On StocS: a Stochastic extension of SCEL ⋆

Diego Latella[1], Michele Loreti[2], Mieke Massink[1], and Valerio Senni[3]

[1] Istituto di Scienza e Tecnologie dell'Informazione 'A. Faedo', CNR, Italy
[2] Università di Firenze, Italy
[3] IMT-Lucca, Italy

Abstract. 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 allows for quantitative modelling and analysis of system behaviour (e.g. performance) but rises a number of non-trivial design and formal semantics issues with different options as possible solutions at different levels of abstraction.

1 Introduction

SCEL (Software Component Ensemble Language) [5, 8], is a kernel language that is equipped with programming abstractions for the specification of system models within the framework of the *autonomic computing* paradigm, and for programming such systems. These abstractions are specifically designed for representing behaviours, knowledge, and aggregations according to specific policies, and to support programming context-awareness, self-awareness, and adaptation.

The main focus of the SCEL language is on supporting the development of autonomous, loosely-coupled, component-based software systems. For this purpose, a number of underlying assumptions are made on the kind of peculiarities of these software systems, among which adaptivity, open-endedness, ensemble-orientedness, high ability of reconfiguration, and support for heterogeneity. Two novel key aspects of SCEL, that distinguish it from other languages, are designed to support these peculiarities: predicate-based communication and the role of the component knowledge-base. 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 destination interface. The component knowledge-base allows to realise various adaptation patterns, by explicit separation of adaptation data in the spirit of [3], and to model components view

⋆ This research has been partially funded by the EU projects ASCENS (nr. 257414) and QUANTICOL (nr. 600708), and the IT MIUR project CINA.

on (and awareness of) the environment. SCEL has been developed in the EU ASCENS project[4] and it has been used to specify many scenarios related to the project case studies [12, 10, 14, 13]. These specifications witness how SCEL primitives simplify the programming of autonomous and adaptive systems.

In [11] we addressed the problem of enriching SCEL with information about action durations, which results in a stochastic semantics for the language. In fact, our goal is to provide a formal, language based, framework for quantitative (e.g. performance) modelling and analysis of autonomic computing systems. Even if there exist various stochastic process languages, including some which incorporate notions of spatial distribution (see [4, 9] and references therein) and frameworks that support the systematic development of stochastic languages (see [7] and references therein), 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. Our contribution in [11] was the proposal of four variants of StocS, a Markovian extension of a significant fragment of SCEL. These variants adopt *the same syntax* of SCEL but denote different underlying stochastic models, having a different level of granularity.

StocS, and its support framework extend SCEL by providing the system modeller with means for characterising relevant delays—related to the execution of SCEL actions—modelling them as random variables (RVs) with negative exponential distributions. The resulting models are continuous time Markov chains (CTMCs).

In the design of StocS, we deliberately omit to incorporate certain advanced features of SCEL, such as the presence and role of policies.

In this book we focus on the *network oriented* variant of StocS briefly introduced in [11]. The semantics of this variant entails that actions are *non-atomic*. Indeed, they are executed through several intermediate steps, each of which requires appropriate time duration.

The work we present in this book is only the latest step of a long journey started more than ten years ago with the AGILE EU project, and carried on first within the SENSORIA EU project and later within the ASCENS EU project. The collaboration with Martin Wirsing, who acted as Coordinator of all these projects, gave us the possibility to study the specific formal tools to use for providing stochastic semantics of domain specific languages as well as to investigate general issues concerning such tools. We would like to thank Martin for the many stimulating discussions we had and for his excellent coordination work. Thank you Martin.

The outline of this chapter is as follows. Section 2 discusses the intuitions behind the stochastic extension of SCEL which is presented in 4 after some preliminary definitions are recalled in Section 3. In Section 5 we present a simple case study to illustrate the use of the various language primitives of StocS.

Concluding remarks and lines for possible future research are presented in Section 6.

[4] http://www.ascens-ist.eu/

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)

2 StocS: a Stochastic extension of SCEL

In this section we present the main features of STOCS. We start by illustrating its main syntactic ingredients. Then, we discuss the stochastically timed semantics we present in this chapter.

2.1 Syntax

The syntax of STOCS is presented in Table 1. The basic category defines PROCESSES that are used to specify the order in which ACTIONS can be performed. Sets of processes are used to define the behavior of COMPONENTS, that in turn 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.

We define the following domains for variables and for defining functions signature: $\mathbb{A}$ is the a of attribute names (which include the constant id used to indicate the component identifier), $\mathbb{V}$ is a set of values, $\mathbb{K}$ is a set of possible knowledge states, $\mathbb{I}$ is a set of knowledge items, $\mathbb{T}$ is a set of knowledge templates. So, in Table 1, $\mathsf{a} \in \mathbb{A}$, $v \in \mathbb{V}$, $K \in \mathbb{K}$, $t \in \mathbb{I}$, $T \in \mathbb{T}$.

Example 1 (Items and Templates as Tuples and Patterns). Consider a signature $(\mathcal{V}, \mathcal{F})$ where $\mathcal{V}$ is a set of variables and $\mathcal{F}$ is a set of function symbols with arity (we indicate by f/n a function symbol f with arity n) such that $\langle\rangle/i \in \mathcal{F}$ for $i = 0, 1, 2, ...$ We denote by $Terms(\mathcal{V}, \mathcal{F})$ the set of all possible finite terms on the given signature (i.e. the terms with variables, constructed respecting function symbols arities) and by $Terms(\mathcal{F})$ the set of all possible finite ground terms. A *pattern* is a term of the form $\langle t_1, \ldots, t_n\rangle$, with $t_i \in Terms(\mathcal{V}, \mathcal{F})$ for $i = 1, \ldots, n$. A *tuple* is a term of the form $\langle t_1, \ldots, t_n\rangle$, with $t_i \in Terms(\mathcal{F})$ for $i = 1, \ldots, n$. In this example we have defined the set of Templates $\mathbb{T}$ as the set of patterns and the set of Items $\mathbb{I}$ as the set of tuples.

Systems and components We let *Sys*, ranged over by S, S_1,..., S'... denote the set of systems defined by the syntax in Table 1. A system S consists of an aggregation of COMPONENTS obtained via the (parallel) *composition* operator $_ \parallel _$. A component $I\,[\,K,\,P\,]$ consists of:

1. An *interface*, which is a function $I : \mathbb{K} \to (\mathbb{A} \to \mathbb{V})$ used for publishing information about the component's state in the form of attribute 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 modelled. 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 $\mathbb{E}$.
2. A *knowledge repository* K, managing both application data and awareness data (following the approach of [3]), together with the specific handling mechanism.
3. A *process* P, together with a set of process definitions. Processes may execute local computations, coordinate local and remote interaction with a knowledge repository, or perform adaptation and reconfiguration.

Processes PROCESSES are the active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* $(a.P)$, *nondeterministic choice* $(P_1 + P_2)$, *parallel composition* $(P_1|P_2)$, *process variable* (X), and *parameterised process invocation* $(A(\bar{p}))$. We feel free to omit trailing occurrences of **nil**, writing e.g. a instead of $a.\mathbf{nil}$, whenever there is no confusion arising. Process variables can be used in templates so that processes can also be stored in / retrieved from knowledge repositories.

We assume that A ranges over a set of parameterised *process identifiers* that are used in recursive process definitions. We also assume that each process identifier A has a *single* definition of the form $A(\bar{f}) \triangleq P$ where all free variables in P are contained in $\bar{f}$ and all occurrences of process identifiers in P are within the scope of an action prefixing. $\bar{p}$ and $\bar{f}$ denote lists of actual and formal parameters, respectively. In the sequel we will use *Proc* to denote the set of processes, ranged over by variables P, Q,..., P_1,Q_1..., P',Q',....

Actions and targets 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.

These actions exploit templates T as patterns to select knowledge items t in the repositories. The precise syntax of templates and knowledge items depends on the specific instance of *knowledge repository* that is used. Indeed, in Example 1 we provided the syntax for items ($\mathbb{I}$) and templates ($\mathbb{T}$) for one possible instance of the repository. In the next section we show how STOCS is in fact parametric with respect to different types of *knowledge repository*.

For the sake of simplicity, in this book 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.

The informal, abstract, semantics of the actions is the following:

- $\mathbf{put}(t)@c$ is non-blocking, its execution causes knowledge item t be added to the knowledge repository of *all* the components (the interface of which is) identified by c, *if any*;
- $\mathbf{get}(T)@c$ ($\mathbf{qry}(T)@c$, respectively) is blocking, it causes a knowledge item t matching pattern T be withdrawn (retrieved, respectively) from the knowledge repository of *any of* the components (the interface of which is) identified by c, *if any*. If no such component/item is available, the process executing it is *blocked* in a waiting state. The two actions differ for the fact that **get** removes the requested item from the knowledge repository while **qry** leaves the target repository unchanged.

The set of components satisfying a given target c of a communication action can be considered as the *ensemble* with which the process performing the action intends to interact.

Knowledge behavior Since StocS is parametric w.r.t. the specific knowledge repository used in a specification, we provide no specific syntax/semantics for knowledge repositories. We only require that 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, $\ldots$, K', $\ldots$ 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 to have 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. These functions have the following signature, where $\mathsf{Dist}(X)$ denotes the class of probability distributions on set X with finite support:

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

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 modelled in a deterministic way, then the distribution π is a Dirac function. One advantage of allowing a probabilistic item addition operation is, for example, the ability of modeling possible.

Function $\ominus$ is *partial* and computes the result of withdrawing a template from a knowledge state in terms of a probability distribution $K \ominus T$ over the set of 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 $K \ominus T = \perp$.

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 $K \vdash T = \perp$.

2.2 Informal Timed Semantics

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 Labelled Transition System (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 away 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;
3. when the process executing the action resumes execution (i.e. becomes P);

and their consequent time relationship. If we want to extend SCEL with an explicit notion of (stochastic) time, we need to take into account the time-related issues mentioned above. These issues can be addressed at different levels of abstraction, reflecting a different choice of details that are considered in modeling SCEL actions.

In the following, the process/component initiating an action will be often called the *source* of the action execution, while the other components involved in the execution will be the *destinations*.

Point (1) above does not require particular comments. Point (2) requires to define *when* a component satisfies p with respect to a process executing an action, when time and possibly space are taken into consideration. We assume that source components are not aware of which are the components satisfying predicate p. Therefore, we define the notion of *observation* of the component

by the process, the result of which allows to establish whether the component satisfies the predicate or not. In the context of distributed systems this is often realised by means of a message, called an *envelope*, carrying the actual data item, sent by the process to the other components. According to this view, the check whether a component satisfies predicate p is performed *when the message reaches it*. This means that, as e.g. in PALOMA [9], a StocS action may require broadcast communication to be executed, even if its effect involves a few and possibly no components. In distributed systems different components may have different response times depending on different network conditions.

Finally, point (3) rises the issue on when source component execution is to be resumed. In particular, it is necessary to identify how the source component is made aware that its role in the communication has been completed. Get/query actions are blocking and they terminate when the source receives a knowledge item from any component. A reasonable choice is that further responses received are ignored. We assume appropriate mechanisms that ensure no confusion arises between distinct actions and corresponding messages. Put actions are non-blocking, so it is sufficient that the source component is aware that all reachable components are involved in the evaluation of the predicate. A possible choice is to set-up the transmission of one request of predicate evaluation for each component and then terminate the execution on the source side immediately. On the destination side, it is necessary to model the reception time as well as subsequent evaluation and corresponding knowledge repository modification.

In this book, we assume a *network-oriented* (NET-OR) view on the system, i.e. the execution of the various phases sketched above is explicitly modelled in detail by the operational semantics, which entails that actions are *non-atomic*. Indeed, they are executed through several intermediate steps, each of which requires appropriate time duration modelling. This kind of semantics is appropriate for models with spatial aspects, where distribution is a sensible aspect influencing the duration of communications on the basis of the location of components.

In order to obtain an underlying CTMC semantics, in StocS relevant delays—related to the execution of SCEL actions—are modelled as random RVs with negative exponential distributions. Therefore, in the following, whenever we associate a rate λ with a duration, the duration is exponentially distributed with rate λ. Non-determinism in process behaviour gives raise to race-conditions.

2.3 Explanatory example

Let us consider three components (see Fig. 1): $C_1 = I_1\,[\,K_1,\ P_1\,]$, $C_2 = I_2\,[\,K_2,\ P_2\,]$, and $C_3 = I_3\,[\,K_3,\ P_3\,]$ and let us assume process P_1 is defined as $\mathbf{put}(t)@\mathsf{p}\,.\,Q$[5]. Note that different components may be in different locations.

The execution of $\mathbf{put}(t)@\mathsf{p}$ starts in C_1 with the first phase in which one copy of the envelope message $\overline{\{t@\mathsf{p}\}}$ is sent, on behalf of P_1, *to each* other component

[5] For the sake of notational simplicity, in this book we assume that predicate p in process actions implicitly refers only to the *other* components, excluding the one where the process is in execution.

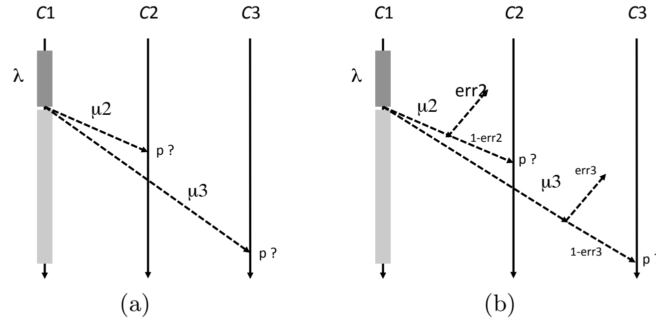

Fig. 1. Dynamics of the **put** action.

of the system[6]. In our example two copies are created/sent, one for/to C_2 and one for/to C_3. The time required for this phase (denoted in grey in Fig. 1 (a)) is modelled by a RV with rate λ: this value is computed as a function of several factors, among which is (the size of) t. Each envelope travels in the system and reaches the component it is associated with. Different envelopes may experience different transmission delays; therefore, *distinct* rates μ_2 and μ_3 are associated to each target (in Fig. 1 (a) this is illustrated by two arrows) and each rate may depend on t as well as other parameters like the distance between C_1 and the destination component. After message creation, P_1 can proceed—since **put** actions are non-blocking—behaving like Q; the light-grey stripe in Fig. 1 (a) illustrates the resumed execution of P_1 in C_1. The evaluation of predicate p is performed in each destination component C_j *when the message arrives at* C_j, and appropriate actions are taken on K_j. For example, it may happen that C_2 satisfies p *at the time the message reaches* C_2, which causes item t to be added to K_2—while C_3 does not satisfy p *at the time the message reaches* C_3—so that K_3 is left unchanged.

In practice, one can be interested in modeling also the event of failed delivery of the envelopes. This is interesting for instance for producing more realistic models with unreliable network communication. Furthermore, the inclusion of additional branches for failure modelling helps reducing discontinuities, which may facilitate the application of advanced analysis techniques based on fluid approximation [2], such as fluid model-checking [1]. Therefore, we add an error probability to the envelopes delivery, which we indicate as p_{err} (or simply err, in the figure). This more detailed semantics of the $\mathbf{put}(t)@\mathsf{p}$ action is illustrated in Fig. 1 (b).

[6] In an implementation of SCEL, this corresponds to a *request* sent either via a *broadcast*, that is not really efficient, or via a *multicast* to all the components potentially involved in the operation. jRESP, the *runtime-environment* of SCEL, provides both of these communication mechanisms [8].

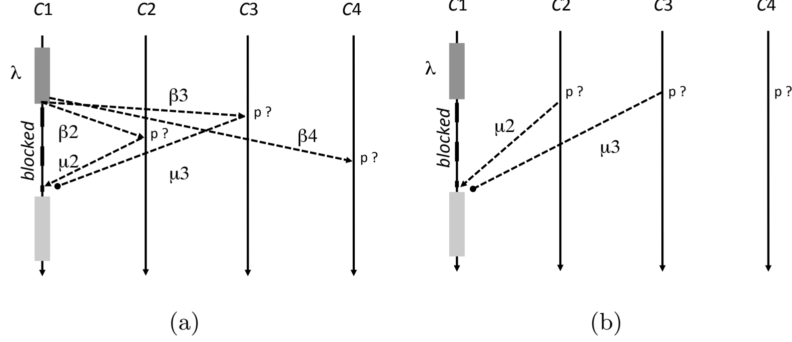

Fig. 2. Dynamics of the **get** action.

Let us now consider a scenario with four components $C_1 = I_1\,[\,K_1,\,P_1\,]$, $C_2 = I_2\,[\,K_2,\,P_2\,]$, $C_3 = I_3\,[\,K_3,\,P_3\,]$, and $C_4 = I_4\,[\,K_4,\,P_4\,]$ with P_1 of the form $\mathbf{get}(T)@\mathsf{p}.Q$ (or $\mathbf{qry}(T)@\mathsf{p}.Q$).

Similarly to the execution of $\mathbf{put}(t)@\mathsf{p}$, the first phase consists in the creation of the envelope messages—with rate λ; this is represented by the grey stripe in Fig. 2 (a). Since the **get** (resp. **qry**) action is blocking, P_1 is then put into a waiting state (denoted by a dashed line in the figure). Each copy of the message is sent to the corresponding component C_j, with transmission rate β_j. Upon envelope arrival, each component checks for satisfaction of predicate p and availability of an item t matching template T. Those components for which such a check gives a positive result, say C_2 and C_3, are eligible to answer the request with item t_2 and t_3 respectively, and a race condition takes place, so that *only one* component, say C_2, succeeds in providing the item, as required by SCEL semantics. Once the item (t_2) reaches C_1, P_1 can restart its execution from $Q\vartheta$, with a suitable variable binding $\mathsf{match}(T, t) = \vartheta$; when the other item ($t_3$) will reach C_1 it will be disregarded. Transmission rate from C_2 (C_3 respectively) is μ_2 (μ_3).

In order to simplify the semantics of the **get**/**qry** actions and to make it more similar to the two-steps semantics of the **put** action, we decided to model the two phases of envelope delivery and response collection as a single one. So, on message creation, the source (P_1) is blocked on waiting for some destination to synchronise with it on the exchange of the retrieved item t matching the template T, as illustrated in Fig. 2 (b). During this synchronization, the predicate p is also checked, on the side of the destination, and the knowledge is changed accordingly. The synchronization attempt of all other candidates (C_3, in the example) is simply lost. In terms of the underlying stochastic model, we are replacing a phase-type distribution, consisting of the sequence of two exponential RVs, with an exponential RV. This choice is also convenient for simplifying the definition of the formal semantics, since it avoids the need of giving a unique id to envelope messages, to be used in the subsequent response collection phase.

3 Preliminary Definitions for Operational Semantics

In this section we provide preliminary notions to support the presentation of the semantics of StocS formalising the ideas described in the previous section. The semantics definition is given in the FuTSs style [7] and, in particular, using its Rate Transition Systems (RTS) instantiation [6].

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*[7] that associates a real non-negative value with each state P'. A non-zero value represents the rate of the exponential distribution characterising 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 chapter, after having recalled some basic notation, and we define them in an abstract way, with respect to a generic sets X, X_1, X_2, ...

Let $\mathbf{TF}(X, \mathbb{R}_{\geq 0})$ denote the set of *total* functions from X to $\mathbb{R}_{\geq 0}$, and $\mathscr{F}$, $\mathscr{P}$, $\mathscr{Q}$, $\mathscr{R}$, ... 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*: $\mathscr{F}$ is an element of $\mathbf{FTF}(X, \mathbb{R}_{\geq 0})$ if and only if there exist $\{d_1, \ldots, d_m\} \subseteq X$, 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.

Definition 1.

1. *For elements* $d_1, \ldots, d_m \in X$ *and* $\gamma_1, \ldots, \gamma_m \in \mathbb{R}_{\geq 0}$ *we use the notation* $[d_1 \mapsto \gamma_1, \ldots, d_m \mapsto \gamma_m]$ *for denoting the following function:*

$$[d_1 \mapsto \gamma_1, \ldots, d_m \mapsto \gamma_m]\, d \;=_{\text{def}}\; \begin{cases} \gamma_i & \text{if } d = d_i \in \{d_1, \ldots, d_m\}, \\ 0 & \text{otherwise.} \end{cases}$$

   the 0 *constant function in* $\mathbf{FTF}(X, \mathbb{R}_{\geq 0})$ *is denoted by* $[]$;
2. *We define addition on* $\mathbf{FTF}(X, \mathbb{R}_{\geq 0})$ *as the point-wise extension of* $+$ *on* $\mathbb{R}$, *i.e.* $(\mathscr{F}_1 + \mathscr{F}_2)\, d \;=_{\text{def}}\; (\mathscr{F}_1\, d) + (\mathscr{F}_2\, d)$;
3. *For any injective binary operator* $\bullet : X_1 \times X_2 \to X$ *we define its* lifting *to* $\mathbf{FTF}(X_1, \mathbb{R}_{\geq 0}) \times \mathbf{FTF}(X_2, \mathbb{R}_{\geq 0}) \to \mathbf{FTF}(X, \mathbb{R}_{\geq 0})$ *by letting*

$$(\mathscr{F}_1 \bullet \mathscr{F}_2)d \;=_{\text{def}}\; \begin{cases} (\mathscr{F}_1\, d_1) \cdot (\mathscr{F}_2\, d_2) & \text{if } \exists d_1 \in X_1, d_2 \in X_2.\ d = d_1 \bullet d_2, \\ 0 & \text{otherwise.} \end{cases}$$

4. *We use the characteristic function* $\mathcal{X}$ *on* X *with* $\mathcal{X} : X \to \mathbf{FTF}(X, \mathbb{R}_{\geq 0})$ *such that* $\mathcal{X}\, d \;=_{\text{def}}\; [d \mapsto 1]$

[7] In the sequel, Currying will be used for continuation function application.

Definition 2. *An A-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 relation $\rightarrowtail \subseteq S \times A \times \mathbf{FTF}(S, \mathbb{R}_{\geq 0})$ is the A-labelled* transition relation.

In order to distinguish and identify the rules of the semantics definition, we label them by unique names. Note that a rule with name r may have one or more associated blocking rules r_{B} which have the role of allowing the execution of no actions other than those explicitly allowed by existing inference rules. These B-rules will not be further commented in the following sections.

4 Network-oriented Operational Semantics

We recall that 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 $\mathbb{E}$. Interface evaluations are used within the so-called *rate function* $\mathcal{R} : \mathbb{E} \times \mathit{Act} \times \mathbb{E} \rightarrow \mathbb{R}_{\geq 0}$, which defines the rates of actions depending on the interface evaluation of the *source* of the action, the action itself (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 synchronisation 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 take 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* $f_{\mathsf{err}} : \mathbb{E} \times \mathit{Act} \times \mathbb{E} \rightarrow [0, 1]$ computing the probability of an error in message delivery. In the semantics, we distinguish between output actions (those issued by a source component) and input actions (those accepted by a destination component). To simplify the synchronisation of input and output actions, we assume input actions are *probabilistic*, and output actions are *stochastic*, therefore their composition is directly performed through multiplication.

In order to realise this semantics we extend the set of labels of actions performed by processes and systems as described in the following.

4.1 Operational semantics of processes

The NET-OR semantics of StocS processes is the RTS $(\mathit{Proc}, \mathit{Act}_{\mathit{Proc}}, \mathbb{R}_{\geq 0}, \rightharpoondown_e)$. Proc is the set of process terms defined according to the syntax of StocS given in Table 1 $\mathit{Act}_{\mathit{Proc}}$ is the set of labels defined according to the grammar below (where $t \in \mathbb{I}$, $T \in \mathbb{T}$, $\mathbf{gq} \in \{\mathbf{get}, \mathbf{qry}\}$, c is a Target, and e is the evaluation of an interface) and it is ranged over by $\alpha, \alpha', \ldots$:

$$\mathit{Act}_{\mathit{Proc}} \quad ::= \quad \tau \;\;\Big|\;\; \overline{\{t@\mathsf{p}\}} \;\;\Big|\;\; \overline{e : \mathbf{put}(t)@c} \;\;\Big|\;\; \overline{e : \mathbf{gq}(T : t)@c}$$

The transition relation $\rightharpoondown \subseteq \mathit{Proc} \times \mathit{Act}_{\mathit{Proc}} \times \mathbf{FTF}(\mathit{Proc}, \mathbb{R}_{\geq 0})$ is the least relation satisfying the rules of Table 2. $\rightharpoondown_e$ is parametrized by e, which is the interface

Inactive process and envelopes:

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

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

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

$$\frac{\mathsf{match}(T,t) = \vartheta \quad \lambda = \mathcal{R}(\sigma, \mathbf{gq}(T:t)@\mathsf{self}, _)}{\mathbf{gq}(T)@\mathsf{self}.P \xrightarrow{\overline{\sigma : \mathbf{gq}(T:t)@\mathsf{self}}}_{\sigma} [P\vartheta \mapsto \lambda]}\ (\text{GQL})$$

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

$$\frac{\lambda = \mathcal{R}(\sigma, \mathbf{gq}(T:_)@\mathsf{p}, _)}{\mathbf{gq}(T)@\mathsf{p}.P \xrightarrow{\tau}_{\sigma} [\{\mathbf{gq}(T)@\mathsf{p}\}.P \mapsto \lambda]}\ (\text{GQW}) \qquad \frac{\alpha \neq \tau}{\mathbf{gq}(T)@\mathsf{p}.P \xrightarrow{\alpha} []}\ (\text{GQW}_\text{B})$$

$$\frac{\mathsf{match}(T,t) = \vartheta \quad \beta = \mathcal{R}(\sigma, \{\mathbf{gq}(T:t)@\mathsf{p}\}, \delta)}{\{\mathbf{gq}(T)@\mathsf{p}\}.P \xrightarrow{\overline{\delta : \{\mathbf{gq}(T:t)@\mathsf{p}\}}}_{\sigma} [P\vartheta \mapsto \beta]}\ (\text{GQD})$$

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

Choice, definition, and parallel composition:

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

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

Table 2. Operational semantics of STOCS processes.

evaluation of the component in which the process resides: we feel free to omit the parameter, if not used in the rule.

We now briefly illustrate the rules of Table 2. We assume to have additional syntactical terms (not available at the user syntax level) which we call *envelopes*. They are of the form $\{t@\mathsf{p}\}_\mu$, can be put in parallel with processes, and denote

messages that are currently traveling towards targets. A second syntactical construct we introduce is $\{\mathbf{get}(T)@\mathsf{p}\}$ ($\{\mathbf{qry}(T)@\mathsf{p}\}$, respectively) which denotes a *waiting* state of the process and it is treated as an action.

(NIL) **nil** is the terminated process, since no process is reachable from it via any action;

(ENV) allows to complete envelope delivery with duration specified by μ;

(PUT)/(PUT$_\mathrm{B}$) 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 $\mathcal{R}$. The execution of a $\mathbf{put}(t)@c$ action depends on the source component and *all* the other components 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 $\mathcal{R}$. On the contrary, rule (PUT$_\mathrm{B}$) states that $\mathbf{put}(t)@c.P$ cannot reach any process after a transition with a label that is different from $\overline{\mathbf{put}(t)@c}$.

(GQL) allows a process to issue a **get** (**qry**, respectively) action over the local knowledge repository (i.e. with target self). The rule models the execution of action $\mathbf{get}(T)@\mathsf{self}$ ($\mathbf{qry}(T)@\mathsf{self}$, respectively) by process $\mathbf{get}(T)@\mathsf{self}.P$ ($\mathbf{qry}(T)@\mathsf{self}.P$, respectively). The duration of this action is described by a rate λ computed using the function $\mathcal{R}$ depending on the interface evaluation of the source σ (i.e. the container component) and on the action; the continuation associates λ with $P\vartheta$, i.e. the process obtained by applying to P the substitution ϑ resulting from match-ing template T against item t;

(GQW) realises the first step of a **get** (**qry**, respectively) action over a remote knowledge in a component satisfying a predicate p, which consists in preparing an envelope $\{\mathbf{get}(T)@\mathsf{p}\}$ ($\{\mathbf{qry}(T)@\mathsf{p}\}$, respectively), which takes a time interval exponentially distributed with rate λ, and brings process P to a *wait* state $\{\mathbf{get}(T)@\mathsf{p}\}.P$ ($\{\mathbf{qry}(T)@\mathsf{p}\}.P$, respectively). Recall that **get**/**qry** actions are *blocking* and the execution of P is resumed only when a counterpart satisfying p has a knowledge item t matching T available and the delivery of t is completed. The duration of this first step is described by a rate λ computed using the function $\mathcal{R}$ depending only on the interface evaluation of the source σ (i.e. the container component) and the sent template T;

(GQD) realises the second step of a **get** (**qry**, respectively) action, which consists in the *delivery* of the knowledge item t matching T and has a duration described by a rate β computed by the function $\mathcal{R}$. Note that in this case the function $\mathcal{R}$ is computed considering interface evaluation of the source σ and the destination δ, as well as the sent item t, which means that this rate can be made dependent (for example) on the distance of the two parties.

(CHO) cumulates the relevant rates by means of the application of the *choice* operator $+$ on the continuation of P ($\mathscr{P}$) and that of Q ($\mathscr{Q}$), thus conforming to the race condition principle of CTMCs;

put actions:

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

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

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

$$\frac{P \xrightarrow{\overline{\{t@\mathsf{p}\}}} \mathscr{P} \quad I(K) \models \mathsf{p} \quad K \oplus t = \pi}{I\,[\,K,\,P\,] \xrightarrow{\overline{\{t@\mathsf{p}\}}} I[\pi, \mathscr{P}]} \text{ (C-ENVA)}$$

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

Table 3. Operational semantics of StocS components (Part 1).

(DEF) is the rule for process instantiation;

(PAR) realises process parallel composition $P \mid Q$ and uses Def. 1, item (3) applied to the process parallel composition syntactic constructor $\mid$ (which is obviously injective). Therefore, given two functions $\mathscr{R}_1$ and $\mathscr{R}_2$, the function $\mathscr{R}_1 \mid \mathscr{R}_2$ applied to process term R returns the product $(\mathscr{R}_1\, R_1) \cdot (\mathscr{R}_2\, R_2)$, whenever R is of the form $R_1 \mid R_2$, for some terms R_1 and R_2, and 0 otherwise. In the rule, also the characteristic function $\mathcal{X}$ is used. Function $\mathscr{P} \mid (\mathcal{X}Q)$ applied to R returns $\mathscr{P}\, R'$ if $R = R' \mid Q$ for some R' and 0 otherwise; i.e. the function behaves as the continuation of P ($\mathscr{P}$) for terms where Q does not progress (for one step). In conclusion, $\mathscr{P} \mid (\mathcal{X}\, Q) + (\mathcal{X}\, P) \mid \mathscr{Q}$ correctly represents process interleaving, keeping track of the relevant rates.

4.2 Operational semantics of components and systems

The NET-OR semantics of StocS systems is the RTS $(Sys, Act_{Sys}, \mathbb{R}_{\geq 0}, \rightarrow)$. Sys is the set of system terms defined according to the syntax of StocS given in Table 1. Set Act_{Sys} of labels is defined according to the grammar below (where $\mathbf{gq} \in \{\mathbf{get}, \mathbf{qry}\}$, $t \in \mathbb{I}$, $T \in \mathbb{T}$, p is a Predicate, and e is the evaluation of an interface):

get/**qry** actions (where, $\mathbf{gq} \in \{\mathbf{get}, \mathbf{qry}\}$):

$$\frac{\sigma = I(K) \quad P \xrightarrow{\overline{\sigma : \mathbf{get}(T{:}t)@\mathsf{self}}}_{\sigma} \mathscr{P} \quad K \ominus T = \pi}{I\,[\,K,\,P\,] \xrightarrow{\overleftrightarrow{\sigma : \mathbf{get}(T{:}t)@\mathsf{self}}} I[\pi(t), \mathscr{P}]} \;(\text{C-GETL})$$

$$\frac{\sigma = I(K) \quad P \xrightarrow{\overline{\sigma : \mathbf{qry}(T{:}t)@\mathsf{self}}}_{\sigma} \mathscr{P} \quad K \vdash T = \pi}{I\,[\,K,\,P\,] \xrightarrow{\overleftrightarrow{\sigma : \mathbf{qry}(T{:}t)@\mathsf{self}}} I[(\mathcal{X}K) \cdot \pi(t), \mathscr{P}]} \;(\text{C-QRYL})$$

$$\frac{K \ominus T = \bot}{I\,[\,K,\,P\,] \xrightarrow{\overleftrightarrow{\sigma : \mathbf{get}(T{:}t)@\mathsf{self}}} []} \;(\text{C-GETL}_{\text{B}}) \qquad \frac{K \vdash T = \bot}{I\,[\,K,\,P\,] \xrightarrow{\overleftrightarrow{\sigma : \mathbf{qry}(T{:}t)@\mathsf{self}}} []} \;(\text{C-QRYL}_{\text{B}})$$

$$\frac{\sigma = I(K) \quad P \xrightarrow{\overline{\delta : \{\mathbf{gq}(T{:}t)@\mathsf{p}\}}}_{\sigma} \mathscr{P}}{I\,[\,K,\,P\,] \xrightarrow{\overline{\delta : \{\mathbf{gq}(T{:}t)@\mathsf{p}\}}} I[(\mathcal{X}K), \mathscr{P}]} \;(\text{C-GQO})$$

$$\frac{\delta = I(K) \quad \delta \models \mathsf{p} \quad K \ominus T = \pi}{I\,[\,K,\,P\,] \xrightarrow{\delta : \{\mathbf{get}(T{:}t)@\mathsf{p}\}} I[\pi(t), (\mathcal{X}P)]} \;(\text{C-GETI})$$

$$\frac{\delta \neq I(K) \quad \vee \quad I(K) \not\models \mathsf{p} \quad \vee \quad K \ominus T = \bot}{I\,[\,K,\,P\,] \xrightarrow{_ : \{\mathbf{get}(T{:}t)@\mathsf{p}\}} []} \;(\text{C-GETI}_{\text{B}})$$

$$\frac{\delta = I(K) \quad \delta \models \mathsf{p} \quad K \vdash T = \pi}{I\,[\,K,\,P\,] \xrightarrow{\delta : \{\mathbf{qry}(T{:}t)@\mathsf{p}\}} [I\,[\,K,\,P\,] \mapsto \pi(t)]} \;(\text{C-QRYI})$$

$$\frac{\delta \neq I(K) \quad \vee \quad I(K) \not\models \mathsf{p} \quad \vee \quad K \vdash T = \bot}{I\,[\,K,\,P\,] \xrightarrow{_ : \{\mathbf{qry}(T{:}t)@\mathsf{p}\}} []} \;(\text{C-QRYI}_{\text{B}})$$

τ actions:

$$\frac{\rho = I(K) \quad P \xrightarrow{\tau}_{\rho} \mathscr{P}}{I\,[\,K,\,P\,] \xrightarrow{\tau} I[(\mathcal{X}K), \mathscr{P}]} \;(\text{C-TAU})$$

Table 4. Operational semantics of StocS components (Part 2).

$$
\begin{array}{rcll}
Act_{Sys} & ::= & e : \mathbf{put}(t)@\mathsf{p} \;\big|\; e : \{\mathbf{gq}(T:t)@\mathsf{p}\} \;\big| & \text{(input actions)} \\
& & \overline{e : \mathbf{put}(t)@\mathsf{p}} \;\big|\; \overline{e : \{\mathbf{gq}(T:t)@\mathsf{p}\}} \;\big| & \text{(output actions)} \\
& & \tau \;\big|\; \overleftrightarrow{e : \{\mathbf{gq}(T:t)@\mathsf{p}\}} \;\big| & \text{(synchronisations)} \\
& & \overline{\{t@\mathsf{p}\}} & \text{(envelopes)}
\end{array}
$$

The transition relation $\rightarrow \subseteq Sys \times Act_{Sys} \times \mathbf{FTF}(Sys, \mathbb{R}_{\geq 0})$ is the least relation satisfying the rules of Tables 3, 4 and 5, where the process relation defined in Table 2 is also used.

---

put synchronization:

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

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

---

get/**qry** synchronization ($\mathbf{gq} \in \{\mathbf{get}, \mathbf{qry}\}$):

$$
\frac{\begin{array}{c} S_1 \xrightarrow{\overleftrightarrow{\delta:\{\mathbf{gq}(T:t)@\mathsf{p}\}}} \mathscr{S}_1^s \; S_1 \xrightarrow{\overline{\delta:\{\mathbf{gq}(T:t)@\mathsf{p}\}}} \mathscr{S}_1^o \; S_1 \xrightarrow{\delta:\{\mathbf{gq}(T:t)@\mathsf{p}\}} \mathscr{S}_1^i \\ S_2 \xrightarrow{\overleftrightarrow{\delta:\{\mathbf{gq}(T:t)@\mathsf{p}\}}} \mathscr{S}_2^s \; S_2 \xrightarrow{\overline{\delta:\{\mathbf{gq}(T:t)@\mathsf{p}\}}} \mathscr{S}_2^o \; S_2 \xrightarrow{\delta:\{\mathbf{gq}(T:t)@\mathsf{p}\}} \mathscr{S}_2^i \end{array}}{S_1 \parallel S_2 \xrightarrow{\overleftrightarrow{\delta:\{\mathbf{gq}(T:t)@\mathsf{p}\}}} \mathscr{S}_1^s \parallel (\mathcal{X}\, S_2) + \mathscr{S}_1^o \parallel \mathscr{S}_2^i + \mathscr{S}_1^i \parallel \mathscr{S}_2^o + (\mathcal{X}\, S_1) \parallel \mathscr{S}_2^s} \; (\textsc{s-gqs})
$$

$$
\frac{S_1 \xrightarrow{\delta:\{\mathbf{gq}(T:t)@\mathsf{p}\}} \mathscr{S}_1 \quad S_2 \xrightarrow{\delta:\{\mathbf{gq}(T:t)@\mathsf{p}\}} \mathscr{S}_2}{S_1 \parallel S_2 \xrightarrow{\delta:\{\mathbf{gq}(T:t)@\mathsf{p}\}} \mathscr{S}_1 \parallel (\mathcal{X}\, S_2) + (\mathcal{X}\, S_1) \parallel \mathscr{S}_2} \; (\textsc{s-gqi})
$$

Internal actions, for $\alpha \in \{\, \tau, \overleftrightarrow{e \,:\, \mathbf{put}(t)@\mathsf{self}}, \overleftrightarrow{e \,:\, \mathbf{gq}(T:t)@\mathsf{self}}, \overline{\{t@\mathsf{p}\}}\,\}$:

$$
\frac{S_1 \xrightarrow{\alpha} \mathscr{S}_1 \quad S_2 \xrightarrow{\alpha} \mathscr{S}_2}{S_1 \parallel S_2 \xrightarrow{\alpha} \mathscr{S}_1 \parallel (\mathcal{X}\, S_2) + (\mathcal{X}\, S_1) \parallel \mathscr{S}_2} \; (\textsc{s-spl})
$$

Table 5. Operational semantics of StocS systems.

The definition of the semantics of system parallel composition $S_1 \parallel S_2$ uses Def. 1, item (3) applied to the system parallel composition constructor $\parallel$, which is injective. As usual, interleaving is modelled as a combination of lifted $\parallel$, $+$ on

functions and the characteristic function. In the rules, we also use Def. 1, item (3) applied to the component syntactic constructors $I[\cdot,\cdot]$, which is injective.

In Table 3 and Table 4, rules are grouped to illustrate how the various action types are realised.

(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.

(C-PUTO) this rule is used when the target of a **put** is not self but a predicate p; the 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.

(C-PUTI) models the *initiation* of the execution of action $\mathbf{put}(t)@c$, which requires several steps to complete, it allows the reception of a **put** action, and it is responsible for the creation of the envelope (carrying the incoming message) in parallel to the local process of a component, thus modeling its travel towards that component in terms of the time necessary to reach it, 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 is executed with rate λ, computed using the function $\mathcal{R}$ 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 realised at system level by the broadcast rules of Table 5.

(C-ENVA)/(C-ENVR) realise envelope delivery by specifying the conditions under which a component *accepts* or *refuses*, respectively, an arriving envelope;

(C-GETL)/(C-QRYL) realise the local **get** (**qry**) action retrieving an item $t \in \mathbb{I}$ matching the pattern T, if possible, in the execution of process P (in the label of the process action we include the item t and we include the interface evaluation σ of the component for computing the action rate) and, since the **get** (**qry**) action may result in several distinct knowledge bases, these need to be summed together considering all possibilities: π is a distribution over pairs (knowledge base and knowledge item) and the possible components in the continuation are weighted by using π;

(C-GQO) realises an output **get**/**qry** action as in the ACT-OR semantics, but with a different label $(\{\ldots\})$ which denotes the synchronization on a *waiting state*;

(C-GETI)/(C-QRYI) realise an input **get**/**qry** action, again as in the ACT-OR semantics, but with a different label ($\{\ldots\}$);

(C-TAU) allows a component to make a τ whenever its process makes such an action.

Finally, we discuss the rules in Table 5:

(S-PO)/(S-PI) realise the broadcast communication of **put**: (S-PO) ensures that if any subsystem executes an output **put** action (i.e. it executes a transition with label $\overline{\sigma : \mathbf{put}(t)@\mathsf{p}}$), then the remaining subsystem must execute the corresponding input **put** action (i.e. it should execute 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; while (S-PI) allows an input **put** action forcing *all* of the components of a sub-system to perform that action;

(S-GQS) realises one-to-one synchronization of **get**/**qry** actions (which are not broadcast), denoted by a $\overleftrightarrow{e : \mathbf{gq}(T : t)@\mathsf{p}}$ label, and performs aggregation of: (1) the synchronization rate of the left (right) sub-system, with the right (resp. left) subsystem that must not progress (this is realised using the $\mathcal{X}$ characteristic function), and (2) output rates of the left sub-system and input rates of the right subsystem (as well as the symmetric case), combined with the $\|$ operator;

(S-GQI) realises the input **get** (resp. **qry**) action for systems in which *one* component, among those satisfying the target predicate and having a matching knowledge item, can answer;

(S-SPL/S-SGQL) allow a system to execute an internal action and exposes the label denoting the type of action to allow appropriate aggregation of the observed rates.

5 StocS at work

In this section we use a simple example to show how StocS can be used to specify and verify quantitative properties of adaptive systems. We consider a *cloud scenario* where users can execute their task in a distributed environment. A *QoS* profile is associated to each user. Possible profiles are: *basic*, *standard*, *premium* and *super premium*. Our system has two main requirements. First, we have to reduce the number of tasks submitted by *premium* and *super premium* users that are waiting for the execution. Moreover, we have to minimise the number of computational resources allocated for the execution of tasks.

In StocS both *users* and *computational resources* can be rendered via *components*. We refer to the first group of components as *user components*, while the components in the second group are referred as *computational components*.

Components associated to users publish in the interface the user QoS level. Moreover, users communicate the need to execute a task via their knowledge. Computational components retrieve the corresponding knowledge element. The following process, executed at users components, models this behaviour:

$$\mathsf{UTask}() \stackrel{def}{=} \mathbf{put}(\text{“}TASK\text{”})@\mathsf{self}.\mathsf{UTask}()$$

We assume that tasks requests arrive at a rate $\lambda_{tr} = 50.0$. We also assume that 40% of the requests come from *basic* users; 35% come from *standard* users while 15% and 10% arrive from *premium* and *super premium* users respectively.

Computational resources can execute at most k tasks at the same time (in the following we consider $k = 15$). This value corresponds to the number of ServiceAgents that are executed at each computational component. These processes retrieve and execute tasks. We can vary the method used by ServiceAgent to retrieve pending tasks from users to obtain different kinds of specifications. In particular, we consider three possible approaches: *static allocation*, *progressive allocation* and *dynamic allocation*. When *static allocation* is used, each computational component only handles tasks from users of a given level. In the *progressive allocation*, like in the previous case, each computational component is associated with a QoS level. However, differently from the *static allocation*, each process is able to handle tasks from users with a QoS level that is equal or higher than the associated one. Finally, in the *dynamic allocation*, the class of users that a component can handle depends on the *computation load*: the higher is the number of executed tasks in a component, the higher is the QoS level that the same component can handle.

Process ServiceAgent is defined as follows:

$$\begin{aligned}\mathsf{ServiceAgent}() \stackrel{def}{=}\ & \mathbf{get}(\text{“}TASK\text{”})@c. \\ & \mathbf{put}(\text{“}EXECUTE\text{”})@\mathsf{self}. \\ & \mathsf{ServiceAgent}()\end{aligned}$$

The **get** action is activated with rate $\lambda = 1$ that is also the rate of data transmission. The execution time of action $\mathbf{put}(\text{“}EXECUTE\text{”})@\mathsf{self}$ mimics the task execution time and it is exponentially distributed with rate $\lambda_e = \frac{1}{3}$. All the above mentioned rates do not appear explicitly in the syntax of the specifications. These are obtained via the appropriate rate function $\mathcal{R}$ according to the kind of action performed, the data transmitted and the interfaces of the involved components, provides the actual action rate. Due to lack of space we omit the explicit definition of function $\mathcal{R}$ that can be easily inferred from the informal description.

Note that in process ServiceAgent, the term c varies according to the considered *allocation method*:

static allocation:

$$\mathsf{this}.level == \mathcal{I}.level$$

progressive allocation:

$$\mathsf{this}.level <= \mathcal{I}.level$$

dynamic allocation:

$$((\mathsf{this}.load <= 50\%) \wedge \mathcal{I}.level >= \mathsf{base}) \vee$$
$$((\mathsf{this}.load <= 66\%) \wedge \mathcal{I}.level >= \mathsf{standard}) \vee$$
$$((\mathsf{this}.load <= 88\%) \wedge \mathcal{I}.level >= \mathsf{premium}) \vee$$
$$(\mathcal{I}.level == \mathsf{superpremium})$$

The formulas listed above identifies the specific predicates used by ServiceAgent to retrieve requests from *user components*. Above, this is used to refer to the interface of the local component (i.e. the *computational component* executing a ServiceAgent) while $\mathcal{I}$ refers to the target interface, i.e. the component from which the request is retrieved. Two attributes are used in the considered predicates: level and load. These identify the user *QoS* level and the workload of a component, respectively.

Note that in the case of *dynamic allocation*, *self-awareness* is rendered directly in the target predicated used to retrieve user requests. A ServiceAgent handles base (resp. standard, premium) users only when the component's load is under 50% (resp. 66%, 88%), while super premium users are always executed. The actual value of attribute *load* is transparently published on the component interface and dynamically computed according to the number of executed tasks.

To perform analyses of the considered system we use jRESP [8]. This is a Java environment that provides a simulation environment that, while implementing the stochastic semantics presented in Section 4, can be used to analyse StocS specifications.

We consider three configurations:

C1 *static allocation* with 16 components: 6 for *base*, 5 *standard*, 3 *premium* and 2 *super premium*;
C2 *progressive allocation* with 13 components: 7 for *base*, 4 *standard*, 2 *premium*;
C3 *dynamic allocation* with 12 components.

The results of simulations are reported in Figure 3 and Figure 4. If we compare the three configurations with respect to the average number of waiting tasks, the model **C1** is the one guaranteeing better quality of service (see Figure 3). However, if we consider the workload, **C3** is the one that, by using a less number of computational resources, guarantees a well balanced use of resources (see Figure 4).

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. The proposed semantics models the execution

[8] http://jresp.sourceforge.org

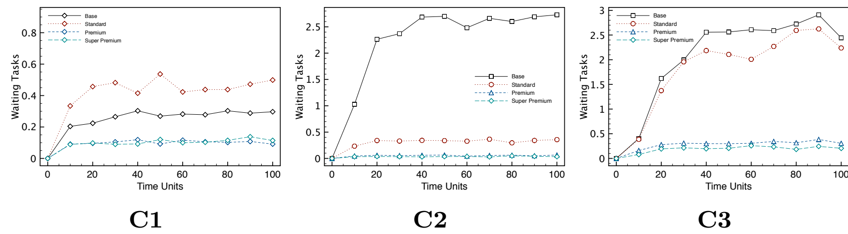

Fig. 3. Simulation results: Number of waiting tasks

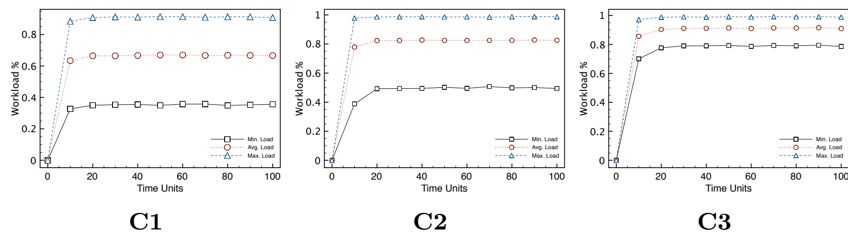

Fig. 4. Simulation results: Workload

StocS actions through several intermediate steps modelling the behaviour of an underling framework providing the machinery for realising the StocS communication primitives. A case study concerning an abstract model of a cloud system was presented to illustrate the use of the various language primitives of StocS. As a future work we plan to develop fluid semantics of StocS together with the related verification techniques that can be used to address the analysis of large scale collective systems along the lines of work in [1, 2].

References

1. Luca Bortolussi and Jane Hillston. Fluid model checking. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2012.
2. Luca Bortolussi, Jane Hillston, Diego Latella, and Mieke Massink. Continuous approximation of collective system behaviour: A tutorial. *Perform. Eval.*, 70(5):317–349, 2013.
3. Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. A conceptual framework for adaptation. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2012.
4. R. De Nicola, J.-P. Katoen, D. Latella, M. Loreti, and M. Massink. Model Checking Mobile Stochastic Logic. *Theoretical Computer Science. Elsevier*, 382(1):42–70, 2007. http://dx.doi.org/10.1016/j.tcs.2007.05.008; DOI 10.1016/j.tcs.2007.05.008.

5. Rocco De Nicola, Gian Luigi Ferrari, Michele Loreti, and Rosario Pugliese. A language-based approach to autonomic computing. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48. Springer, 2011.
6. Rocco De Nicola, Diego Latella, Michele Loreti, and Mieke Massink. Rate-based transition systems for stochastic process calculi. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris E. Nikoletseas, and Wolfgang Thomas, editors, *ICALP (2)*, volume 5556 of *Lecture Notes in Computer Science*, pages 435–446. Springer, 2009.
7. Rocco De Nicola, Diego Latella, Michele Loreti, and Mieke Massink. A uniform definition of stochastic process calculi. *ACM Comput. Surv.*, 46(1):5:1–5:35, July 2013.
8. Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *TAAS*, 9(2):7, 2014.
9. Cheng Feng and Jane Hillston. PALOMA: A process algebra for located markovian agents. In Gethin Norman and William H. Sanders, editors, *Quantitative Evaluation of Systems - 11th International Conference, QEST 2014, Florence, Italy, September 8-10, 2014. Proceedings*, volume 8657 of *Lecture Notes in Computer Science*, pages 265–280. Springer, 2014.
10. Nora Koch, Matthias Hölzl, Annabelle Klarl, Philip Mayer, Tomas Bures, Jaques Combaz, Alberto Lluch Lafuente, Rocco De Nicola, Stefano Sebastio, Francesco Tiezzi, Andrea Vandin, Fabio Gadducci, Valentina Monreale, Ugo Montanari, Michele Loreti, Carlo Pinciroli, Mariachiara Puviani, Franco Zambonelli, Nikola Šerbedžija, and Emil Vassev. JD3.2: Software engineering for self-aware SCEs, 2013. ASCENS Deliverable JD3.2.
11. Diego Latella, Michele Loreti, Mieke Massink, and Valerio Senni. Stochastically timed predicate-based communication primitives for autonomic computing. In Nathalie Bertrand and Luca Bortolussi, editors, *Proceedings Twelfth International Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2014, Grenoble, France, 12-13 April 2014*, volume 154 of *EPTCS*, pages 1–16, 2014.
12. Rocco De Nicola, Matthias Hölzl, Michele Loreti, Alberto Lluch Lafuente, Ugo Montanari, Emil Vassev, and Franco Zambonelli. JD2.1: Languages and knowledge models for self-awareness and self-expression, 2012. ASCENS Deliverable JD2.1.
13. Nikola Šerbedžija, Nicklas Hoch, Carlo Pinciroli, Michal Kit, Tomas Bures, Valentina Monreale, Ugo Montanari, Philip Mayer, and José Velasco. D7.3: Third report on wp7 - integration and simulation report for the ascens case studies, 2013. ASCENS Deliverable D7.3.
14. Nikola Šerbedžija, Mieke Massink, Carlo Pinciroli, Manuele Brambilla, Diego Latella, Marco Dorigo, Mauro Birattari, Philip Mayer, José Angel Velasco, Nicklas Hoch, Henry P. Bensler, Dhaminda Abeywickrama, Jaroslav Keznikl, Ilias Gerostathopoulos, Tomas Bures, Rocco De Nicola, and Michele Loreti. D7.2: Second report on wp7 - integration and simulation report for the ascens case studies, 2012. ASCENS Deliverable D7.2.