they are identical. If more than one tuple match a given template, one of them is arbitrarily chosen.

Processes and Actions. PROCESSES are the active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* ($a.P$), *conditional choice* ($\mathbf{if}\ (e)\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2$), *parallel composition* ($P_1 \mid P_2$), and *parametrized process invocation* ($A(\bar{u})$). Processes can perform four different kinds of ACTIONS. Actions $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$ and $\mathbf{put}(t)@c$ are used to manage shared knowledge repositories by withdrawing/retrieving/adding information items from/to the knowledge repository identified by c. These actions exploit templates T to select knowledge items t from the repositories. They are implemented by invoking

the handling operations provided by the knowledge repository. Action $attr := e$ atomically assigns the value of e to $attr$ and, differently from the other actions, it is not indexed with an address because it always acts locally. Actions **get** and **qry** are blocking and, thus, may cause the process executing them to wait for the wanted element if it is not (yet) available in the knowledge repository. The two actions differ for the fact that **get** removes the retrieved item from the target repository while **qry** leaves the repository unchanged. Actions **put** and := are instead immediately executed.

Different entities may be used as the target c of an action, namely a component name n (in case of *point-to-point* communication) or a *predicate* $\mathcal{P}$ (in case of *group-oriented* communication). In fact, in an action using a predicate $\mathcal{P}$ to indicate the target, the predicate acts as a 'guard' specifying *all* components that may be affected by the execution of the action, i.e. a component must satisfy $\mathcal{P}$ to be the target of the action. Thus, the set of components satisfying a given predicate used as the target of a communication action can be considered as the *ensemble* with which the process performing the action intends to interact. A predicate is a boolean-valued expression obtained by applying standard operators to the results returned by the evaluation of relations between components' attributes and expressions. Notably, an attribute name occurring in a predicate refers to an attribute within the interface of the *object* components (i.e., components that are target of the communication action).

The service provision scenario in SCELight. The application scenario introduced in Section 2 can be formalized in SCELight as the following specification

$$\langle D, \mathcal{I}_{c_1}[\mathcal{K}_{c_1}, P_{c_1}] \parallel \ldots \parallel \mathcal{I}_{c_n}[\mathcal{K}_{c_n}, P_{c_n}] \parallel \mathcal{I}_{p_1}[\mathcal{K}_{p_1}, A_{p_1}] \parallel \ldots \parallel \mathcal{I}_{p_m}[\mathcal{K}_{p_m}, A_{p_m}] \rangle$$

consisting of a composition of n clients $\mathcal{I}_{c_h}[\mathcal{K}_{c_h}, P_{c_h}]$ and m providers $\mathcal{I}_{p_j}[\mathcal{K}_{p_j}, A_{p_j}]$. The latter ones are dynamically organised in ensembles according to requirements expressed in terms of suitable attributes exposed in the components' interfaces. In particular, we assume that attributes named *hw* and *load* are provided by each component. The former can take an integer value from 0 to 10 that gives an indication of the capacity of the hardware configuration of the component, while the latter can take an integer value from 0 to 100 that estimates the actual computational load of the component. The values of such attributes can be dynamically changed through actions $hw := e_1$ and $load := e_2$. Each service component also stores in its knowledge repository a collection of items indicating the available services, together with their component identifier. For example, the provider p_j offering the $factorial$ service stores in its local repository the item $\langle \text{“}service\text{”}, \text{“}factorial\text{”}, i_{p_j} \rangle$. Note that including the identifier in the tuple publishing the service is fundamental as the group-oriented communication primitives are completely anonymous, i.e. the actual objects of a group-oriented communication action are not known to the subject.

The three ensembles of *gold*, *silver* and *base* service providers are characterized by the following predicates:

$$\begin{aligned}
\mathcal{P}_g &\triangleq (\quad hw \geq 7) \\
\mathcal{P}_s &\triangleq (4 \leq hw < 7) \vee (\mathcal{P}_g \wedge load < 40) \\
\mathcal{P}_b &\triangleq (\quad hw < 4) \vee (\mathcal{P}_s \wedge load < 40) \vee (\mathcal{P}_g \wedge load < 20)
\end{aligned}$$

Each client component c_h runs the process P_{c_h}, that takes care of the interaction with the *factorial* service and is of the form

$$\begin{aligned}
&\mathbf{qry}(\text{“}service\text{”}, \text{“}factorial\text{”}, ?x)@\mathcal{P}_k. \\
&\mathbf{put}(\text{“}invoke\text{”}, \text{“}factorial\text{”}, v, i_{c_h})@x. \\
&\mathbf{get}(\text{“}result\text{”}, \text{“}factorial\text{”}, ?y)@i_{c_h}. P'_{c_h}
\end{aligned}$$

for some service level k in $\{b, s, g\}$ and some argument v for the factorial function the client would like the server to execute.

In words, such process first searches, via a **qry** action, among the components belonging to the ensemble identified by predicate $\mathcal{P}_k$, an item matching the template (“*service*”, “*factorial*”, $?x$). In this way, by taking advantage of group-oriented communication, the client is able to dynamically identify a component x that provides the *factorial* service at the desired service level k. Then, via a **put** action, the process invokes the selected service, in a point-to-point fashion, by providing the actual parameter v of the request. After issuing the invocation, the process waits for the result (recall that action **get** is blocking). Whenever the result of the service invocation is made available, the process can withdraw it from the local repository and continue as process P'_{c_h}.

Each server i_{p_j} runs the process A_{p_j} defined in D as:

$$\begin{aligned}
A_{p_j} \triangleq\ & \mathbf{get}(\text{“}invoke\text{”}, \text{“}factorial\text{”}, ?x, ?y)@i_{p_j}. \\
& load := load + 20. \\
& (A_{p_j} \mid Q(x, y))
\end{aligned}$$

The process is triggered by a client request. Whenever this happens, the computational load is updated; we assume that each service instance uses 20% of the sever's capacity. Then, the $factorial$ service becomes again ready to serve other client requests, and the process Q, which actually computes the result of the invoked service for the current request, is executed. We assume that, before its termination, process Q updates the value of attribute *load*, and puts the result of the computation in the repository of the client.

4 Translating SCELight into Promela

In this section we introduce the translation of SCELight specifications into Promela in order to verify ensemble-based scenarios with the model checker SPIN. The translation is formally defined by a family of functions $[\![\cdot]\!]$.

$[\![\langle D, S\rangle]\!]$ =
```
/* The type of the interface as a struct of attributes */
typedef interface{
  int attr_1;
  ...
  int attr_w;
}

/* A component-indexed array of interfaces */
interface I[cNum(S)];

/* Component-indexed array of knowledge repositories */
chan K[cNum(S)] = [capacity] of {int, ..., int}
                                 \_____max(S,D)_____/
int initialized = 0;

/* process definitions */
[[D]]^{max(S,D),cNum(S)-1}

/* Component specifications */
[[S]]^{max(S,D),cNum(S)-1}
```

Fig. 3. Translation of SCELight specifications

Specifications. Given a SCELight specification $\langle D, S\rangle$, function $[\![\cdot]\!]$ in Figure 3 returns a Promela specification containing the declaration of the necessary data structures for representing interfaces, knowledge, components and processes. Data structures representing interfaces and knowledge repositories are declared with a global scope; in this way, attributes and knowledge items can be directly accessed by Promela processes.

Interfaces. The translation declares a structured type `interface` as a collection of (integer) variables, one for each attribute; we assume that all components expose the same set of attributes $\{attr_1, \ldots, attr_w\}$. All interfaces are then recorded in the array `I`, whose size is computed by function $cNum(S)$, which returns the number of components in S.

Repositories. All knowledge repositories are grouped together in the array `K`. Each repository is implemented as a channel of tuples of length $max(S, D)$, which corresponds to the maximum length of items used in the definitions D and system S. To simplify message management in Promela, all tuples have the same length and are composed only of integer values. To fulfil this assumption, messages representing shorter items are completed by using dummy values (see Figure 8), while string values are converted into integers in a pre-processing phase. The dimension of repositories is set by means of the parameter *capacity* (its value depends on the application domain).

Initialization and Process -Definitions. The translation also initializes a counter (`initialized`) used to implement a barrier that guarantees that all processes

$$[\![D_1, D_2]\!]^{m,\ell} = [\![D_1]\!]^{m,\ell}\ \ [\![D_2]\!]^{m,\ell} \qquad [\![S_1 \parallel S_2]\!]^{m,\ell} = [\![S_1]\!]^{m,\ell}\ \ [\![S_2]\!]^{m,\ell}$$

$$[\![A(\bar{f}) \triangleq P]\!]^{m,\ell} = \texttt{proctype A(f) \{ run A\_0(f) \}}\ [\![P]\!]^{m,\ell,i,\bar{\mathtt{f}} \cup var(P)}_{\mathtt{A\_0}}$$

$$[\![\mathcal{I}_i[\mathcal{K}_i, P_i]]\!]^{m,\ell} =$$

```
active proctype c_i {
  atomic {
    /* Attribute initialization */
    I[i].attr_1 = I_i.attr_1; ... I[i].attr_w = I_i.attr_w;

    /* Knowledge repository initialization */
    ∀t ∈ K_i : K[i]![[t]];

    /* Increment initialization counter */
    initialized++;
  }
  /* Start when all components are initialized */
  initialized == ℓ + 1 -> run c_i_0(0̄)
}
```

$[\![P_i]\!]^{m,\ell,i,var(P_i)}_{\mathtt{c\_i\_0}}$

Fig. 4. Translation of definitions and system components

start their execution when all initializations of interface attributes and knowledge repositories is terminated. Finally, an auxiliary function $[\![\cdot]\!]^{m,\ell}$ is used to individually translate the process definitions and the system components. This function is parameterized by the maximum length of items m and the highest component index (ranged from 0 to $cNum(S)-1$) necessary to properly translate SCELight processes in D and S.

Process Definitions. The translation of process definitions and system components is reported in Figure 4. A definition $A(\bar{f}) \triangleq P$ is rendered as a declaration of a Promela process (via the `proctype` construct) with the same name and parameters `A(f)`, and followed by the translation of P. As clarified later, the latter is another process declaration that will be activated by the `run` operator within the body of the process declaration `A`.

Components. The translation of a component $\mathcal{I}_i[\mathcal{K}_i, P_i]$ corresponds again to a process declaration, with name `c_`i, that initializes the data structures modelling the component attributes and its knowledge repositories with values in $\mathcal{I}_i$ and $\mathcal{K}_i$. Notably, differently from all other process definitions, component translations are automatically instantiated in the initial system state (by means of the keyword `active`). Since the repository is modelled as a channel `K`$[i]$, the insertion of (the translation of) an item is performed by means of a *send* operation (`!`). When all initializations are completed, the execution of the translation of P_i, defined immediately after `c_`i, is triggered. Such translation is defined as a function $[\![\cdot]\!]^{m,\ell,i,\bar{\mathtt{x}}}_{\mathsf{p}}$ parameterized, besides by m and ℓ, also by the process index i, the set $\bar{\mathtt{x}}$ of variables used in the SCELight process (identified by functions $var(\cdot)$ and

$$[\![\mathbf{nil}]\!]_{\mathtt{p}}^{m,\ell,i,\bar{\mathtt{x}}} = \texttt{proctype p(}\bar{\mathtt{x}}\texttt{) \{ \}}$$

$$[\![a.P]\!]_{\mathtt{p}}^{m,\ell,i,\bar{\mathtt{x}}} = \texttt{proctype p(}\bar{\mathtt{x}}\texttt{) \{ }[\![a]\!]^{m,\ell}\texttt{;run p0 \}}\ [\![P]\!]_{\mathtt{p0}}^{m,\ell,i,\bar{\mathtt{x}}}$$

$$[\![\mathbf{if}\ (e)\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2]\!]_{\mathtt{p}}^{m,\ell,i,\bar{\mathtt{x}}} = \texttt{proctype p(}\bar{\mathtt{x}}\texttt{) \{ if :: atomic\{ }e\texttt{ -> run pt \} :: atomic\{ else -> run pf \} fi \}}\ [\![P_1]\!]_{\mathtt{pt}}^{m,\ell,i,\bar{\mathtt{x}}}\ [\![P_2]\!]_{\mathtt{pf}}^{m,\ell,i,\bar{\mathtt{x}}}$$

$$[\![P_1 \mid P_2]\!]_{\mathtt{p}}^{m,\ell,i,\bar{\mathtt{x}}} = \texttt{proctype p(}\bar{\mathtt{x}}\texttt{) \{ atomic \{ run pl; run pr \} \}}\ [\![P_1]\!]_{\mathtt{pl}}^{m,\ell,i,\bar{\mathtt{x}}}\ [\![P_2]\!]_{\mathtt{pr}}^{m,\ell,i,\bar{\mathtt{x}}}$$

$$[\![A(\bar{u})]\!]_{\mathtt{p}}^{m,\ell,i,\bar{\mathtt{x}}} = \texttt{proctype p(}\bar{\mathtt{x}}\texttt{) \{ run A(}\bar{\mathtt{u}}\texttt{) \}}$$

Fig. 5. Translation of processes

passed as parameters from a process to another) and the name `p` to be used for the process declaration to generate unique process names. To guarantee the uniqueness of process declaration names, for each component i the names of its declarations are prefixed by `c_i_0` and are built by adding a character for each translated construct: `0` for action prefix, `t` or `f` for conditional choice (depending on the branch), `l` or `r` for parallel composition (depending on the side).

Processes. The translation of processes is reported in Figure 5. Each SCELight process is naturally translated into a Promela process declaration. The base cases are the translations of the empty process **nil** and call $A(\bar{u})$, which consist of an empty declaration and a declaration containing only a run statement (see the translation of definitions in Figure 4), respectively. In case an action prefixing $a.P$, the process declaration contains the translation of a, which models the action execution, while the translation of the continuation P is outside the declaration and is activated only after the termination of the action execution. The translation of the other constructs, namely conditional choice and parallel composition, is similar and straightforwardly relies on the Promela constructs for selection (`if ... fi`) and for the parallel execution of processes (via multiple `run` statements). Both cases use an `atomic` block: in case of conditional choice, it just aims at reducing the complexity of the verification model (by restricting the amount of interleaving), while in case of parallel execution this ensures the simultaneous activation of the parallel processes.

Actions. Translation $[\![\cdot]\!]^{m,\ell,i}$ of actions is defined in Figure 6. It is worth noticing that in most cases `atomic` blocks are used to guarantee atomic execution

$[\![\mathbf{get}(T)@n]\!]^{m,\ell,i}$ = `atomic{ K[`n`]???`$[\![T]\!]^m$ `}`

$[\![\mathbf{get}(T)@\mathcal{P}]\!]^{m,\ell,i}$ =
```
if
  :: atomic {P|0 && K[0]??[[T]^m] -> K[0]???[T]^m }
  ...
  :: atomic {P|ℓ && K[ℓ]??[[T]^m] -> K[ℓ]???[T]^m }
fi
```

$[\![\mathbf{qry}(T)@n]\!]^{m,\ell,i}$ = `atomic{ K[`n`]???<`$[\![T]\!]^m$`> }`

$[\![\mathbf{qry}(T)@\mathcal{P}]\!]^{m,\ell,i}$ =
```
if
  :: atomic {P|0 && K[0]??[[T]^m] -> K[0]???<[T]^m> }
  ...
  :: atomic {P|ℓ && K[ℓ]??[[T]^m] -> K[ℓ]???<[T]^m> }
fi
```

$[\![\mathbf{put}(t)@n]\!]^{m,\ell,i}$ = `K[eval(`n`)]!`$[\![t]\!]$`;`

$[\![\mathbf{put}(t)@\mathcal{P}]\!]^{m,\ell,i}$ =
```
atomic{
  int j=0;
  do
    :: j == ℓ -> break
    :: P|j    -> K[j]![[t]]; j++
    :: else   -> j++
  od
}
```

$[\![attr := e]\!]^{m,\ell,i}$ = `I[`i`].attr = e;`

Fig. 6. Translation of actions

of the actions. We also recall that the FIFO receive operations of Promela on asynchronous channels are `q?m` (remove the first message from channel `q` if it matches `m` and update the variables in `m` accordingly); `q?<m>` (test if the first message on channel `q` matches `m` and update the variables in `m` accordingly); and `q?[m]` (test if the first message on channel `q` matches `m` without side-effects on the variables of `m`). In addition, Promela provides three so-called random receive variants of the previous ones (denoted with `??` in place of `?`), which remove/test the oldest tuple matching the pattern instead of the first one.

A point-to-point action $\mathbf{get}(T)@n$ is basically modeled as a (pattern-matching-based) receive operation (`???`) on the channel `K[eval(`n`)]` corresponding to the knowledge repository of the component identified by n. Note that `q???m` is not the primitive Promela operation `q??m` but an abbreviation defined in Figure 7. The receive operation `q??m` does encode the semantics we need since it removes the

```
i = len(q);
do
  :: q??m -> break
  :: i>0  -> q??m; q!m; i--
od
```

Fig. 7. Abbreviation `q???m`

$$[\![T]\!]^m = [\![T]\!], \underbrace{\_, \ldots, \_}_{m-|T|} \qquad [\![v]\!] = \texttt{v} \qquad [\![x]\!] = \texttt{eval(x)} \qquad [\![?\,x]\!] = \texttt{x;} \qquad [\![T_1, T_2]\!] = [\![T_1]\!], [\![T_2]\!]$$

$$[\![e]\!] = \texttt{e} \qquad [\![t_1, t_2]\!] = [\![t_1]\!], [\![t_2]\!]$$

Fig. 8. Translation of templates and items

oldest tuple in the channel among those matching the template `m` and not *any* of them as required by the semantics of **get**. The abbreviation `q???m` ensures a non-deterministic removal by non-deterministically choosing between (i) removing the oldest matched item and (ii) looping after reinserting the oldest matched item in the queue so that it becomes the newest such item. The latter can be attempted as many times as the size of `q` to ensure termination and it guarantees that all possible messages matching `m` will be considered.

A group-oriented action $\mathbf{get}(T)@\mathcal{P}$ is translated as a non-deterministic choice among a set of input actions on each repository. In particular, for each repository i there is a branch guarded by $\mathcal{P}|_i$ `&& K[`i`]??[`$[\![T]\!]^m$`]` which will ensure the transition to fire only if the target predicate holds for component i and i has a matching item in its repository. If that is the case the item is indeed removed using again the non-deterministic input operation in `K[`i`]???`$[\![T]\!]^m$.

Actions $\mathbf{qry}(T)@n$ and $\mathbf{qry}(T)@\mathcal{P}$ are translated in the same way, except for the use of the non-consuming variant (`???<...>`) of the receive operation, while a point-to-point **put** action is simply translated as a send operation (`!`) on the appropriate channel, while the group-oriented one consists of a loop that sends the tuple to the repositories of all components satisfying the target predicate. The selection statement permits ignoring the components that do not satisfy the predicate (in fact, the **put** action is non-blocking).

Action $attr := e$ is straightforwardly translated as an assignment of expression e to the attribute $attr$ exposed in the interface of the proper component (the latter is identified by the parameter i of the translation function).

Templates and Items. Function $[\![\cdot]\!]^m$ (Figure 8) returns a template of length m by concatenating the translation of the template given as argument with a sequence of so-called *hidden* variables (denoted by "`_`"). The translation functions $[\![T]\!]$ and $[\![t]\!]$ are straightforward. Filling the tuple with dummy values is not needed in the translation of items; this is automatically done by SPIN. It is worth to recall as well that function `eval( · )`, instead, is used for evaluating variables and protecting them from assignments in the matching mechanism.

5 Verification

We illustrate in this section some examples of how SPIN can be used to check and verify properties of SCELight specifications, by resorting to the translation of the SCELight specification of the scenario presented in Section 3.

Checking Deadlock Absence. One first property one would like to check is absence of deadlocks. Obviously not every instance of our scenario is deadlock free. Indeed, if the instance contains clients requiring a service that is not offered by any server or that cannot be served at the required quality level, deadlocks may arise since SCEL input operations have blocking semantics. Notably, the system can have valid terminal states as well, since clients gracefully terminate after successfully receiving the results from servers.

Below, we report an example result of invoking SPIN for checking deadlock absence in an instance of our scenario with 3 servers with different hardware configurations and 5 clients invoking the services offered by the servers:

```
State-vector 1828 byte, depth reached 81, errors: 0
  3849511 states, stored
```

The result is positive (no errors) and SPIN explores a few millions of states.

Checking Server Overload. Another typical use of SPIN that is very convenient for our purposes is to look for interesting executions by characterizing them by means of an LTL formula and asking SPIN for a counterexample. For example, in our scenario, to obtain system runs overloading the server s_i we can specify a formula $\Box\, \mathcal{I}_{s_i}.load \leq 100$, which states that server s_i will never be overloaded.

Indeed, if we check the above invariant in an instance of our scenario with one *gold* server and 6 clients requiring a *gold* service, SPIN returns a counterexample

```
pan:1: assertion violated  !( !((I[0].load<=100))) (at depth 145)
pan: wrote client-server-scenario.pml.trail
```

which consists of an execution of the system, i.e. a trail (stored in the file `client-server-scenario.pml.trail`), in which the server accepts and executes the six requests concurrently, which causes its load to be $6 \times 20\% = 120\%$. One may think that if the clients request a *base* service, it would not be possible to overload the server, as the *gold* server will accept to serve only a few *base* requests concurrently. Actually, this is not true. The reason is that even if a *gold* server will belong to the *base* ensemble only if its load is below 20%, it may be identified as a target by several concurrent clients before actually accepting any service request (and hence updating its load). Indeed, SPIN provides a counterexample also for the above property for a configuration with *gold* server and 6 clients sending *base* requests. Of course, the problem raised by this verification result can be easily fixed by changing the servers specification in order to check the *load* value before accepting additonal requests.

Checking Responsiveness. Finally, we show an example of a typical liveness property expressing the fact that clients are guaranteed responsiveness: whenever a client invokes the factorial service, it will eventually get a result. This can be formalized with the usual LTL formulae of the form $\Box(\mathit{request} \rightarrow \Diamond \mathit{response})$. SPIN provides positive answers for all possible instances of our scenario, since once a client finds an appropriate server for the required service, the server cannot avoid providing the service.

6 Concluding Remarks

We have presented a formal approach to the specification and verification of ensemble-based systems, by providing a translation of SCELight specifications into Promela, the specification language of the SPIN model checker. SCELight is a dialect of the SCEL specification language specifically devised in the EU project Ascens [11] for modelling autonomic, ensemble-based, systems. We have illustrated our approach by verifying a few properties of a service provision scenario. The presented approach enriches the toolset support for SCEL-based engineering of ensemble systems, which currently includes statistical model checking in MiScel [12], the Maude-based SCEL interpreter, and run-time testing with jRESP [13], the Java-based run-time environment for SCEL.

As future work, we plan to continue our programme to verify ensemble-based systems by pursuing different lines of research. The proposed approach will be enhanced by optimizing the generated Promela code to enable a more efficient verification, e.g. by reducing the number of process declarations and invocations, which is actually only required to deal with parallel composition and recursion. Moreover, to foster the practical application of the approach, the SCELight to Promela translation will be implemented in a standard programming language, like Java, by resorting to supporting framework specifically devised for this purpose like Xtext [14]. From a more theoretical perspective, we intend to formally prove that the presented encoding is sound and complete with respect to the operational semantics of SCEL and Promela.

We also plan to extend the work by considering the SCEL constructs not included in SCELight. The main challenge will be to treat the dynamic creation of new names and components for which SPIN does not offer any (efficient) verification support. Some techniques have been proposed to deal with dynamic aspects of software in SPIN (see e.g. [15,16]), but they are not included in the official SPIN distribution. To deal with dynamicity, we plan to investigate the use of other verification tools that provide a better support to these features. We plan also to consider the possibility of using the operational semantics of SCEL as a starting point to generate systems descriptions that can be provided as input to the BIP toolset. The challenge here is understanding if the dynamic part of full SCEL specifications can be "constrained" to provide a full model to be analyzed in BIP and if Dy-BIP [17], the extension of BIP [18] to deal with dynamic architectures, will do a better service.

Finally, another promising line of research that we intend to explore concerns the extension of Promela and BIP, with primitives for group-oriented communication. In fact, on the one hand, the suitability of SCEL to model ensemble-based systems points out the benefits of such form of communication in this application domain. On the other hand, avoiding specification translations would improve efficiency of the verification and, hence, its effectiveness.

References

1. Project InterLink (2007), `http://interlink.ics.forth.gr`

2. Sommerville, I., Cliff, D., Calinescu, R., Keen, J., Kelly, T., Kwiatkowska, M.Z., McDermid, J.A., Paige, R.F.: Large-scale complex IT systems. Commun. ACM 55(7), 71–77 (2012)
3. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: the SCEL Language. ACM Transactions on Autonomous and Adaptive Systems (to appear, 2014), available as Technical Report from `http://eprints.imtlucca.it/2117/`
4. Cesari, L., De Nicola, R., Pugliese, R., Puviani, M., Tiezzi, F., Zambonelli, F.: Formalising Adaptation Patterns for Autonomic Ensembles. In: Proc. of the 10th International Symposium on Formal Aspects of Component Software (FACS 2013). LNCS, Springer, Heidelberg (2014)
5. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: A Language-Based Approach to Autonomic Computing. In: Beckert, B., Damiani, F., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 7542, pp. 25–48. Springer, Heidelberg (2012), `http://rap.dsi.unifi.it/scel/`
6. Margheri, A., Pugliese, R., Tiezzi, F.: Linguistic Abstractions for Programming and Policing Autonomic Computing Systems. In: Proc. of the 10th IEEE International Conference on Autonomic and Trusted Computing (ATC 2013). IEEE Computer Society (2014)
7. Mayer, P., Klarl, A., Hennicker, R., Puviani, M., Tiezzi, F., Pugliese, R., Keznikl, J., Bures, T.: The Autonomic Cloud: A vision of voluntary, peer-2-peer cloud computing. In: Proc. of the 2013 IEEE Seventh International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2013). IEEE Computer Society (2014)
8. Bures, T., De Nicola, R., Gerostathopoulos, I., Hoch, N., Kit, M., Koch, N., Monreale, G., Montanari, U., Pugliese, R., Serbedzija, N., Wirsing, M., Zambonelli, F.: A Life Cycle for the Development of Autonomic Systems: The e-mobility showcase. In: Proc. of the 2013 IEEE Seventh International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2013). IEEE Computer Society (2014)
9. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. 23(5), 279–295 (1997)
10. De Nicola, R., Ferrari, G., Pugliese, R.: Klaim: A Kernel Language for Agents Interaction and Mobility. IEEE Trans. Software Eng. 24(5), 315–330 (1998)
11. ASCENS: Autonomic service-component ensembles, `http://www.ascens-ist.eu/`
12. Belzner, L., De Nicola, R., Vandin, A., Wirsing, M.: Reasoning (on) Service Component Ensembles in Rewriting Logic. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi, SAS 2014 (to appear, April 2014)
13. jRESP, `http://code.google.com/p/jresp/`
14. Xtext, `http://www.eclipse.org/Xtext/`
15. Demartini, C., Iosif, R., Sisto, R.: dSPIN: A Dynamic Extension of SPIN. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 261–276. Springer, Heidelberg (1999)
16. Iosif, R.: Symmetry reductions for model checking of concurrent dynamic software. STTT 6(4), 302–319 (2004)
17. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling Dynamic Architectures Using Dy-BIP. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (eds.) SC 2012. LNCS, vol. 7306, pp. 1–16. Springer, Heidelberg (2012)
18. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)