A Formal Approach to Modelling Time Properties of Service-Oriented Systems*

L. Bocchi[1], J. Fiadeiro[1], S. Gilmore[2], J. Abreu[1], M. Solanki[1], and V. Vankayala[1]

[1] Department of Computer Science, University of Leicester
{lb148,jose,jpad2,ms491}@mcs.le.ac.uk
[2] School of Informatics, The University of Edinburgh
stg@staffmail.ed.ac.uk

Abstract. We provide a formal model for expressing and analysing time-related properties of service-oriented systems. Our approach extends SRML, a high-level modelling language that we have been developing in the SENSORIA project. We introduce new primitives for SRML that capture several kinds of delays that can occur during service provision (e.g., the time taken by components to process events and perform computations, the time taken by the SOA middleware for discovering, selecting and binding services, etc.). Finally, we show how we can use the stochastic process algebra PEPA and its development environment to represent and analyse time properties of SRML models.

1 Introduction

Service-oriented systems have come a long way in the past decade and are now considered one of the key technologies for building new generations of digital business systems. Service-based enterprise applications are usually composed "on-the-fly" from services individually provided by different providers. The run-time provision of composed services is governed by negotiations/agreements defining expected quality-of-service metrics. The conformance of such systems to mutually negotiated timing-related policies is crucial when analysing the fulfilment of their contractual obligations.

The work presented in this paper addresses the modelling and analysis of time-related properties of service-oriented models. For instance, we have in mind the ability to certify that a mortgage-brokerage service satisfies properties of the form *"In at least* 80% *of the cases, a reply to a request for a mortgage proposal will be sent within 7 seconds"*. Properties of this kind are extremely important in a number of application domains and are usually part of the service level agreements (SLAs) that are negotiated between clients and providers. However, to the best of our knowledge, we are still lacking a formal model of how such timing issues arise in service-oriented architectures (SOAs) as well as modelling and analysis tools that can be used by designers to develop services that can be certified to meet given timing constraints.

To this aim, we extended SRML, a modelling language for service-oriented computing (SOC) developed by the SENSORIA consortium, with primitives that allow us to capture, model and analyse typical time-related SLAs. SRML is a "prototype language"

* This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

developed to capture the foundational aspects of SOC (including service composition, dynamic binding/reconfiguration and service level agreements) while remaining simple enough to represent these aspects within one integrated formal framework. Details on the language and associated methodology can be found in [12, 5] (see also [1, 13] for a formal semantics of its static and dynamic aspects). Qualitative analysis using model-checking techniques is addressed in [2].

In SRML, the orchestration of services is expressed in terms of a number of components that are connected to each other through interaction protocols and jointly execute a (distributed) business process. The configuration of this business process may change at run time as the discovery of required services is triggered. Interactions are expressed in terms of a number of events that occur according to a conversational protocol that captures typical peer-to-peer business interconnections. SRML allows modellers to describe a service-oriented application by defining the structure of the model, including components and abstract references to services that will be possibly discovered at run-time. The architectural definition has been inspired by the Service Component Architecture [11], a set of specifications proposed by an industrial consortium that describe a middleware-independent model for building software applications over SOAs. Differently from SCA, SRML offers primitives to provide a high-level behavioural description of each element of the model.

The extensions that we propose in this paper allow us to capture different kinds of delays that can occur during service provision: the time that components take to process events and perform computations on their local states, the time that wires take to transmit events between parties, and the delays with which the SOA middleware performs discovery, selection and binding of services. SRML offers a number of advantages in the analysis of the time-related properties of a service-oriented model, namely in helping to (i) identify the delays deriving from the structure of the model, (ii) identify the delays due to the behaviour of each single entity and (iii) investigate the inter-relation between (i) and (ii), determining the influence of the delays of each part of the system in the whole business process. From a methodological point of view, the aim is to validate a model against a number of time-related requirements (e.g., upper bounds on the delays between interactions) and to suggest improvements in both the overall structure (e.g., adding or substituting components) and the individual components themselves to meet such requirements. The analysis offers a modular specification of the time-related properties that each component/service should meet.

In order to analyse our models for the properties that can be ensured of the services that implement them, we use the markovian process algebra PEPA [17]. PEPA was adopted in SENSORIA to perform quantitative analysis in general because it has a well-defined meaning and is supported by a range of efficient software tools that can perform the analysis needed for service-level agreements. Our experience showed that PEPA is indeed well suited to give a timed account of behaviour as specified in SRML and evaluate the service-level agreements that are of interest to us here.

We illustrate our approach by using a simplified version of a case study on mortgage-brokerage services that we have developed within SENSORIA [5, 13]. Another example was developed around stock trading as part of an MSc thesis [23].

2 Preliminaries

Section 2.1 illustrates the architectural and behavioural aspects of SRML that are necessary to understand the delays modelled in Section 4. PEPA provides a very concise set of constructs, which are briefly summarized in Section 2.2.

2.1 SRML

SRML provides primitives for modelling composite service-oriented applications by orchestrating interactions among components and services provided by external parties. The SRML unit of design is called a "module". A module M consists of:

- A set $nodes(M)$ and a set $edges(M)$ where each edge w is a set of two nodes $w : n \leftrightarrow m$. Edges are also called "wire-interfaces".
- $nodes(M)$ consists of four partitions: (1) a set $provides(M)$ consisting of one "provides-interface", (2) a set $uses(M)$ of "uses-interfaces", (3) a set $requires(M)$ of "requires-interfaces", and (4) a set $components(M)$ of "components".
- An internal configuration policy that, for every $n \in requires(M)$, consists of $trigger(n)$ – a pair (m, e) where $m \in nodes(M)$ and e is an event published by m as part of an interaction between the components that bind to n and m.
- An external configuration policy represented as a constraint system $cs(M)$ involving a set $sla(M)$ of constraints.

A labelling function $label_M$ associates a specification with every node and edge of M. All the specifications involve a signature $sign(label_M(n))$ that consists of the set of interactions that the entity n can engage in, and a definition of the behavioural protocol that is required of those interactions. Different formalisms are used for defining the protocol depending on the nature of the node, as described below. The c-semiring approach to constraint satisfaction and optimisation [4] is used for the external configuration policy. Constraints are used for modelling the SLAs involved in service discovery and selection. They involve a number of quality-of-service parameters, including delays.

Business Roles. The behavioural description of every node $n \in components(M)$ is given as a 'business role' — which describe an *orchestration* of the events associated with the interactions in $sign(label_M(n))$. Orchestrations are modelled as state machines and represented as UML statecharts involving two types of nodes, 'state-nodes' and 'transition-nodes', as illustrated in Fig 3. *SRML transitions* are represented in the statechart by a node reflecting the fact that, according to our computational model, a transition involves two steps. The first step is executed by the trigger of the transition, which leads to an intermediate state where the component processes the reaction to the trigger. Depending on the trigger, the SRML transition may branch to a number of exit states. We denote by $(trans(label_M(n)))$ the set of state transitions of the node n.

Business Protocols. The specifications used for provides- and requires-interfaces are called 'business protocols' and consist of collections of *statements* that capture typical patterns of conversational behaviour that arise in service provision. The semantics of such statements is expressed as sentences of the temporal logic UCTL [18], which facilitates the use of model-checking techniques for analysing functional properties of

the services specified through the modules [2]. Notice that by $e?$ we refer to the processing of event e and, by $e!$ to its publication. If we want to refer generically to either the processing or the publication of event e we use $e*$.

- **initiallyEnabled** $e?$: the event e is never discarded until it is executed.
- e_1* **enables** $e_2?$ **until** e_3*: after e_1* happens, and while e_3* does not happen, e_2 will not be discarded. Also, e_2 cannot be executed neither before e_1* nor after e_3*.
- e_1* **ensures** $e_2!$: e_2 will be published after, but not before, e_1* happens.

Layer Protocols. The specifications associated with uses-interfaces are called 'layer protocols'. They specify synchronous interactions with persistent components, for instance to obtain or store information in database systems.

Interaction Protocols. Every edge $n \leftrightarrow m$ is associated with a 'connector' $\langle \mu_A, P, \mu_B \rangle$, where P is an 'interaction protocol' with two 'roles' $roleA_P$ and $roleB_P$, and μ_A (resp. μ_B) is an attachment between $roleA_P$ and $sign(label_M(n))$ (resp. $roleB_P$ and $sign(label_M(m))$). Interaction protocols are specifications of the way wires coordinate interactions between parties. In order to simplify the treatment of timing issues, we restrict ourselves to linear protocols, which ensure that events are transmitted to the corresponding co-party in the order that they are received, but with a possible delay. More specifically, we restrict ourselves to the case where a connector is a collection $pairs(w)$ of pairs $\langle a, b \rangle$ where a is an interaction of $sign(label_M(n))$ and b is an interaction of $sign(label_M(m))$ with opposite types, i.e. the types of a and b must be one of $\langle$**s&r**, **r&s**$\rangle$, $\langle$**snd**, **rcv**$\rangle$, $\langle$**ask**, **rpl**$\rangle$, $\langle$**tll**, **prf**$\rangle$ or their dual.

The pair $\langle$**s&r**, **r&s**$\rangle$ and its dual capture asynchronous conversational interactions; $\langle$**snd**, **rcv**$\rangle$ is for asynchronous one-way interactions, i.e. when a reply is not expected; $\langle$**ask**, **rpl**$\rangle$ and $\langle$**tll**, **prf**$\rangle$ are for synchronous interactions, the first for queries and the second for updates.

A number of events are associated with an asynchronous conversation interaction a: a🕭 is the initiation event, a✉ is the reply, a✓ is the commit, a✘ is the cancel event, and a✞ is the compensation.

2.2 PEPA

We perform quantitative analysis of SRML models by mapping to PEPA and using the PEPA analysis tools [22, 10]. PEPA is a timed process algebra in which model components perform activities with exponentially-distributed rates.

In practical applications, it is rarely the case that it is possible to obtain a complete response-time distribution of all services in the problem under study. It is far more likely that one will only know the average response time. In this setting, it is indeed correct to capture the inherent stochasticity in the system through a *exponential distribution*. The exponential distribution requires only a single parameter, the average response time.[3] We apply our modelling only in settings where the average response time is a meaningful quantity to use. For example, we do not model systems that have

[3] Other distributions would require knowledge of higher moments and other parameters which we do not have. We take care not to require too many parameters because finding each one accurately may require careful measurement or estimation.

a substantial component requiring a response from a single human participant because the great variance in human response time makes knowledge of the average response time alone insignificant for analysis purposes. This setting connects us to the rich theory of stochastic process including continuous-time Markov chains (CTMC), and a wealth of efficient numerical procedures for their analysis.

PEPA models consist of a parallel composition of sequential components and thus are finite-state by construction. Activities may be performed in isolation or in cooperation with another component. The term $(\alpha, r).P$ denotes a component which performs activity α at rate r and evolves to P. This can be performed in cooperation with a component $(\alpha, \top).Q$ which allows the other partner in the cooperation to determine the rate of the shared activity (the symbol $\top$ denotes this permission). In the term $P \underset{L}{\bowtie} Q$ components P and Q cooperate on the activities in the set L, but are free to proceed with other activities independently. Finally, a choice such as $(\alpha, p\lambda).P_1 + (\alpha, (1-p)\lambda).P_2$ performs activity α with rate λ and evolves to P_1 with probability p and P_2 with probability $1 - p$.

3 The Mortgage-Brockerage Case Study

In this section we present the case study that we use to illustrate our approach. Figure 1 illustrates the structure of the SRML module $GetMortgage$ that models a complex service able to find the best mortgage according to the personal data and preferences obtained from the client application.

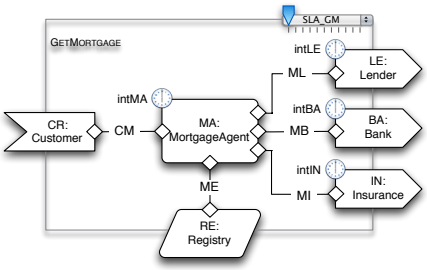

Fig. 1. Structure of the SRMLmodule $GetMortgage$

$GetMortgage$ includes (1) the provides-interface CR, which describes the interface of the service that is offered by the module, (2) one uses-interface RE that describes the interface to a persistent resource (a registry of trusted lenders that is consulted when selecting the lender), (3) three requires-interfaces, LE, BA, and IN, specifying the interfaces to the services that may need to be procured on the fly from external parties (the loan, the bank account and the insurance, respectively), and (4) one interface MA to the components that orchestrates the interactions among the parties. A number of wires establish interaction protocols between the different entities involved in the service. Each requires-interfaces has a $trigger$ event (e.g. $intLE$ for LE) that

launches the run-time discovery of a service that matches with the requires-interface. The discovery and selection process involves the negotiation of service level agreements according to the external configuration policy SLA_GM.

Figure 2 illustrates a fragment of the statechart diagram that defines the behaviour of *MortgageAgent*, involving two transitions $GetClientRequest$ and $GetProposal$. As an example of a business protocol, the statement $\mathbf{initiallyEnabled}\ askProposal🔔?$ is used in the business protocol $Lender$ to express that the lender should be ready to accept a request for a proposal from the moment it is bound to the broker.

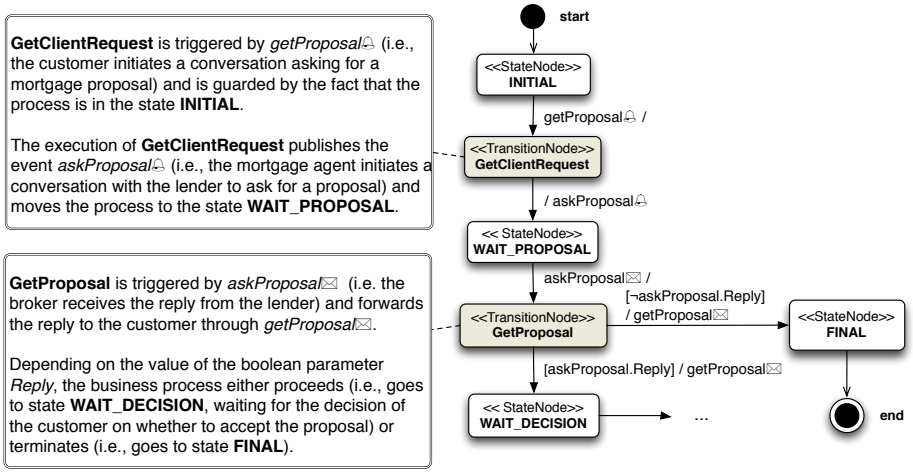

Fig. 2. Textual and graphical representation of a fragment of the behaviour of $MortgageAgent$

4 Timing Issues in SRML Models

In this section we show how SRML can be extended in order to model the delays involved in the business process through which a service is provided. We analyse properties such as the one discussed in Section 1 (i.e., certify that $GetMortgage$ ensures, up to a certain percentage, an upper bound to the delay of a mortgage proposal requested by a customer). This approach draws from the work reported in [23] and builds on the computational and coordination model that was presented in [1].

Given two events e_1 and e_2, we denote by $Delay(e_1, e_2)$ the time that separates their occurrences, e.g. $Delay(getProposal🔔, getProposal✉)$ in the example above. We follow the approach discussed in Section 2.2 and assume that such delays follow an exponential distribution of the form $F_{Delay(e_1,e_2)}(t) = 1 - e^{-rt}$. The rate r is associated with the entity that processes and publishes the events, and used as a modelling primitive in the proposed extension of SRML. Event-based selection of continuations in SRML becomes probabilistic choice in PEPA. We estimate the probability of the relative outcomes and use the resulting probabilities to weight the rates in the PEPA

model to ensure the correct distribution across the continuations. In this way all number distributions remain exponential and thus we can achieve probabilistic branching while remaining in the continuous-time Markovian realm.

We report below a number of delays that, according to the computation and coordination model of SRML, can affect service execution. The rates can be negotiated as SLAs with service providers in the constraint systems mentioned in Section 2.

Delays in components. Because they may be busy, components store the events they receive in a buffer where they wait until they are processed, at which point they are either executed or discarded. Two kinds of rates are involved in this process:

processingRate. This rate represents the time taken by the component to remove an event from the buffer. Different components may have different processing rates but all events are treated equally by the same component.

executionRate. This represents the time taken by the component to perform the transition triggered by the event, i.e. making changes to the state and publishing events. We assume that discarding an event does not take time. Each transition declared in a business role has its own execution rate, which should be chosen taking into account the specific effects of that transition.

Delays of requires-interfaces. As already mentioned, requires-interfaces represent parties that have to be discovered at run time when the corresponding trigger becomes true. Two kinds of rates are involved in this process:

compositionRate. This rate applies to the run-time discovery, selection and binding processes as performed by the middleware, i.e. (1) the time to connect to a broker, (2) the time for matchmaking, ranking and selection, and (3) the time to bind the selected service. We chose to let different requires-interfaces have different composition rates in order to reflect the fact that different brokers may be involved, depending on the nature of the required external services.

responseRate. These are rates that apply to the responses that the business protocol requires of the external service through statements of the form $e_1 * \ ensures\ e_2!$. More specifically, we consider a rate $responseRate(e_1, e_2)$ for each such pair of events, which include $responseRate(a🔔, a✉)$ for every interaction a of type **`r&s`** declared in the business protocol.

Delays in wires. Each wire of a module has an associated transfer rate. As mentioned in Section 2, we are considering only interaction protocols that affect a linear transmission from one party to its co-party, and do not involve complex data transformation.

Delays in synchronous communication and resource contention. The interface of a resource consists of a number of synchronous interactions . We define a synchronisation rate for each such interactions and associate it with the events that resolve synchronisation requests by replying to a query or executing an operation.

In summary, we extend every module M with a time policy P that consists of several collections of rates. Each rate is a term of type $\mathbb{R}^+ \cup \{\top\}$, where $\top$ is the passive rate (i.e., the event with a passive rate occurs only in collaboration with another event, when this second event is ready):

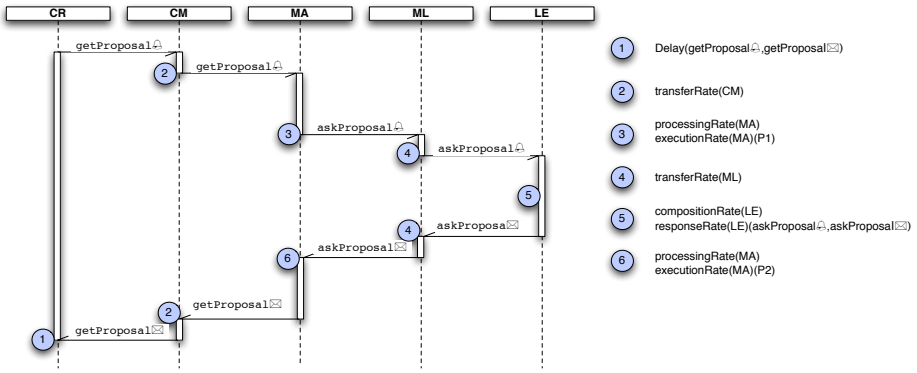

Fig. 3. Cascade of delays in a fragment of $GetMortgage$

- For every requires-interface $n \in requires(M)$
  - $compositionRate(n)$
  - $responseRate(n)(e_1, e_2)$ for every statement $(e_1 * \textbf{ensures } e_2!)$
- For every $w \in edges(M)$, $transferRate(w)$.
- For every $n \in components(M)$
  - $processingRate(n)$
  - $executionRate(n, P)$ for every transition $P \in trans(label_M(n))$
- for every $n \in components(M) \cup serves(M) \cup uses(M)$ and interaction i of type **`rpl`** and **`prf`**, $synchronisationRate(n)(i)$.

The sequence diagram in Figure 3 illustrates how the response time associated with $getProposal$ depends on the delays associated with the rates discussed in this section. The value of the rates that apply to components and wires to other components or uses-interfaces are fixed when the module is instantiated, i.e. when the interfaces are bound to components or network connections. The rates that involve requires-interfaces are fixed at run time, subject to SLAs.

5 Representing SRML Timing Issues as Stochastic Processes

In this section, we explain how a SRML module can be coded as a stochastic process so that the timing properties that derive from the timing policy of the module can be analysed using PEPA. This encoding involves several steps. First, the structure of the SRML module is decomposed into a PEPA configuration consisting of a number of PEPA terms. Each PEPA term corresponds to either a node or a wire of the original SRML model. In this way we can easily map the results of the quantitative analysis back to the original SRML specification. Second, the behavioural interface of each entity of the SRML model is encoded into a PEPA term, enabling to analyze the delays due to each single component. We use $\langle\langle m \rangle\rangle = t$ to express that the encoding of the SRML element m is the PEPA term t.

Encoding the signature. In SRML the signatures (sets of interactions) associated with specifications of different entities involved in a module are not assumed to be mutually disjoint. This is because we want to promote reuse, which is also why interconnections are established explicitly through wires. Therefore, because in PEPA interconnections are based on shared names, the first step of our encoding consists in renaming all the interactions to guarantee that the interconnections of the SRML model are properly represented by the scopes of action names in PEPA. We do so by defining, for every node n, its encoding signature $esign_M(n)$ obtained by prefixing each interaction name in $sign(label_M(n))$ with n.

The overall encoding $\langle\langle M \rangle\rangle$ of a module M is a cooperation process that includes one sequential component for each node of M, one sequential component for each edge of M, and one additional sequential component for each requires-interface:

$$\langle\langle M \rangle\rangle = \Big\|_{n \in nodes(M)} \langle\langle n \rangle\rangle \underset{L_1}{\bowtie} \Big\|_{w \in edges(M)} \langle\langle w \rangle\rangle \underset{L_2}{\bowtie} \Big\|_{n \in requires(M)} \langle\langle trigger\_n \rangle\rangle$$

The cooperation set L_1 includes all the interaction events associated with all the interaction names of all the nodes (note that the synchronisation event associated with synchronous interaction types has the same name as the interaction):

$$L_1 = \bigcup_{n \in nodes(M)} \bigcup_{i \in esign_M(n)} \{i, i\text{✉}, i\text{✓}, i\text{✘}, i\text{✞}, i\}$$

The cooperation set L_2 includes all the interaction events that act as triggers for requires-interfaces and, for each requires interface n, an event that controls the discovery process associated with n.

$$L_2 = \{m\_e : trigger(n) = (m, e), n \in requires(M)\} \cup \{discovery\_n : n \in requires(M)\}$$

Let n be a requires-interface with $trigger(n) = (m, e)$.

$$\langle\langle trigger\_n \rangle\rangle = P \quad \text{where} \quad P = (m\_e, \top).(discovery\_n, compositionRate(n)).P$$

This term models the delay due to the discovery process that occurs when the trigger becomes true. As shown in Section 5, a wire connecting a node to a requires-interface n must wait for the activity $discovery\_n$ before enacting any interaction with n.

Encoding components. The PEPA term corresponding to a component-interface n is obtained in two steps: (1) we refine the statechart that defines the business role associated with n, (2) we apply the translation provided by the PEPA toolset [9] to obtain the corresponding PEPA term.

The refinement of the statechart is performed in three substeps. First, the events that occur in the SRML statechart are translated using $esign_M(n)$ as defined in the previous paragraph. That is, given an asynchronous interaction a, $\langle\langle a\text{✉} \rangle\rangle = n\_a\text{✉}$, $\langle\langle a\text{✉} \rangle\rangle = n\_a\text{✉}$, $\langle\langle a\text{✓} \rangle\rangle = n\_a\text{✓}$, $\langle\langle a\text{✘} \rangle\rangle = n\_a\text{✘}$ and $\langle\langle a\text{✞} \rangle\rangle = n\_a\text{✞}$. For every synchronous interaction i, $\langle\langle i \rangle\rangle = n\_i$. The second step assigns a probability to the branches of the statechart that are associated with each SRML transition. More precisely, given a transition P with n branches P_{c_i}, we associate a probability p_{c_i} with each branch such that $\sum_{i=1..n} p_{c_i} = 1$. The designer can assign these probabilities taking into account

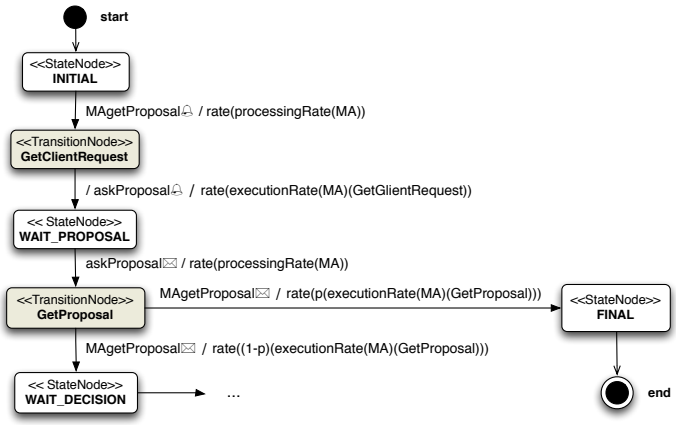

Fig. 4. Statechart for MA with the notation for performance analysis with PEPA

specific knowledge of the application domain, or decide for an equal probability $1/n$, or yet experiment with different values to analyse different possible behaviours. The third step consists in adding the rates. For every SRML transition P of a component n, the incoming arrow is assigned the rate $processingRate(n)$ and each branch P_{c_i} is assigned the rate $p_{c_i} * executionRate(n, P)$. Figure 4 illustrates the statechart diagram for the orchestration of MA, annotated with information on $executionRate$ for each transition.

Wires and Interaction Protocols. In order to encode a SRML edge $w : n \leftrightarrow m$, we consider first the case when none of the nodes involved is a requires-interface. In this case, all we have to do is to model the transfer of the events from one component to the other. As discussed in Section 2, every wire w defines a set of pairs of interactions $pairs(w)$. We define

$$\langle\!\langle w \rangle\!\rangle = \Big\|_{\langle a,b\rangle \in pairs(w)} \langle\!\langle \langle a, b\rangle \rangle\!\rangle$$

The encoding of the pairs of interactions depends on their types. Consider the case of $\langle$ **s&r** , **r&s** $\rangle$. In this case, the wire forwards the initiation, commit, cancel and revoke events from n to m and the reply back from m to n. We assign the delay $r = transferRate(w)$ to the second leg (delivery to the target):

$$\begin{aligned}
&\langle\!\langle \langle a, b\rangle \rangle\!\rangle = Q \\
&Q = (n_a, \top).(m_b, r).Q + (n_a\checkmark, \top).(m_b\checkmark, r).Q + (n_a\boldsymbol{\times}, \top).(m_b\boldsymbol{\times}, r).Q \\
&\quad +(n_a, \top).(m_b, r).Q + (m_b, \top).(n_a, r).Q
\end{aligned}$$

The encoding that applies to the other types of interaction is defined in a similar way. In the case of a one-way asynchronous protocol $\langle$ **snd** , **rcv** $\rangle$ we have:

$$\langle\!\langle \langle a, b\rangle \rangle\!\rangle = Q \quad \text{where} \quad Q = (n\_a, \top).(m\_b, transferRate(w)).Q$$

In the case of a synchronous interaction $\langle$ **ask** , **rpl** $\rangle$ we have:

$$\langle\!\langle \langle a, b\rangle \rangle\!\rangle = Q \quad \text{where} \quad Q = (n\_a, \top).(m\_b, synchronisationRate(m, b)).Q$$

The case of $\langle$ **tll** , **prf** $\rangle$ is identical. In the case of an edge connecting a requires-interface n, the encoding is:

$$\langle\!\langle w \rangle\!\rangle = (discovery\_n, \top). \Big\|_{\langle a,b\rangle \in pairs(w)} \langle\!\langle \langle a, b\rangle \rangle\!\rangle.$$

External-Interfaces. The encoding of requires-interfaces n is defined in terms of two processes that cooperate over the set L including all the events in $esign_M(n)$ and, for each $e \in esign_M(n)$, the actions $enables\_e$ and $disables\_e$. One of the processes (represented by the term S_n) encodes the statements that define the required behaviour of the external party. The other process (represented by the term E_n) controls the enabling and disabling of the interaction events in which the external party can be involved. That is,

$$\langle\!\langle n \rangle\!\rangle = S_n \underset{L}{\bowtie} E_n.$$

Let us consider each process in turn, starting with E_n. We need to control the incoming events, i.e. those received by the external party, all of which have a passive rate. The outgoing events are controlled by the components that receive them through the use of guards as discussed before.

$$E_n = \Big\|_{\substack{\text{type(i)}\\ \texttt{=rcv}}} E(n\_i\text{🔔}) \Big\|_{\substack{\text{type(i)}\\ \texttt{=s\&r}}} E(n\_i\text{✉}) \Big\|_{\substack{\text{type(i)}\\ \texttt{=r\&s}}} E(n\_i\text{🔔}) \mid E(n\_i\checkmark) \mid E(n\_i\texttt{✘}) \mid E(n\_i\text{✟})$$

$$E(e) = (enables\_e, \top).E'(e) \quad \text{where} \quad E'(e) = (e, \top).E(e) + (disables\_e, \top).E(e)$$

That is, $E(e)$ synchronises with the enabling of the event, after which it either executes it or disables it again. Consider now the term S_n. We have seen in Section 2.1 that the business protocol associated with a requires-interface n defines a set of statements $statements_M(n)$. We distinguish three kinds of statements: those that use the connective $initiallyEnabled$ – the set of which we denote by A_1; those that use $enablesUntil$ – the set of which we denote by A_2; and those of the form $ensures$ – the set of which we denote by A_3. Each kind of statement is encoded separately, leading to:

$$S_n = \Big\|_{s_1 \in A_1} \langle\!\langle s_1 \rangle\!\rangle \Big\|_{s_2 \in A_2} \langle\!\langle s_2 \rangle\!\rangle \Big\|_{s_3 \in A_3} \langle\!\langle s_3 \rangle\!\rangle \quad \text{where} \quad \langle\!\langle initiallyEnabled\, e_1? \rangle\!\rangle = enables\_\langle\!\langle e_1 \rangle\!\rangle$$

The enabling action for e_1 has no associated rate (i.e., it is an immediate action, as defined in [3]), because the activity does not involve any of the delays of a SRML module we want to analyze.

$$\langle\!\langle e_1 * \ enables\ e_2?\ until\ e_3 * \rangle\!\rangle = (\langle\!\langle e_1 \rangle\!\rangle, \top).P_1 + (\langle\!\langle e_3 \rangle\!\rangle, \top).P_2$$
$$P_1 = enables\_\langle\!\langle e_2 \rangle\!\rangle.(\langle\!\langle e_3 \rangle\!\rangle, \top).disables\_\langle\!\langle e_2 \rangle\!\rangle.\langle\!\langle e_1 * \ enables\ e_2?\ until\ e_3 * \rangle\!\rangle$$
$$P_2 = disables\_\langle\!\langle e_2 \rangle\!\rangle.\langle\!\langle e_1 * \ enables\ e_2?\ until\ e_3 * \rangle\!\rangle$$

We distinguish between the situation in which e_3 occurs first, disabling e_2, or e_1 occurs first, enabling e_2 until e_3 occurs. The enabling/disabling are immediate actions.

$$\langle\!\langle e_1 * \ ensures\ e_2! \rangle\!\rangle = (\langle\!\langle e_1 \rangle\!\rangle, \top).(\langle\!\langle e_2 \rangle\!\rangle, responseRate(n)(e_1, e_2))$$

That is, the execution of e_1 is followed by that of e_2 with a delay whose rate is given by an SLA variable as discussed in Section 3.

Uses-interfaces. As explained in Section 2.1, uses-interfaces provide synchronous interactions with components that offer a certain degree of persistence. For the nodes $n \in serves(M)$ (notice that synchronous interactions can occur more than once during one module instance):

$$\langle\langle n \rangle\rangle = \sum_{\forall i \in sign(label_M(n))} P_{n_i} \qquad \text{where} \qquad P_{n_i} = (n\_i, synchronisationRate(i)).P_{n_i}$$

6 Quantitative Analysis of Time Properties

In this section we discuss the quantitative analysis that we are able to perform on a SRML module by using the PEPA Eclipse Plug-in [22] and IPC [10], formal analysis components of the SENSORIA Development Environment[4]. First, we use the PEPA Eclipse Plug-in tool to generate the statespace of the derived PEPA configuration. We used the static analyser and qualitative analysis capabilites of this tool to determine that the configuration is deadlock free and has no unreachable local states in any component (no "dead code" in the model).

The analysis of a PEPA term encoding a SRML module is inexpensive because the statespace of the model is relatively small, meaning that the number of states of a module grows linearly with respect to the number of nodes. The reason is that the nodes of a SRML module do not execute independently but they wait for one another (i.e., typically not more than one at a time is active).

We performed the passage time analysis of the example illustrated in Figure 3, to investigate the probability of each possible delay between $CRgetProposal$🔔 and $CRgetProposal$✉. We conducted a series of experiments on our PEPA model to determine the answers to the following questions:

1. Is the advertised SLA *"80% of requests receive a response within 7 seconds"* satisfied by the system at present?
2. What is the bottleneck activity in the system at present (i.e. where is it best to invest effort in making one of the activities more efficient?)

The first question is answered by computing the cumulative distribution function (CDF) for the passage from request to response and determining the value at time $t = 10$. The second question is answered by performing a *sensitivity analysis*. That is, we vary each of the rates used in the model (both up from the true value, and down from it) and evaluate the CDF repeatedly over this range of values. The resulting graphs are shown in Figure 5 (the plus denotes the coordinate for 7 seconds and 80%).

Each of the graphs is a CDF which plots the probability of having completed the passage of interest by a given time bound. To determine whether the stated SLA is satisfied we need only inspect the value of this probability at the time bound. For the given values of the rates we find that it is the case that this SLA is not satisfied (Figure 5(a)).

In performing sensitivity analysis we vary each rate through a fixed number of possible values to see if we can identify an improvement which satisfies the SLA. We have

[4] Our aim is to discuss the proposed method rather than focusing on the results related to the specific case study, which is used for illustrative purposes.

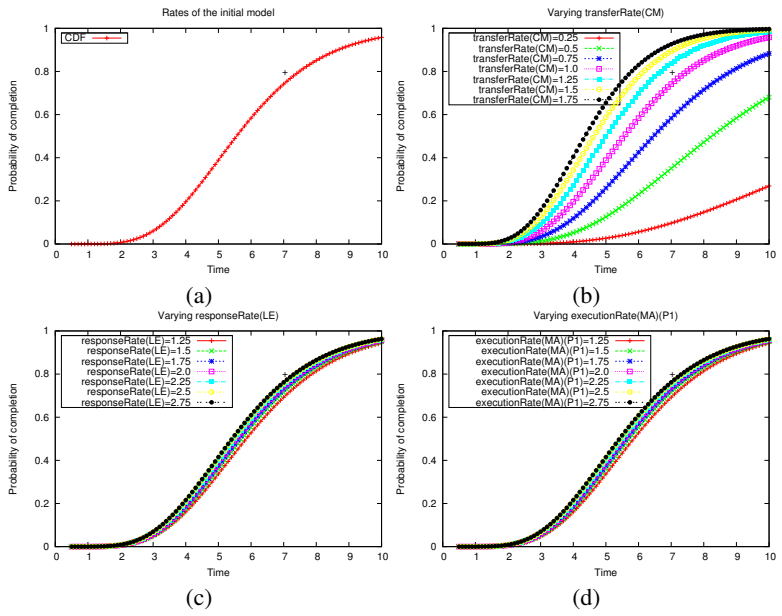

Fig. 5. Sensitivity analysis of response time distributions

begun by considering seven possible values here. Three of these are above the true value (i.e. the activity is being performed faster) and three are below (i.e. the activity is being performed slower). From the sensitivity analysis we determine (from Figure 5(b)) that variations in rate parameter $transferRate(CM)$ have the greatest impact on the passage of interest. Due to the structure of the model this rate controls the entry into the passage from request to response so delays here have a greater impact further through the passage. In contrast variations in rate parameter $responseRate(LE)$ (seen in Figure 5(c)) and $executionRate(MA)(P1)$ (seen in Figure 5(d)) have the least impact overall. Thus if seeking to improve the performance of the system we should invest in improving $coTransferRate$ before trying to improve $responseTime(LE)$. Figure 5(b) illustrates, for example, how the advertised SLA is satisfied by improving the value of $transferRate(CM)$ to 1.25. It is entirely possible that the sensitivity analysis will identify several ways in which the SLA can be satisfied. In this case the service stakeholders can evaluate these in terms of implementation cost or time and identify the most cost-effective way to improve the service in order to meet the SLA.

7 Related Work

Performance evaluation of service-oriented systems is of interest to many authors. We can only survey a few related papers here. In [15] the authors focus on tools for business process composition, performance engineering and dynamic system architectures. In this approach the authors may describe systems in Business Process Modelling No-

tation (BPMN) [24], their own high-level modelling notation or at a lower level of abstraction in a performance modelling formalism. The authors' tool suite generates code to implement this performance evaluation as a Java application, an Apache JMeter script, or in other formats. Stub code is generated for services with either delays according to a stochastic model or according to historical measurement data.

Our process calculus models are obtained by mapping from SRML but other authors prefer to sometimes work with timed process calculi directly (untimed calculi are used in [8, 6], timed calculi are used in [20, 19]). In other work process calculi models are generated from formalisms such as BPMN [21], WS-CDL [14, 7] or BPEL [16].

8 Concluding Remarks

We presented an encoding from SRML, which allows high level modelling of structural/behaviour aspects of service-oriented applications, into PEPA, which enables quantitative analysis of timing properties. The aim is to certify SLAs of complex services modelled in SRML, defining an upper bound, up to a certain probability, for the delay between pairs of events. Through sensitivity analysis of response time distributions, the tools offered by PEPA enable to vary rates for efficiency to improve the overall performance. We tested the proposed approach on a financial case study. A formal proof of correctness of the encoding based on the semantics of both languages is also under way, which should be quite straightforward as the delays were placed on SRML transitions according to the model presented in [1].

At the moment, the encoding is intended as a guideline for analysis performed by humans. As future work, we plan to increase the automation of the analysis by integrating the tools currently available for PEPA with the SRML editing environment. This would allow us, for example, to define orchestrations as statechart diagrams with the SRML editor and automatically transform them into PEPA terms.

We also plan to further investigate the implications of delays in expressing SLA constraints in SRML. We are currently working on a more accurate representation of interaction parameters, namely when they influence the choice of a branch in an orchestration. The aim is to represent them as probabilities, specifically the probability of receiving such a value through an interaction, and to associate them with the rates in a way that does not alter the analysis by introducing unwanted delays.

References

1. J. Abreu and J. L. Fiadeiro. A coordination model for service-oriented interactions. In *Coordination Languages and Models*, volume 5052 of *LNCS*, pages 1–16. Springer, 2008.
2. J. Abreu, F. Mazzanti, J. Fiadeiro, and S. Gnesi. A model-checking approach for service component architectures. In *Formal Methods for Open Object-Based Distributed Systems*, LNCS. Springer, 2009.
3. A. Argent-Katwala, J. Bradley, A. Clark, and S. Gilmore. Location-aware quality of service measurements for service-level agreements. In *Trustworthy Global Computing (TGC'07)*, volume 4912 of *LNCS*, pages 222–239. Springer, 2008.
4. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.

5. L. Bocchi, A. Lopes, and J. Fiadeiro. A use-case driven approach to formal service-oriented modelling. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'08)*, volume 17 of *CCIS*, pages 155–169. Springer, 2008.
6. M. Boreale, R. Bruni, R. D. Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.
7. M. Bravetti, I. Lanese, and G. Zavattaro. Contract-driven implementation of choreographies. In *TGC*, volume 5474 of *LNCS*, pages 1–18. Springer, 2008.
8. M. Bravetti and G. Zavattaro. Service oriented computing from a process algebraic perspective. *J. Log. Algebr. Program.*, 70(1):3–14, 2007.
9. C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens. Performance modelling with the unified modelling language and stochastic process algebras. In *Computers and Digital Techniques, IEE Proceedings*, volume 150, pages 107–120. IEEE, 2003.
10. A. Clark. The ipclib PEPA Library. In M. Harchol-Balter, M. Kwiatkowska, and M. Telek, editors, *Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST)*, pages 55–56. IEEE, Sept. 2007.
11. S. Consortium. Building Systems using a Service Oriented Architecture. Whitepaper, 2005.
12. J. L. Fiadeiro, A. Lopes, and L. Bocchi. A Formal Approach to Service Component Architecture. In *Web Services and Formal Methods*, volume 4184 of *LNCS*, pages 193–213. Springer, 2006.
13. J. L. Fiadeiro, A. Lopes, and L. Bocchi. An abstract model of service discovery and binding, 2009. Submitted (available from www.cs.le.ac.uk/people/jfiadeiro).
14. R. Gorrieri, C. Guidi, and R. Lucchi. Reasoning about interaction patterns in choreography. In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2005.
15. J. Grundy, J. Hosking, L. Li, and N. Liu. Performance engineering of service compositions. In *Service-Oriented Software Engineering*, pages 26–32, New York, NY, USA, 2006. ACM.
16. C. Guidi, R. Lucchi, N. Busi, R. Gorrieri, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *Proc. of International Conference on Service Oriented Computing'06*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.
17. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
18. M. Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An Action/State-Based Model-Checking Approach for the Analysis of Communication Protocols for Service-Oriented Applications. In *Formal Methods for Industrial Critical Systems*, volume 4916 of *LNCS*, pages 133–148. Springer, 2008.
19. R. D. Nicola, D. Latella, M. Loreti, and M. Massink. MarCaSPiS: a Markovian extension of a calculus for services. In *Proceedings of the workshop on Structural Operational Semantics*, Satellite workshop of ICALP, Reykjavik, Iceland, 2008.
20. D. Prandi and P. Quaglia. Stochastic COWS. In B. J. Krämer, K.-J. Lin, and P. Narasimhan, editors, *International Conference on Service Oriented Computing*, volume 4749 of *Lecture Notes in Computer Science*, pages 245–256. Springer, 2007.
21. D. Prandi, P. Quaglia, and N. Zannone. Formal analysis of BPMN via a translation into COWS. In *COORDINATION*, volume 5052 of *LNCS*, pages 249–263. Springer, 2008.
22. M. Tribastone. The PEPA Plug-in Project. In *Quantitative Evaluation of SysTems*, pages 53–54. IEEE, 2007.
23. V. Vankayala. Business process modelling using SRML (Advanced System Design - Project Dissertation), 2008.
24. S. A. White and D. Miers. *BPMN Modeling and Reference Guide*. Perfect Paperback, 2008.