Stochastically timed predicate-based communication primitives for autonomic computing

Diego Latella
ISTI - CNR

Michele Loreti
Università di Firenze

Mieke Massink
ISTI - CNR

Valerio Senni
IMT Lucca

Predicate-based communication allows components of a system to send messages and requests to ensembles of components that are determined at execution time through the evaluation of a predicate, in a multicast fashion. Predicate-based communication can greatly simplify the programming of autonomous and adaptive systems. We present a stochastically timed extension of the Software Component Ensemble Language (SCEL) that was introduced in previous work. Such an extension raises a number of non-trivial design and formal semantics issues with different options as possible solutions at different levels of abstraction. We discuss four of these options, of which two in more detail. We provide a formal semantics and an illustration of the use of the language modeling a variant of a bike sharing system, together with some preliminary analysis of the system performance.

1 Introduction

The next generation of software-intensive distributed computing systems has to deal with issues that arise from the presence of possibly large numbers of heterogeneous components, featuring complex interactions, and operating in open and non-deterministic environments. A further challenge is to deal with dynamic adaptation as response to evolving requirements and changes in the working environment [13, 14, 19]. Applications with the above characteristics are already being built and can be found in, for example, smart spaces, sensor networks, and large online cloud systems. Devising appropriate abstractions and linguistic primitives together with a consistent programming methodology is essential for the structured and reliable design of these complex systems. One proposal that has been put forward to this aim is the notion of *ensemble*, and in particular that of *autonomic service components* (AC) and *autonomic service-component ensembles* (ACE). The former are autonomic entities whereas the latter are collections of ACs with dedicated knowledge units and resources, and with a goal-oriented execution. Both notions play a central role in the recently developed kernel language SCEL (Software Component Ensemble Language) [5, 8] together with a number of abstractions that are specifically designed for representing behaviours, knowledge, and aggregations according to specific policies, and to support programming context-awareness, self-awareness, and dynamic adaptation. SCEL shares some features with KLAIM [4] but there are two novel key aspects of SCEL, that distinguish it from KLAIM and other languages. They are *predicate-based communication* and the notion of *general component knowledge-base*, and they are specifically designed to support the development of autonomous, loosely-coupled, component-based software systems. Predicate-based communication, allows to send messages to *ensembles* of components that are not predetermined at modeling time, but are defined at execution time, depending on how the communication predicate evaluates w.r.t. the receiver interface. The component knowledge-base provides the realisation of various adaptation patterns, by explicit separation of adaptation data in the spirit of [3], and to model the components view on (and awareness of) the environment. SCEL has been used to specify many scenarios related to the Case Studies of the ASCENS project [9]. These specifications witness how

Submitted to:
QAPL 2014

© D. Latella, M. Loreti, M. Massink & V. Senni
This work is licensed under the
Creative Commons Attribution License.

SCEL primitives, and in particular the property-based interaction paradigm, simplify the programming of autonomous and adaptive systems.

In this paper we address the problem of enriching SCEL with information about action *duration* by providing a stochastic semantics for the language. Such a semantics is important for the analysis of the performance aspects of ensemble based systems. We focus mainly on the issues concerning predicate-based communication, which are definitely more difficult to deal with then those related to the knowledge-bases. There exist various frameworks that support the systematic development of stochastic languages, such as [7]. However, the main challenge in developing a stochastic semantics for SCEL is in making appropriate modeling choices, both taking into account the specific application needs and allowing to manage model complexity and size. The main contribution in this work is the proposal of four variants of STOCS, a Markovian extension of a significant fragment of SCEL, that can be used to support quantitative analysis of adaptive systems composed of *ensembles of cooperating components*. Providing suitable Markovian semantics to predicate based ensemble languages poses a number of design challenges regarding the temporal ordering of multicast and information request actions that differ considerably from traditional process algebras. The four variants adopt the same language syntax of SCEL, or restrictions thereof, but denote different underlying stochastic models, having a different level of granularity. We obtain these variants by modifying labels and relations used to construct the transition systems. Finally, an important aspect in a modeling language concerns the need of devising an appropriate syntax for the environment model. In STOCS and SCEL the point of contact with the environment is the knowledge base, which contains both internal information and externally-sensed events. In our approach, the knowledge is the most appropriate part of the language to specify environment models. In summary, STOCS is essentially a *modeling language* which inherits the purpose and focus of SCEL. STOCS extends SCEL by modeling the average time duration of state-permanence and by replacing non-determinism by a probability distribution over outgoing transitions, thus adopting a CTMC-based operational semantics [6]. In the current phase of the design of STOCS, we deliberately omit to incorporate certain advanced features of SCEL, such as the presence and role of policies. In the same vein, we limited our investigation to a CTMC-based semantics at this stage, leaving for further study variants with a clear separation between stochastically timed actions and observable instantaneous actions, leading to a semantics based on IMCs or CTMDPs [11].

The outline of the paper is as follows. Section 3 discusses the trade-offs between four stochastic variants of SCEL, followed by the presentation of the key aspects of the formal semantics of two of them in Section 4. Section 5 introduces a case study to illustrate various aspects of the use of the design language in the context of a smart bike sharing system. Due to space limitation, in this paper we cannot report all the details of the relevant definitions and examples; they can be found in [15], which the interested reader is referred to.

2 Related work

An overview of related work on a language-based approach to autonomic computing beyond the works cited in the introduction can be found in [8] and the references therein. Due to space limitations, here we briefly mention a selection of directly relevant work that has not yet been addressed in the introduction. To the best of our knowledge there is no work available in the literature that addresses the stochastic extension of predicate based communication in the context of ensemble languages. There are a number of formal core languages that address dynamically changing network topologies in the context of mobile systems. This feature is directly relevant also to ensemble based systems. Examples

are the calculus for wireless systems [16] and the ω-calculus [18], a calculus for mobile ad-hoc networks, which is a conservative extension of the π-calculus and which captures the ability of nodes to broadcast messages to other nodes that are within its physical transmission range. The calculus does not support general predicate-based communication and autonomic aspects of components.

3 STOCS: a Stochastic extension of SCEL

SCEL (Software Component Ensemble Language) [8] is a kernel language that takes a holistic approach to programming autonomic computing systems and aims at providing programmers with a complete set of linguistic abstractions for programming the behavior of Autonomic Components (ACs) and the formation of Autonomic Component Ensables (ACEs), and for controlling the interaction among different ACs. A SCEL program consists of a set of components of the form $I[K, P]$. Each component provides: a *knowledge repository* K, an interface I, and a *process* P.

The *knowledge repository* K manages both *application data* and *awareness data*. Application data is used for enabling the progress of ACs' computations, while awareness data provides information about the environment in which the ACs are running (e.g. monitored data from sensors) or about the status of an AC (e.g. its current location). The definition of SCEL abstracts from a specific implementation of knowledge repository. It is only assumed that there are specific operations for adding *knowledge items* in a repository ($K \oplus t$), for removing elements from a repository ($K \ominus T$), and for inferring elements from a repository ($K \vdash T$). In the sequel we let $\mathbb{V}$ denote the set of values, $\mathbb{K}$ denote the set of possible knowledge states, $\mathbb{I}$ denote the set of knowledge items, $\mathbb{T}$ denote the set of knowledge templates. The latter are used to retrieve data from the knowledge repository. We refer to [5, 8] for a detailed discussion on motivations and role of knowledge repositories in SCEL.

The component *interface* I is used to publish and to make available structural and behavioral information about the component in the form of *attributes*, i.e. names acting as references to information stored in the component's knowledge repository. Let $\mathbb{A}$ be the set of attribute names (which include the constant id used to indicate the component identifier); an interface I is a function in the set $\mathbb{K} \to (\mathbb{A} \to \mathbb{V})$. An interface defines a (partial) function from a pair knowledge-base and attribute-name to the domain of values. Among the possible attributes, id is mandatory and is bound to the name of the component. Component names are not required to be unique, so that replicated service components can be modeled. The *evaluation* of an interface I in a knowledge state K is denoted as $I(K)$. The set of possible interface evaluations is denoted by $\mathbb{E}$.

A *process* P, together with a set of process definitions, 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 ACs to access knowledge in the repositories of other ACs. Processes can perform three different kinds of ACTIONS: $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$ and $\mathbf{put}(t)@c$, used to act over shared knowledge repositories by, respectively, withdrawing, retrieving, and adding information items from/to the knowledge repository identified by c. We restrict targets c to the distinguished variable self, that is used by processes to refer to the component hosting it, and to component *predicates* p, i.e. formulas on component attributes. A component $I[K, P]$ is *identified* by a predicate p if $I(K) \models \mathsf{p}$, that is, the interpretation defined by the evaluation of I in the knowledge state K is a model of the formula p. Note that here we are assuming a fixed interpretation for functions and predicate symbols that are not within the attributes ($\mathbb{A}$). E.g. $battery < 3$ is a possible predicate, where $<$ and 3 have a fixed interpretation, while the value of *battery* depends on the specific component addressed.[1]

[1] For the sake of notational simplicity, in the present paper we assume that predicate p in process actions implicitly refers

SYSTEMS:	S	::=	$C \mid S \parallel S$
COMPONENTS:	C	::=	$I[K,P]$
PROCESSES:	P	::=	$\mathbf{nil} \mid a.P \mid P+P \mid P \mid P \mid X \mid A(\bar{p})$
ACTIONS:	a	::=	$\mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c$
TARGETS:	c	::=	$\mathsf{self} \mid \mathsf{p}$
ENSEMBLE PREDICATES:	p	::=	$tt \mid e \bowtie e \mid \neg\mathsf{p} \mid \mathsf{p} \wedge \mathsf{p}$ with $\bowtie \in \{<,\leq,>,\geq\}$
EXPRESSIONS:	e	::=	$v \mid \mathsf{x} \mid \mathsf{a} \mid \ldots$

Table 1: STOCS syntax (KNOWLEDGE K, TEMPLATES T, and ITEMS t are parameters)

The syntax of STOCS, a Stochastic Extension of SCEL, is presented in Table 1, where the syntactic categories of the language are defined. The basic category defines PROCESSES, used to specify the order in which ACTIONS can be performed. Sets of processes are used to define the behavior of COMPONENTS, that are used to define SYSTEMS. ACTIONS operate on local or remote knowledge-bases and have a TARGET to determine which other components are involved in the action. As we mentioned in the Introduction, for the sake of simplicity, in this version of STOCS we do not include POLICIES, whereas, like SCEL, STOCS is parametric w.r.t. KNOWLEDGE, TEMPLATES and ITEMS.

3.1 From SCEL to STOCS

The semantics of SCEL does not consider any time related aspect of computation. More specifically, the execution of an action of the form $\mathbf{act}(T)@c.P$ (for **put/get/qry** actions) is described by a *single* transition of the underlying SCEL LTS semantics. In the system state reached by such a transition it is guaranteed that the process which executed the action is in its local state P and that the knowledge repositories of all components involved in the action execution have been modified accordingly. In particular, SCEL abstracts from details concerning: (1) when the execution of the action starts, (2) if c is a predicate p, when the possible destination components are required to satisfy p, and (3) when the process executing the action resumes execution (i.e. becomes P).

In the extension of SCEL with an explicit notion of (stochastic) time, the time-related issues mentioned above can be addressed at different levels of abstraction, reflecting different choices of details in modeling SCEL actions. In this section, we discuss and motivate several design choices of STOCS. In order to obtain an underlying CTMC semantics, we model state residence times in a Markovian way. Therefore, whenever we indicate that an action has rate λ, we mean that the duration of the action (or, equivalently, the state residence time before action execution) is modeled by a random variable (RV, in the sequel) with negative exponential distribution having rate λ. Indeed, the actual residence-time depends also on other conflicting actions the process may be engaged in, and the resulting race-condition.

Depending on the degree of detail in modeling these aspects, we have four different semantics: *network-oriented* (NET-OR), *action-oriented* (ACT-OR), *interaction-oriented* (INT-OR), and *activity-oriented* (ACTIV-OR). These semantics have an increasing level of abstraction, facilitating the management of the complexity of the model according to the application of interest. In the remaining part of this section we briefly discuss these four variants of the stochastic semantics and their motivations. In Section 4 we will present the formal definition of the ACT-OR and NET-OR semantics for a substan-

only to the *other* components, excluding the one where the process is in execution.

tial fragment of STOCS[2]. The complete formalization of the four variants can be found in [15]. We selected the ACT-OR and NET-OR semantics for ease of presentation, given their intuitive flavour.

Network-oriented Semantics. This semantics takes into account points (1)–(3) mentioned previously and provides the most detailed modeling among the four semantics, which entails that actions are *non-atomic*. Indeed, they are executed through several intermediate steps, each of which requires appropriate time duration modeling. In particular, **put** actions are realized in two steps: (1) an envelope preparation and shipping (one for each component in the system, other than the source), (2) envelope delivery, with its own delivery time, test of the truth value of the communication predicate, and update of the knowledge-state. Also the actions **get/qry** are realized in two steps: (1) initiation of the item retrieval by a source component by entering in a waiting state, (2) synchronization with a destination component and exchange of the retrieved item. Since actions are not executed atomically, their (partial) execution is interleaved with that of other actions executed in parallel.

Action-oriented Semantics. In this simplified semantics, an action of the form $\mathbf{act}(T)@c$ (for **put/get/qry** actions) is described by a *single* transition and has a state residence time provided by taking into account the *source* component interface, the *target* component interface, the *cost* of retrieved/transmitted knowledge item, and possibly other parameters. For example, this rate may take into account the components locations and different response times of components. Upon item retrieval (by a **get/qry** action), eligible components (in terms of predicate satisfaction and availability of requested item) are in a race for response, with rate assigned component-wise. The underlying stochastic semantics drives the outcome of the race, with appropriate weighting depending on the rates. Even if this semantics does not consider all the realistic aspects of predicate-based communication, it is simpler, it generates a smaller CTMC, and can be used in scenarios where actions average execution time does not depend on the number of components involved in the communication.

Interaction-oriented Semantics. This is based on the action-oriented semantics and distinguishes local and remote actions, by assuming that local actions are executed instantaneously. In some scenarios, local actions happen in a time-scale which is very different (usually much smaller) from that of remote actions. In these situations it is reasonable to consider as instantaneous the execution of local actions, which is the idea we realize in the definition of the INT-OR semantics. As a useful side effect of ignoring the duration of local actions, we obtain more concise models. This approximation can be considered as an approach to reducing multi-scale models to single-scale models. In the latter, the macro-scale of inter-component communication drives the execution of macro-actions. A similar idea is explored for Bio-PEPA models in [10] and used to abstract away from fast reactions in biochemical networks. There, under the so-called Quasi-Steady-State Assumption of the system, it is also defined a form of bisimilarity between the abstract and the concrete model. The assumption we make for defining this semantics is that each remote (stochastically timed) action is followed by a (possibly empty) sequence of local (probabilistic) actions. We ensure this assumption is satisfied by imposing syntactic restrictions on processes. Then, by realizing a form of maximal progress [12] we execute a timed action and all of its subsequent probabilistic actions in a single transition of the STOCS LTS.

Activity-oriented Semantics. This semantics is very abstract and allows to explicitly declare as atomic an entire sequence of actions, by assigning to it an execution rate of the entire sequence. Since the execution of the sequence of actions is atomic, it allows no interleaving of other actions. As an interesting consequence of this, we have a significant reduction in the state-space of the system. This variant of the semantics is motivated by the fact that STOCS only provides primitives for asynchronous communication. Synchronization, if needed, has to be encoded through a protocol using

[2]Due to lack of space, in the present paper we cannot provide all the details for all the variants.

asynchronous communication primitives [17]. Whatever is the adopted semantics among the previous ones (for example, the Action-oriented semantics), the protocol execution for the synchronization action is interleaved with the execution of other actions. This leads to unclear dependencies of protocol execution times from the environment. The Activity-oriented semantics allows us to declare as atomic an entire sequence of actions and to assign a rate to it. More in general, the purpose of this semantics is to have a very high-level abstraction of the interaction mechanisms. This must be handled with care as potentially relevant system behaviors (and interleavings) may be no longer present in the model. Therefore, properties of the model are not necessarily satisfied also by the system.

4 Stochastic Semantics of SCEL

In this section we present a significative fragment of the formal semantics rules for the *action-* and *network-* oriented variants of stochastic SCEL. The interested reader is referred to [15] where all the details of all the abstractions are presented.

In all of the four semantics, interface evaluations are used within the so-called *rate function* $\mathscr{R} : \mathbb{E} \times Act \times \mathbb{E} \to \mathbb{R}_{\geq 0}$, which defines the rates of actions depending on the interface evaluation of the *source* of the action, the action (where *Act* denotes the set of possible actions), and the interface evaluation of the *destination*. For this purpose, interface evaluations will be embedded within the transition labels to exchange information about source/destination components in a synchronization action. The rate function is not fixed but it is a parameter of the language. Considering interface evaluations in the rate functions, together with the executed action, allows us to keep into account, in the computation of actions rates, various aspects depending on the component state such as the position/distance, as well as other time-dependent parameters. We also assume to have a *loss probability function* $f_{\mathsf{err}} : \mathbb{E} \times Act \times \mathbb{E} \to [0,1]$ computing the probability of an error in message delivery.

In the semantics, we distinguish between output actions (those issued by a component) and input actions (those accepted by a component). To simplify the synchronization of input and output actions, we assume input actions are *probabilistic*, and output actions are *stochastic*, therefore their composition is directly performed through a multiplication.

4.1 Preliminaries

Operational semantics of STOCS is given in the FUTSs style [7] and, in particular, using its Rate Transition Systems (RTS) instantiation [6]. We now briefly recall preliminary definitions and notations.

In RTSs a transition is a triple of the form $(P, \alpha, \mathscr{P})$, the first and second components of which are the source state and the transition label, as usual, and the third component $\mathscr{P}$ is the *continuation function* that associates a real non-negative value with each state P'. A non-zero value represents the rate of the exponential distribution characterizing the time needed for the execution of the action represented by α, necessary to reach P' from P via the transition. Whenever $\mathscr{P}(P') = 0$, this means that P' is not reachable from P via α. RTS continuation functions are equipped with a rich set of operations that help to define these functions over sets of processes, components, and systems. Below we show the definition of those functions that we use in this paper, after having recalled some basic notation, and we define them in an abstract way, with respect to a generic set X.

Let $\mathbf{TF}(X, \mathbb{R}_{\geq 0})$ denote the set of *total* functions from X to $\mathbb{R}_{\geq 0}$, and $\mathscr{F}, \mathscr{P}, \mathscr{Q}, \mathscr{S}$ range over it. We define $\mathbf{FTF}(X, \mathbb{R}_{\geq 0})$ as the subset of $\mathbf{TF}(X, \mathbb{R}_{\geq 0})$ containing only functions with *finite support*: function $\mathscr{F}$ is an element of $\mathbf{FTF}(X, \mathbb{R}_{\geq 0})$ if and only if there exists $\{d_1, \ldots, d_m\} \subseteq X$, the *support* of $\mathscr{F}$, such that $\mathscr{F}\, d_i \neq 0$ for $i = 1 \ldots m$ and $\mathscr{F}\, d = 0$ for all $d \in X \setminus \{d_1, \ldots, d_m\}$. We equip $\mathbf{FTF}(X, \mathbb{R}_{\geq 0})$

with the operators defined below. The resulting *algebraic structure* of the set of finite support functions will be crucial for the compositional features of our approach.

Def. 4.1 *Let X be a set, and $d, d_1, \ldots, d_m$ be distinct elements of X, $\gamma_1, \ldots, \gamma_m \in \mathbb{R}_{\geq 0}$, $\bullet : X \times X \to X$ be an injective binary operator, $\mathscr{F}_1$ and $\mathscr{F}_2$ in $\mathbf{FTF}(X, \mathbb{R}_{\geq 0})$:*

1. *$[d_1 \mapsto \gamma_1, \ldots, d_m \mapsto \gamma_m]$ denotes the function associating γ_i to d_i and 0 to the other elements; the 0 constant function in $\mathbf{FTF}(X, \mathbb{R}_{\geq 0})$ is denoted by $[]$;*
2. *Function $+$ is defined as $(\mathscr{F}_1 + \mathscr{F}_2)\, d \;=_{\text{def}}\; (\mathscr{F}_1\, d) + (\mathscr{F}_2\, d)$;*
3. *Function $(\mathscr{F}_1 \bullet \mathscr{F}_2)$ maps terms of the form $d_1 \bullet d_2$ to $(\mathscr{F}_1\, d_1) \cdot (\mathscr{F}_2\, d_2)$ and the other terms to 0;*
4. *The characteristic function $\mathscr{X} : X \to \mathbf{FTF}(X, \mathbb{R}_{\geq 0})$ with $\mathscr{X}\, d \;=_{\text{def}}\; [d \mapsto 1]$.*

Note that all the summations above are over *finite* sets, due to the definition of $\mathbf{FTF}(S, \mathbb{R}_{\geq 0})$.

Def. 4.2 *An A-*labelled Rate Transition System *(RTS) is a tuple $(S, A, \mathbb{R}_{\geq 0}, \rightarrowtail)$ where S and A are countable, non-empty, sets of* states *and transition* labels*, respectively, and $\rightarrowtail \subseteq S \times A \times \mathbf{FTF}(S, \mathbb{R}_{\geq 0})$ is the A-labelled* transition relation*.*

4.2 Knowledge repositories in STOCS

In STOCS, like in SCEL, no specific knowledge repository is defined. A *knowledge repository type* is completely described by a tuple $(\mathbb{K}, \mathbb{I}, \mathbb{T}, \oplus, \ominus, \vdash)$ where $\mathbb{K}$ is the set of possible *knowledge states* (the variables K, K_1, ..., K', ...range over $\mathbb{K}$), $\mathbb{I}$ is the set of *knowledge items* (the variables t, t_1,...,t',...range over $\mathbb{I}$) and $\mathbb{T}$ is the set of *knowledge templates* (the variables T, T_1,..., T',...range over $\mathbb{T}$). Knowledge items have no variable, while knowledge templates have. We assume a partial function $\mathsf{match} : \mathbb{T} \times \mathbb{I} \to \mathsf{Subst}(\mathbb{I})$ (where $\mathsf{Subst}(X)$ is the set of substitutions with range in X) and we denote as $\mathsf{match}(T, t) = \vartheta$ the substitution obtained by matching the pattern T against the item t, if any. By a small abuse of notation, we write $\neg\mathsf{match}(T, t)$ to denote that $\mathsf{match}(T, t)$ is undefined.

The operators $\oplus$, $\ominus$, $\vdash$ are used to add, withdraw, and infer knowledge items to/from knowledge repositories in $\mathbb{K}$, respectively. In this paper, we give a probabilistic interpretation of the above operators. This provides a uniform treatment of all the ingredients of the semantics definition and a simple way for modelling probabilistic aspects of local computations (e.g. occurrence of errors); of course, deterministic behavior of a knowledge repository is readily represented by using Dirac probability distributions. The following are sufficient requirements on knowledge repository operators, for the purposes of the present paper. The operators have the following signature, where $\mathsf{Dist}(X)$ denotes the class of probability distributions on a set X with finite support:

$$\oplus : \mathbb{K} \times \mathbb{I} \to \mathsf{Dist}(\mathbb{K}), \quad \ominus : \mathbb{K} \times \mathbb{T} \hookrightarrow \mathsf{Dist}(\mathbb{K} \times \mathbb{I}), \quad \vdash : \mathbb{K} \times \mathbb{T} \hookrightarrow \mathsf{Dist}(\mathbb{I}).$$

Function $\oplus$ is *total* and defines how a knowledge item can be inserted into a knowledge repository: $K \oplus t = \pi$ is the probability distribution over knowledge states obtained as the effect of adding t. If the item addition operation is modeled in a deterministic way, then the distribution π is a Dirac function.

Function $\ominus$ is *partial* and computes the result of withdrawing a template from a knowledge state as a probability distribution $K \ominus T$ over all pairs $(K, t) \in (\mathbb{K} \times \mathbb{I})$ such that the item t matches the template T. Intuitively, if $K \ominus T = \pi$ and $\pi(K', t) = p$ then, when one tries to remove an item matching template T from K, with probability p item t is obtained and the resulting knowledge state is K'. If a tuple matching template T is not found in K then $K \ominus T$ is undefined, which is indicated by $K \ominus T = \bot$.

Function $\vdash$ is *partial* and computes (similarly to $\ominus$) a probability distribution over the possible knowledge items matching template T that can be inferred from K. Thus, if $K \vdash T = \pi$ and $\pi(t) = p$ then the probability of inferring t when one tries to infer from K a tuple matching T is p. If no tuple matching T can be inferred from K then $K \vdash T$ is undefined, which is indicated by $K \vdash T = \bot$.

4.3 Action-oriented Operational Semantics

This variant of the STOCS operational semantics (called ACT-OR) is defined according to the classical principle adopted for the definition of stochastic variants of Process Algebras: the execution of each action takes time, which is modeled by a RV exponentially distributed according to a rate λ.

In this semantics the rate associated to an action depends on the type of action performed (e.g. **put**, **get** or **qry**), on the knowledge item involved in the action and on the evaluation interfaces of the interacting components according to the rate function $\mathscr{R} : \mathbb{E} \times Act \times \mathbb{E} \to \mathbb{R}_{\geq 0}$ which takes the interface evaluation of the *source*, an action in the set of labels

$$Act = \{\mathbf{put}(t)@c,\ \mathbf{get}(T:t)@c,\ \mathbf{qry}(T:t)@c \mid t \in \mathbb{I} \text{ and } T \in \mathbb{T} \text{ and } c \in \text{TARGET}\}$$

and the interface evaluation of the *destination*, and returns a value in $\mathbb{R}_{\geq 0}$, which is the rate of execution of the given action with counterparts having those interface evaluations. Note that **get/qry** action labels have argument $T:t$ (rather than T as in Table 1) because the *labels* of the **get/qry** transition will contain also the matching/retrieved term t.

4.3.1 Operational semantics of processes

The ACT-OR semantics of STOCS *processes* is the RTS $(Proc, Act_{Proc}, \mathbb{R}_{\geq 0}, \multimap_e)$ where $Proc$ is the set of process terms defined according to the syntax of STOCS given in Table 1. The set Act_{Proc} of labels is defined according to the grammar below (where e' is the evaluation of an interface, $t \in \mathbb{I}$, $T \in \mathbb{T}$, and c is a TARGET) and it is ranged over by $\alpha, \alpha', \ldots$:

$$Act_{Proc} \ ::= \ \overline{\mathbf{put}(t)@c} \ \mid \ \overline{e' : \mathbf{get}(T:t)@c} \ \mid \ \overline{e' : \mathbf{qry}(T:t)@c}$$

The transition relation $\multimap_e \subseteq Proc \times Act_{Proc} \times \mathbf{FTF}(Proc, \mathbb{R}_{\geq 0})$ is the least relation satisfying the rules of Table 2. This relation describes how a process evolves when one of the STOCS actions is executed and is parameterized by an interface evaluation e that is the one associated to the component where the process is running. In the rest of this paper, we will omit the parameter if unnecessary.

Rule (NIL) states that **nil** is the terminated process, since no process is reachable from it via any action. Rules (PUT) and (PUT_B) describe possible transitions of a process of the form $\mathbf{put}(t)@c.P$. The first rule states that $\mathbf{put}(t)@c.P$ evolves with rate λ to P after a transition labeled $\overline{\mathbf{put}(t)@c}$. This rate is computed by using rate function $\mathscr{R}$. The execution of a $\mathbf{put}(t)@c$ action depends on the source component and *all* the other componens in the system, which are involved as potential destinations. Consequently, the execution rate λ can be seen as a function of the action and of the source component (interface evaluation) only; in particular, the action rate *does not* depend on (the interface evaluation of) a specific (destination) component; this is represented by using the symbol _ in the destination argument of $\mathscr{R}$. On the contrary, rule (PUT_B) states that $\mathbf{put}(t)@c.P$ cannot reach any process after a transition with a label that is different from $\overline{\mathbf{put}(t)@c}$.

Rules (GQ), (GQ_B1) and (GQ_B2) are similar and describe the evolution of a process of the form $\mathbf{gq}(T)@c.P$, where $\overline{\mathbf{gq} \in \{\mathbf{get}, \mathbf{qry}\}}$. In this case, a process is reachable from $\mathbf{gq}(T)@c.P$ only after a transition labeled $\overline{\delta : \mathbf{gq}(T:t)@c}$. The latter indicates a request for a knowledge item t matching template T from a component identified by c. Note that, in the case of $\mathbf{gq}(T)@c$ the execution rate depends also on the destination interface evaluation. This is because only one destination will be involved in the completion of the execution of the action. Rules (CHO), (DEF) and (PAR) are standard.

4.3.2 Operational semantics of components and systems

The stochastic behaviour of STOCS *systems* is defined by the RTS $(Sys, Act_{Sys}, \mathbb{R}_{\geq 0}, \to)$ where Sys is the set of system terms defined according to the syntax of STOCS given in Table 1. The set Act_{Sys} of

Inactive process:

$$\frac{}{\mathbf{nil} \xrightarrow{\alpha}_{\curlywedge} []} \text{(NIL)}$$

Actions (where, $\mathbf{gq} \in \{\mathbf{get}, \mathbf{qry}\}$ and c is a TARGET):

$$\frac{\lambda = \mathscr{R}(\sigma, \mathbf{put}(t)@c, \_)}{\mathbf{put}(t)@c\,.\,P \xrightarrow{\overline{\mathbf{put}(t)@c}}_{\curlywedge\sigma} [P \mapsto \lambda]} \text{(PUT)} \qquad \frac{\alpha \neq \overline{\mathbf{put}(t)@c}}{\mathbf{put}(t)@c.P \xrightarrow{\alpha}_{\curlywedge} []} \text{(PUT}_{\text{B}}\text{)}$$

$$\frac{\mathsf{match}(T,t) = \vartheta \quad \lambda = \mathscr{R}(\sigma, \mathbf{gq}(T:t)@c, \delta)}{\mathbf{gq}(T)@c\,.\,P \xrightarrow{\overline{\delta:\mathbf{gq}(T:t)@c}}_{\curlywedge\sigma} [P\vartheta \mapsto \lambda]} \text{(GQ)}$$

$$\frac{\neg\mathsf{match}(T,t)}{\mathbf{gq}(T)@c\,.\,P \xrightarrow{\overline{\_:\mathbf{gq}(T:t)@c}}_{\curlywedge} []} \text{(GQ}_{\text{B1}}\text{)} \qquad \frac{\alpha \neq \overline{\_:\mathbf{gq}(T:t)@c}}{\mathbf{gq}(T)@c\,.\,P \xrightarrow{\alpha}_{\curlywedge} []} \text{(GQ}_{\text{B2}}\text{)}$$

Choice, definition, and parallel composition:

$$\frac{P \xrightarrow{\alpha}_{e} \mathscr{P} \quad Q \xrightarrow{\alpha}_{e} \mathscr{Q}}{P+Q \xrightarrow{\alpha}_{e} \mathscr{P}+\mathscr{Q}} \text{(CHO)} \qquad \frac{A(\overrightarrow{x}) \stackrel{def}{=} P \quad P[\overrightarrow{v}/\overrightarrow{x}] \xrightarrow{\alpha}_{e} \mathscr{P}}{A(\overrightarrow{v}) \xrightarrow{\alpha}_{e} \mathscr{P}} \text{(DEF)}$$

$$\frac{P \xrightarrow{\alpha}_{e} \mathscr{P} \quad Q \xrightarrow{\alpha}_{e} \mathscr{Q}}{P \mid Q \xrightarrow{\alpha}_{e} \mathscr{P} \mid (\mathscr{X}\,Q) + (\mathscr{X}\,P) \mid \mathscr{Q}} \text{(PAR)}$$

Table 2: Operational semantics of STOCS processes (ACT-OR).

labels consists of three groups of labels of the form $\overline{\alpha}$, α and $\overleftrightarrow{\alpha}$ formally defined according to the grammar below (where $\mathbf{gq} \in \{\mathbf{get}, \mathbf{qry}\}$, e' is the evaluation of an interface, $t \in \mathbb{I}$, $T \in \mathbb{T}$, and p is a PREDICATE):

Act_{Sys} ::=	$e' : \mathbf{put}(t)@\mathsf{p}$	$\mid$	$e' : \mathbf{gq}(T:t)@\mathsf{p}$	$\mid$ (input actions)
	$\overline{e' : \mathbf{put}(t)@\mathsf{p}}$	$\mid$	$\overline{e' : \mathbf{gq}(T:t)@\mathsf{p}}$	$\mid$ (output actions)
	$\overleftrightarrow{e' : \mathbf{put}(t)@\mathsf{self}}$	$\mid$	$\overleftrightarrow{e' : \mathbf{gq}(T:t)@c}$	(synchronizations)

whereas $\rightarrow \subseteq Sys \times Act_{Sys} \times \mathbf{FTF}(Sys, \mathbb{R}_{\geq 0})$. Due to limited space, in Table 3 we present only the rules governing the execution of **put** actions at the level of components and systems. The complete definition of the formal semantics can be found in [15].

Rule (C-PUTL) describes the execution of **put** actions operating at self. Let $I[K,P]$ be a component; this rule states that P executes action $\mathbf{put}(t)@\mathsf{self}$ with local interface evaluation $\sigma = I(K)$ and evolves to $\mathscr{P}$, then a local execution of the action can occur and the entire component evolves with label $\overleftrightarrow{\sigma : \mathbf{put}(t)@\mathsf{self}}$ to $I[\pi, \mathscr{P}]$, where $\pi = K \oplus t$ is a probability distribution over the possible knowledge states obtained from K by adding the knowledge item t, while $I[\pi, \mathscr{P}]$ is the function which maps any term of the form $I[K,P]$ to $(\pi K) \cdot (\mathscr{P}P)$ and any other term to 0.

When the target of a **put** is not self but a predicate p, rule (C-PUTO) is used. This rule simply lifts an output **put** action from the process level to the component level and transmits to its counterpart its current interface evaluation σ by including it in the transition label.

The fact that a component *accepts* a **put** is modelled via rules (C-PUTI) and C-PUTIR). The first rule is applied when the component satisfies the predicate p. When the predicate is *not* satisfied the second rule is applied.

$$\frac{\sigma = I(K) \quad P \xrightarrow{\overline{\mathbf{put}(t)@\mathsf{self}}}_{\sigma} \mathscr{P} \quad K \oplus t = \pi}{I[K,P] \xrightarrow{\overleftrightarrow{\sigma:\mathbf{put}(t)@\mathsf{self}}} I[\pi,\mathscr{P}]} \text{ (C-PUTL)}$$

$$\frac{\sigma = I(K) \quad P \xrightarrow{\overline{\mathbf{put}(t)@\mathsf{p}}}_{\sigma} \mathscr{P}}{I[K,P] \xrightarrow{\overline{\sigma:\mathbf{put}(t)@\mathsf{p}}} I[(\mathscr{X}K),\mathscr{P}]} \text{ (C-PUTO)}$$

$$\frac{\delta = I(K) \quad \delta \models \mathsf{p} \quad K \oplus t = \pi \quad p_{\mathsf{err}} = f_{\mathsf{err}}(\sigma, \mathbf{put}(t)@\mathsf{p}, \delta)}{I[K,P] \xrightarrow{\sigma:\mathbf{put}(t)@\mathsf{p}} [I[K,P] \mapsto p_{\mathsf{err}}] + I[\pi,(\mathscr{X}P)] \cdot (1 - p_{\mathsf{err}})} \text{ (C-PUTI)}$$

$$\frac{I(K) \not\models \mathsf{p}}{I[K,P] \xrightarrow{\sigma:\mathbf{put}(t)@\mathsf{p}} [I[K,P] \mapsto 1]} \text{ (C-PUTIR)}$$

Table 3: Operational semantics of STOCS components, **put** rules (ACT-OR).

$$\frac{S_1 \xrightarrow{\overline{\sigma:\mathbf{put}(t)@\mathsf{p}}} \mathscr{S}_1^o \quad S_1 \xrightarrow{\sigma:\mathbf{put}(t)@\mathsf{p}} \mathscr{S}_1^i \quad S_2 \xrightarrow{\overline{\sigma:\mathbf{put}(t)@\mathsf{p}}} \mathscr{S}_2^o \quad S_2 \xrightarrow{\sigma:\mathbf{put}(t)@\mathsf{p}} \mathscr{S}_2^i}{S_1 \parallel S_2 \xrightarrow{\overline{\sigma:\mathbf{put}(t)@\mathsf{p}}} \mathscr{S}_1^o \parallel \mathscr{S}_2^i + \mathscr{S}_1^i \parallel \mathscr{S}_2^o} \text{ (S-PO)}$$

$$\frac{S_1 \xrightarrow{\sigma:\mathbf{put}(t)@\mathsf{p}} \mathscr{S}_1 \quad S_2 \xrightarrow{\sigma:\mathbf{put}(t)@\mathsf{p}} \mathscr{S}_2}{S_1 \parallel S_2 \xrightarrow{\sigma:\mathbf{put}(t)@\mathsf{p}} \mathscr{S}_1 \parallel \mathscr{S}_2} \text{ (S-PI)}$$

Table 4: Operational semantics of STOCS systems, **put** synchronization (ACT-OR).

When predicate p is satisfied by $I(K)$ and $K \oplus t = \pi$, component $I[K,P]$ accepts the **put** action and evolves to $[I[K,P] \mapsto p_{\mathsf{err}}] + I[\pi,(\mathscr{X}P)] \cdot (1 - p_{\mathsf{err}})$. The first term, that is selected with probability p_{err}, models a failure in the action execution, for instance due to a communication error. Value p_{err} is computed by the function f_{err} taking into account the source, the action performed, and the destination. The second term identifies the different configurations the component can reach when knowledge item t is added to the knowledge repository K. Finally, if p is not satisfied by $I(K)$, the component accepts an input **put** action producing no effect. It is worth noting that rules (C-PUTI) and (C-PUTIR) deal with probability only. In fact, the actual rate of the action is the one which will result from system synchronization (Rules (S-PO) and (S-PI) in Table 4) on the basis of the rates settled by the rule (PUT) of Table 2.

Rule (S-PO) ensures that if any subsystem executes an output **put** action (i.e. a $\overline{\sigma : \mathbf{put}(t)@\mathsf{p}}$ labeled transition), the remaining subsystem must execute the corresponding input **put** action (i.e. a $\sigma : \mathbf{put}(t)@\mathsf{p}$ labeled transition); the composed system does not exhibit a synchronization label, but it rather propagates the output $\overline{\sigma : \mathbf{put}(t)@\mathsf{p}}$ to allow further synchronization with all the other components in parallel; in the computation of the final rate it is necessary to consider output on the left sub-system and input on the right as well as the symmetric case.

Rule (S-PI) ensures that all subcomponents of a system synchronize, all together, on a (specific) input **put** action, completing the broadcast communication. Note that each component is constantly enabled on the input label for any **put** action (rules (C-PUTI) and (C-PUTO).

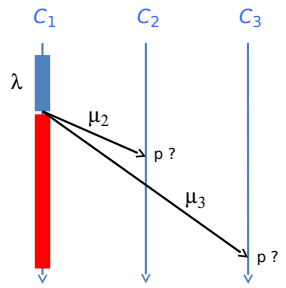

Figure 1: Dynamics of the **put** action.

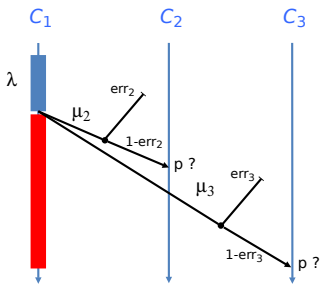

Figure 2: Actual model of **put**.

4.4 Network-oriented Operational Semantics

The Action-oriented operational semantics considered in the previous section completely abstracts from the network structure and topology underlying a given STOCS specification. To make explicit the relevant interactions occurring when one-to-many SCEL communications are performed, we introduce Network-oriented operational semantics (called NET-OR).

Let us consider a process P, of the form $\mathbf{put}(v)@\mathsf{p}\,.\,Q$, and the execution of action $\mathbf{put}(v)@\mathsf{p}$, as illustrated in Figure 1. If we consider all the interactions occurring at the network level, the execution of this action begins with the creation of an envelope message that is shipped, typically in broadcast, to *all* the system components. After this message is shipped, P can proceed behaving like Q. We can assume that the time needed to send this message is exponentially distributed according to a rate λ. When this message is received by a component, the latter first checks if its interface satisfies p, and if so, it delivers t in its knowledge repository. We can assume that the time it takes the envelope message to reach a component is exponentially distributed with rate μ, which may depend on t as well as other parameters like, e.g., the distance between the sender and the receiver component. To model this complex interaction we extend the syntax of processes by adding the new term $\{t@\mathsf{p}\}_\mu$. This term, that is not available at the user syntax level, identifies a pending request for a $\mathbf{put}(t)@\mathsf{p}$; the parameter μ models the rate for transmission, predicate evaluation and repository update as discussed previously. When the request is activated, satisfaction of predicate p is checked and, in the positive case, knowledge item t is added to the local knowledge repository.

Let us consider three components: $C_1 = I_1[K_1, P_1]$, $C_2 = I_2[K_2, P_2]$, and $C_3 = I_3[K_3, P_3]$ and assume process P_1 is defined as $\mathbf{put}(v)@\mathsf{p}\,.\,Q$ as described above. Note that different components may be in different locations. The interaction we illustrate starts with process P_1 executing the first phase of $\mathbf{put}(v)@\mathsf{p}$, i.e. creating two copies of the special message $\{v@\mathsf{p}\}$, one for component C_2 and one for component C_3, and sending these messages. The time required for this phase (denoted in blue in the figure) is modeled by a rate λ. The time for each message to arrive at component C_j ($j = 2,3$), be evaluated against $I_j(K_j)$ and possibly cause the update of K_j is modeled by RVs with rates μ_j (in Figure 1 this is illustrated by two arrows). The delivery of the two messages fails with probability err_2 and err_3, respectively, and succeeds with their complement (see Figure 2). The execution of component C_1 restarts as soon as the copies of the messages are sent, without waiting for their arrival at the destination components (the red stripe in the figure illustrates the resumed execution of C_1). The evaluation of predicate p is performed when the message arrives at the corresponding component so, for example, it may happen that C_2 satisfies p at the time the message arrives (so K_2 is updated accordingly), while C_3 does not satisfy p (thus leaving K_3 unchanged).

The operational semantics of processes already presented in Section 4.3.1 is extended with the

$$\frac{}{\{t@\mathsf{p}\}_\mu \xrightarrow{\overline{\{t@\mathsf{p}\}}} [\mathbf{nil} \mapsto \mu]}\ (\text{ENV}) \qquad \frac{\alpha \neq \overline{\{t@\mathsf{p}\}}}{\{t@\mathsf{p}\}_\mu \xrightarrow{\alpha} []}\ (\text{ENV}_\text{B})$$

Table 5: Operational semantics of STOCS processes, **put** rules (NET-OR).

$$\frac{\delta = I(K) \quad \mu = \mathscr{R}(\sigma, \{t@\mathsf{p}\}, \delta) \quad p_{\mathsf{err}} = f_{\mathsf{err}}(\sigma, \{t@\mathsf{p}\}, \delta)}{I[K,P] \xrightarrow{\sigma:\mathbf{put}(t)@\mathsf{p}} [I[K,P] \mapsto p_{\mathsf{err}},\ I[K,P|\{t@\mathsf{p}\}_\mu] \mapsto (1-p_{\mathsf{err}})]}\ (\text{C-PUTI})$$

$$\frac{P \xrightarrow{\overline{\{t@\mathsf{p}\}}} \mathscr{P} \quad I(K) \models \mathsf{p} \quad K \oplus t = \pi}{I[K,P] \xrightarrow{\overline{\{t@\mathsf{p}\}}} I[\pi, \mathscr{P}]}\ (\text{C-ENVA}) \qquad \frac{P \xrightarrow{\overline{\{t@\mathsf{p}\}}} \mathscr{P} \quad I(K) \not\models \mathsf{p}}{I[K,P] \xrightarrow{\overline{\{t@\mathsf{p}\}}} I[(\mathscr{X}K), \mathscr{P}]}\ (\text{C-ENVR})$$

Table 6: Operational semantics of STOCS components, **put** rules (NET-OR).

rules in Table 5. These rules operate in combination with the new rules for components reported in Table 6. Rule (C-PUTI) models the *initiation* of the execution of action $\mathbf{put}(t)@c$; it allows the reception of a **put** action, and it is responsible for the creation of the envelope (carrying the incoming message) thus modeling its travel towards that component parametrized by rate μ. The fact that the envelope is in parallel with the process of the potential receiver component by no means should be interpreted as the representation of the fact that the message reached the component; simply, the association between the message and the component is represented by means of a parallel composition term; in other words, the fact that a specific message is 'addressed' to a component is represented syntactically by such a parallel composition; this action will be executed with rate λ, computed using the function $\mathscr{R}$ depending on the interface evaluation of the source σ (i.e. the container component) and the sent item t. This is postulated by the rule (PUT) and realized at system level by the broadcast rules of Table 4 that are part of the (NET-OR) semantics of the **put** operation. Rule (C-ENVA)/(C-ENVR) realizes envelope delivery by specifying the conditions under which a component *accepts* or *refuses*, respectively, an arriving envelope.

5 Case Study

We develop a model of a *bike sharing service*, where we assume a city with m *parking stations*, each one with his *location* $\ell_i \in Loc = \{\ell_1, \ldots, \ell_m\}$, a number of *available bikes* b_i, and a number of *available parking slots* s_i (for $i = 1, \ldots, m$). Parking stations are in one-to-one correspondence with the set of possible locations, which should be considered as (disjoint) areas of influence in the city. We also assume to have n *users* of the bike sharing service: at any time, each user is positioned in *one* location and can be in one of the two states *Pedestrian* and *Biker*. In each of the two states, the user moves around the city (with speed depending on the state) according to its preferences, modeled by two *probability* transition matrices Q_b and Q_p of size $m \times m$ for the biker and the pedestrian state, respectively. Then, the user becomes a *Biker* or a *Pedestrian* by transitions named *Borrow* and *Return*.

A user in a location ℓ can borrow (or return) a bike by issuing a request (e.g. by means of a mobile phone application) to the bike sharing system for a parking with an available bike (or slot) within a neighborhood of ℓ. The bike sharing system answers with the location of a parking station having an available bike or an available parking slot, within a neighborhood of ℓ specified by a neighborhood condition $\varphi_n(\ell, \ell')$ (modeling, for example, that a parking station ℓ' is easily reachable from ℓ). This flexibility allows some control on which parking station is selected among those that

$P_u \triangleq$ *Pedestrian*

Pedestrian $\triangleq$	**get**(p_next, L)@self. *Borrow*	*Biker* $\triangleq$	**get**(b_next, L)@self. *Return*
Borrow $\triangleq$	**qry**(loc, L)@self. **get**(bike_res, ID)@near(L). **put**(go, ID)@self. **get**(bike)@loc(ID). **put**(b)@self. *Biker*	*Return* $\triangleq$	**qry**(loc, L)@self. **get**(slot_res, ID)@near(L). **put**(go, ID)@self. **put**(bike)@loc(ID). **put**(p)@self. *Pedestrian*

Figure 3: User behavior as a STOCS process.

are in the neighborhood of the current user location (including itself), which can be used to re-balance slot/bike availability by redirecting users to parking station that have many available bikes (or slots). Re-balancing performed by users can help reducing the cost of bike reallocation by means of trucks.

A single user is represented as a component $I_u[K_u, P_u]$, whose knowledge state K_u is an element $\langle s, \ell \rangle$ in $\{\mathtt{b}, \mathtt{p}\} \times Loc$ denoting the user state (i.e. either being a pedestrian or a biker) and the user location, and whose interface I_u, which defines the predicates biker, pedestrian, and $\mathsf{loc}(\ell)$ as follows: $I_u(\langle \mathtt{b}, \ell \rangle) \models \mathsf{biker}$, $I_u(\langle \mathtt{p}, \ell \rangle) \models \mathsf{pedestrian}$, and $I_u(\langle s, \ell \rangle) \models \mathsf{loc}(\ell)$, for every $\ell \in Loc$ and $s \in \{\mathtt{b}, \mathtt{p}\}$. Let us summarize the role of the user knowledge operators (see [15] for details). The $\oplus$ operator allows: to change state by $\oplus(\mathtt{b})$ (change to biker state) and by $\oplus(\mathtt{p})$ (change to pedestrian state), and to move to a specified location ℓ' by $\oplus(\mathtt{go}, \ell')$. The $\ominus$ operator allows to move to a location according to the average user behavior in the pedestrian state, by $\ominus(\mathtt{p\_next}, L)$, and in the biker state, by $\ominus(\mathtt{b\_next}, L)$. Finally, the $\vdash$ operator allows to retrieve the current user location.

The users behaviour is given in Figure 3. Each user starts in the state *Pedestrian*, where movement is possible through a local **get** of the item p_next. The effect of this action is to change user location into ℓ_j. The latter is also returned as a binding for the variable L. This information will be used to compute the rate of the action (i.e. of the movement) by a suitable rate function $\mathscr{R}$[3]. The process *Borrow* first retrieves the current location L then performs a *bike* reservation (bike_res) from a parking station *ID* satisfying predicate $\mathsf{near}(L)$. The actual rate of this action *depends on available bikes*: the higher is the number of available bikes, the higher is the execution rate. As an effect of this race condition, the *ID* of the near station containing *more bikes* is received by the user with a *higher probability* than a near station with *fewer bikes*, causing a more balanced distribution of bikes in the system. When the parking lot is reserved, the user moves towards the parking stations. The rate of this action depends on the distance between the user and the parking station. After that process *Borrow* takes a bike; this operation is performed via a **get** action that decrements the bikes available and increments the slots available. Finally, the user status is updated to biker b. A biker moves around the city and, then, leaves his bike in a parking station by executing the process *Return*. Its behavior is similar to that of a pedestrian, except for the fact it reserves *parking slots* instead of bikes.

A parking station is represented as a component $I_p[K_p, \mathbf{nil}]$ that has no behavior (it is passive). Its knowledge state is a vector $\langle b_a, b_r, s_a, s_r, \ell \rangle \in \mathbb{N}^4 \times Loc$ denoting the number of available bikes (b_a), of reserved bikes (b_r), of available parking slots (s_a), and of reserved parking slots (s_r), as well as the parking location ℓ. The parking station interface I_p defines the predicates $\mathsf{loc}(\ell)$ and $\mathsf{near}(\ell)$ as follows [15]: $I_p(\langle b_a, b_r, s_a, s_r, \ell \rangle) \models \mathsf{loc}(\ell)$ and $I_p(\langle b_a, b_r, s_a, s_r, \ell \rangle) \models \mathsf{near}(\ell')$ if $\varphi_n(\ell, \ell')$ holds, for every $\ell, \ell' \in Loc$ and $b_a, b_r, s_a, s_r \in \mathbb{N}$.An initial state of this model is a term

$$\|_{i=1}^{m} \ (\ (I_u[\langle \ell_i, \mathtt{p} \rangle, P_u])[k_i] \ \| \ I_p[\langle b_i, 0, s_i, 0, \ell_i \rangle, \mathbf{nil}] \ )$$

which denotes, for $i = 1, \ldots, m$: (i) k_i pedestrians in locations ℓ_i, and (ii) b_i available bikes and s_i

[3] Due to space limitation, we leave out the definition of $\mathscr{R}$, which can be found in [15].

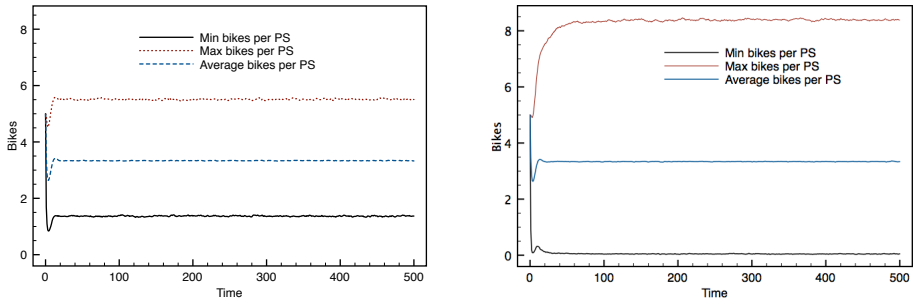

Figure 4: Simulation of bike sharing service.

available parking slots in parking station at location ℓ_i. Note that the number of reserved bikes as well as the number of reserved slots is set to zero at the initial state of the system in every parking station. The overall number of bikes in the system is preserved by the knowledge-update rules. Some simple simulation analyses of the considered system are reported in Figure 4. These simulations, based on action oriented semantics, have been performed with jRESP[4]. This is a Java framework that can be used to execute and simulate SCEL/STOCS specifications. Figure 4 compares the simulation results of the considered case study where rates of bikes and slots reservations depend on the number of available resources (on the left) with the one where these rates are constant[5] (on the right). We can notice that the average number of available bikes in parking stations is similar in the two simulations. However, when the bikes/slots reservation rate depends on the available resources, the bikes are more evenly distributed over the different parking stations.

6 Conclusions and Future Work

We have introduced STOCS, a stochastic extension of SCEL, for the modeling and analysis of performance aspects of ensemble based autonomous systems. One of the original features of the language is the use of stochastic predicate based multi-cast communication which poses particular challenges concerning stochastically timed semantics. Four variants of the semantics, considering different levels of abstraction, have been discussed and for two of them the main aspects of the formal semantics have been provided. A case study concerning shared bikes systems was presented to illustrate the use of the various language primitives of STOCS. The development of both numeric and statistical model-checking tools for STOCS is in progress. In particular, (ACT-OR) and (INT-OR) semantics are well suited for formal analysis techniques (e.g. probabilistic model-checking), while the more detailed and complex (NET-OR) can be used for simulation-based techniques (e.g. statistical model-checking). The formal relationship between the different semantics is a non trivial issue and it is left for future work, as well as the development of fluid semantics and verification techniques to address large scale collective systems along the lines of work in [1, 2].

7 Acknowledgments

The research presented in this report has been partially funded by the EU projects ASCENS (nr.257414) and QUANTICOL (nr.600708), and by the Italian MIUR PRIN project CINA (2010LHT4KM).

[4] `http://jresp.sourceforge.org`

[5] We consider 40 users moving over a grid of 4×4 locations. Each parking station starts with 5 bikes and 5 empty slots.

References

[1] Luca Bortolussi & Jane Hillston (2012): *Fluid Model Checking*. In Maciej Koutny & Irek Ulidowski, editors: *CONCUR*, *Lecture Notes in Computer Science* 7454, Springer, pp. 333–347. Available at http://dx.doi.org/10.1007/978-3-642-32940-1_24.

[2] Luca Bortolussi, Jane Hillston, Diego Latella & Mieke Massink (2013): *Continuous approximation of collective system behaviour: A tutorial*. *Perform. Eval.* 70(5), pp. 317–349. Available at http://dx.doi.org/10.1016/j.peva.2013.01.001.

[3] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente & Andrea Vandin (2012): *A Conceptual Framework for Adaptation*. In Juan de Lara & Andrea Zisman, editors: *FASE*, *Lecture Notes in Computer Science* 7212, Springer, pp. 240–254. Available at http://dx.doi.org/10.1007/978-3-642-28872-2_17.

[4] R. De Nicola, G. Ferrari & R. Pugliese (1998): *KLAIM: A Kernel Language for Agents Interaction and Mobility*. *IEEE Transactions on Software Engineering* 24(5), pp. 315–329.

[5] Rocco De Nicola, Gian Luigi Ferrari, Michele Loreti & Rosario Pugliese (2011): *A Language-Based Approach to Autonomic Computing*. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer & Marcello M. Bonsangue, editors: *FMCO*, *Lecture Notes in Computer Science* 7542, Springer, pp. 25–48. Available at http://dx.doi.org/10.1007/978-3-642-35887-6_2.

[6] Rocco De Nicola, Diego Latella, Michele Loreti & Mieke Massink (2009): *Rate-Based Transition Systems for Stochastic Process Calculi*. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas & Wolfgang Thomas, editors: *ICALP (2)*, *Lecture Notes in Computer Science* 5556, Springer, pp. 435–446. Available at http://dx.doi.org/10.1007/978-3-642-02930-1_36.

[7] Rocco De Nicola, Diego Latella, Michele Loreti & Mieke Massink (2013): *A Uniform Definition of Stochastic Process Calculi*. *ACM Comput. Surv.* 46(1), pp. 5:1–5:35, doi:10.1145/2522968.2522973. Available at http://doi.acm.org/10.1145/2522968.2522973.

[8] Rocco De Nicola, Michele Loreti, Rosario Pugliese & Francesco Tiezzi (2013): *A formal approach to autonomic systems programming: The SCEL Language*. *ACM Transactions on Autonomous and Adaptive Systems* To Appear. Available at http://rap.dsi.unifi.it/scel/.

[9] ASCENS: Autonomic Service component Ensembles: *http://ascens-ist.eu*.

[10] Vashti Galpin, Jane Hillston & Federica Ciocchetta (2011): *A semi-quantitative equivalence for abstracting from fast reactions*. In Ion Petre & Erik P. de Vink, editors: *CompMod*, *EPTCS* 67, pp. 34–49. Available at http://dx.doi.org/10.4204/EPTCS.67.5.

[11] Holger Hermanns & Sven Johr (2007): *Uniformity by Construction in the Analysis of Nondeterministic Stochastic Systems*. In: *DSN*, IEEE Computer Society, pp. 718–728. Available at http://doi.ieeecomputersociety.org/10.1109/DSN.2007.96.

[12] Holger Hermanns & Joost-Pieter Katoen (2009): *The How and Why of Interactive Markov Chains*. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede & Michael Leuschel, editors: *FMCO*, *Lecture Notes in Computer Science* 6286, Springer, pp. 311–337. Available at http://dx.doi.org/10.1007/978-3-642-17071-3_16.

[13] Matthias Hölzl, Axel Rauschmayer & Martin Wirsing (2008): *Software Engineering for Ensembles*. In: *Software-Intensive Systems and New Computing Paradigms*, Springer, pp. 45–63.

[14] Project InterLink (2007): *http://interlink.ics.forth.gr/central.aspx*.

[15] Diego Latella, Michele Loreti, Mieke Massink & Valerio Senni (2014): *Stochastically timed predicate-based communication primitives for autonomic computing - Full Paper*. Technical Report TR-QC-03-2014, QUANTICOL Project. http://www.quanticol.eu/.

[16] Nicola Mezzetti & Davide Sangiorgi (2006): *Towards a Calculus For Wireless Systems*. *Electr. Notes Theor. Comput. Sci.* 158, pp. 331–353. Available at http://dx.doi.org/10.1016/j.entcs.2006.04.017.

[17] Catuscia Palamidessi (2003): *Comparing The Expressive Power Of The Synchronous And Asynchronous Pi-Calculi*. *Mathematical Structures in Computer Science* 13(5), pp. 685–719. Available at `http://dx.doi.org/10.1017/S0960129503004043`.

[18] Anu Singh, C. R. Ramakrishnan & Scott A. Smolka (2010): *A process calculus for Mobile Ad Hoc Networks*. *Sci. Comput. Program.* 75(6), pp. 440–469. Available at `http://dx.doi.org/10.1016/j.scico.2009.07.008`.

[19] Roy Want, Eve Schooler, Lenka Jelinek, Jaeyeon Jung, Dan Dahle & Uttam Sengupta (2010): *Ensemble Computing: Opportunities And Challenges*. *Intel Technology Journal* 14(1), pp. 118–141. Available at `http://doi.acm.org/10.1145/1592761.1592785`.