PhD Dissertation

International Doctorate School in Information and Communication Technologies

DIT - University of Trento

Statistical model checking of Web Services

Stefano Schivo

Advisor:

Prof. Paola Quaglia

Università degli Studi di Trento

2010

Imagination, not intelligence, made us human.

Terry Pratchett

Abstract

In recent years, the increasing interest on service-oriented paradigm has given rise to a series of supporting tools and languages. In particular, COWS (Calculus for Orchestration of Web Services) has been attracting the attention of part of the scientific community for its peculiar effort in formalising the semantics of the de facto standard Web Services orchestration language WS-BPEL. The purpose of the present work is to provide the tools for representing and evaluating the performance of Web Services modelled through COWS. In order to do this, a stochastic version of COWS is proposed: such a language allows us to describe the performance of the modelled systems and thus to represent Web Services both from the qualitative and quantitative points of view. In particular, we provide COWS with an extension which maintains the polyadic matching mechanism: this way, the language will still provide the capability to explicitly model the use of session identifiers. The resulting Scows *is then equipped with a software tool which allows us to effectively perform model checking without incurring into the problem of state-space explosion, which would otherwise thwart the computation efforts even when checking relatively small models. In order to obtain this result, the proposed tool relies on the statistical analysis of simulation traces, which allows us to deal with large state-spaces without the actual need to completely explore them. Such an improvement in model checking performances comes at the price of accuracy in the answers provided: for this reason, users can trade-off speed against accuracy by modifying a series of parameters. In order to assess the efficiency of the proposed technique, our tool is compared with a number of existing model checking softwares.*

Keywords
stochastic process calculi, statistical model checking, performance evaluation, discrete-event simulation

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Context

In recent years the world of Web Services has been attracting the interest of part of the scientific community because of the success of the service oriented interaction paradigm on which it is based. A number of existing formal languages have been successfully applied for modelling Web Services [10, 7, 34], while some others have been proposed specifically for this aim [9, 8, 17]. Among these formalisms, COWS (Calculus for Orchestration of Web Services [30]) has been recently attracting the attention of a part of the scientific community for its peculiar effort in formalising the semantics of the de facto standard Web Services orchestration language WS-BPEL [6]. The calculus allows us to model different and typical aspects of services and Web Services technologies, such as multiple start activities, receive conflicts, routing of correlated messages, service instances and interactions among them. COWS can express the most common workflow patterns and has been shown to encode other process and orchestration languages [30]. One of the peculiarities of COWS is its *matching* mechanism, through which the concept of session identifier is modelled, effectively providing a way to correlate interactions belonging to the same session. We will now present a brief example illustrating the polyadic matching communication paradigm as defined in COWS. Consider the following service (i.e., COWS process) definition, in which sub-services have been labelled for convenience:

$$S = \underbrace{p!\langle m, n\rangle}_{s_1} \mid \underbrace{p?\langle x, y\rangle.\, s_2'}_{s_2} \mid \underbrace{p?\langle x, n\rangle.\, s_3'}_{s_3}$$

Here, three services are composed in parallel: s_1 represents a service trying to send a pair of names $\langle m, n\rangle$ via endpoint (i.e., channel) p, while s_2 tries to substitute variables x and y for the names received via a communication on endpoint p before continuing as service s_2', and s_3 is asking for a communication on endpoint p in which the second name is fixed to the predetermined name n in order to continue as service s_3'. Service s_3 can be thought of as being the continuation of a previously started service, in which the name n has been agreed as a session identifier (i.e., it will be included in all communications between services belonging to

the same session). The communication paradigm defined in COWS is based on the idea of *best matching*, which means that any given sending action (which will be called from now on *invoke action*) can communicate only with one of the receiving actions (named *request actions*) whose parameters match the best way with the ones being sent. This means that the number of variable substitutions generated by a communication involving invoke action $p!\langle m,n\rangle$ has to be minimal w.r.t. all request actions in the whole service. In our case this translates into the possibility to have *only one* communication happening in S: the one involving services s_1 and s_3. In fact, the communication between s_1 and s_2 would generate two substitutions (replacing x with m and y with n), while the communication between s_1 and s_3 generates only one substitution (replacing x with m and matching n with n). Thus, we can think of services s_1 and s_3 as belonging to the same session, identified by name n.
As a final remark, notice that if a third request action in the form $p?\langle n,y\rangle$ would have been available, it would not have been able to participate to the communication either, because it requires the name n to be in the first position of the tuple, while service s_1 offers m as first parameter.

1.2 The Problem

The purpose of this thesis is to provide the tools for representing and evaluating the performance of Web Services modelled through COWS. In order to do this, a stochastic version of COWS is required: such a language would allow dealing with the performance of the modelled systems, thus fastening the analysis of Web Services both from the qualitative and quantitative points of view.

Actually, a stochastic extension of COWS has already been proposed in [35], but the version presented there is based on a monadic fragment of the language (i.e., communications allow exchanging only one name at a time). One of the objectives of our work is to provide COWS with an extension which keeps the polyadic matching mechanism; this will in turn allow us to maintaining the capability to explicitly model the use of session identifiers. Moreover, while we are confident that a monadic version of COWS would maintain the same expressivity as the original calculus (in Appendix A we provide an encoding from COWS to its monadic fragment), the same relation between a monadic stochastic extension of COWS and its polyadic counterpart would be troublesome to be proven. Intuitively, at a certain point of the proof a single stochastic execution step (in the polyadic calculus) should be matched with a series of stochastic execution steps (in the monadic calculus). However, as a sequence of exponentially distributed delays is not exponentially distributed itself, the rates of the two cases would not correspond, and thus the correspondence could not be accomplished.

Once a stochastic version of COWS is available, a tool is needed in order to perform model checking on the models generated with such a calculus. An approach common to a number of stochastic calculi is to deduce the Continuous Time Markov Chain (CTMC [26]) from the transition system of a given model, and then use standard numerical solution and probabilistic model checking methods to obtain the required performance measures. However, a common problem which needs to be faced when generating a transition system of a process calculus term is the so-called state-space explosion. This problem comes from the fact that the composition of x parallel processes each with y different states in its transition system gives rise to a transition system with at most y^x states, thus requiring (as an upper bound) exponential time and resources to be computed. A solution often applied is to generate the state-spaces of the subprocesses in isolation, and then, taking advantage of the characteristics of some compact representation (e.g., in the case of stochastic calculi, Multi Terminal Binary Decision Diagrams [15] are sometimes employed), merge the resulting transition systems, obtaining a description of the whole state-space of the starting process using less-than-exponential time and memory resources. However, such compositional approach cannot be applied in our case, because the stochastic calculus presented in this thesis inherits the matching communication paradigm directly from COWS. As illustrated before, this paradigm requires the knowledge of the complete process in order to compute any transition. There are however other ways of mitigating the state-space explosion problem in the case of our calculus: see for example the Scows_lts tool by Igor Cappello [11, 12], in which the application of a congruence notion largely improves the tractability of the problem. An objective of this work is to explore the possibility to apply a different approach to the model checking of stochastic COWS models.

1.3 The Solution

As anticipated in Section 1.2, our work consists in the design of a stochastic extension of COWS which maintains the polyadic communication paradigm peculiar to the calculus. The resulting Scows is a stochastic calculus whose syntax adheres to the original version of COWS, to which only minor adjustments are made in order not to obtain infinitely branching transition systems from Scows models. This eventuality would in fact bar the possibility to obtain proper CTMCs from Scows models. The most important extension introduced in Scows is the association of a stochastic rate to each basic action of the language: these rates are the parameters of exponentially distributed random variables each describing the duration of the associated action. The stochastic operational semantics of Scows allows to obtain the possible actions available to a starting service, while the actual rates of such actions are computed through a formula which takes into account the communication capabilities of all entities involved in each interaction. Even if communication actions are binary operations, involving only two distinct services, other services can be indirectly involved into the rate computation: these are the services offering communication endpoints defined to be *in competition* with the ones actually communicating. Briefly explained, a service s_1 competes with service s_2 for a particular communication if s_1 can take the place of s_2 in the same communication. Notice that this way of computing transition rates is directly based on COWS communication paradigm, thus allowing us to keep intact as much as possible the foundational ideas of COWS.

In order to avoid the problem of state-space explosion, we present a model checking tool which does not require to compute the whole state space of a Scows model; instead, it works through a direct simulation-based approach inspired by [40] and [19]. What we do in practice is to generate a number of simulation traces starting from a given Scows model, and from these we obtain the required performance measures through statistical reasoning. In particular, the traces are generated by iteratively applying the stochastic operational semantics to the starting model, while the number of simulation traces needed to obtain a desired confidence is derived from the user-defined error thresholds. These thresholds can be tuned via a trade-off between execution time and approximation, thus allowing to obtain more precise results when time is not an (important) issue, while allowing to get likely estimations after a few minutes (or even seconds) of computation.

1.4 Innovative Aspects

The introduction of Scows allows to describe the quantitative aspects of Web Services, while still maintaining the capabilities provided by the polyadic communication paradigm peculiar to COWS. The proposed tool, providing a way to perform model checking on Scows models, and thus derive performance measures on the modelled Web Services, allows to fully exploit the potentials of the language without having to deal with complete state-space generation (and thus with the problem of state-space explosion). The approach used in our tool turns out to be even more useful than the generation of the whole transition system of a model (from which a CTMC is then to be derived) when the number of parallel components contained the Scows model at hand reaches high values. In practice, this allows us to extend the number of tractable Scows models, which has already been successfully increased by the Scows_amc tool.

The performance of the proposed tool is compared with other existing model checking tools. In order to have a common field of comparison between our tool and existing model checking tools, we first generate a CTMC from the Scows model at hand, and then give it as input to other model checking tools. As CTMCs are the favored input language for a substantial number of model checking tools, this choice enables us to compare our work with some of the most widespread tools. This comparison comes at the price of running into the state-space explosion problem while generating the CTMC, even if with greatly reduced effects, thanks to Scows_lts. This problem considerably increases the combined model checking time. However, the step of CTMC generation has to be taken into account when computing the total model checking time, as the comparisons we make are about model checking Scows models.

1.5 Overview of the thesis

The core of the thesis is composed of three parts, represented by Chapters 3, 4 and 5. In Chapter 3 we will introduce Scows, the stochastic extension of COWS, explaining in detail its operational semantics and the computation of stochastic rates for transition steps. Chapter 4 will deal with the Scows_amc tool which we introduce with the aim to permit model checking

Scows models. In that chapter the internals of the tool will be explained, explaining the theories on which the tool is based, and proposing some improvements which allow us to increase its performances. Finally, Chapter 5 will show a comparison between the Scows_amc tool and some model checking approaches based on CTMCs, thus allowing us to make some observations regarding the most useful situations in which the tool can be employed. As usual, the core of the thesis will be preceded by an introductory chapter in which we will explain the state of the art. The thesis will close with Chapter 6, in which conclusions on the present work will be drawn and some possible directions of development will be proposed. This thesis comprises also an appendix (Appendix A), in which an encoding from COWS to its monadic fragment is presented and shown to be operationally equivalent to the full version of the calculus. The objective of the work in Appendix A is to explain the cause of the evident complexity of the Scows language, which mainly comes from the choice of maintaining a polyadic matching mechanism when computing the stochastic rates of actions. Another appendix is included in the thesis (Appendix B), with the intent of showing a non-trivial Web Service modelled with Scows, ready for input to Scows_amc.

Chapter 2

State of the art

In this chapter we will present two of the main languages for service-oriented modelling: WS-BPEL [6] and BPMN [33]. Taking hint from the lack of formal definition in these languages, we will show an attempt to remedy to their shortcomings with the process algebra COWS [30]. Furthermore, we will present a series of extensions to the COWS language, together with a tool for the qualitative assessment of COWS models. We will then proceed with the presentation of some methods and tools for testing quantitative properties of systems. The quantitative testing of a system comprises the evaluation of the performances of the system after having assigned numerical duration to the actions represented by a model. The typical quantitative reasoning tools are based on continuous time Markov chains (CTMC, [26]), a mathematical formalism used to represent systems with stochastic behaviour.

2.1 Web service modelling languages

2.1.1 WS-BPEL

WS-BPEL [6] (Web Services - Business Process Execution Language - sometimes called BPEL) is the *de-facto* standard language for the modelling of web service orchestrations. It is an XML-based language (as nearly all web services-related languages), to guarantee the typical benefits of XML which are needed in the SOC ambient. However, not many WS-BPEL models can claim a great readability, because XML documents are generally easily manageable by humans only in the short size. In fact, although XML languages are often advertised as *human readable*, this characteristic rarely finds realisation, and only automatic manipulation is actually feasible. Thanks to a number of visual modelling tools, the actual WS-BPEL code rarely needs to be dealt with, allowing the designers to concentrate on the orchestration and coordination of their systems.

2.1.2 BPMN

BPMN [33] (Business Process Modelling Notation) is a graphical representation for web services orchestrations, whose purpose is to allow business-process technicians to easily model web service interactions. The notation is equipped with an intuitive set of symbols which represent WS-BPEL functionalities, but it suffers from lack of ontological completeness and clarity [37]. Moreover, it has been recently pointed out that BPMN and WS-BPEL suffer from conceptual mismatch in the representation of some control flow patterns [38]. Concluding, we do not consider the language mature enough to come out useful in real business process modelling, and thus it cannot be safely used as an effective visual representation of WS-BPEL models.

2.1.3 COWS

COWS is a process calculus first introduced in [30] as an evolution of ws-calculus [31], with the aim to model service-oriented orchestrations. The language is particularly interesting because its predecessor was created with the purpose to give a formal semantics to WS-BPEL (through the translation presented in [31]), and more work is being carried out with the aim to obtain a complete mapping of WS-BPEL into COWS [32]. The interest roused by COWS is also due to its mainly web services-oriented primitives, since many of its basic constructs come from the SOC world.

2.2 Stochastic modelling

2.2.1 Continuous-time Markov chains

Continuous-time Markov chains (CTMCs [26]) are a widely used formalism generated by the semantics of SPAs such as Stochastic COWS, and are at the basis of performance evaluation in many different areas.

Definition 1. *A continuous time Markov chain (CTMC) is a pair $C = (S, \mathbf{R})$ where*

- *S is a countable set of states*
- *$\mathbf{R} : S \times S \to \mathbb{R}_{\geqslant 0}$ such that $\sum_{s' \in S} \mathbf{R}(s, s')$ converges is a* rate matrix

Taken a CTMC $C = (S, \mathbf{R})$, the *exit rate* for state $s \in S$ is

$$E(s) = \sum_{s' \in S} \mathbf{R}(s, s') < \infty$$

We can compute the probability to move from state $s \in S$ to a state $s' \in S$ (such that $\mathbf{R}(s, s') > 0$) before t time units using the exponential distribution function:

$$P[s \to s' \text{ by } t \text{ time units}] = 1 - e^{-\lambda t}$$

where $\lambda = \mathbf{R}(s, s')$.
The probability that the transition $s \to s'$ is chosen between all transitions exiting from s (that is, the probability that the delay for going from s to s' "finishes before" any other delay of going from s to another state $s'' \neq s'$) is

$$P[s \to s'] = \frac{\mathbf{R}(s, s')}{E(s)}$$

2.3 Quantitative model checking

Probabilistic model checking of a stochastic model is the technique which allows us to measure the quantitative aspects of the model through probabilistic reasoning. In particular, if the model can be represented as a CTMC, a common probabilistic model checking approach allows us to define some properties against which the model is to be checked. Such properties are defined using a specification language suitable to the model: for example, in the case of CTMC, a suggested property specification language is CSL (Continuous Stochastic Logic, [5]), whose BNF grammar is presented in Table 2.1.

(state formula)	Φ	::=	$\mathtt{true} \mid \mathtt{false} \mid a \mid \neg\,\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid$
			$\mathcal{S}_{\bowtie\theta}\,[\Phi] \mid \mathcal{P}_{\bowtie\theta}\,[\varphi] \mid \mathcal{S}_{=?}\,[\Phi] \mid \mathcal{P}_{=?}\,[\varphi]$
(path formula)	φ	::=	$\mathcal{X}\,\Phi \mid \Phi\,\mathcal{U}\,\Phi \mid \Phi\,\mathcal{U}^{[t_{\min},t_{\max}]}\,\Phi$

Table 2.1: BNF grammar of the Continuous Stochastic Logic (CSL). Here, a is an atomic formula, $\theta \in [0, 1]$, $\bowtie \in \{<, \leqslant, >, \geqslant\}$ and $t_{\min}, t_{\max} \in \mathbb{R}$.

The semantics of CSL formulas defines how to verify if a particular state s of the CTMC satisfies a CSL state formula Φ (written $s \models \Phi$), and is presented in Table 2.2.

$s \models \mathtt{true}$	
$s \not\models \mathtt{false}$	
$s \models a$	iff a is an atomic formula evaluating to true in s
$s \models \neg\Phi$	iff $s \not\models \Phi$
$s \models \Phi_1 \wedge \Phi_2$	iff $s \models \Phi_1$ and $s \models \Phi_2$
$s \models \Phi_1 \vee \Phi_2$	iff $s \models \Phi_1$ or $s \models \Phi_2$
$s \models \mathcal{P}_{\bowtie\theta}\,[\varphi]$	iff $\mathit{Prob}\{\pi \in \mathit{Path}(s) \mid \pi \models \varphi\} \bowtie \theta$
$\pi \models \mathcal{X}\,\Phi$	iff Φ is true on the second state of π
$\pi \models \Phi_1\,\mathcal{U}\,\Phi_2$	iff $\exists x \in \mathbb{R}_{\geqslant 0}\,.\,(\pi(x) \models \Phi_2$ and $\forall y \in [0, x)\,.\,\pi(y) \models \Phi_1)$
$\pi \models \Phi_1\,\mathcal{U}^{[t_0,t_1]}\,\Phi_2$	iff $\exists x \in [t_0, t_1]\,.\,(\pi(x) \models \Phi_2$ and $\forall y \in [t_0, x)\,.\,\pi(y) \models \Phi_1)$

Table 2.2: Semantics of CSL formulas. The writing $s \models \Phi$ (respectively $\pi \models \varphi$) is read "state s satisfies state-formula Φ" (resp. "path π satisfies path-formula φ"), while $\pi(t)$ equals to the state which in path π is active at time t.

Chapter 3

Scows

In this chapter we introduce a stochastic extension of COWS, which will be used in the following as a base for the statistical model checking of Web Services.

The objective of the stochastic extension is to provide the modelling language COWS with a way to represent also quantitative aspects of Web Services, allowing thus to test their performance and draw inferences on the efficiency of the modelled entities. In order to do this, the base actions of COWS are enriched with a real value, called rate. Such rates are the parameters of exponentially distributed random variables, which model the duration of the corresponding actions.

As we will see, maintaining the matching mechanism peculiar to COWS will be one of the major challenges in adding stochastic capabilities to the language. This happens because we want to make sure that the communication capabilities of individual services (which in COWS are called "services") are taken into account when computing the rate of an interaction happening inside a composite service.

3.1 Stochastic semantics for COWS

Scows services are represented as basic activities composed following the grammar shown in Table 3.1, and based on three countable and disjoint sets of *entities*: the set of names $\mathcal{N}$ (with metavariables $m, n, o, p, m', n', o', p'$), the set of variables $\mathcal{V}$ (with metavariables x, y, z, x', y', z') and the set of killer labels $\mathcal{K}$ (with metavariables k, k'). Adhering to the conventions adopted in [30], we shall indicate elements of $\mathcal{N} \cup \mathcal{V}$ with u, v, w, u', v', w', and elements of $\mathcal{N} \cup \mathcal{V} \cup \mathcal{K}$ with d, d'. Notice that, differently from the original version of the language, we do not consider the class of "values" in our grammar, as in our case the management of values and expressions would lead to a more complicated operational semantics without adding significance to the stochastic extension.

We will indicate tuples by adding the characterising sign $\tilde{}$ to the metavariables: e.g., a tuple of input entities will be called $\tilde{u}$. A tuple made up by a single element can be written as the element itself: i.e., if $\tilde{u} = \langle x \rangle$, then x can be used instead of $\tilde{u}$. In a similar fashion we set that a sequence of delimiters can be rewritten as a delimiter for a tuple of entities: i.e., $[\,d_1\,][\,d_2\,]\ldots[\,d_j\,]s = [\,\tilde{d}\,]s$, where $\tilde{d} = \langle d_1, d_2, \ldots d_j\rangle$. Furthermore, we define the (possibly empty) delimiter tuple as

$$\tilde{t} ::= \tilde{n} \mid \varepsilon$$

The writing of the form $[\,\tilde{t}\,]s$ in the residual services of rule com (cfr. Table 3.2) will be intended as s alone if $\tilde{t} = \varepsilon$, and as $[\,\tilde{n}\,]s$ if $\tilde{t} = \tilde{n}$. Finally, the substitution of entities in tuples is written $\tilde{w}\{^n/_x\}$, and defined as the substitution of all occurrences of x in $\tilde{w}$ with n.

$$\begin{array}{rcl} s & ::= & (u.u!\tilde{u}, \delta) \mid g \mid s \mid s \mid \{\!|s|\!\} \mid (\mathbf{kill}(k), \lambda) \mid [\,d\,]s \mid \mathsf{S}(m_1, \ldots, m_j) \\ g & ::= & \mathbf{0} \mid (p.o?\tilde{u}, \gamma).\,s \mid g + g \end{array}$$

Table 3.1: The grammar of Scows.

Following a common approach in stochastic extensions [23, 36, 35], we associate a stochastic delay to each basic action of the language. These delays are determined via exponentially distributed random variables, the rates of which are included in the syntax of basic actions. The three basic actions are invoke $(u.u'!\tilde{u}, \delta)$, request $(p.o?\tilde{u}, \gamma)$ and kill $(\mathbf{kill}(k), \lambda)$, and represent respectively the capacity to send an output tuple $\tilde{u}$ through endpoint $u.u'$, receive an input on tuple $\tilde{u}$ through endpoint p, and request the termination of all (unprotected) services under the scope of killer label k; the rates of these activities are respectively δ, γ and λ. The basic activities are assembled through the classical combinators of parallel composition $s \mid s$, *guarded* choice $g + g$ (the classical nondeterministic choice, in which each service has to start with a request action) and service identifier $\mathsf{S}(m_1, \ldots, m_j)$. The last allows to express recursive behaviour and is associated to a corresponding service definition of the form $\mathsf{S}(n_1, \ldots, n_j) =$ s included in the model definition. The employment of *delimiters* $[\,d\,]s$ allows to define the scope of entities, and thus to delimit the extent of variable substitution, the validity of names and the limits for service termination. An entity under the scope of a delimiter is said to be *bound* by that delimiter. For example, in the service $S = [\,x\,]((p.o?x, \gamma).\,(p.o!x, \delta))$ the variable x is bound by the delimiter $[\,x\,]$, while the name p is not bound. A Scows service S is termed *closed* if every entity appearing in P is bound by a delimiter. In order to delimit the effects of service termination, in addition to killing label termination and adhering to the features present in COWS, we provide the protection construct $\{\!|\_|\!\}$ with the same semantics as in the original version of the language.

As the objective of this work is the automatic manipulation and model checking of Scows models, we need to constrain these models to produce finitely branching transition systems. It is for this reason that we diverge from the original syntax by using service identifiers instead of service replication; moreover, we assume that each service identifier is guarded in a valid starting service, i.e. it has to be contained in a service starting with a request action. In order to avoid name captures due to variable substitutions, we require that in any valid service no homonymy exists between both bound and free entities. This condition is supposed to hold in the starting service and kept valid through the use of functions s_dec and l_dec in rule $\mathsf{ser_{id}}$ (cfr. Tables 3.2 and 3.3). These functions are directly drawn from [35] and decorate bound names respectively in the transition label and in the residual service by adding to each entity a finite number of zeroes as superscript.

The form of a generic transition obtained from the application of the operational semantics rules of Tables 3.2 and 3.3 to a (closed) Scows service is $s \xrightarrow[\rho]{\vartheta(\alpha)} s'$. Here, α is the action label; ρ is the rate associated to the basic action, or a pair formed by the rates of the two interacting services if the action is a communication; and ϑ is a string of "$+_0$" and "$+_1$" created through the application of rules $\mathsf{choice_0}$ and $\mathsf{choice_1}$. The last approach is drawn from the one presented in [35] in order to distinguish between transitions which would otherwise be identical. As an example, consider the following service:

$$S = (p.o!n, \delta) \mid (p.o?n, \gamma).\,\mathbf{0} + (p.o?n, \gamma).\,\mathbf{0}$$

From S, two transitions lead to the same residual:

$$S \xrightarrow[{[\gamma,\delta]}]{+_0(p.o\ \lfloor\emptyset\rfloor\ n\ n)} \mathbf{0} \mid \mathbf{0} \ \text{ and } \ S \xrightarrow[{[\gamma,\delta]}]{+_1(p.o\ \lfloor\emptyset\rfloor\ n\ n)} \mathbf{0} \mid \mathbf{0}.$$

Notice that, if we had not established a way to distinguish between the two transitions, the total rate of the transitions labelled with $p.o\ \lfloor\emptyset\rfloor\ n\ n$ exiting from S would have been computed taking into account only one transition, and not two, thus leading to an inconsistent representation of the model.

In addition to the metavariable conventions mentioned before, in the operational semantics rules we will use a, b to range over $u, u', (u), (u')$ (i.e. possibly *bound* input entities) and h, i to range over $m, n, (m), (n)$ (i.e. possibly *bound* output names). In operational semantics labels, we call an entity "bound" if it falls under the scope of a delimiter in the service on which the semantics rules are being applied. For example, in the transition $[\,x\,](p.o?x, \gamma).\,s \xrightarrow[\gamma]{(p.o?(x))} s$ we use (x) in the label to state that the scope of x has been opened (by applying rule open), and thus that x is bound in the starting service.

Finally, in order not to obtain infinitely branching transition systems, we have to avoid the use of a congruence relation within the operational semantics rules. For this reason we deal with opening and closing of scopes with specific semantic rules: open, com and $\mathsf{del_{cl}}$. Rule open uses the function op (see Table 3.4) to actually modify transition labels, opening the scope of bound entities, and signaling their bound status in the resulting transition label. Rules com and $\mathsf{del_{cl}}$ handle the closing of scopes for names sent in a communication.

Notice that the given definition for rule open allows to extrude the scope of names for all three communicating labels. This allows us to take into account also the case in which the scope of the input variable of the request action contains the scope of the output name from the invoke action, which in turn contains *both* communicating services. As an example, consider the following service:

$$s = [\,x\,]([\,n\,]((p.o?x, \gamma).\,\mathbf{0} \mid (p.o!n, \delta_1)) \mid (q.m!x, \delta_2))$$

$$\text{(kill)}\ \frac{-}{(\mathbf{kill}(k), \lambda) \xrightarrow[\lambda]{(\dagger k)} \mathbf{0}} \qquad \text{(req)}\ \frac{-}{(p.o?\tilde{u}, \gamma).\, s \xrightarrow[\gamma]{(p.o?\tilde{u})} s} \qquad \text{(inv)}\ \frac{-}{(p.o!\tilde{n}, \delta) \xrightarrow[\delta]{(p.o!\tilde{n})} \mathbf{0}}$$

$$\text{(prot)}\ \frac{s \xrightarrow[\rho]{\vartheta(\alpha)} s'}{\{\!|s|\!\} \xrightarrow[\rho]{\vartheta(\alpha)} \{\!|s'|\!\}} \qquad \text{(choice}_0\text{)}\ \frac{g_1 \xrightarrow[\rho]{\vartheta(\alpha)} s}{g_1 + g_2 \xrightarrow[\rho]{+_0\vartheta(\alpha)} s} \qquad \text{(choice}_1\text{)}\ \frac{g_2 \xrightarrow[\rho]{\vartheta(\alpha)} s}{g_1 + g_2 \xrightarrow[\rho]{+_1\vartheta(\alpha)} s}$$

$$\text{(open)}\ \frac{s \xrightarrow[\rho]{\vartheta(\alpha)} s' \quad u \in d(\alpha)}{[\,u\,]s \xrightarrow[\rho]{\vartheta(\mathsf{op}(\alpha,u))} s'}$$

$$\text{(com)}\ \frac{s_1 \xrightarrow[\rho_1]{\vartheta_1(p.o?\tilde{a})} s_1{}' \quad s_2 \xrightarrow[\rho_2]{\vartheta_2(p.o!\tilde{h})} s_2{}' \quad \mathcal{M}\big(\tilde{a}, \tilde{h}\big) = (\sigma', \sigma, \tilde{t}) \quad \neg\Big(s_1 \mid s_2 \downarrow^{|\sigma'|+|\sigma|}_{p.o,\tilde{h}}\Big)}{s_1 \mid s_2 \xrightarrow[[\rho_1,\rho_2]]{(\vartheta_1,\vartheta_2)(p.o\, \lfloor\sigma'\rfloor\, \tilde{a}\, \tilde{h})} [\,\tilde{t}\,](s_1{}'\sigma \mid s_2{}')}$$

Table 3.2: Semantics for Scows (Part 1). We omit the symmetric counterpart for rule com.

$$(\mathsf{par_{conf}})\ \frac{s_1 \xrightarrow[\rho]{\vartheta(p.o\,\lfloor\sigma\rfloor\,\tilde{a}\,\tilde{h})} {s_1}' \quad \neg\left(s_2 \downarrow^{\#(\tilde{a},\tilde{h})}_{p.o,\tilde{h}}\right)}{s_1 \mid s_2 \xrightarrow[\rho]{\vartheta(p.o\,\lfloor\sigma\rfloor\,\tilde{a}\,\tilde{h})} {s_1}' \mid s_2} \qquad (\mathsf{par_{pass}})\ \frac{s_1 \xrightarrow[\rho]{\vartheta(\alpha)} {s_1}' \quad \alpha \neq p.o\,\lfloor\sigma\rfloor\,\tilde{a}\,\tilde{h} \quad \alpha \neq \dagger k}{s_1 \mid s_2 \xrightarrow[\rho]{\vartheta(\alpha)} {s_1}' \mid s_2}$$

$$(\mathsf{par_{kill}})\ \frac{s_1 \xrightarrow[\rho]{\vartheta(\dagger k)} {s_1}'}{s_1 \mid s_2 \xrightarrow[\rho]{\vartheta(\dagger k)} {s_1}' \mid halt(s_2)} \qquad (\mathsf{del_{pass}})\ \frac{s \xrightarrow[\rho]{\vartheta(\alpha)} s' \quad d \notin d(\alpha) \quad s{\downarrow_{kill}} \Rightarrow (\alpha = \dagger \text{ or } \alpha = \dagger k)}{[\,d\,]s \xrightarrow[\rho]{\vartheta(\alpha)} [\,d\,]s'}$$

$$(\mathsf{del_{cl}})\ \frac{s \xrightarrow[\rho]{\vartheta(p.o\,\lfloor\sigma \uplus \{(n)/x\}\rfloor\,\tilde{a}\,\tilde{h})} s'}{[\,x\,]s \xrightarrow[\rho]{\vartheta(p.o\,\lfloor\sigma\rfloor\,\tilde{a}\,\tilde{h})} [\,n\,]s'\{^{n}/_{x}\}} \qquad (\mathsf{del_{sub}})\ \frac{s \xrightarrow[\rho]{\vartheta(p.o\,\lfloor\sigma \uplus \{n/x\}\rfloor\,\tilde{a}\,\tilde{h})} s'}{[\,x\,]s \xrightarrow[\rho]{\vartheta(p.o\,\lfloor\sigma\rfloor\,\tilde{a}\,\tilde{h})} s'\{^{n}/_{x}\}} \qquad (\mathsf{del_{kill}})\ \frac{s \xrightarrow[\rho]{\vartheta(\dagger k)} s'}{[\,k\,]s \xrightarrow[\rho]{\vartheta(\dagger)} [\,k\,]s'}$$

$$(\mathsf{ser_{id}})\ \frac{s\{^{m_1,\ldots,m_j}/_{n_1,\ldots,n_j}\} \xrightarrow[\rho]{\vartheta(\alpha)} s' \quad \mathsf{S}(n_1,\ldots,n_j) = s}{\mathsf{S}(m_1,\ldots,m_j) \xrightarrow[\rho]{\vartheta(\mathrm{l\_dec}(\alpha))} \mathrm{s\_dec}(\alpha, s')}$$

Table 3.3: Semantics for Scows (Part 2). We omit the symmetric counterparts for rules $\mathsf{par_{conf}}$, $\mathsf{par_{pass}}$ and $\mathsf{par_{kill}}$.

$$
\begin{array}{rcl}
\mathsf{op}(p.o!\tilde{h}, n) & = & p.o!\tilde{h}' \text{ where } \tilde{h}' = \tilde{h}\{{}^{(n)}/_{n}\} \\
\mathsf{op}(p.o?\tilde{a}, x) & = & p.o?\tilde{a}' \text{ where } \tilde{a}' = \tilde{a}\{{}^{(x)}/_{x}\} \\
\mathsf{op}(p.o\ \lfloor\sigma\rfloor\ \tilde{a}\ \tilde{h}, n) & = & p.o\ \lfloor \mathsf{op_s}(\sigma, n)\rfloor\ \tilde{a}\ \tilde{h}' \text{ where } \tilde{h}' = \tilde{h}\{{}^{(n)}/_{n}\} \\
 & & \\
\mathsf{op_s}(\emptyset, n) & = & \emptyset \\
\mathsf{op_s}(\{{}^{n}/_{x}\} \uplus \sigma, m) & = & \begin{cases} \{{}^{(n)}/_{x}\} \uplus \mathsf{op_s}(\sigma, m) & \text{if } n = m \\ \{{}^{n}/_{x}\} \uplus \mathsf{op_s}(\sigma, m) & \text{if } n \neq m \end{cases}
\end{array}
$$

Table 3.4: Definition of functions op and $\mathsf{op_s}$ used for the opening of input, output and communication labels.

Here, the scope of n can be correctly extended via the application of rule open, generating the transition $s \xrightarrow[\ [\gamma,\delta_1]\ ]{\vartheta(p.o\ \lfloor\emptyset\rfloor\ x\ (n))} [\,n\,](\mathbf{0} \mid \mathbf{0} \mid (q.m!n, \delta_2))$. The derivation of such transition is shown in Table 3.5, where the application of rule open is marked by $(*)$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{-}{(p.o?x,\gamma).\,\mathbf{0} \xrightarrow[\gamma]{(p.o?x)} \mathbf{0}} \qquad \dfrac{-}{(p.o!n,\delta_1) \xrightarrow[\delta_1]{(p.o!n)} \mathbf{0}}}{(p.o?x,\gamma).\,\mathbf{0} \mid (p.o!n,\delta_1) \xrightarrow[\ [\gamma,\delta_1]\ ]{(p.o\ \lfloor\{n/x\}\rfloor\ x\ n)} \mathbf{0} \mid \mathbf{0}}\ (*)}{[\,n\,]((p.o?x,\gamma).\,\mathbf{0} \mid (p.o!n,\delta_1)) \xrightarrow[\ [\gamma,\delta_1]\ ]{(p.o\ \lfloor\{(n)/x\}\rfloor\ x\ (n))} \mathbf{0} \mid \mathbf{0}}}{[\,n\,]((p.o?x,\gamma).\,\mathbf{0} \mid (p.o!n,\delta_1)) \mid (q.m!x,\delta_2) \xrightarrow[\ [\gamma,\delta_1]\ ]{(p.o\ \lfloor\{(n)/x\}\rfloor\ x\ (n))} \mathbf{0} \mid \mathbf{0} \mid (q.m!x,\delta_2)}}{[\,x\,]([\,n\,]((p.o?x,\gamma).\,\mathbf{0} \mid (p.o!n,\delta_1)) \mid (q.m!x,\delta_2)) \xrightarrow[\ [\gamma,\delta_1]\ ]{(p.o\ \lfloor\emptyset\rfloor\ x\ (n))} [\,n\,](\mathbf{0} \mid \mathbf{0} \mid (q.m!n,\delta_2))}
$$

Table 3.5: Derivation of the example showing the application of rule open.

In the conclusion of rule com we set that the label for communication is of the form $p.o\ \lfloor\sigma'\rfloor\ \tilde{a}\ \tilde{h}$. Its meaning is as follows:

- $p.o$ is the endpoint on which the communication is happening;
- σ' is a partial function on the communication entities representing the possible substitution(s) to be applied when the appropriate delimiters are encountered (see part on $\mathcal{M}$ below);
- $\tilde{a}$ is the tuple of (possibly bound) input variables/names involved in the communication;
- $\tilde{h}$ is the tuple of (possibly bound) output names involved in the communication.

Notice that the fact that entities in $\tilde{a}$ and $\tilde{h}$ are bound influences only the application of rules $\mathsf{del_{cl}}$ and $\mathsf{del_{sub}}$, being transparent to all other semantics rules.

In order to determine which names or variables are involved in an action labelled by α, we use function $d(\alpha)$, which is defined in the same way as function $d(\alpha)$ in [30]. For example, we have that $d(p.o!\langle n,m\rangle) = \{p,n,m\}$, and $d(p.o\ \lfloor\{n/x\} \uplus \{m/y\}\rfloor\ \langle x,y\rangle\ \langle n,m\rangle) = \{n,m,x,y\}$ (notice the absence of p in the case of communication).

Function $\mathcal{M}$ (see Table 3.6) is also drawn from [30], and has been extended in order to correctly deal with scope extrusion.

$$\mathcal{M}(n,n) = (\emptyset,\emptyset,\varepsilon) \qquad \mathcal{M}(x,n) = (\{^n/_x\},\emptyset,\varepsilon)$$

$$\mathcal{M}((x),n) = (\emptyset,\{^n/_x\},\varepsilon) \qquad \mathcal{M}(x,(n)) = (\{^{(n)}/_x\},\emptyset,\varepsilon)$$

$$\mathcal{M}((x),(n)) = (\emptyset,\{^n/_x\},n)$$

$$\frac{\mathcal{M}(a_1,h_1) = (\sigma_1{}',\sigma_1,\tilde{t_1}) \quad \mathcal{M}\left(\tilde{a_2},\tilde{h_2}\right) = (\sigma_2{}',\sigma_2,\tilde{t_2})}{\mathcal{M}\left(\langle a_1\rangle :: \tilde{a_2}, \langle h_1\rangle :: \tilde{h_2}\right) = (\sigma_1{}' \uplus \sigma_2{}', \sigma_1 \uplus \sigma_2, \tilde{t_1} :: \tilde{t_2})}$$

Table 3.6: Definition of the matching function $\mathcal{M}$. Here $\uplus$ stands for *disjoint union* of substitution functions, and :: is used for concatenating two tuples.

Notice that the return value of $\mathcal{M}$ is a tuple $(\sigma',\sigma,\tilde{t})$ composed by:

- σ': the substitution(s) to be carried out as soon as the corresponding delimiter is encountered;
- σ: the substitution(s) to be carried out directly on the residual service of the communication;
- $\tilde{t}$: the name(s) to be restricted in the residual service.

A brief explanation of each base case tackled by $\mathcal{M}$ is given below:

- $\mathcal{M}(n,n)$: no substitution is needed, as input and output names coincide;
- $\mathcal{M}(x,n)$: a substitution will be performed when the delimiter for variable x will be encountered (rule $\mathsf{del_{sub}}$), so that the value of x will be substituted on its entire scope;
- $\mathcal{M}((x),n)$: input variable x is bound (i.e. its delimiter has already been encountered, and thus the scope of x is contained into the scope of n), so the substitution $\{^n/_x\}$ is performed directly on the residual service and no delimiter needs to be added;

- $\mathcal{M}(x,(n))$: output name n is bound (i.e. its scope is contained in the scope of x), so the substitution needs to be performed when the delimiter for x is found, and that delimiter will be substituted by a delimiter for n (rule $\mathsf{del_{cl}}$), effectively closing the scope of n;
- $\mathcal{M}((x),(n))$: both delimiters have already been encountered, so the substitution has to be performed directly on the residual service and the scope of n needs to be closed at the same time (so a delimiter for n is added to the residual service).

Cases in which the input *name* is bound ($\mathcal{M}((n),n)$ and $\mathcal{M}((n),(n))$) are not considered as they are not possible, while cases of the form $\mathcal{M}(n,m)$ with $n \neq m$ are not defined as no match applies.

To make notation shorter, we add a utility function for counting the number of substitutions needed for the match of two tuples, defined as follows:

$$\#(\tilde{a},\tilde{h}) = \begin{cases} |\sigma'|+|\sigma| & \text{if } \mathcal{M}\big(\tilde{a},\tilde{h}\big) = (\sigma',\sigma,\tilde{t}) \\ \infty & \text{otherwise} \end{cases}$$

where $|\sigma|$ is the number of substitutions performed by σ.

In order to correctly check that the matching components in a communication are the best possible, we make use of an adapted version of the predicate $s \downarrow^{c}_{p.o,\tilde{h}}$ defined in [30], meaning "service s performs a request action on endpoint $p.o$ which matches tuple $\tilde{h}$ with strictly less than c substitutions". Finally, predicate $s \downarrow_{kill}$ retains the original meaning of "service s can perform a kill action".

3.2 Stochastic rates

The stochastic execution step of a closed service is

$$s \xrightarrow[\rho]{\vartheta(\alpha)} s'$$

with either $\alpha = \dagger$ or $\alpha = p.o\ \lfloor\emptyset\rfloor\ \tilde{a}\ \tilde{h}$ (i.e., we accept as execution steps only killing and communication actions), ϑ is as described above (cfr. Sec. 3.1), and ρ represents the rates of the two participants to the communication or the rate of the kill action. As we will see in this section, the computation of the rate of a communication action involves the *apparent* rates of individual actions, i.e. the rates at which these actions are seen happening by an external observer (an observer that cannot distinguish between individual actions of the same kind). The computation of apparent rates is in turn based on the respective basic rates.

Calculating the rate of a communication

In [30] the communication between two COWS services is presented as an asynchronous, asymmetric event. Asynchronous because invoke actions are independent services, and asymmetric since a request action is adapted to its corresponding invoke action through the matching function $\mathcal{M}$. Following this same approach, we assume that request actions are dependent on invoke actions: i.e., the choice of the "transmitting" invoke action determines the set of possible "receiving" request actions. This happens because the set of request actions matching with the smallest number of substitutions (and thus eligible for communication with a given invoke action) is different from one invoke action to another. As this concept is foundational in our method for the computation of communication rates, we make use of an example to better explain it. Consider the service

$$\begin{aligned} S = [\,m,n,o,x,y\,]((&\underbrace{(p.q!\langle m,n\rangle,\delta_1)}_{\text{A}} \mid \underbrace{(p.q!\langle m,o\rangle,\delta_2)}_{\text{B}} \mid \underbrace{(p.q!\langle n,o\rangle,\delta_3)}_{\text{C}} \\ &\mid \underbrace{(p.q?\langle m,x\rangle,\gamma_1).\,\mathbf{0}}_{\text{D}} \mid \underbrace{(p.q?\langle y,o\rangle,\gamma_2).\,\mathbf{0}}_{\text{E}} \mid \underbrace{(p.q!\langle n,n\rangle,\delta_4)}_{\text{F}}) \end{aligned}$$

Using the notation P $\rhd$ Q to mean that services P and Q communicate (with P sending and Q receiving), we list below all communications made possible in S by the matching paradigm:

- A $\rhd$ D (matching m with m and substituting x by n)
- B $\rhd$ D (matching m with m and substituting x by o)
- B $\rhd$ E (substituting y by m and matching o with o)
- C $\rhd$ E (substituting y by n and matching o with o)
- (no service is able to receive F's tuple, so F does not communicate)

We can see from the example that the choice of the invoke action also determines the set of possible communications: when choosing to make a communication involving service A, there is only one possible communication (A $\rhd$ D), and the same holds for service C (with communication C $\rhd$ E), while when choosing service B the possible communications are two: B $\rhd$ D and B $\rhd$ E. Note that, as each of the two communications B $\rhd$ D and B $\rhd$ E makes one substitution, they are equally viable from the non-stochastic point of view, while, as we will see, the stochastic rates of the two transitions add a bias to the choice between the two actions.

The rate of a communication event is obtained by multiplying the apparent rate of the communication by the probability to choose the two participants among all possible competitors for the same communication. As mentioned before, the apparent rate of the communication is computed taking into account the rates of the actions which would be considered as *of the same kind* by an external observer. We will consider two actions to be indistinguishable from the point of view of an external observer if these actions are competing to participate with the same role in the same communication. Adopting a classical way of approximating exponential rates [23], we take the apparent rate of a communication to be the minimum between the apparent rates of the participating services (i.e., the communication proceeds at the speed of the "slowest" of the participants). So, the ideal formula for the rate of a communication between two hypothetical services P and Q has the following form:

$$\texttt{rate}(\text{communication}) = \mathcal{P}(\mathrm{P} \cap \mathrm{Q}) \cdot \texttt{min}(\texttt{appRate}(\mathrm{P}), \texttt{appRate}(\mathrm{Q})) \tag{3.1}$$

where $\mathcal{P}(\mathrm{P} \cap \mathrm{Q})$ is the probability to have both services P and Q involved in the communication, and appRate (P) (resp. appRate (Q)) is to be intended as the apparent rate of the invoke (request) action computed considering the whole service containing P and Q.

As we consider request actions to be dependent on invoke actions, we compute the probability to choose a pair invoke-request via the conditional probability formula:

$$\mathcal{P}(\mathrm{P} \cap \mathrm{Q}) = \mathcal{P}(\mathrm{P}) \cdot \mathcal{P}(\mathrm{Q} \mid \mathrm{P})$$

This means that the probability to have a communication between invoke $\mathrm{P} = (p.o!\tilde{n}, \delta)$ and (best matching) request $\mathrm{Q} = (p.o?\tilde{u}, \gamma)$ is given by the product of the probability to have chosen P among all possible invoke actions on endpoint p and the probability to choose Q among the request actions made available by the choice of P (i.e., the request actions matching with P through the minimal number of substitutions). In the example above, the probability that a communication happens between B and D is calculated as follows:

$$\begin{aligned} \mathcal{P}(\mathrm{B} \cap \mathrm{D}) &= \mathcal{P}(\mathrm{B}) \cdot \mathcal{P}(\mathrm{D} \mid \mathrm{B}) \\ &= \frac{\delta_2}{\delta_1 + \delta_2 + \delta_3} \cdot \frac{\gamma_1}{\gamma_1 + \gamma_2} \end{aligned} \tag{3.2}$$

(notice that we do not take into account the rate δ_4 of service F, as F cannot take part into any communication), while the probability to choose communication between A and D is basically the probability to choose A (as D is the only request action matching with A):

$$\begin{aligned} \mathcal{P}(\mathrm{A} \cap \mathrm{D}) &= \mathcal{P}(\mathrm{A}) \cdot \mathcal{P}(\mathrm{D} \mid \mathrm{A}) \\ &= \frac{\delta_1}{\delta_1 + \delta_2} \cdot 1 \end{aligned}$$

Notice that we do not take into account the rate δ_3 of service C, as C cannot communicate with D ($n \neq m$) and thus cannot influence the communication.

Now we will explain how to compute the apparent rates of invoke and request actions involved in a communication, defined as the sum of all rates of invoke (request) actions in competition with the selected one. It is important to remember that we have to take into account the communication capabilities of request and invoke actions, as the matching rules determine which are to be considered the "competing" actions.

Apparent rate of an invoke action

The apparent rate of an invoke action taking part in a communication internal to service s is given by the sum of the rates of all invoke actions in s able to perform a communication on the same endpoint on which the communication happens. Function `inv` (see Table 3.7) is used to this aim. This function is applied to the whole service in which the communication happens, and, through recursive calls of the auxiliary function `inv'`, sums up the rates of all unguarded invoke actions on the given endpoint. Notice that the reference to function `sumRates` is used in order to determine if the invoke action under inspection activates at least one request, so that only actions with the actual capacity to communicate are taken into account. To this same end, the condition $\tilde{u} = \tilde{n}$ ensures that the invoke action can send its output tuple: indeed, the operational semantics rule inv (cfr. Table 3.2) allows only fully instantiated output tuples to be sent. As an example of application, referring to the example presented at the beginning of Subsection 3.2, we show the result of the computation of the apparent rate for service B involved in the communication B $\triangleright$ D:

$$\mathtt{appRate}(\mathrm{B}) = \delta_1 + \delta_2 + \delta_3 \tag{3.3}$$

Notice once more that δ_4 does not appear in the sum, as service F activates no request action.

$$\begin{array}{rcl}
\mathtt{inv}(s, p.o) & = & \mathtt{inv}'(s, s, p.o) \\[1ex]
\multicolumn{3}{c}{\mathtt{inv}'(s, (\mathbf{kill}(k), \lambda), p.o) = \mathtt{inv}'(s, (p'.o'?\tilde{u}, \gamma).\, s', p.o) = \mathtt{inv}'(s, \mathbf{0}, p.o) = 0} \\[1ex]
\mathtt{inv}'(s, (p'.o'!\tilde{u}, \delta), p.o) & = & \begin{cases} \delta & \text{if } \; p = p', o = o', \tilde{u} = \tilde{n} \text{ and} \\ & \quad \mathtt{sumRates}(s, \tilde{n}, p.o) \neq (0, \infty) \\ 0 & \text{otherwise} \end{cases} \\[1ex]
\mathtt{inv}'(s, s_1 \mid s_2, p.o) & = & \mathtt{inv}'(s, s_1, p.o) + \mathtt{inv}'(s, s_2, p.o) \\[1ex]
\mathtt{inv}'(s, g_1 + g_2, p.o) & = & \mathtt{inv}'(s, g_1, p.o) + \mathtt{inv}'(s, g_2, p.o) \\[1ex]
\mathtt{inv}'(s, [\, d\,]s', p.o) & = & \mathtt{inv}'(s, s', p.o) \\[1ex]
\mathtt{inv}'(s, \{\!|s'|\!\}, p.o) & = & \mathtt{inv}'(s, s', p.o) \\[1ex]
\mathtt{inv}'(s, \mathsf{S}(m_1, \ldots, m_j), p.o) & = & \mathtt{inv}'(s, s'\{{}^{m_1, \ldots, m_j}/_{n_1, \ldots, n_j}\}, p.o) \quad \text{if } \mathsf{S}(n_1, \ldots, n_j) = s'
\end{array}$$

Table 3.7: Definition of function `inv`.

Apparent rate of a request action

Definition 1. *Request action* $(p'.o'?\tilde{u}, \gamma)$ *is* activated *by invoke action* $(p.o!\tilde{n}, \delta)$ *if the communication between the services performing such actions is possible.*

So, in a specific service s, an unguarded request action $(p'.o'?\tilde{u}, \gamma)$ is activated by invoke action $(p.o!\tilde{n}, \delta)$ if $p = p'$, $o = o'$ and $\#(\tilde{u}, \tilde{n})$ is minimal w.r.t. any other unguarded request action $(p'.o'?\tilde{u}', \gamma')$ in s for which $\mathcal{M}(\tilde{u}', \tilde{n})$ is defined.

Definition 2. *The* best-matching set *for invoke action* $(p.o!\tilde{n}, \delta)$ *in service* s *is the set of all request actions activated by invoke action* $(p.o!\tilde{n}, \delta)$.

The apparent rate of a request action $(p.o?\tilde{u}, \gamma)$ involved in a communication inside service s is calculated as follows:

- consider a set $\mathcal{B}$ of best-matching sets, each relative to a different unguarded invoke action in s on endpoint p, and such that $(p.o?\tilde{u}, \gamma)$ is contained in all sets in $\mathcal{B}$;
- for each best-matching set $\mathcal{S} \in \mathcal{B}$, sum up the rates of all (request) actions contained in $\mathcal{S}$;
- multiply each of such sums by the probability to select the corresponding activating invoke action;
- sum all results obtained this way to obtain the apparent rate of $(p.o?\tilde{u}, \gamma)$ in s.

Notice that we compute the apparent rate of a request action averaging it through the probabilities to choose each of its activating invoke actions.

As an example to clarify the computation explained above, consider again communication B $\rhd$ D from the example introduced at the beginning of Subsection 3.2. The apparent rate of request action D depends from two best-matching sets: {D} and {D, E}, activated respectively by A and B. The apparent rate of D results therefore as follows:

$$\texttt{appRate}\,(\mathrm{D}) = \underbrace{\frac{\delta_1}{\delta_1 + \delta_2}}_{\mathcal{P}(\mathrm{A})} \cdot \gamma_1 + \underbrace{\frac{\delta_2}{\delta_1 + \delta_2}}_{\mathcal{P}(\mathrm{B})} \cdot (\gamma_1 + \gamma_2) \tag{3.4}$$

where we have highlighted the probabilities to select the activating invoke actions for each best-matching set to which D belongs. Please notice that, as service C is incompatible with D, its rate is not taken into account when computing $\mathcal{P}(\mathrm{A})$ and $\mathcal{P}(\mathrm{B})$, as C would not influence the communication.

Putting together formulae (3.1), (3.2), (3.3) and (3.4), the rate of communication B $\rhd$ D results as follows:

$$\begin{aligned}\texttt{rate}\left(S \xrightarrow[{[\gamma_1,\delta_2]}]{\vartheta(p.q\,\lfloor\emptyset\rfloor\,\langle m,x\rangle\,\langle m,o\rangle)} S'\right) &= \frac{\delta_2}{\delta_1+\delta_2+\delta_3} \cdot \frac{\gamma_1}{\gamma_1+\gamma_2} \cdot \texttt{min}\left(\delta_1+\delta_2+\delta_3, \frac{\delta_1}{\delta_1+\delta_2} \cdot \gamma_1 + \frac{\delta_2}{\delta_1+\delta_2} \cdot (\gamma_1+\gamma_2)\right) \\ &= \frac{\delta_2}{\delta_1+\delta_2+\delta_3} \cdot \frac{\gamma_1}{\gamma_1+\gamma_2} \cdot \texttt{min}\left(\delta_1+\delta_2+\delta_3, \frac{\delta_1 \cdot \gamma_1 + \delta_2 \cdot (\gamma_1+\gamma_2)}{\delta_1+\delta_2}\right)\end{aligned}$$

where $S' = [\,m, n, o, x, y\,](\mathrm{A} \mid \mathbf{0} \mid \mathrm{C} \mid \mathbf{0} \mid \mathrm{E} \mid \mathrm{F})\{^{o}\!/\!_{x}\}$.

Rate of a communication

Generalising from the example presented above, we can finally state the formula for the calculation of the rate of a communication happening in service s on endpoint p between the receiving service performing request action $p.o?\tilde{a}$ and the sending service performing invoke action $p.o!\tilde{h}$:

$$\texttt{rate}\left(s \xrightarrow[{[\gamma,\delta]}]{\vartheta(p.o\,\lfloor\emptyset\rfloor\,\tilde{a}\,\tilde{h})} s'\right) = \frac{\delta}{\texttt{inv}(s, p.o)} \cdot \frac{\gamma}{\texttt{req}(s, p.o, \tilde{h}, \#(\tilde{a}, \tilde{h}))} \cdot \texttt{min}\left(\texttt{inv}(s, p.o), \frac{\texttt{aR}(\texttt{bms}(s), p.o, \tilde{a})}{\texttt{aInv}(s, p.o, \tilde{a})}\right) \tag{3.5}$$

where γ and δ are respectively the basic rate of request and invoke actions, while functions `inv`, `aInv`, `req`, `bms`, `sumRates` and `aR` are briefly explained here below and more precisely defined in the respective tables.

$\mathtt{inv}(s, p.o)$ (cfr. Table 3.7) computes the sum of all the rates of invoke actions in s which can perform a communication on endpoint p. The actions activating no request action are not taken into account, as they do not influence any communication. Through this function we compute the probability to choose action $(p.o!\tilde{h}, \delta)$ for communication inside service s as $\frac{\delta}{\mathtt{inv}(s,p.o)}$.

$\mathtt{req}(s, p.o, \tilde{h}, c)$ (cfr. Table 3.8) is built in a way similar to $\mathtt{inv}$, and returns the sum of all rates of unguarded request actions in s on endpoint p matching with given tuple $\tilde{h}$ through exactly c substitutions. These actions are the ones competing with the selected request action $(p.o?\tilde{a}, \gamma)$ for the communication at hand. Such sum allows us to compute the probability to choose that specific request action among all competitors: $\frac{\gamma}{\mathtt{req}(s,p.o,\tilde{h},c)}$. Notice that the fact that c is minimal (and thus that the selected request action is best-matching) comes directly from the existence of the communication transition itself.

$\mathtt{bms}(s)$ (cfr. Table 3.9) for each unguarded invoke action $(p.o!\tilde{n}, \delta)$ in s, returns a pair of the form $((\Gamma, c), (p.o!\tilde{n}, \delta))$, where (Γ, c) is the result of $\mathtt{sumRates}(s, \tilde{n}, p.o)$ (see below), with Γ standing for the sum of all rates of the best-matching set for action $p!\tilde{n}$, and c representing the number of substitutions performed by the request actions belonging to such best-matching set. The resulting set of pairs $B = \mathtt{bms}(s)$ is used in $\mathtt{aR}(B, p.o, \tilde{a})$ in order to compute the apparent rate of request action $p?\tilde{a}$ as described in Subsection 3.2.

$\mathtt{sumRates}(s, \tilde{n}, p.o)$ (cfr. Table 3.11) analyses service s and returns a pair of numbers (Γ, c), where Γ is the sum of the rates of all request actions in the best-matching set for invoke action $p.o!\tilde{n}$ and c is the number of substitutions through which these request actions match. Notice that, by the definition of best-matching set, c is the minimal number of substitutions with which any request action can match with $p.o!\tilde{n}$, and thus the sum of these rates is the apparent rate of a request action able to communicate only with $p.o!\tilde{n}$.

$\mathtt{aR}(B, p.o, \tilde{a})$ (cfr. Table 3.10) computes the sum of the product $\Gamma \cdot \delta$ for each pair of the form $((\Gamma, c), (p.o!\tilde{n}, \delta))$ in B and is used in order to compute the total apparent rate of request action $p?\tilde{a}$ defined as the average of the apparent rate of the request action (the sum of the rates of all request actions which could take its place in a communication) for each invoke action activating it. The average is weighted with the probabilities to choose each activating invoke action: e.g., the probability to choose $(p.o!\tilde{n}, \delta)$ is $\frac{\delta}{\mathtt{aInv}(s,p.o,\tilde{a})}$, where s is the whole service.

$\mathtt{aInv}(s, p.o, \tilde{a})$ (cfr. Table 3.12) computes the sum of the rates of all invoke actions in s whose best matching set contains the request action $p.o?\tilde{a}$. Please notice that, although it behaves similarly to $\mathtt{inv}(s, p.o)$, $\mathtt{aInv}$ uses a more specific clause when encountering an invoke action: the corresponding rate is added to the sum only if that action activates request action $p.o?\tilde{a}$.

$$\mathtt{req}((\mathbf{kill}(k),\lambda), p.o, \tilde{h}, c) = \mathtt{req}((p'.o'!\tilde{u},\delta), p.o, \tilde{h}, c) = \mathtt{req}(\mathbf{0}, p.o, \tilde{h}, c) = 0$$

$$\begin{aligned}
\mathtt{req}((p'.o'?\tilde{u},\gamma).\,s, p.o, \tilde{h}, c) &= \begin{cases} \gamma & \text{if } p = p', o = o' \text{ and } c = \#(\tilde{u},\tilde{h}) \\ 0 & \text{otherwise} \end{cases} \\
\mathtt{req}(s_1 \mid s_2, p.o, \tilde{h}, c) &= \mathtt{req}(s_1, p.o, \tilde{h}, c) + \mathtt{req}(s_2, p.o, \tilde{h}, c) \\
\mathtt{req}(g_1 + g_2, p.o, \tilde{h}, c) &= \mathtt{req}(g_1, p.o, \tilde{h}, c) + \mathtt{req}(g_2, p.o, \tilde{h}, c) \\
\mathtt{req}([\,d\,]s, p.o, \tilde{h}, c) &= \mathtt{req}(s, p.o, \tilde{h}, c) \\
\mathtt{req}(\{\!|s|\!\}, p.o, \tilde{h}, c) &= \mathtt{req}(s, p.o, \tilde{h}, c) \\
\mathtt{req}(\mathsf{S}(m_1,\ldots,m_j), p.o, \tilde{h}, c) &= \mathtt{req}(s\{^{m_1,\ldots,m_j}/_{n_1,\ldots,n_j}\}, p.o, \tilde{h}, c) \quad \text{if } \mathsf{S}(n_1,\ldots,n_j) = s
\end{aligned}$$

Table 3.8: Definition of function req.

$$\begin{aligned}
\mathtt{bms}(s) &= \mathtt{bms}'(s,s) \\
\mathtt{bms}'((p.o!\tilde{u},\delta), s) &= \begin{cases} \{(\mathtt{sumRates}(s,\tilde{n},p.o),(p.o!\tilde{n},\delta))\} & \text{if } \tilde{u} = \tilde{n} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

$$\mathtt{bms}'((p.o?\tilde{u},\gamma).\,s', s) = \mathtt{bms}'((\mathbf{kill}(k),\lambda), s) = \mathtt{bms}'(g_1 + g_2, s) = \mathtt{bms}'(\mathbf{0}, s) = \emptyset$$

$$\begin{aligned}
\mathtt{bms}'(s_1 \mid s_2, s) &= \mathtt{bms}'(s_1, s) \cup \mathtt{bms}'(s_2, s) \\
\mathtt{bms}'([\,d\,]s', s) &= \mathtt{bms}'(s', s) \\
\mathtt{bms}'(\{\!|s|\!\}, s) &= \mathtt{bms}'(s', s) \\
\mathtt{bms}'(\mathsf{S}(m_1,\ldots,m_j), s) &= \mathtt{bms}'(s_1\{^{m_1,\ldots,m_j}/_{n_1,\ldots,n_j}\}, s) \;\text{ if }\; \mathsf{S}(n_1,\ldots,n_j) = s_1
\end{aligned}$$

Table 3.9: Definition of function bms.

$$\begin{aligned}
\mathtt{aR}(\emptyset, p'.o', \tilde{a}) &= 0 \\
\mathtt{aR}(\{((\Gamma,c),(p.o!\tilde{n},\delta))\} \cup M, p'.o', \tilde{a}) &= \begin{cases} \mathtt{aR}(M, p'.o', \tilde{a}) + \Gamma\cdot\delta & \text{if } C \\ \mathtt{aR}(M, p'.o', \tilde{a}) & \text{otherwise} \end{cases}
\end{aligned}$$

$$\text{where } C = (p = p' \text{ and } \#(\tilde{a},\tilde{n}) = c)$$

Table 3.10: Definition of function aR. Condition C is used in order to check whether request action under consideration belongs to a best-matching set.

$$\texttt{sumRates}((p'.o'?\tilde{u},\gamma).\,s,\tilde{n},p.o) = \begin{cases} (\gamma,\#(\tilde{u},\tilde{n})) & \text{if } p = p' \text{ and } o = o' \\ (0,\infty) & \text{otherwise} \end{cases}$$

$$\texttt{sumRates}((p'.o'!\tilde{u},\delta),\tilde{n},p.o) = \texttt{sumRates}((\mathbf{kill}(k),\lambda),\tilde{n},p.o) = \texttt{sumRates}(\mathbf{0},\tilde{n},p.o) = (0,\infty)$$

$$\texttt{sumRates}([\,d\,]s,\tilde{n},p.o) = \texttt{sumRates}(s,\tilde{n},p.o)$$

$$\texttt{sumRates}(\{\!|s|\!\},\tilde{n},p.o) = \texttt{sumRates}(s,\tilde{n},p.o)$$

$$\frac{\texttt{sumRates}(s_1,\tilde{n},p.o) = (\gamma_1,c_1) \quad \texttt{sumRates}(s_2,\tilde{n},p.o) = (\gamma_2,c_2)}{\texttt{sumRates}(s_1 \mid s_2,\tilde{n},p.o) = (\gamma_3,c_3)} \ (*)$$

$$\frac{\texttt{sumRates}(g_1,\tilde{n},p.o) = (\gamma_1,c_1) \quad \texttt{sumRates}(g_2,\tilde{n},p.o) = (\gamma_2,c_2)}{\texttt{sumRates}(g_1 + g_2,\tilde{n},p.o) = (\gamma_3,c_3)} \ (*)$$

$$(*) \text{ where } c_3 = \texttt{min}(c_1,c_2) \text{ and } \gamma_3 = \begin{cases} \gamma_1 & \text{if } c_1 < c_2 \\ \gamma_2 & \text{if } c_2 < c_1 \\ \gamma_1 + \gamma_2 & \text{if } c_1 = c_2 \end{cases}$$

$$\texttt{sumRates}(\mathsf{S}(m_1,\ldots,m_j),\tilde{n},p.o) = \texttt{sumRates}(s\{^{m_1,\ldots,m_j}/_{n_1,\ldots,n_j}\},\tilde{n},p.o) \text{ if } \mathsf{S}(n_1,\ldots,n_j) = s$$

Table 3.11: Definition of function sumRates for the calculation of the sum of rates in best-matching sets.

$$\texttt{aInv}(s,p.o,\tilde{a}) = \texttt{aInv}'(s,s,p.o,\tilde{a})$$

$$\texttt{aInv}'(s,(p'.o'?\tilde{u},\gamma).\,s',p.o,\tilde{a}) = \texttt{aInv}'(s,(\mathbf{kill}(k),\lambda),p.o,\tilde{a}) = \texttt{aInv}'(s,\mathbf{0},p.o,\tilde{a}) = 0$$

$$\texttt{aInv}'(s,(p'.o'!\tilde{u},\delta),p.o,\tilde{a}) = \begin{cases} \delta & \text{if } p = p' \text{ and } \texttt{sumRates}(s,\tilde{n},p.o) = (\Gamma,\#(\tilde{a},\tilde{n})), \\ & \text{where } \Gamma > 0 \text{ and } \tilde{u} = \tilde{n} \\ 0 & \text{otherwise} \end{cases}$$

$$\texttt{aInv}'(s,s_1 \mid s_2,p.o,\tilde{a}) = \texttt{aInv}'(s,s_1,p.o,\tilde{a}) + \texttt{aInv}'(s,s_2,p.o,\tilde{a})$$

$$\texttt{aInv}'(s,g_1 + g_2,p.o,\tilde{a}) = \texttt{aInv}'(s,g_1,p.o,\tilde{a}) + \texttt{aInv}'(s,g_2,p.o,\tilde{a})$$

$$\texttt{aInv}'(s,[\,d\,]s',p.o,\tilde{a}) = \texttt{aInv}'(s,s',p.o,\tilde{a})$$

$$\texttt{aInv}'(s,\{\!|s'|\!\},p.o,\tilde{a}) = \texttt{aInv}'(s,s',p.o,\tilde{a})$$

$$\texttt{aInv}'(s,\mathsf{S}(m_1,\ldots,m_j),p.o,\tilde{a}) = \texttt{aInv}'(s,s'\{^{m_1,\ldots,m_j}/_{n_1,\ldots,n_j}\},p.o,\tilde{a}) \text{ if } \mathsf{S}(n_1,\ldots,n_j) = s'$$

Table 3.12: Definition of function aInv.

Resuming, the rate of a stochastic execution step of a closed Scows service is computed as follows:

$$\mathtt{rate}\left(s \xrightarrow[\rho]{\vartheta(\alpha)} s'\right) = \begin{cases} \frac{\delta}{\mathtt{inv}(s,p.o)} \cdot \frac{\gamma}{\mathtt{req}(s,p.o,\tilde{h},\#(\tilde{a},\tilde{h}))} \cdot \mathtt{min}\left(\mathtt{inv}(s, p.o), \frac{\mathtt{aR}(\mathtt{bms}(s),p.o,\tilde{a})}{\mathtt{aInv}(s,p.o,\tilde{a})}\right) & \text{if } \alpha = p.o \lfloor\emptyset\rfloor\, \tilde{a}\, \tilde{h} \\ & \text{and } \rho = [\gamma, \delta] \\ \rho & \text{if } \alpha = \dagger \end{cases}$$

3.3 Structural congruence in Scows

In this section, we will introduce a notion of structural congruence in Scows. This congruence is not to be applied directly in the operational semantics, as the introduction of a rule for the application of structural congruence goes against our objective to generate finitely branching transition systems. However, such a notion of congruence could become useful to identify services which, although grammatically different, originate the same behaviour.

Definition 3. *The structural congruence relation of* Scows *is the least congruence relation* $\equiv$ *satisfying all laws in Table 3.13.*

(prot$_1$)	$\{\!\!\{\mathbf{0}\}\!\!\} \equiv \mathbf{0}$	(prot$_2$)	$\{\!\!\{\{\!\!\{s\}\!\!\}\}\!\!\} \equiv \{\!\!\{s\}\!\!\}$
(prot$_3$)	$\{\!\!\{[\,d\,]s\}\!\!\} \equiv [\,d\,]\{\!\!\{s\}\!\!\}$	(delim$_1$)	$[\,d\,]\mathbf{0} \equiv \mathbf{0}$
(delim$_2$)	$[\,d_I\,][\,d_2\,]s \equiv [\,d_2\,][\,d_I\,]s$	(delim$_3$)	$s_1 \mid [\,d\,]s_2 \equiv [\,d\,](s_1 \mid s_2)$ if $d \notin \mathrm{fk}(s_2)$
(par$_{\mathsf{assoc}}$)	$s_1 \mid (s_2 \mid s_3) \equiv (s_1 \mid s_2) \mid s_3$	(par$_{\mathsf{comm}}$)	$s_1 \mid s_2 \equiv s_2 \mid s_1$
(par$_{\mathbf{0}}$)	$s \mid \mathbf{0} \equiv s$	(cho$_{\mathbf{0}}$)	$(p.o?\tilde{u},\gamma).\,s + \mathbf{0} \equiv (p.o?\tilde{u},\gamma).\,s$
(cho$_{\mathsf{comm}}$)	$g_1 + g_2 \equiv g_2 + g_1$	(cho$_{\mathsf{assoc}}$)	$(g_1 + g_2) + g_3 \equiv g_1 + (g_2 + g_3)$

Table 3.13: Structural congruence rules for Scows.

Notice that the rules are standard, with the only notable exception of a rule for idempotence of guarded sum, which does not apply in our case: as stated before (cfr. Sec. 3.1), we wish to distinguish between two services performing the same actions in a choice.

In [27] the authors show that the classical way of computing stochastic rates [36] for communication actions is in some cases imprecise, as it does not take into account the complete service in which the communication happens. This leads to the fact that two services differing only in the parenthesisation of their subcomponents, as could be $(P \mid Q) \mid R$ and $P \mid (Q \mid R)$ (which should be congruent by the associativity of parallel composition), give rise to transitions with *different* stochastic rates: such a problem negates the congruence law for parallel associativity. Nevertheless, our language still retains the associativity of parallel composition, as the communication rates are computed taking into account the whole service. This is due to the fact that COWS itself, through its polyadic matching rule, requires an analysis of the whole service before a transition is computed: the stochastic semantics follows this approach and the resulting rate computation method preserves the congruence relation.

We can show via a proof based on the structure of the functions used in (3.5) that our way of calculating the rate of communications is not affected by the problem presented in [27].

Theorem 1. *Given two* Scows *services P and Q such that $P \equiv Q$ by the structural congruence law* $\mathsf{par}_{\mathsf{assoc}}$*, and such that $P \xrightarrow[\rho]{\vartheta_1(\alpha)} P'$ and $Q \xrightarrow[\rho]{\vartheta_2(\alpha)} Q'$, we have that* $\mathtt{rate}\left(P \xrightarrow[\rho]{\vartheta_1(\alpha)} P'\right) = \mathtt{rate}\left(Q \xrightarrow[\rho]{\vartheta_2(\alpha)} Q'\right)$.

Proof. As the congruence rule (assoc) states that $(s_1 \mid s_2) \mid s_3 \equiv s_1 \mid (s_2 \mid s_3)$, we can suppose that $P \equiv (s_1 \mid s_2) \mid s_3$ and $Q \equiv s_1 \mid (s_2 \mid s_3)$. Then, there are two cases based on the form of α:

- $\alpha = \dagger$. Supposing that both P and Q have chosen the same action $(\mathbf{kill}(k), \lambda)$, we have that $\rho = \lambda$ and, by the computation of the rate for killing actions, the thesis.

- $\alpha = p.o\ \lfloor\emptyset\rfloor\ \tilde{a}\ \tilde{h}$. In this case we have that $\rho = [\gamma, \delta]$ (where γ is the rate of the request action and δ is the rate of the invoke action involved in the communication) and the rate is computed following equation (3.5):

$$\mathtt{rate}\left(P \xrightarrow[{[\gamma,\delta]}]{\vartheta_1(\alpha)} P'\right) = \frac{\delta}{\mathtt{inv}(P,p.o)} \cdot \frac{\gamma}{\mathtt{req}(P,p.o,\tilde{h},\#(\tilde{a},\tilde{h}))} \cdot \mathtt{min}\left(\mathtt{inv}(P, p.o), \frac{\mathtt{aR}(\mathtt{bms}(P),p.o,\tilde{a})}{\mathtt{aInv}(P,p.o,\tilde{a})}\right).$$

  Now, for each of the functions involved in the computation of the rate we will show that the function preserves the associativity of parallel composition.

`sumRates` Given that

$$\begin{aligned}\mathtt{sumRates}(s_1, \tilde{n}, p.o) &= (\gamma_1, c_1)\\ \mathtt{sumRates}(s_2, \tilde{n}, p.o) &= (\gamma_2, c_2)\\ \mathtt{sumRates}(s_3, \tilde{n}, p.o) &= (\gamma_3, c_3)\end{aligned}$$

we have that

$$\mathtt{sumRates}(s_1 \mid s_2, \tilde{n}, p.o) = (\gamma_{1,2}, c_{1,2})$$
$$\text{where } c_{1,2} = \mathtt{min}(c_1, c_2) \text{ and } \gamma_{1,2} = \begin{cases} \gamma_1 & \text{if } c_1 < c_2 \\ \gamma_2 & \text{if } c_2 < c_1 \\ \gamma_1 + \gamma_2 & \text{if } c_1 = c_2 \end{cases}$$

$$\mathtt{sumRates}((s_1 \mid s_2) \mid s_3, \tilde{n}, p.o) = (\gamma, c)$$
$$\text{where } c = \mathtt{min}(c_{1,2}, c_3) \text{ and } \gamma = \begin{cases} \gamma_{1,2} & \text{if } c_{1,2} < c_3 \\ \gamma_3 & \text{if } c_3 < c_{1,2} \\ \gamma_{1,2} + \gamma_3 & \text{if } c_{1,2} = c_3 \end{cases}$$

while

$$\begin{array}{l}\mathtt{sumRates}(s_2 \mid s_3, \tilde{n}, p.o) = (\gamma_{2,3}, c_{2,3}) \\ \text{where } c_{2,3} = \mathtt{min}(c_2, c_3) \text{ and } \gamma_{2,3} = \begin{cases} \gamma_2 & \text{if } c_2 < c_3 \\ \gamma_3 & \text{if } c_3 < c_2 \\ \gamma_2 + \gamma 3 & \text{if } c_2 = c_3 \end{cases}\end{array}$$

$$\begin{array}{l}\mathtt{sumRates}(s_1 \mid (s_2 \mid s_3), \tilde{n}, p.o) = (\gamma', c') \\ \text{where } c' = \mathtt{min}(c_1, c_{2,3}) \text{ and } \gamma' = \begin{cases} \gamma_1 & \text{if } c_1 < c_{2,3} \\ \gamma_{2,3} & \text{if } c_{2,3} < c_1 \\ \gamma_1 + \gamma_{2,3} & \text{if } c_1 = c_{2,3} \end{cases}\end{array}$$

Noticing that

$$c = \mathtt{min}(\mathtt{min}(c_1, c_2), c_3) = \mathtt{min}(c_1, \mathtt{min}(c_2, c_3)) = c'$$

and that

$$\gamma = \left\{ \begin{array}{ll} \gamma_1 & \text{if } c_1 < c_2 \text{ and } c_1 < c_3 \\ \gamma_2 & \text{if } c_2 < c_1 \text{ and } c_2 < c_3 \\ \gamma_3 & \text{if } c_3 < c_1 \text{ and } c_3 < c_2 \\ \gamma_1 + \gamma_2 & \text{if } c_1 = c_2 < c_3 \\ \gamma_1 + \gamma_3 & \text{if } c_1 = c_3 < c_2 \\ \gamma_2 + \gamma_3 & \text{if } c_2 = c_3 < c_1 \\ \gamma_1 + \gamma_2 + \gamma_3 & \text{if } c_1 = c_2 = c_3 \end{array} \right\} = \gamma'$$

we can conclude that

$$\mathtt{sumRates}((s_1 \mid s_2) \mid s_3, \tilde{n}, p.o) = \mathtt{sumRates}(s_1 \mid (s_2 \mid s_3), \tilde{n}, p.o).$$

`inv` $\mathtt{inv}((s_1 \mid s_2) \mid s_3, p.o) = \mathtt{inv}'((s_1 \mid s_2) \mid s_3, (s_1 \mid s_2) \mid s_3, p.o)$ Noticing that `inv'`
$= \mathtt{inv}'((s_1 \mid s_2) \mid s_3, s_1 \mid s_2, p.o) + \mathtt{inv}'((s_1 \mid s_2) \mid s_3, s_3, p.o)$
$= \mathtt{inv}'((s_1 \mid s_2) \mid s_3, s_1, p.o) + \mathtt{inv}'((s_1 \mid s_2) \mid s_3, s_2, p.o) + \mathtt{inv}'((s_1 \mid s_2) \mid s_3, s_3, p.o)$
returns a sum of integer numbers, and that in the base case where a non-zero value is returned it invokes function `sumRates` on the starting process P, we rely on the proof for `sumRates` to complete the case for `inv`.

`aInv` This case is dealt the same way as the previous one.

`req` $\mathtt{req}((s_1 \mid s_2) \mid s_3, p.o, \tilde{h}, c) = \mathtt{req}(s_1 \mid s_2, p.o, \tilde{h}, c) + \mathtt{req}(s_3, p.o, \tilde{h}, c)$
$= \mathtt{req}(s_1, p.o, \tilde{h}, c) + \mathtt{req}(s_2, p.o, \tilde{h}, c) + \mathtt{req}(s_3, p.o, \tilde{h}, c)$
$= \mathtt{req}(s_1, p.o, \tilde{h}, c) + \mathtt{req}(s_2 \mid s_3, p.o, \tilde{h}, c) = \mathtt{req}(s_1 \mid (s_2 \mid s_3), p.o, \tilde{h}, c)$

`bms` $\mathtt{bms}((s_1 \mid s_2) \mid s_3) = \mathtt{bms}'((s_1 \mid s_2) \mid s_3, (s_1 \mid s_2) \mid s_3)$
$= \mathtt{bms}'(s_1 \mid s_2, (s_1 \mid s_2) \mid s_3) \cup \mathtt{bms}'(s_3, (s_1 \mid s_2) \mid s_3)$
$= \mathtt{bms}'(s_1, (s_1 \mid s_2) \mid s_3) \cup \mathtt{bms}'(s_2, (s_1 \mid s_2) \mid s_3) \cup \mathtt{bms}'(s_3, (s_1 \mid s_2) \mid s_3)$
The case is similar to `inv`, using this time union between sets as associative operator with $\emptyset$ as 0-element.

aR As can be derived from the definition of aR, no reference to services is made, so the function does not need to be taken into account.

We can then conclude that

$$\mathtt{rate}\left(P \xrightarrow[{[\gamma,\delta]}]{\vartheta_1(\alpha)} P'\right) = \mathtt{rate}\left(Q \xrightarrow[{[\gamma,\delta]}]{\vartheta_2(\alpha)} Q'\right).$$

□

3.4 Final considerations on Scows

We acknowledge the work by Prandi and Quaglia [35], in which an extension to COWS is proposed. However, Scows differs from the version presented in [35] because our extension is based on a different fragment of the original language. In particular, preserving the polyadic communication paradigm, we give more importance to the communication capabilities of services when computing transition rates. Therefore we obtain a different (and, from the aspect of stochastic rate computation, more precise) stochastic version of COWS. The downside of our approach is that generating the transition system of a Scows model can take a time exponential in the number of parallel services the model contains. This happens because of the notorious problem of state space explosion, which is most evident in models of loosely coupled interacting entities (as is the case of Web Services): the transition system of such a model contains a number of states which can be up to exponential in the number of its parallel sub-models. As the model checking of a stochastic model typically requires its translation in a more widespread formalism such as Continuous-Time Markov Chains (CTMC) or Petri Nets, in order to perform model checking on a Scows model, we would need to obtain a CTMC from it. Unfortunately enough, this would require the entire transition system of the model to be generated: therefore, the generation of a CTMC from a Scows model (and the subsequent model checking) can become a troublesome task and, on models complex enough, even infeasible. As we will see in the next chapter, model checking Scows models is viable trough an alternative approach.

Chapter 4

A tool for model checking Scows

In the conclusion of Chapter 3 we highlighted the fact that generating a CTMC from a Scows model can be a computationally costly task, as it would require the generation of the complete transition system of the model. The main problem that the generation of a complete transition system for a non-trivial Scows model would cause is the notorious state space explosion problem. This kind of problem is most evident when a model comprises a number of loosely coupled components, as is often the case when dealing with distributed systems. The problem comes from the observation that the transition graph of a system composed of x processes in parallel, each of which with y states, can have (in the case that no communication happens among the processes) up to y^x states. A compositional generation of the transition system (and thus, of the underlying CTMC) would allow to avoid the problem of state space explosion, thanks to the fact that parallel components are considered in isolation and then merged with less-than-exponential execution time. Unfortunately, this type of approach can not be applied in the case of Scows. This is because of the name-passing matching paradigm adopted by the language, which requires to know an entire model to calculate a single communication action, preventing de facto any compositional approach to the transition system generation. In languages without the name-passing feature such as PEPA [22], the compositional approach can be applied and is in fact feasible: an approach of this kind is used for the generation of a CTMC from a PEPA model in PRISM [24], which is based on MTBDD (multi-terminal binary decision diagrams, [15, 20]) representations. Another example of application of the same principle is the CASPA tool [29], which generates a MTBDD representation from YAMPA, a stochastic process calculus based on TIPP [21].

In this chapter we will present our proposed solution to the problem of model checking Scows models avoiding the generation of the full transition system: Scows_amc. This tool distinguishes itself from the other stochastic model checking tools because it makes use of a simulation-based approach for the verification of properties without generating the full transition system of a model. This allows us to avoid the problem of state space explosion, while still maintaining acceptable computation time and approximation values.

The implementation language chosen for the tool is Java [1]. The reason behind this choice is that an integration with other tools from the SENSORIA project [2] is planned, and having a common implementation language would greatly simplify the integration process.

4.1 Overview on the approach

In order not to generate a complete transition system, and thus avoid the state space explosion problem, we base our model checking approach on the direct simulation of Scows models. This means that we generate a number of simulation traces by applying the operational semantics presented in Chapter 3 directly to Scows services, and then perform the computations necessary to check the desired formula against these traces. As a single execution trace of a stochastic model is by definition a random walk on the transition system of the model, we have to resort to statistical reasoning in order to estimate the size of the error we make in evaluating the requested property through a finite number of random walks. The theories behind the reasoning on which the approach used by the Scows_amc tool is based are the one adopted in the Ymer tool [18] (mainly consisting of Wald's *sequential probability ratio test* [39]) and the one adopted in the APMC tool [19]. Scows_amc differentiates from both because it is based on Scows models, while the others are based on CTMCs.

The CSL [5] formulas on which we will concentrate our efforts can be expressed in one of the following forms:

$$\mathcal{P}_{\bowtie\theta}\,[\Psi_1\ \mathcal{U}^{[t_0,t_1]}\ \Psi_2],\ \text{where}\ \bowtie\ \in \{<, \leqslant, >, \geqslant\}\ \text{and}\ \theta \in [0,1] \tag{4.1}$$

or

$$\mathcal{P}_{=?}\,[\Psi_1\ \mathcal{U}^{[t_0,t_1]}\ \Psi_2] \tag{4.2}$$

and can be read respectively as "Is there a probability $p \bowtie \theta$ that state formula Ψ_1 will hold until, inside the time interval $[t_0 t_1]$, state formula Ψ_2 holds?" and "What is the probability that state formula Ψ_1 will hold until, inside the time interval $[t_0 t_1]$, state formula Ψ_2 holds?". State formulae Ψ_1 and Ψ_2 are to be intended as state labels or as formulae of type 3/4 themselves. As we will see, the approach used in Ymer will be employed to decide the truth value of CSL formulas of type (4.1), while the method applied in APMC will be used in the estimation of formulas of type (4.2).

In the following subsections we will give an explanation of the theory and the observations on which the Scows_amc tool is based.

4.2 Statistical hypothesis testing

The approach employed in the Ymer tool is based on a theory of statistical hypothesis testing, which we will resume here.

First of all, notice that the property $\mathcal{P}_{\geqslant\theta}\,[\Phi\ \mathcal{U}^{[t_0,t_1]}\ \Psi]$ is equivalent to $\neg\mathcal{P}_{<\theta}\,[\Phi\ \mathcal{U}^{[t_0,t_1]}\ \Psi]$, and that, as the interval $[0,1]$ is continuous, the conditions $p \geqslant \theta$ or $p > \theta$ are indistinguishable.

For this reason, we will explain the statistical hypothesis testing theory taking into account only the formula $\mathcal{P}_{\geqslant\theta}\,[\Phi\ \mathcal{U}^{[t_0,t_1]}\ \Psi]$ without losing generality w.r.t. the other forms described in (4.1). The formula, which corresponds to the assertion "the probability that formula $\Phi\ \mathcal{U}^{[t_0,t_1]}\ \Psi$ is true if tested on a random walk starting from the initial state of the given Scows model is at least θ", will be represented from now on by the hypothesis H = "$p \geqslant \theta$".

The hypothesis testing problem takes into account the probability to have an error in the observation of an event, and thus to find the hypothesis to be false while it is actually true (*false negative*), or to state that the property is true when it is false (*false positive*). We will choose limits to the occurrence of the two types of error, and in particular we set the probability to have a false negative (i.e., to reject H when it is true) to be at most α, and the probability to have a false positive (i.e., to accept H when it is false) to be at most β, with $\alpha + \beta \leqslant 1$.

In Figure 4.1 we show the graph of the ideal testing environment, where the probability of false negative is exactly α and the probability of false positive is exactly β. The graph presents a

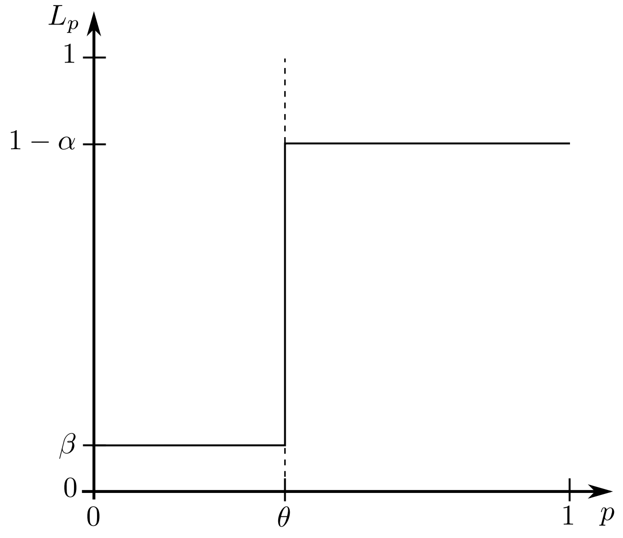


Figure 4.1: The probability L_p to accept hypothesis H = "$p \geqslant \theta$" as a function of p. Ideal case.

plot of the probability to accept hypothesis H as a function of the probability p. As can be seen, when $p < \theta$ (and thus, the hypothesis "$p \geqslant \theta$" is false), the probability to accept the hypothesis equals β (as it is the probability of a false positive), while when $p > \theta$ this probability becomes $1 - \alpha$ (as α is the probability to have a false negative). In the point where $p = \theta$, we request the probability to accept H to be both equal to $1 - \alpha$ and to β. Thus, it must be that $\beta = 1 - \alpha$, preventing us to choose the values of α and β independently.

In order to be able to set the error probabilities α and β independently from each other, a relaxation of the threshold becomes necessary: in particular, an *indifference region* is introduced. This region corresponds to all values of p between two thresholds p_1 and p_0 for which $\theta - \delta = p_1 < \theta < p_0 = \theta + \delta$ holds. The value δ determines the amplitude of the region in which the value of the hypothesis is considered not to be reliably asserted. The introduction of the indifference region determines a change in the hypotheses to test: we move from testing H

against its negate to testing H_0 = "$p \geqslant p_0$" against H_1 = "$p \leqslant p_1$". Maintaining the conditions on α and β, we present in Figure 4.2 a realistic hypothesis testing case. In this case, the original

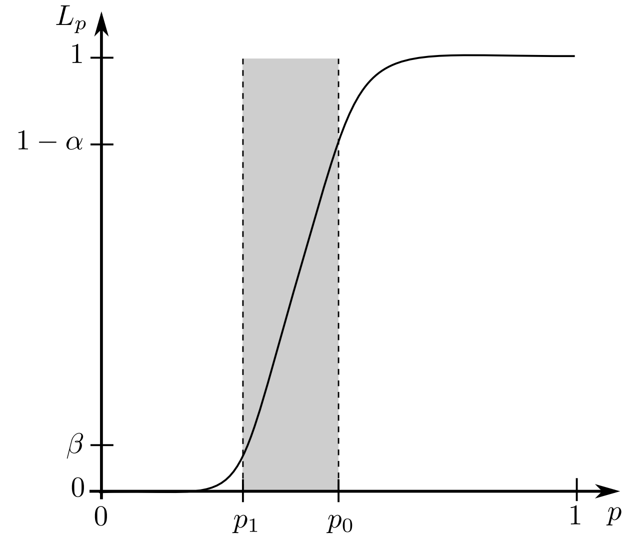


Figure 4.2: The probability L_p to accept hypothesis H_0 = "$p \geqslant p_0$" as a function of p. Real case.

hypothesis H = "$p \geqslant \theta$" is accepted with probability at least $1-\alpha$ if $p \geqslant \theta + \delta$ (because hypothesis H_0 is accepted), and is rejected with probability at most β if $p \leqslant \theta - \delta$ (hypothesis H_1 is accepted). If $p \in (p_1, p_0)$, both hypotheses H_0 and H_1 are false, as in the indifference region it is indifferent which of the two hypotheses is accepted. In Figure 4.2 this region is marked by a gray area. Notice that decreasing the value of δ, and thus narrowing the indifference region, it is possible to get closer to the ideal hypothesis testing situation of Figure 4.1.

In the next section we will present a statistical method which allows to test the described hypothesis H with the chosen probabilities of error α and β.

4.3 The sequential probability ratio test

The sequential probability ratio test has been introduced by Wald in [39], and is a statistical test where the number of observations is not determined *a priori*. Instead, the test allows to decide whether the given hypothesis is true or false when the error evaluation crosses a predetermined threshold. If a decision cannot be made, a new observation is made.

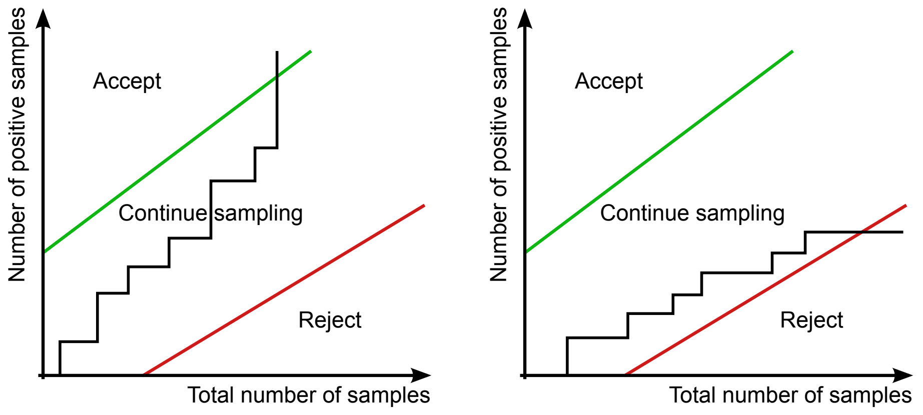
In our case, remembering that we have defined an indifference region between probabilities $p_1 = \theta - \delta$ and $p_0 = \theta + \delta$, the hypothesis to be tested at each iteration is H_0 = "$p \geqslant p_0$", and its converse is H_1 = "$p \leqslant p_1$". After having performed the m-th observation o_m, a probability ratio is computed:

$$R = \frac{p_{1m}}{p_{0m}}$$

where p_{im} is the probability to have obtained the (independent) observations $o_1, \ldots, o_m$ when the probability to have a positive response from each observation is p_i. The ratio thus obtained can be read as a ratio between the probability to have obtained the series of observations $o_1, \ldots, o_m$ from a series of identical, independent experiments whose success rate is p_1 and the probability to obtain the same series of observations when the success rate is p_0.

Hypothesis H_0 will be accepted if $R \leqslant B$, while H_1 will be accepted (thus rejecting H_0) if $R \geqslant A$, where A and B are such that $A > B$ and are chosen in order to respect the limits α and β we have established for the error probabilities. Thus, it must hold that if $R \leqslant B$, then the probability to accept H_0 while H_1 actually holds is at most β, and it must also be true that if $R \geqslant A$, then the probability to reject H_0 when it actually holds is at most α. As there is not a trivial way to find the values for A and B, an heuristic proposed in [39] in order to get error estimations close to the wanted ones is to set $A = \frac{1-\beta}{\alpha}$ and $B = \frac{\beta}{1-\alpha}$.

The sequential probability ratio test has been implemented into the Scows_amc tool for the evaluation of CSL formulae of type (4.1). The algorithm which performs the kernel of the task for the evaluation of such properties is illustrated in Algorithm 1, while Figure 4.3 graphically represents two possible execution paths for the algorithm.



(a) Accept (b) Reject

Figure 4.3: Two possible execution paths of the sequential probability ratio test algorithm implemented in Scows_amc. The two colored lines represent the bounds defined by the parameters α and β. As the number of samples increases, we move towards one of the two bounds: when one is crossed, the sampling stops and the hypothesis is either accepted or rejected (i.e., the formula is either decided to be true or false).

4.4 Probability estimation method

In the model checking of type (4.2) formulae we want to give an approximated result without having to take into account the whole transition system of the model at hand. In order to obtain such an approximation with a given error bound, we rely on a method presented in [19] and based on [25].

Algorithm 1 The algorithm used to perform the sequential probability ratio test. In order to facilitate the understanding of the method, we present a simplified version of the algorithm. The algorithm we show here does not consider the possibility of having probabilistic sub-formulas inside formula φ, nor does take into account the cases in which $\theta - \delta < 0$ or $\theta + \delta > 1$. The actual implementation of the algorithm provides support for both situations.

```
input $\alpha, \beta, \delta, \Phi = \mathcal{P}_{\bowtie\theta}[\varphi]$
$p_0 \leftarrow \theta + \delta$
$p_1 \leftarrow \theta - \delta$
$logA \leftarrow \ln \frac{1-\beta}{\alpha}$
$logB \leftarrow \ln \frac{\beta}{1-\alpha}$
$nObservations \leftarrow 0$
$d \leftarrow 0$
while $logB < d \wedge d < logA$ do
  generate a random walk $\sigma$
  if $\sigma \models \varphi$ then
    $d \leftarrow d + \ln \frac{p_1}{p_0}$
  else
    $d \leftarrow d + \ln \frac{1-p_1}{1-p_0}$
  end if
  $nObservations \leftarrow nObservations + 1$
end while
if $\bowtie \in \{>, \geqslant\}$ then
  return $d \leqslant logB$
else
  return $d > logB$
end if
```

The concept underlying the method presented in [19] is that also the value of CSL formulae of type (4.2) can be estimated through a series of observations on random walks over the transition system; but, differently from the method applied to type (4.1) formulae, this approach allows to determine the number of observations necessary to obtain the desired approximation level *before* the observations are made. This allows us to make trade-offs between speed and approximation with a deeper insight on the effects of our choices.

The error estimation tools provided by the method presented in [19] are the *approximation parameter* ε and the *confidence parameter* δ. In particular, it can be shown that the evaluation of the given formula on $O\left(\frac{1}{\varepsilon^2} \cdot \log \frac{1}{\delta}\right)$ random walks brings to a probability estimation differing from the real value by less than ε with probability $1-\delta$. The number of random walks necessary to obtain the desired result is given by the following formula:

$$nObservations = 4 \times \frac{\log \frac{2}{\delta}}{\varepsilon^2}. \tag{4.3}$$

Algorithm 2 is used in the Scows_amc tool for the estimation of type (4.2) formulae and is in fact the one presented in [19]. The idea on which the algorithm is based in order to compute the

Algorithm 2 The algorithm used in the Scows_amc tool for the estimation of a CSL formula in the form $\mathcal{P}_{=?}[\varphi]$.

```
input $\delta$, $\varepsilon$
$nObservations \leftarrow 4\log\left(\frac{2}{\delta}\right)/\varepsilon^2$
$count \leftarrow 0$
for $i = 1$ to $nObservations$ do
  generate a random walk $\sigma$
  if $\sigma \models \varphi$ then
    $count \leftarrow count + 1$
  end if
end for
return $count/nObservations$
```

probability estimation $\mathcal{P}_{=?}[\varphi]$ is to execute a fixed number of observations of the truth value of the formula φ, counting the number of positive results. The probability that formula φ is true is then computed dividing the number of positive observations by the total number of observations.

4.5 Speeding up the verification of bounded until formulae

The single observation of an event in both the approaches presented in Subsections 4.3 and 4.4 implies the verification of a path formula on a simulation trace (a random walk on the transition system of the model). This could become a lengthy task if the path formula at hand is a (time-bounded) until clause, because the Scows_amc tool generates every simulation path working directly on the Scows model. In fact, having to generate the set of possible candidate next states at each simulation step causes a degradation of performances w.r.t. the computation of simulation paths on a CTMC model (where the whole transition system has been already created), and such degradation is much more evident when the verification of a formula of type (4.2) is made in a parametric way. For example, consider the formula (4.4):

$$\Phi = \mathcal{P}_{=?} [\Phi_1 \; \mathcal{U}^{[t_0,t_1]} \; \Phi_2] \tag{4.4}$$

If we want to obtain a graph plotting the variation of the evaluation of Φ against the variation of (one or both) time bounds t_0 or t_1, the formula Φ has to be evaluated applying the algorithm described in Subsection 4.4 once for every different value of the varying time bound(s).

$$\begin{aligned} & t_0 = 0 \\ & t_1 \in \{0, 1, 2, \ldots, 20\}, \\ & \varepsilon = 0.01 \text{ (approximation)} \\ & \delta = 0.1 \text{ (confidence set to 0.9)} \end{aligned} \tag{4.5}$$

Thus, if e.g. we want to plot a graph for the formula for the values set as indicated in (4.5), the total number of simulation traces to be computed is

$$21 \times 4 \times \frac{\log \frac{2}{\delta}}{\varepsilon^2} = 21 \times 52,042 = 1,092,882$$

which would generally require a significant amount of time to be processed. Of course, parallel computation could be a solution to this problem, as each point of the plot can be computed by a single CPU (or core), but we propose a solution still more efficient, based on the reuse of already computed simulation traces for the verification of the until formula for all the variations of the time bounds. In the case of the example, our method would compute only 52,042 simulations for the generation of the graph with the given parameters. We will now present the basic algorithm for the verification of until path formulae, then proceed with the introduction of the modifications needed to obtain the described improvement.

4.5.1 Evaluating an until formula with constant time bounds

In order to obtain the truth value of an until path formula on a single simulation trace of the model, an algorithm similar to Algorithm 3 is used. Notice that such algorithm performs the checking of the formula as the generation of the trace goes along. This is a solution which has a better average execution time w.r.t. an approach in which a complete simulation trace is generated and then checked against the path formula. This is possible thanks to the fact that the until formula has time bounds, which allows us to stop the generation of a trace when "the time is up" or when a truth value for the formula has been found, even if the simulation could proceed further. We will now give an explanation on the less trivial points of the algorithm.

The function computeTransitions (lines 5 and 27) applies the operational semantics to the Scows model represented by the current state in the simulation: this allows us to obtain a set of candidate next states, each associated to the transition leading into it.

Algorithm 3 Verification of a bounded until formula.

```
1: input $\alpha, \beta, \delta$, Scowsmodel, $\varphi = \Phi_1 \ \mathcal{U}^{[tmin,tmax]} \ \Phi_2$
2: $totalTime \leftarrow 0$
3: $nextTime \leftarrow 0$
4: $currState \leftarrow$ initialState(Scowsmodel)
5: $transitions \leftarrow$ computeTransitions($currState$)
6: while $\neg$isEmpty($transitions$) do
7:    ($nextState$, $\tau$) $\leftarrow$ computeNextState($transitions$)
8:    $nextTime \leftarrow totalTime + \tau$
9:    if $tmin \leqslant totalTime$ then
10:      if verify($\Phi_2$, $currState$, $\alpha, \beta, \delta$) then
11:         return true
12:      else if $\neg$verify($\Phi_1$, $currState$, $\alpha, \beta, \delta$) then
13:         return false
14:      end if
15:   else
16:      if $\neg$verify($\Phi_1$, $currState$, $\alpha, \beta, \delta$) then
17:         return false
18:      else if $tmin < nextTime \wedge$ verify($\Phi_2$, $currState$, $\alpha, \beta, \delta$) then
19:         return true
20:      end if
21:   end if
22:   $currState \leftarrow nextState$
23:   $totalTime \leftarrow nextTime$
24:   if $tmax < totalTime$ then
25:      return false
26:   end if
27:   $transitions \leftarrow$ computeTransitions($currState$)
28:   if isEmpty($transitions$) then
29:      return verify($\Phi_2$, $currState$, $\alpha, \beta, \delta$)
30:   end if
31: end while
32: return false
```

The function computeNextState (line 7) makes a weighted randomised choice among all candidate next states, returning the chosen next state and the time required to reach it. The computation is weighted on the rates associated to the transitions leading to the given states. In particular, the choice is made through the application of a kinetic Monte Carlo algorithm (see Subsection 4.6.2).

The generation of the trace will end when a truth value for the formula has been found. Another way to end the generation cycle would be the case in which no next state can be computed, because the current state has no exiting transitions (the condition at line 6): in this case, the formula would be evaluated as false.

The time bounds of the until formula are checked in lines 9-21. More in detail, we cover all possible situations in the following 6 cases.

1. The current time (*totalTime*) is inside the interval $[tmin, tmax]$ (line 9) and the current state satisfies the post-condition Φ_2 (line 10): in this case, the formula is true because we have from case 3 that in each state for which $totalTime < tmin$ the pre-condition Φ_1 is fulfilled.

2. The current time is inside the interval $[tmin, tmax]$ (line 9) and the current state does not satisfy either of the two conditions Φ_1 and Φ_2 (line 16): the formula is false, because this is the first state for which Φ_1 does not hold (again, we have from case 3 that Φ_1 is true for all states before the current one) and Φ_2 does not hold either.

3. The current time is before *tmin*: in this case, it is sufficient that formula Φ_1 is verified in the current state. If it is not the case, (line 16), the whole formula is false.

4. The current time is before *tmin* and formula Φ_1 is true. If at the next step the total time will be inside the interval $[tmin, tmax]$ and formula Φ_2 will be verified (line 18), then the whole formula is true. Otherwise, the formula remains undecided and the computation can continue.

5. If at the next step the time will exceed the upper bound *tmax* (line 24), the formula is false, because even if the pre-condition Φ_1 has been kept true up to the current time (by case 3), the post-condition Φ_2 has never become true during the interval $[tmin, tmax]$.

6. If all other cases do not apply to the current situation, and the next state has no exiting transitions (line 28), then the truth value of the whole until formula corresponds to the truth value of the post-condition Φ_2 evaluated on the next state, because at time *tmax* we will still be at that same state and the only chance for the until formula to be verified is that the post-condition turns out to be true in the last state reached before exiting from the interval $[tmin, tmax]$.

Notice that in cases 1, 2 and 4 the current time is guaranteed to be not after the interval $[tmin, tmax]$ ($totalTime > tmax$): this check is performed before reaching the next state (case 5).

4.5.2 Evaluating an until formula with varying time bounds

In order to improve the execution time of the evaluation of a CSL until formula with a set of different time bounds, we propose to reuse the same set of simulation traces for the model checking of each variation of the formula. In the case of the evaluation of a CSL probabilistic formula of type (4.2), this translates into generating a number of simulation traces equal to the one obtained by formula (4.3), instead of that number multiplied by the number of variations of the until formula.

In the example presented in formula (4.4) with the settings defined by (4.5), the idea is to keep an ordered list of all the different values for t_1. During the generation of the simulation trace, we will change the value of t_1 when its old value is *not useful* any more. The value of the upper time bound t_1 is considered to be not useful when the current simulation time has crossed it (i.e., when a decision on the truth value of the current formula can be made). So, if for example the current value for t_1 is 5, the current total simulation time is 3 time units and the next simulation step requires 2.5 time units, then at the next step the total simulation time will be greater than t_1. In that case, by the way in which Algorithm 3 is constructed, the checking of the formula $\Phi_1 \; \mathcal{U}^{[0,5]} \; \Phi_2$ will have surely come to an answer, either positive or negative, and thus the simulation would terminate. Instead of terminating the simulation, we store the result obtained with the current value of t_1 and continue the checking with the next value $t_1 = 6$. Before computing a new simulation step, however, it is necessary to check again the formula using the updated value for t_1, in order to verify if the next value of t_1 would allow to immediately decide the truth value of the formula. This check on the changing value of t_1 continues until we find a value of t_1 which does not allow to directly determine the truth value of the formula: only then we proceed to the computation of the next simulation step. If there are no more values for t_1, the simulation is terminated and all the truth values of the until formula are returned, one for each different value of t_1.

As an example application of the described approach, in Algorithm 4 we show the modifications we have introduced in order to speed up the evaluation of an until formula with different upper time bounds. Notice that the evaluation of the path formula in this case does not generate a single boolean value (the truth value of the formula), but an *array* of such values (*result*[]), one for each of the different time bounds given as input.

In this version of the algorithm we need to know when the upper time bound *tmax* is crossed by the value of the current time (line 27) or, more generally, when a decision has been reached on the truth value of the formula with the current value for *tmax*: we keep track of this with the flag *done*[*tmaxIndex*]. When the flag indicates that the truth value for the formula has been decided, a new value for *tmax* is taken from the ordered list of available values and used in the next iteration of the cycle in lines 12-35. Through this new cycle we keep on verifying the truth value of the formula, remaining on the same state while a decision on it can be made immediately; when it is not possible to make such a decision, the computation of the simulation trace continues with the next state.

Managing the varying values for the upper time bound of the until path formula in the way

Algorithm 4 Verification of a bounded until formula with varying upper time bound.

```
1: input $\alpha, \beta, \delta$, Scows$model$, $\varphi = \Phi_1\ \mathcal{U}^{[tmin,tmax]}\ \Phi_2$,
       $tmaxValues[] = [tmax_1, tmax_2, \ldots, tmax_n]$
2: $totalTime$, $nextTime \leftarrow 0$
3: $tmaxIndex \leftarrow 1$
4: $tmax \leftarrow tmaxValues[tmaxIndex]$
5: $done[1,\ldots,n] \leftarrow$ [false, ..., false]
6: $result[1,\ldots,n] \leftarrow$ [false, ..., false]
7: $currState \leftarrow$ initialState(Scows$model$)
8: $transitions \leftarrow$ computeTransitions($currState$)
9: while $\neg done[n] \wedge \neg$isEmpty($transitions$) do
10:    ($nextState$, $\tau$) $\leftarrow$ computeNextState($transitions$)
11:    $nextTime \leftarrow totalTime + \tau$
12:    repeat
13:        $redo \leftarrow$ false
14:        if $tmin \leqslant totalTime$ then
15:            if verify($\Phi_2$, $currState$, $\alpha, \beta, \delta$) then
16:                $result[tmaxIndex] \leftarrow$ true   $done[tmaxIndex] \leftarrow$ true
17:            else if $\neg$verify($\Phi_1$, $currState$, $\alpha, \beta, \delta$) then
18:                $result[tmaxIndex] \leftarrow$ false   $done[tmaxIndex] \leftarrow$ true
19:            end if
20:        else
21:            if $\neg$verify($\Phi_1$, $currState$, $\alpha, \beta, \delta$) then
22:                $result[tmaxIndex] \leftarrow$ false   $done[tmaxIndex] \leftarrow$ true
23:            else if $tmin < nextTime \wedge$ verify($\Phi_2$, $currState$, $\alpha, \beta, \delta$) then
24:                $result[tmaxIndex] \leftarrow$ true   $done[tmaxIndex] \leftarrow$ true
25:            end if
26:        end if
27:        if $\neg done[tmaxIndex] \wedge tmax < nextTime$ then
28:            $result[tmaxIndex] \leftarrow$ false   $done[tmaxIndex] \leftarrow$ true
29:        end if
30:        if $done[tmaxIndex]$ then
31:            $tmaxIndex \leftarrow tmaxIndex + 1$
32:            $tmax \leftarrow tmaxValues[tmaxIndex]$
33:            $redo \leftarrow$ true
34:        end if
35:    until $\neg redo$
36:    $currState \leftarrow nextState$
37:    $totalTime \leftarrow nextTime$
38:    $transitions \leftarrow$ computeTransitions($currState$)
39:    if $\neg done[n] \wedge$ isEmpty($transitions$) then
40:        $result[tmaxIndex,\ldots,n] \leftarrow$ verify($\Phi_2$, $currState$, $\alpha, \beta, \delta$)
41:    end if
42: end while
43: return $result[]$
```

presented in Algorithm 4, we obtain all the truth values of the formula corresponding to the different values of *tmax*. This way of dealing with the problem is demonstrably equivalent to having computed a new simulation path for each value of *tmax*, and thus represents an improvement in the performance of the tool in all the cases in which a plot of the value of a formula of type 4.2 over varying time bounds is needed.

Finally, notice that the approach presented here can be trivially extended in order to deal with varying values of both lower and upper time bounds.

4.6 Architecture

The tool is organised in the following modules:

- Parser for Scows models and CSL formulas
- Simulator engine for the generation of traces
- Model checking engine for the evaluation of formulae
- Textual user interface
- Graphical user interface

We will explain the details about each module in the following subsections.

4.6.1 Parser

The parser module handles the generation of abstract syntax tree from the model and formula input files, and has been created through the tools JFlex [3] (for the lexical analyzer part) and CUP [4] (for the syntax analyzer part). An example input model is shown in Table 4.1: as can be seen, the syntax of the input language adheres to the syntax of Scows as presented in Chapter 3.

Table 4.2 presents two example CSL input formulas based on the model in Table 4.1. Notice that the syntax of the formulas is self-explicative, as it is directly derived from the syntax of CSL formulas as explained in Chapter 2, and is based on a grammar close to the one used in PRISM.

4.6.2 Simulator engine: generation of random walks

The simulation engine employed for the generation of random walks on the transition system of a Scows model is based on an algorithm similar to Gillespie's Stochastic Simulation Algorithm (SSA, [16]), which is a randomised algorithm proved to be a sound application in a biochemical environment where particular conditions hold. The use of the same approach in the case of distributed system models could at first glance appear a less sound idea, but it nevertheless makes sense, as the SSA is a Monte Carlo method for the simulation of stochastic systems, and is in essence a Kinetic Monte Carlo algorithm [28], whose field of application varies widely.

```
//Agents
Portal() =
    //customer login
    [vU][vP][vI](
        (p#.cl#?<vU, vP, vI>,1.0).(
            //lookup in the login repository
            [lkey#][ndc#](
                (ndc#.ndc#!<ndc#>,1.0)
                |(
                    (ndc#.ndc#?<ndc#>,failRate).(p#.ko#!<vI>,1.0)
                    + (ndc#.ndc#?<ndc#>,okRate).(p#.lok#!<vI, lkey#>,1.0)
                )
            )
            | Portal()
        )
    );

Customer() =
    //initialId is used to give the response to _this_ particular
    //customer after the information lookup for the login.
    [u#] [pwd#] [inid#](
        (p#.cl#!<u#, pwd#, inid#>,1.0)
        | (
            //login not successful
            (p#.ko#?<inid#>,1.0).nil

            //login successful
            //myKey is the variable that will contain the "session id" of the customer
            + [myKey](p#.lok#?<inid#,myKey>,1.0).( (myInitialId.clok#!<lok#>,1.0) )
        )
    );
$
//initial process
[verification#][infoUpload#][yes#][no#]( Portal() | Customer() | Customer() | Customer() )
$
//counter definitions
finished : [ 0 .. 3 ];
$
//cows actions <-> counter modifications
p# . lok# <*>: finished < 3 : (finished' = finished + 1) ;
```

Table 4.1: An example input model for Scows_amc. Notice that this is a Scows representation of the example explained in detail in Section 4.7.

```
P =? [ true U [ 5 , 20 ] finished >= 2 ]
P >= 0.3 [ X finished > 0]
```

Table 4.2: Two example CSL formulas against which the model in Table 4.1 can be checked.

The Kinetic Monte Carlo algorithm is employed in our case in a “next-state” function. This function takes as input the current state and the corresponding exiting transitions, and outputs a pseudo-randomly chosen next state plus the time required to make the transition from the current to the next state. The pseudo-random decision is based on a weighted Monte Carlo choice, using the stochastic rates of the transitions exiting from the current state as weights. Algorithm 5 represents the steps performed in order to perform the pseudo-random choice of the next state.

Algorithm 5 The steps performed by the simulator engine in order to compute the next-state function.

```
1:  input $currentState$, $transitions[]$
2:  $totalRate \leftarrow 0$
3:  for all $t_i \in transitions$ do
4:      $totalRate \leftarrow totalRate + \texttt{rate}(t_i) \times \text{multiplicty}(t_i)$
5:  end for
6:  generate random number $p_1 \in (0, 1]$
7:  find $i$ such that $\sum_{j=1}^{i-1} \texttt{rate}\left(t_j\right) < p_1 \times totalRate \leqslant \sum_{j=1}^{i} \texttt{rate}\left(t_j\right)$
8:  update state variables according to the actions associated to transition $t_i$
9:  $nextState \leftarrow \text{targetState}(t_i)$
10: generate random number $p_2 \in (0, 1]$
11: $\tau \leftarrow -\frac{\ln p_2}{totalRate}$
12: return $\langle nextState, \tau \rangle$
```

The algorithm chooses the next transition weighting the choice with the rates of the transitions. After having generated a random number p_1 from a uniform distribution in the interval $(0, 1]$ (line 6), the next transition is determined in a way such that the probability for each transition t_j to occur is proportional to its rate $\texttt{rate}\left(t_j\right)$. In Figure 4.4 we show a graphical representation of this step: the transition to be chosen is t_i such that $p_1 \times totalRate$ falls inside the segment representing $\texttt{rate}(t_i)$ (line 7).

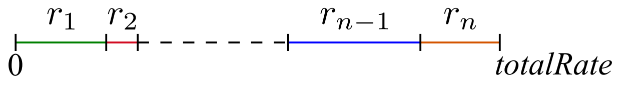


Figure 4.4: Choosing the next transition step when generating a simulation trace. Please notice that in this graphical representation we shorten the rate notation with the convention $r_j = \texttt{rate}\left(t_j\right)$ for $j \in \{1, 2, \ldots, n\}$.

Finally, a pseudo-random number $p_2 \in [0, 1]$ is chosen and used in computing the time τ needed in order to move from the current to the next state. The time τ is obtained by generating a random variate (i.e., a possible outcome of a random variable) using the exponential distribution having *totalRate* as parameter. Indeed, by the property of exponential distribution, the rate at which the current state is left is the sum of the rates of all transitions exiting from it.

A random walk is obtained by iteratively applying Algorithm 5 starting with the initial state of the model. The transitions exiting from each reached state are obtained by applying to the current state the operational semantics described in Chapter 3.

4.6.3 Model checking engine

The part of the tool which allows to perform the model checking of Scows models implements the algorithms described in Sections 4.3, 4.4 and 4.5.

The structure of the CSL formulae is exploited in their evaluation: the model checking algorithms are implemented as methods of the classes representing CSL formulae. Thus, all path formulae allow to perform a `sample` action, which generates a simulation trace and produces the boolean value resulting from the evaluation of the path formula on the trace, while state formulae can perform either `modelCheck` (which decides on the truth value of the formula) or `estimateProbability` (which returns an estimation on the probability of the sub-formula to be true). The actual availability of the `modelCheck` and `estimateProbability` methods is declared by the state formula itself and is based on its type and parameters. For example, $\mathcal{P}_{>p}[\varphi]$ allows only `modelCheck` to be made, while $\mathcal{P}_{=?}[\varphi]$ allows us to perform only `estimateProbability`.

The implementation of the model checking algorithms adds also the support for a client-server architecture in the evaluation of path formulae, thus allowing to exploit distributed computing in the evaluation of probabilistic state formulae. Basically, a path formula can act both as a client and as a server. In the former case, the `sample` method keeps on calculating simulation paths and sending the result of the formula evaluation to the server. In the latter case, the server keeps a queue of client results to which truth values are added each time a new result is received from a client, and from which truth values are extracted each time the model checking algorithm requests a new path evaluation.

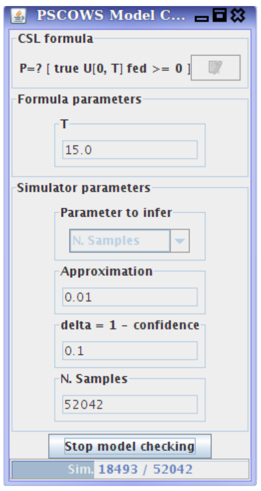
4.6.4 User interfaces

The tool provides both graphical and textual user interfaces (see Figure 4.5), the latter providing support for both interactive and automatic operative modes.

These interfaces have been designed in order to support an user in two phases of the model checking process:

- thanks to the interactive interface, the user can devise a CSL formula and try its evaluation/estimation in a restricted context in order to reach the desired trade-off between execution time and accuracy;
- exploiting the support for execution in a shell script, the user can define a range of parameters for the chosen formula(e) and run them automatically one after the other.

Furthermore, the textual interface allows the user to define a client-server architecture through which to distribute model checking on various machines.



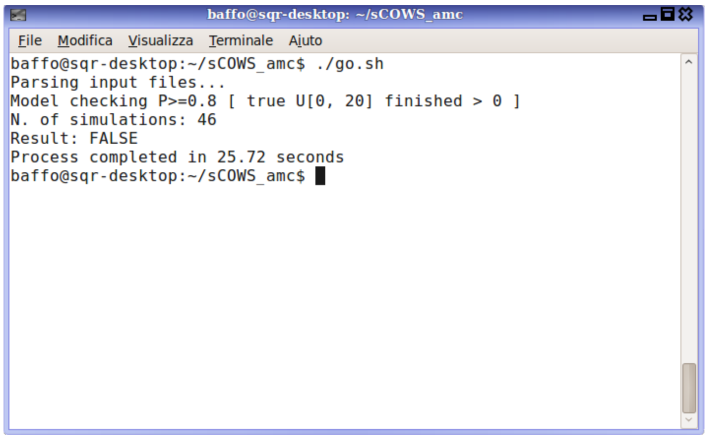
CSL formula: P=? [true U[0, T] fed >= 0]

Formula parameters
T: 15.0

Simulator parameters
Parameter to infer: N. Samples
Approximation: 0.01
delta = 1 - confidence: 0.1
N. Samples: 52042

Sim. 18493 / 52042

(a) Graphical


baffo@sqr-desktop: ~/sCOWS_amc
File Modifica Visualizza Terminale Aiuto

```
baffo@sqr-desktop:~/sCOWS_amc$ ./go.sh
Parsing input files...
Model checking P>=0.8 [ true U[0, 20] finished > 0 ]
N. of simulations: 46
Result: FALSE
Process completed in 25.72 seconds
baffo@sqr-desktop:~/sCOWS_amc$
```

(b) Textual

Figure 4.5: The user interfaces provided by the Scows_amc tool.

4.7 Example: client log-in attempts

As an example application of our tool, we present a simplified model derived from a case study developed within the SENSORIA European Project [2]. The setting for the scenario consists of a Bank Credit Request service, which can be invoked by customers. The example presents the login phase of the scenario, which is modelled by two services: one representing the portal and the other representing a client (see Table 4.1).

The portal service is persistent, i.e. the service can answer to multiple requests without terminating. This behaviour is achieved using service recursion, ensuring that all the recursive calls are guarded in order to obtain a finitely branching transition system. In the example three customers try to concurrently login. Each customer can attempt only once to perform the login procedure and then its behaviour terminates.

The login procedure begins with an invocation by the customer (modelled via the `Customer` agent) on the `portal.customerLogin` channel; the customer provides a username, a password and a private name used to distinguish between different customer login transactions (i.e., it represents the session identifier). The complementary request is performed by an instance of the `Portal` agent, which we assume to be the only service offering this specific request.

The task to decide whether a login attempt is successful or not is performed directly by the portal service. The choice that determines the outcome of the login attempt is made between two possible communications over the `ndc.ndc` channel. In the case of a positive response, a private name is sent from the portal service to the customer. This private name models a session identifier, which will identify the corresponding logged customer throughout the rest of the interactions between the customer and any additional service provided by the bank. The private name representing the session identifier will be exchanged in any interaction that can involve only successfully logged customers. In the model, this is achieved taking advantage of the matching mechanism used in the communication paradigm of the language, which has been designed to naturally capture this behaviour. Note that, in the definition of `Portal`, the basic rates for the receiving actions on `ndc.ndc` are not numeric values: the rates of the communications are computed in a symbolic way, using *failRate* and *okRate* as placeholders. The actual values for the basic rates will be defined when using the model checking software. In this way different configurations for the same model can be used when performing model checking analysis, without the need to modify the starting model.

We have defined a parametric CSL property (4.6) that will be checked against the considered system:

$$\mathcal{P}_{=?}\,[\mathtt{true}\ \mathcal{U}^{[\mathtt{T},\mathtt{T}]}\ \mathtt{finished} = \mathtt{N}] \tag{4.6}$$

which can be read as “Which is the probability to have exactly `N` successfully logged customers at time `T`?”. Notice that the value of the variable named `finished` is the number of successfully logged customers in the system. This variable is incremented in the Scows model when the transition is given by a communication happening through the `portal.loginOk` endpoint (cfr. Table 4.1).

The graph plotting the results of the evaluation of formula (4.6) against the model presented in Table 4.1 with $T \in \{0, \dots, 30\}$ and $N \in \{0, \dots, 3\}$ is presented in Figure 4.6.

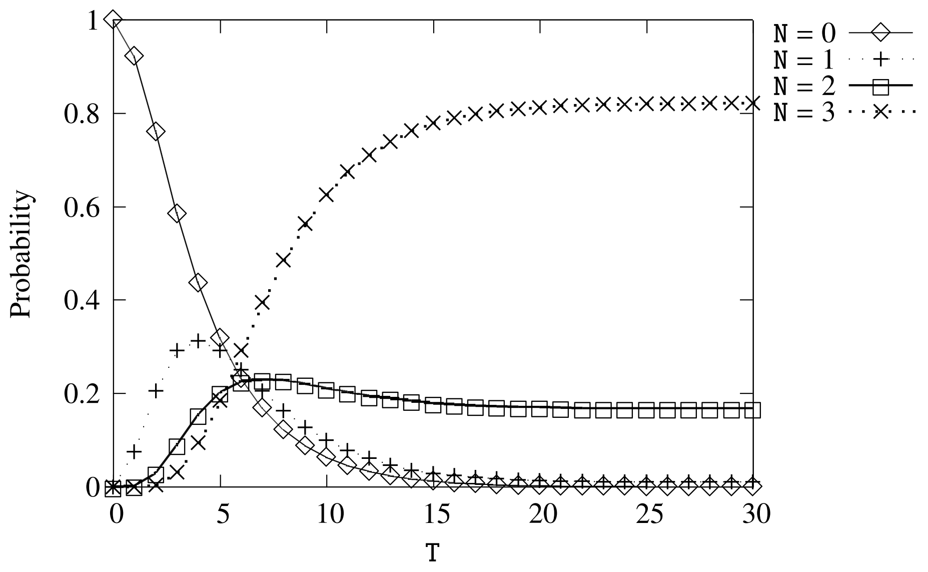


Figure 4.6: Plot of the values obtained from the model checking of the example model against the CSL formula $\mathcal{P}_{=?}\,[\mathtt{true}\ \mathcal{U}^{[\mathtt{T},\mathtt{T}]}\ \mathtt{finished} = \mathtt{N}]$. The parameters used in the simulations are $\varepsilon = \delta = 0.1$ (thus $52,042$ simulation traces are computed).

Chapter 5

Case studies and related work

The Scows_amc tool presented in Chapter 4 allows us to check Web Services modelled through Scows against CSL formulae without requiring to generate the complete state space underlying the models. This is done applying simulation-based statistical methods directly on the Scows models, thus allowing to completely avoid the possibility to face with the problem of state-space explosion.

In this chapter we will evaluate the performances of the proposed model checker compared with other existing model checking tools and then show an example application of the Scows_amc tool to a more complex case study from the SENSORIA European project [2] inside which our work is set.

To our knowledge, the one presented in Chapter 4 is the only model checker based on Scows models. Thus, in order to obtain sensible results, we will compare the performances of our tool with CTMC-based model checkers, where the CTMCs are obtained from Scows models thanks to Igor Cappello's Scows_lts tool [11, 12]. This tool generates the CTMC underlying a Scows model, with the objective of a subsequent employment with CTMC-based model checking tools. The peculiar effort made by the tool allows to partially overcome the state-space explosion problem, thanks to a structural congruence check applied to every generated state, which allows the tool to use memory and CPU far better than it would without this approach.

We want to show when and why our tool resorting to a direct simulation method outperforms the approaches requiring full knowledge of the underlying CTMC. We expect that approaches based on CTMC simulation taken in isolation will often outperform our proposed method, as the direct simulation of Scows models requires much more time than the same approach applied to a CTMC. Whereas, when considering also the time for CTMC generation, we expect our tool to show better performances on higher-dimensioned problems. Furthermore, even if numerical approaches allow us to obtain exact results, they can require more resources than approximated ones when the state-space dimension is in the order of thousands.

5.1 Case studies

The tools will be tested in the model checking of the Scows models of two classical concurrency theory problems: the dining philosophers and the sleeping barber. Both these problems have variants in the number of processes considered (e.g., the number of philosophers sitting at the table, or the number of available chairs in the barber shop): by varying these numbers we will be able to make considerations on the impact of the number of starting parallel processes on the execution time of the tools, and thus on the scalability of the adopted approaches.

5.1.1 The dining philosophers

The problem of the dining philosophers has first been formulated by Dijkstra in [14] in order to explain the problems of deadlock and resource starvation in computer systems. The version of the problem we adopt here represents a number *N* of philosophers sitting at a circular table, where a plate of food is available for each philosopher. A philosopher can either think or eat, but in order to eat he needs to acquire both a fork and a knife, which can be acquired only if the two philosophers at his side have not acquired them themselves. In fact, there is a number of forks and knives each equal to half the number of philosophers, and the cutlery is positioned in an alternating way between the philosophers, following the example schema in Figure 5.1, where 6 dining philosophers are shown. As can be seen in the figure, the philosophers need to be alternated between right-handed and left-handed in order to correctly use the cutlery without changing its position.

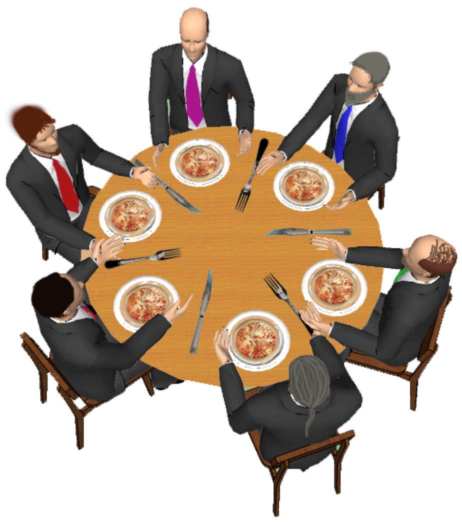


Figure 5.1: An example configuration of the dining philosophers problem, showing how 6 philosophers would look like when placed as described.

The Scows model corresponding to the 6 philosophers configuration of the problem can be found in Table 5.1. In the model we distinguish between left-handed (service `LHphil`) and right-handed (service `RHphil`) philosophers because, as said previously, the cutlery is positioned on the table alternating between knife and fork. Notice also that, as will happen in the case of the other models presented here, we use parametric rate definitions (i.e., we use `r1`, `r2`, `r3`, ... instead of actual real numbers): this will allow us to instantiate the proper rates with values only at the moment of model checking. This way of defining rates will allow us to change rates as desired without the need for additional computations on the model (for instance, the recomputation of a CTMC) and is supported by all the tools used in the comparison.

The models taken into account for the tests described in the sections below comprise the versions of the generic model where the number of philosophers is even and between 2 and 12, and the number of forks/knives is properly adapted. For example, in the model with 8 philosophers, there are also 4 forks and 4 knives.

5.1.2 The sleeping barber

The problem of the sleeping barber has been formulated by Dijkstra in [13] and aims to describe the communication and synchronization challenges that can arise when dealing with multiple operating system processes running concurrently. The version of the model adopted here has four main components: the barber, a single barber chair, a waiting room with a finite number of chairs, and a number of customers. When the barber has no work to do, he simply sits down on his chair and sleeps. When a customer enters the shop, he first checks whether the barber is asleep: in that case, he wakes him up and gets his hair cut; if the barber is already working on another customer, the newcomer tries to find a place in the waiting room. If no chair is available in the waiting room, the customer exits the shop and comes back later after having gone for a walk; conversely, if a chair is available the customer takes a seat and waits the barber to become free. When the barber has finished dealing with a customer, after having been paid, he checks the waiting room for another customer: the first to answer his call is served, the others are left to wait (he is a very exclusive and eccentric barber); in case no customers are present in the waiting room, the barber goes back to sleep on his chair. In Table 5.2 we show the Scows representation of the model as described here. In this example the number of customers is limited to 3 and there are 2 places available in the waiting room. Notice from the model that, thanks to the polyadic matching communication paradigm used in Scows, we have modelled in an efficient way the different priorities of the passages made by a customer entering the shop: if the barber is sleeping, the triple <`me,n#,n#`> is matched by the process `SleepingBarber` with a single substitution; if that process is not active (i.e., the barber is not sleeping), then one of the `Chair` processes can match the triple with only one more substitution (variable `x` by name `n#`); if no chair is available, the `NoChairs` process will obtain the triple (matching with three substitutions) and let the customer know that the shop is full. The checks performed by the barber after having finished dealing with a customer are modelled in a similar way (cfr. the possible communications on channels `queue.call` and `barber.reply`).

The models we will use in the next sections for test beds will contain a number `N` of cus-

```
//Agents

//RHphil: right-handed philosopher
RHphil(right#, left#) =
[fork][knife]( (right#.take#?<fork>,r1) . (left#.take#?<knife>,r2) .
    [eat#][food#]( (eat#.eat#!<food#>,r3) | (eat#.eat#?<food#>,r4) .
         ( (left#.release#!<knife>,r5) | (right#.release#!<fork>,r6) )
    )
);

//LHphil: left-handed philosopher
LHphil(right#, left#) =
[knife][fork]( (left#.take#?<fork>,r7) . (right#.take#?<knife>,r8) .
    [eat#][food#]( (eat#.eat#!<food#>,r9) | (eat#.eat#?<food#>,r10) .
         ( (right#.release#!<knife>,r11) | (left#.release#!<fork>,r12) )
    )
);

Cutlery(f#) = [p#] ( (f#.take#!<p#>,r13)
                   | (f#.release#?<p#>,r14).Cutlery(f#) );

$
//initial process

[fork1#][knife1#][fork2#][knife2#][fork3#][knife3#][take#][release#] (
      RHphil(fork1#, knife1#) | LHphil(knife1#, fork2#)
    | RHphil(fork2#, knife2#) | LHphil(knife2#, fork3#)
    | RHphil(fork3#, knife3#) | LHphil(knife3#, fork1#)
    | Cutlery(fork1#) | Cutlery(knife1#)
    | Cutlery(fork2#) | Cutlery(knife2#)
    | Cutlery(fork3#) | Cutlery(knife3#)
)

$

//counter definitions

fed : [ 0 .. 6 ];

$
//cows actions <-> counter modifications

eat#.eat#<*>: fed < 6 : (fed' = fed + 1);
```

Table 5.1: The problem of 6 dining philosophers modelled with Scows.

tomers varying between 1 and 3, and a number of chairs varying between 0 and 2, leaving out all models in which the number of available chairs is at least as large as the number of customers. Thus, we will consider a model comprising 1 chair and 3 customers, but not a model in which there are 2 chairs and 2 customers.

5.2 Selected model checkers

The tools against which the Scows_amc tool will be compared have been chosen with the aim to obtain a significant evaluation of the performances of our tool.

5.2.1 PRISM - Probabilistic Symbolic Model Checker

PRISM [24] is a tool allowing to adopt various model checking approaches in order to verify properties of both discrete-time (DTMC) and continuous-time Markov chains (CTMC), and of Markov decision processes (MDP). The properties, which can be expressed in a language incorporating temporal logics such as CSL, PCTL and LTL, can also deal with costs and rewards defined in the systems. The tool offers support for both exact and approximate approaches to model checking, implementing advanced techniques for symbolic state-space handling (such as multi terminal binary decision diagrams, MTBDDS [15]) and providing an efficient discrete-event simulator for statistical analysis of model properties. The tool implements also the approach presented in the previous chapter for model checking CSL formulas of type $\mathcal{P}_{=?}\,[\Phi_1\ \mathcal{U}^{[t_0,t_1]}\ \Phi_2]$, which is the one presented in [19], and the same one used in our tool.

5.2.2 Ymer

Ymer [40] is a software developed by H. L. S. Younes which applies the sequential probability ratio test (SPRT, [39]) to CTMC model checking, and to which the Scows_amc tool is partly inspired. Ymer also implements some of the numerical approaches used in PRISM for the exact computation of model checking results. The reason for which Ymer has been included in the tests is to have an element of comparison for the performances of the SPRT implementation in Scows_amc, in order to quantify the advantages of starting from a completely known state space (as is the case of Ymer) versus the approach of direct simulation of a process calculus.

5.3 Experimental setup

The performances of the selected tools will be tested in the model checking of the two case studies, of which a number of variants are proposed, each sporting a different number of interacting processes, in order to study the scalability of the two main approaches: complete state space (and CTMC) generation from the Scows model followed by the application of one of the available model checking techniques versus statistical reasoning on discrete-event simulation

```
//Agents
IdleBarber(wakeUp#) =
(queue#.call#!<n#>,r1)
| [customerID]((barber#.reply#?<customerID>,r2) . WorkingBarber(customerID, wakeUp#)
             + (barber#.reply#?<nobodyPresent#>,r3) . SleepingBarber(wakeUp#)
);

SleepingBarber(wakeUp#) =
[customerID]((barber#.wakeUp#?<customerID, n#, n#>,r4) . WorkingBarber(customerID, wakeUp#));

WorkingBarber(customerID, wakeUp#) =
[newStyle#][money#]((customerID.cutHair#!<newStyle#, money#>,r5)
                    | (barber#.pay#?<money#>,r6) . IdleBarber(wakeUp#));

Chair(findChair#) =
[customerID][x]((barber#.findChair#?<customerID,x,n#>,r7) . (
                (customerID.chairAvailable#!<n#>,r8)
                | (queue#.call#?<n#>,r9) .
                      ((barber#.reply#!<customerID>,r10) | Chair(findChair#))));

Customer(enterShop#) =
[me#] ((barber#.enterShop#!<me#, n#, n#>,r11)
| [newStyle][money]((me#.cutHair#?<newStyle,money>,r12) . (barber#.pay#!<money>,r13))
 + (me#.chairAvailable#?<n#>,r14) .
             [newStyle][money]((me#.cutHair#?<newStyle,money>,r15) . (barber#.pay#!<money>,r16))
 + (me#.shopFull#?<n#>,r17) .
             ([wander#] (me#.wander#!<n>,r18) | (me#.wander#?<n>,r19) . Customer(enterShop#)));

NoChairs(checkChair#) =
[customerID][x][y]((barber#.checkChair#?<customerID,x,y>,r20) .
          ((customerID.shopFull#!<n#>,r21) | NoChairs(checkChair#)));

NoCustomers() =
[x]((queue#.call#?<x>,r22) . ((barber#.reply#!<nobodyPresent#>,r23) | NoCustomers()));

$
//Initial process
[doAction#][barber#][queue#][call#][reply#][chairAvailable#]
[shopFull#][n#][nobodyPresent#][cutHair#][pay#](
     SleepingBarber(doAction#)
   | Chair(doAction#) | Chair(doAction#) | NoChairs(doAction#) | NoCustomers()
   | Customer(doAction#) | Customer(doAction#) | Customer(doAction#)
)

$
//counter definitions
cut : [ 0 .. 3 ];

$
//cows actions <-> counter modifications
me#.cutHair#<*>: cut < 3 : (cut' = cut + 1);
```

Table 5.2: The problem of the sleeping barber with 2 chairs and 3 customers.

based directly on the Scows model. This will allow us to better define the boundaries inside which the employment of our tool becomes more convenient than the CTMC-based approach.

The objective of the test is the model checking of the following CSL formulae

$$\mathcal{P}_{=?}\,[\mathtt{true}\ \mathcal{U}^{[\mathtt{T},\mathtt{T}]}\ \mathtt{fed} = \mathtt{N}] \tag{5.1}$$

$$\mathcal{P}_{\geqslant p}\,[\mathtt{true}\ \mathcal{U}^{[\mathtt{0},\mathtt{20}]}\ \mathtt{fed} \geqslant \mathtt{N}] \tag{5.2}$$

$$\mathcal{P}_{=?}\,[\mathtt{true}\ \mathcal{U}^{[\mathtt{T},\mathtt{T}]}\ \mathtt{cut} = \mathtt{N}] \tag{5.3}$$

$$\mathcal{P}_{\geqslant p}\,[\mathtt{true}\ \mathcal{U}^{[\mathtt{0},\mathtt{20}]}\ \mathtt{cut} \geqslant \mathtt{N}] \tag{5.4}$$

against the dining philosophers models (formulae 5.1 and 5.2) and the sleeping barber models (formulae 5.3 and 5.4), with parameters respectively of $\mathtt{T} \in \{0, \ldots, 40\}$ for formulae 5.1 and 5.3, of $p \in \{0, 0.05, 0.1, 0.15, \ldots, 0.9, 0.95, 1\}$ for formulae 5.2 and 5.4, and $\mathtt{N} \in \{0, \ldots, nPeople\}$ for all formulae. Here, *nPeople* represents the total number of people on which the counting is based (i.e., the number of philosophers in formulae 5.1 and 5.2 and the number of customers in formulae 5.3 and 5.4).

5.3.1 Confidence settings

As the Scows_amc tool is an *approximated* model checker based on statistical reasoning, we will need to define a standard confidence setting for the tools providing an approximated approach, in order to confront execution data obtained under the same conditions. Of course, exact approaches do not need confidence settings.

The confidence settings we impose on approximated approaches for the evaluation of Formulae 5.1 and 5.3 are the *approximation level* of 10^{-2} and the *confidence level* of 0.9; i.e., we require the approximated results to differ from the exact results by less than 10^{-2} with probability of at least 0.9. These settings will apply to the approximated engine of PRISM and to Scows_amc, as Ymer does not allow to deal with formulae of this type. As we have explained in Chapter 4, further increasing the requested confidence settings would likely require much more time to compute the results. However, there are cases in which a better approximation is more important than execution time: in these cases a boost in performances can be gained from parallel computing, which is supported by Scows_amc.

For the case of Formulae 5.2 and 5.4, the approximation parameters apply only to Scows_amc and Ymer, as PRISM can only offer exact model checking for this kind of formulae. These parameters are, as detailed on Chapter 4, α (the probability to have a false negative), β (the probability of false positive) and δ (the semi-width of the indifference region centered on the threshold p), and are all set to a value of 10^{-2}, which allows to obtain an answer in short time with reasonable uncertainty bounds.

5.4 Experimental results

We will now show, for each case study, a table resuming the performance values obtained from the different tools, and will show a graphical comparison between the approaches when they are particularly interesting.

The parameter evaluated for the execution of the tools is the computational time required by the tools to complete the model checking of the selected formulae. Notice that not all tools present a result under their columns in the tables: this can depend from the fact that the tool cannot be employed for model checking the formula at hand (as is the case of Ymer for Formulae 5.1 and 5.3, and of the simulation engine of PRISM for Formulae 5.2 and 5.4), or that the time required to complete the computation would have been beyond human limits. The latter is the case of the dining philosophers models comprising 8 to 12 philosophers: as the Scows_lts tool would require too much time to produce a CTMC from the Scows model, Formulae 5.1 and 5.2 cannot be evaluated in these cases with the CTMC-based approaches.

All tests have been executed on a workstation equipped with an Intel (R) Pentium (R) D 3.40 GHz CPU and 2 Gigabyte of RAM, running Ubuntu Linux 9.10.

5.4.1 Dining philosophers

The peculiar formulation of the dining philosophers problem presented in Section 5.1.1 allows us to test in particular the scalability of the tools, evidencing the difficulty of generating a CTMC when the number of states in the transition system rapidly increases. In fact, because of the limitation of the alternating pattern in cutlery, there must be an even number of philosophers in every variation of the model, thus bringing forth large differences in state-space dimension between adjacent variations.

Table 5.3 shows the computation times required by the tools to produce their output when checking Formula 5.1 in all its variations. Notice the the time required for computing the CTMC severely hampers the model checking possibilities of CTMC-based tools, extending to the point of completely preventing them to be employed. In fact, the models with 8 philosophers or more are estimated to require a time in the order of at least hundreds of years to generate a CTMC. Remember however that the Scows_lts tool requires a large computational effort in order to control the state-space explosion through the application of a congruence checking to each state of the transition system. Thanks to these checks, the memory usage is kept at lower levels than it would have been with a naive approach, thus allowing us to obtain CTMCs for models which would otherwise be intractable.

The case of 6 dining philosophers shows something interesting: in fact, the time required for the computation with Scows_amc is less than the time to perform the same operation with the approximated approach in PRISM. This is possible thanks to the enhancements described in Section 4.5, which allow us to reuse the same simulation traces for more than one configuration of the formula.

Figure 5.2 shows a comparison in the execution times of the tools Scows_amc and Scows_lts.

Philosophers	N. of states	Scows_lts	PRISM		Ymer	Scows_amc
			Exact	Approx.		
2	20	0.9	1.9	95.5	-	395.6
4	249	345.4	153.5	1871.2	-	5537.8
6	3247	523173.0	138749.0	73729.4	-	31109.4
8	-	-	-	-	-	113603.0
10	-	-	-	-	-	309769.1
12	-	-	-	-	-	719487.9

Table 5.3: Computational time results for the model checking of Formula 5.1 (times are expressed in seconds). The two columns for PRISM correspond to the numerical (Exact) and simulation-based (Approx.) model checking approaches. Data for Ymer are not present, as the tool does not provide a method for dealing with formulae of this type.

As can be seen from the graph, a comparison between Scows_amc and any CTMC-based tool would be severely biased by the time needed to compute the CTMC. It can however be noticed that for small state-space sizes (2 and 4 philosophers) it is far more convenient to use a CTMC-based approach.

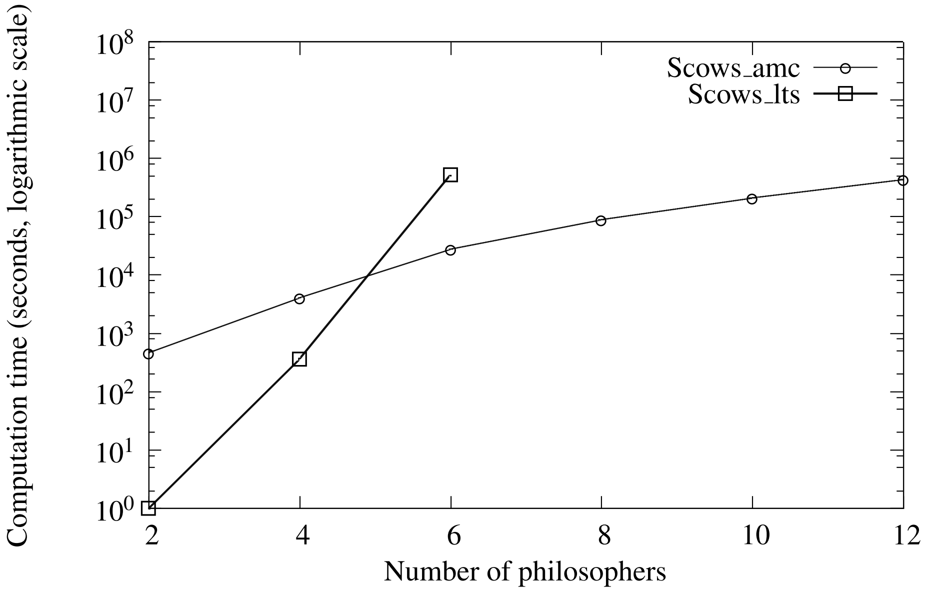


Figure 5.2: Graph comparing the performances of the tools Scows_lts and Scows_amc when applied to the dining philosophers problem. Data for 8 philosophers and up are not available for Scows_lts, as their estimated computation time would be too much large to be actually computed.

The verification of Formula 5.2 shows the same differences between the CTMC-based approaches and the Scows_amc tool. In fact, the time required to compute a CTMC represents again the most part of the time required by the model checking process in these cases. However, an advantage of using the CTMC-based approach appears clearly at this point: when a

number of different formulae are to be checked against the same model, the CTMC needs to be obtained only once, and then all formulae can be checked using the favourite approach.
Notice also that the sequential probability ratio test (applied in Ymer and Scows_amc for the model checking of Formulae 5.2 and 5.4) adds a certain aleatority to the computational time performances, because only the previous history of simulation traces determines the decision to perform another observation (cfr. Figure 4.3). So, if we have been particularly "lucky", the time taken to reach the required error rate can be rather small: it is the case of the 10-philosophers model in Table 5.2, which has taken Scows_amc less time to compute than the 8-philosophers case.

Philosophers	N. of states	Scows_lts	PRISM		Ymer	Scows_amc
			Exact	Approx.		
2	20	0.9	1.68	-	2.1	262.9
4	249	345.4	61.45	-	16.9	2931.0
6	3247	523173.0	50700.4	-	3390.4	20293.9
8	-	-	-	-	-	80153.8
10	-	-	-	-	-	59419.0
12	-	-	-	-	-	131238.0

Table 5.4: Computational time results for the model checking of Formula 5.2 (times are expressed in seconds). Please notice that only the "Exact" column contains time values for PRISM, because the tool does not provide an approximated method for checking formulae of this type. Data for Scows_lts are the same as in Table 5.3 because the models are the same.

5.4.2 Sleeping barber

For the sleeping barber problem, we have kept the number of parallel services at low numbers in order to be able to get an overview of the performances of the different approaches to model checking without running too badly into state-space explosion. As already noticed for the case of the dining philosophers, also Table 5.5 shows that the performance difference between the approximated approach of PRISM and the Scows_amc tool decreases with growing state spaces; when the states space is big enough Scows_amc even outperforms PRISM: again, this happens thanks to the improvement for "recycling" simulation traces presented in Section 4.5.

Table 5.6 shows the computation times required by the model checking of Formula 5.4 with the selected tools. In this case, the comparison between CTMC-based simulation and Scows-based simulation shows how much an advantage the knowledge of the full state space is w.r.t. the generation of all possible transitions at each step of the computation of a simulation trace. As it can be seen, the exploration of a completely known state space allows Ymer to outperform by far the Scows_amc tool. However, the computation of the CTMC has to be taken into account also in this case, slightly evening out the competition: the sum of the times required by both Scows_lts and Ymer is not dramatically smaller than the time required by Scows_amc.

Chairs	Customers	N. of states	Scows_lts	PRISM		Ymer	Scows_amc
				Exact	Approx.		
0	1	6	1.2	1.7	40.4	-	287.4
0	2	50	2.3	2.6	202.3	-	1191.7
0	3	981	295.7	1812.4	4977.2	-	4130.9
1	2	43	2.4	3.3	291.1	-	2007.7
1	3	1253	517.0	3927.1	10963.2	-	6929.4
2	3	1066	792.9	3422.7	10646.6	-	8277.0

Table 5.5: Computational time results for the model checking of Formula 5.3 (times are expressed in seconds). The two columns for PRISM correspond to the numerical (Exact) and simulation-based (Approx.) model checking approaches. Data for Ymer are not present, as the tool does not provide a method for dealing with formulae of this type.

Chairs	Customers	N. of states	Scows_lts	PRISM		Ymer	Scows_amc
				Exact	Approx.		
0	1	6	1.2	2.2	-	0.8	22.9
0	2	50	2.3	2.5	-	0.7	152.1
0	3	981	295.7	872.9	-	18.0	909.0
1	2	43	2.4	2.4	-	0.9	190.0
1	3	1253	517.0	1862.1	-	30.5	1760.1
2	3	1066	792.9	1651.6	-	32.8	1170.2

Table 5.6: Computational time results for the model checking of Formula 5.4 (times are expressed in seconds). Please notice that only the "Exact" column contains time values for PRISM, because the tool does not provide an approximated method for formulae of this type.

Figure 5.3 shows the peculiarity of the sequential probability ratio test: when the threshold p approaches the real probability, the computation time required to obtain an answer on the truth value of a formula of the kind of Formula 5.4 tends to grow steeply. As can be seen in the graph, the maximum computation time for the Scows_amc tool corresponds to the exact value for the formula $\mathcal{P}_{=?}\,[\mathtt{true}\ \mathcal{U}^{[0,20]}\ \mathtt{cut} \geqslant 2]$ (i.e., the point in which Formula 5.4 changes its truth value). For the sake of completeness, we show in Table 5.7 the values of formula $\mathcal{P}_{\geqslant p}\,[\mathtt{true}\ \mathcal{U}^{[0,20]}\ \mathtt{cut} \geqslant 2]$ as computed by the Scows_amc tool.

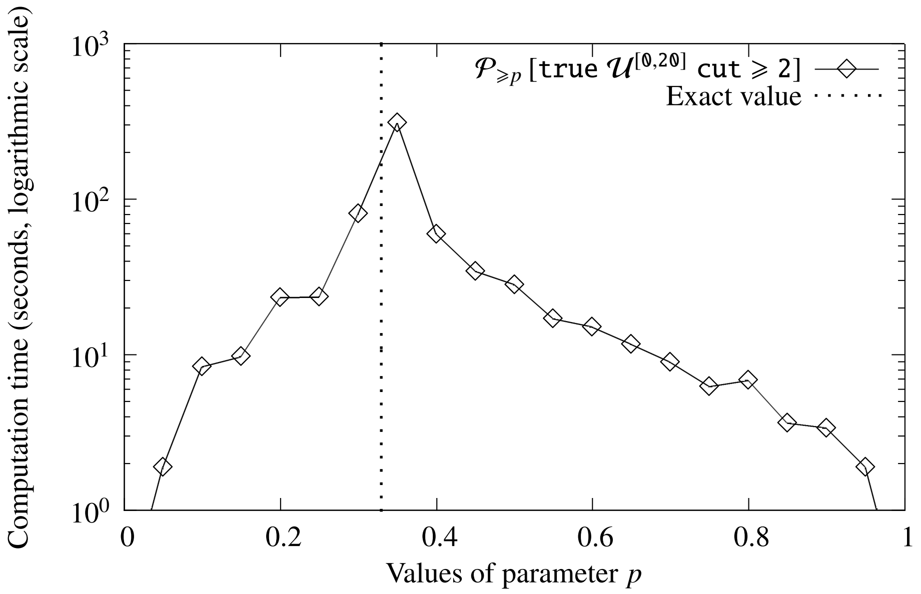


Figure 5.3: Comparing the execution times against the probability parameter for Formula 5.4 when checking the sleeping barber model with 0 chairs and 3 customers. The dotted line denotes the exact value of the formula $\mathcal{P}_{=?}\,[\mathtt{true}\ \mathcal{U}^{[0,20]}\ \mathtt{cut} \geqslant 2]$, and thus the point in which the answer to Formula 5.4 changes.

Value of p	Result	Time (s)	Value of p	Result	Time (s)
0.00	`true`	0.22	0.55	`false`	17.04
0.05	`true`	1.89	0.60	`false`	15.06
0.10	`true`	8.34	0.65	`false`	11.64
0.15	`true`	9.74	0.70	`false`	8.90
0.20	`true`	23.28	0.75	`false`	6.24
0.25	`true`	23.42	0.80	`false`	6.87
0.30	`true`	80.64	0.85	`false`	3.64
0.35	`false`	309.84	0.90	`false`	3.35
0.40	`false`	59.61	0.95	`false`	1.88
0.45	`false`	34.43	1.00	`false`	0.22
0.50	`false`	28.17			

Table 5.7: Values of formula $\mathcal{P}_{\geqslant p}\,[\mathtt{true}\ \mathcal{U}^{[0,20]}\ \mathtt{cut} \geqslant 2]$ as computed by Scows_amc, and their computation times.

5.5 A more complex case study: the bank credit request scenario

We will now present an extended version of the example shown in Section 4.6, which will allow us to give an idea of the employment of the Scows_amc tool in a more suitable environment. The example is a case study taken from the SENSORIA project, of which Table B.1 in Appendix B shows the Scows model.

The model represents a service for granting loans to bank customers. When a customer asks for a loan, the bank processes the balance and amount. If the amount is under a certain threshold, or if the assessment on the balance is particularly favorable (class "A" assessment), an automatic loan offer is provided to the customer. Conversely, if the request cannot be handled automatically, the attention of a clerk is requested (assessment "B"). If the assessment is not positive ("C"), a supervisor is asked to deal with the request instead of the clerk. Clerks and supervisors can either accept or reject the loan request: in the former case, a loan proposal is forwarded to the customer. When the customer receives a loan offer, he can either refuse or accept it, ending the process. The whole process can also be terminated in any moment if the customer decides to cancel the request.

We propose to use our tool exploiting both the model checking approaches it provides. In this example, we check if a probability is above a certain threshold before requesting its approximated value, taking advantage of the rapid convergence times when dealing with formulae in the form $\mathcal{P}_{\geqslant p}[\varphi]$, and requiring a more precise estimation only when the first survey has given a positive result. As an example, we will test the performances of the presented case study using the probability to obtain a response from the system within 40 time units as a performance index:

$$\mathcal{P}_{=?}\,[\texttt{true}\;\mathcal{U}^{[0,40]}\;\texttt{finished} = 1] \tag{5.5}$$

The system will be tested in various configurations, each of which enhances the response time of one of the main components by increasing the corresponding rate:

- log-in phase (rate `loginDecisionRate` in the model)
- balance assessment service (rate `assessmentDecisionRate`)
- bank clerk (rate `clerkDecisionRate`)
- credit supervisor (rate `supDecisionRate`)

When all rates are set to their base values (cfr. Table B.2), the probability to obtain a response in the given time interval is 0.31. We will perform a preliminary test for each configuration by requesting the probability of Formula 5.5 to be greater than 0.35 (i.e., we test Formula 5.6): only when this threshold is crossed we will need to know the actual improvement obtained from the variation, otherwise such an improvement would hardly be useful.

$$\mathcal{P}_{\geqslant 0.35}\,[\texttt{true}\;\mathcal{U}^{[0,40]}\;\texttt{finished} = 1] \tag{5.6}$$

Notice from data in Table 5.8 that the most valuable performance enhancement is obtained by increasing the speed at which clerks evaluate a customer (for example, by taking on more clerks), and its probability is the only one we actually needed to calculate. All other probabilities are included only for the sake of completeness, but did not need to be calculated, as they had failed the preliminary test. The result we obtained had to be expected, as the probability for a customer to be evaluated by a clerk is slightly larger than the probability to get an automatic response or to be evaluated by a supervisor, as can be inferred from the rate settings for the assessment outcomes (rates `assessmentArate`, `assessmentBrate` and `assessmentCrate`).

Thanks to the proposed method, an evaluation requiring on average 8 minutes of computation can be used in order to filter out all ineffectual improvement hypotheses, thus performing the time-consuming (about 8 hours) process of probability estimation only when actually needed.

Enhanced component	Rate value	Value of Formula 5.5
Log-in phase	2.0	(0.33)
Balance assessment	2.0	(0.33)
Clerk	0.8	0.41
Supervisor	1.0	(0.32)

Table 5.8: Results of the model checking on the SENSORIA finance case study. Numbers between parentheses are the actual probability values of cases for which Formula 5.6 was evaluated as false.

Chapter 6

Conclusions

The objectives we posed at the start of the work were to obtain an efficient way of modelling and evaluating the performances of Web Services taking advantage of the formal foundations provided by COWS. In order to reach the objective, we planned to divide the work into two steps: the creation of a suitable stochastic modelling language (preferably based on COWS) and the ideation of a way to evaluate the performance of systems modelled through such language.

Scows, the stochastic extension of COWS, maintains the polyadic matching communication paradigm of the original calculus, thus allowing us to model the use of session identifiers during the course of an interaction in a service oriented environment. In order to maintain the polyadic matching in the calculus, we had to resort to a number of supplemental calculations when computing the rate of a transition. In fact, we need to know an entire Scows term in order to decide not only which transitions are possible (which is already the case of the basic version of the calculus), but also the stochastic rates of such transitions. This non-compositionality in turn gives rise to some difficulties when trying to check quantitative properties of large Scows models: in these cases state-space explosion can become a severe hindrance to model checking Scows terms. The solution we adopted was to implement some statistical model checking approaches in the proposed Scows_amc tool, through which the state-space explosion problem can be avoided while still obtaining valuable performance estimations. The price of this approach is that only approximated results can be obtained, as they are based on a finite number of random walks on the complete state space of a Scows model. Also, the number of formulae which can be dealt with is smaller than if using an approach such as the one applied in Scows_lts [11, 12] (e.g., steady-state analysis is not contemplated in our case). However, a positive aspect of the proposed model checker is that an approximation threshold can be set for each evaluation, thus allowing the user to balance between execution time and confidence levels. This feature enables the user to perform quantitative analysis even when dealing with rather large state-spaces.

In order to obtain an estimation of the performances of Scows_amc, we have made a comparison between the proposed tool and existing model checking tools based on CTMC (which require the knowledge of the complete transition system of a Scows model to be computed). The comparison has shown that, when models are sufficiently small, the generation of the state-space of a Scows model is still advisable, provided that an effort is made for reducing the exponential explosion in the number of states, e.g. partitioning the state-space using a notion of structural congruence (as is done in Scows_lts). When numbers start to grow, the approach requiring an existing model checking tool would be outperformed by Scows_amc, as the time needed to compute a CTMC would be larger than the time needed to perform direct simulation.

The work presented here makes use of some approaches applied before to the statistical model checking of CTMC models, and shows that they are applicable also directly to a process calculus. The loss in performance the direct simulation of a Scows model shows when compared against the corresponding CTMC-based approach can partly be reduced if considering the proposed improvements. This allows us to put forward the application of a method of this kind also to those other stochastic process calculi for which the computation of a complete state-space could raise performance issues.

Future developments involve mainly extensions to the functionalities of the Scows_amc tool. For example, a more complete support could be added to the parametrized formulae, improving the performances in a wider number of situations. Also, it would be advisable to extend the support to more CSL formulae, for example adding the possibility to deal with steady-state formulae, which allow to test the performance of a system in the long run. Finally, it would be interesting to test the approaches used in Scows_amc by applying them to other stochastic process calculi.

Appendix A

From COWS to Monadic COWS

We present an encoding from the original (polyadic) version of distributed systems modelling language COWS to a monadic version thereof, thus limiting the number of transmitted parameters to one for each communication. Through results on the operational correspondence between COWS services and the respective encodings, we show that interactions in the language could be expressed in a monadic form without losing expressive power, while slightly simplifying the communication paradigm. However, the same approach could not be applied to the stochastic version of the language, as it becomes clear from the demonstrations presented here. In fact, the correspondence between a monadic and a polyadic stochastic versions of COWS could not be possibly proven because of the impossibility to maintain both the polyadic matching mechanism and the correct stochastic rates for every execution step without modifying the operational semantics of the language. This impossibility has brought us to maintain the polyadic communication paradigm in our stochastic extension of COWS, which could not be further simplified without losing expressive power w.r.t. the non-stochastic version.

A.1 COWS syntax and semantics

Here we present the syntax and semantics of the language on which we will work in the next sessions. As our objective is to create an encoding from the polyadic to the monadic versions of the same language, we will apply polyadic semantics also to monadic COWS, although better fitting semantics could be built for the specific case.

The syntax of the version of COWS we will use in this paper is defined by the following grammar:

$$\begin{array}{rcl} s & ::= & u.u!\tilde{e} \mid \sum_{i=0}^{r} p_i.o_i?\tilde{w}_i.\, s_i \mid \mathbf{kill}(k) \mid \\ & & s \mid s \mid \{\!|s|\!\} \mid [\,d\,]s \mid \mathsf{S}(n_1,\ldots,n_j) \mid \mathbf{0} \end{array}$$

Notice that service replication $*s$ has been replaced here by service definition and instantiation ($\mathsf{S}(m_1,\ldots m_j) = s$ and $\mathsf{S}(n_1,\ldots,n_j)$) in order to improve the readability of the encoding. For what concerns metavariables, we will divide all *entities* into three disjoint sets: the set of names (which will be represented by identifiers starting with lower-case letters, as

$m, n, o, p, m', n', myName, yourName, \dots$), the set of variables (represented by identifiers all upper-case: X, Y, X', Y', $MYVAR$, $YOURVARIABLE$) and the set of killer labels (represented by $k, k', k_1, \dots$). Adhering to the conventions adopted in [30], we shall use u, w, u', w' as metavariables for names and variables, and d, d' for generic entities. Moreover, the class of expressions (with metavariables $e, e', e_1, \dots$) will include names, variables and (at least) arithmetic-boolean expressions, which can be evaluated through function $\mathcal{E}$. This function, taken as input a tuple $\tilde{e}$ of expressions, produces a tuple $\tilde{v}$ of values, containing at each position i the value v_i corresponding to the evaluation of expression e_i. The set of values (with metavariables $v, v', v_1, \dots$) thus includes both names and evaluated expressions. Finally, services will be identified by metavariables $s, s', s_1, \dots$, while service identifiers (used for defining recursive behaviour) will be referred to using names like $\mathsf{S}, \mathsf{Service}_1, \mathsf{MyService}, \dots$.

The semantics for the language is defined in Table A.2. All the functions used in the semantics are defined the same way as in [30], with the changes needed to adapt them to the proposed syntax. The structural congruence relation $\equiv$ is taken from [30], and adapted to our case (see Table A.1).

(prot_1)	$\{\|\mathbf{0}\|\} \equiv \mathbf{0}$	(prot_2)	$\{\|\{\|s\|\}\|\} \equiv \{\|s\|\}$
(prot_3)	$\{\|[d]s\|\} \equiv [d]\{\|s\|\}$	(delim_1)	$[d]\mathbf{0} \equiv \mathbf{0}$
(delim_2)	$[d_1][d_2]s \equiv [d_2][d_1]s$	(delim_3)	$s_1 \mid [d]s_2 \equiv [d](s_1 \mid s_2)$ if $d \notin \mathrm{fk}(s_2)$
$(\mathsf{par}_{\mathsf{comm}})$	$s_1 \mid s_2 \equiv s_2 \mid s_1$	$(\mathsf{par}_{\mathsf{assoc}})$	$s_1 \mid (s_2 \mid s_3) \equiv (s_1 \mid s_2) \mid s_3$
$(\mathsf{par}_{\mathbf{0}})$	$s \mid \mathbf{0} \equiv s$		
$(\mathsf{cho}_{\mathsf{comm}})$	$s_1 + s_2 \equiv s_2 + s_1$	$(\mathsf{cho}_{\mathsf{assoc}})$	$(s_1 + s_2) + s_3 \equiv s_1 + (s_2 + s_3)$
$(\mathsf{cho}_{\mathbf{0}})$	$s + \mathbf{0} \equiv s$	$(\mathsf{cho}_{\mathsf{ide}})$	$s + s \equiv s$

Table A.1: COWS structural congruence.

A.2 Conventions used in the encoding

Here we will present some notations used in the encoding from polyadic to monadic COWS in order to increase its readability. Notice that all these conventions are only of syntactical nature and do not add limitations to the scope of the encoding.

- We will not always employ composite names for endpoints: i.e., we will sometimes write endpoints in the form p instead of $p.o$ when the use of an operation name could be avoided without generating ambiguity.

$$\text{(kill)}\ \frac{-}{\mathbf{kill}(k) \xrightarrow{\dagger k} \mathbf{0}} \qquad \text{(inv)}\ \frac{\mathcal{E}(\tilde{e}) = \tilde{v}}{p!\tilde{e} \xrightarrow{p!\tilde{v}} \mathbf{0}}$$

$$\text{(choice)}\ \frac{1 \leqslant j \leqslant r}{\sum_{i=1}^{r} p_i?\tilde{w}_i.\, s_i \xrightarrow{p_j?\tilde{w}_j} s_j} \qquad \text{(prot)}\ \frac{s \xrightarrow{\theta} s'}{\{\!|s|\!\} \xrightarrow{\theta} \{\!|s'|\!\}}$$

$$\text{(del}_{\text{sub}}\text{)}\ \frac{s \xrightarrow{p\ \lfloor \sigma \uplus \{v'/x\} \rfloor\ \tilde{w}\ \tilde{v}} s'}{[\,x\,]s \xrightarrow{p\ \lfloor \sigma \rfloor\ \tilde{w}\ \tilde{v}} s'\{v'/x\}} \qquad \text{(del}_{\text{kill}}\text{)}\ \frac{s \xrightarrow{\dagger k} s'}{[\,k\,]s \xrightarrow{\dagger} [\,k\,]s'}$$

$$\text{(del}_{\text{pass}}\text{)}\ \frac{s \xrightarrow{\theta} s' \quad d \notin d(\theta) \quad s{\downarrow_{kill}} \Rightarrow \theta = \dagger, \dagger k}{[\,d\,]s \xrightarrow{\theta} [\,d\,]s'}$$

$$\text{(com)}\ \frac{s_1 \xrightarrow{p?\tilde{w}} s_1' \quad s_2 \xrightarrow{p!\tilde{v}} s_2' \quad \mathcal{M}(\tilde{w}, \tilde{v}) = \sigma \quad \neg(s_1 \mid s_2 \downarrow_{p,\tilde{v}}^{|\sigma|})}{s_1 \mid s_2 \xrightarrow{p\ \lfloor \sigma \rfloor\ \tilde{w}\ \tilde{v}} s_1' \mid s_2'}$$

$$\text{(par}_{\text{conf}}\text{)}\ \frac{s_1 \xrightarrow{p\ \lfloor \sigma \rfloor\ \tilde{w}\ \tilde{v}} s_1' \quad \neg(s_2 \downarrow_{p,\tilde{v}}^{|\mathcal{M}(\tilde{w},\tilde{v})|})}{s_1 \mid s_2 \xrightarrow{p\ \lfloor \sigma \rfloor\ \tilde{w}\ \tilde{v}} s_1' \mid s_2} \qquad \text{(par}_{\text{kill}}\text{)}\ \frac{s_1 \xrightarrow{\dagger k} s_1'}{s_1 \mid s_2 \xrightarrow{\dagger k} s_1' \mid halt(s_2)}$$

$$\text{(par}_{\text{pass}}\text{)}\ \frac{s_1 \xrightarrow{\theta} s_1' \quad \theta \neq (p\ \lfloor \sigma \rfloor\ \tilde{w}\ \tilde{v}), \dagger k}{s_1 \mid s_2 \xrightarrow{\theta} s_1' \mid s_2} \qquad \text{(cong)}\ \frac{s \equiv s_1 \quad s_1 \xrightarrow{\theta} s_2 \quad s_2 \equiv s'}{s \xrightarrow{\theta} s'}$$

$$\text{(ser}_{\text{id}}\text{)}\ \frac{s\{m_1, \ldots, m_j / n_1, \ldots, n_j\} \xrightarrow{\theta} s' \quad \mathsf{S}(n_1, \ldots, n_j) = s}{\mathsf{S}(m_1, \ldots, m_j) \xrightarrow{\theta} s'}$$

Table A.2: COWS operational semantics. Notice that, in order to avoid confusion, we indicate the function for the evaluation of expressions with $\mathcal{E}(\_)$ rather than $[\![\_]\!]$, which is the symbol we use for our encoding function.

- When making a recursive invocation of a service, we will often write all parameters which remain equal to the formal parameters as "…". For example, consider the recursive call to the counting process Count, where the counter n has to be increased by 1, while all other parameters remain the same: $\mathsf{Count}(n + 1, \ldots)$. The employment of "…" is also extended to the usual case of a list of elements in a tuple, as in $\tilde{w} = \langle w_1, \ldots, w_j \rangle$, and in a similar way inside a service definition when a common structure is used for each element of a tuple. For example, see the definition of process Send, where "…" are used to point out that each element of tuple $\tilde{w}$ is sent using P as endpoint.

- We will make frequent use of the symbol "$\circ$" in the encoding: it stands for a parameter which always generates a match. We could have substituted "$\circ$" by a special name and obtain the same result, but this could have reduced the readability of the encoding.

- Even if the semantics and congruence relation would require a renaming when a new name has to be introduced in order to avoid entity capturing, in the execution traces of the services defined by our encoding (see Theorems 2 and 3 further below) we will show decorated versions of the same names. I.e., we will add a number of zeroes as superscript in order to identify different names coming from the same definition. This method is used in order to allow one to better understand the origin of the identifiers.

A.3 Description of the encoding

The encoding of a generic polyadic COWS service s into monadic COWS is as follows:

$$[\![s]\!]^* = [\,lock, unlock\,]([\![s]\!] \mid \mathsf{Lockit}()) \qquad (A.1)$$

The possible execution steps of a service s involve killing and communication. When a **kill** action is available, there is no protocol to be followed, as the same action is performed in the encoded service in one step. All the work in the encoding is thus for the management of the communication steps. As can be seen from (A.1), the service named Lockit is put in parallel with the encoded service, communicating with it through channels *lock, unlock*. This approach is taken in order to ensure that there is only one instance of locking service Lockit, guaranteeing that every encoded communication is executed as an atomic action: at each "communication round", an invoke action is chosen among all unguarded invoke actions via a synchronisation on channel *lock*. The encoding of the chosen action will then engage a protocol at the end of which one of these two conditions hold:

- a best-matching request action has been found and the communication has happened, substituting all involved variables;

- no matching request action (or no request action at all) has been found

$$\llbracket \mathbf{kill}(k) \rrbracket = \mathbf{kill}(k) \qquad \llbracket \lBrace s \rBrace \rrbracket = \lBrace \llbracket s \rrbracket \rBrace \qquad \llbracket s_1 \mid s_2 \rrbracket = \llbracket s_1 \rrbracket \mid \llbracket s_2 \rrbracket \qquad \llbracket \mathbf{0} \rrbracket = \mathbf{0}$$

If $\mathsf{S}(n_1, \dots, n_j) = s$ then $\llbracket \mathsf{S}(m_1, \dots, m_j) \rrbracket = \mathsf{S'}(m_1, \dots, m_j)$ where $\mathsf{S'}(n_1, \dots, n_j) = \llbracket s \rrbracket$

$$\llbracket [\,d\,] s \rrbracket = \begin{cases} [\,d\,]\, \llbracket s \rrbracket & \text{if } d \text{ is a killer label} \\ [\,d, p_d\,](\llbracket s \rrbracket \mid \mathsf{Unknown}(p_d)) & \text{if } d \text{ is a variable} \\ [\,d, p_d\,](\llbracket s \rrbracket \mid \mathsf{Known}(p_d)) & \text{o.w.} \end{cases}$$

$$\llbracket \textstyle\sum_{i=0}^{r} p_i ?\tilde{w}_i .\, s_i \rrbracket \;=\; [\,k\,]\,(\llbracket p_0 ?\tilde{w}_0 .\, s_0 \rrbracket_k \mid \cdots \mid \llbracket p_r ?\tilde{w}_r .\, s_r \rrbracket_k)$$

$$\begin{aligned} \llbracket p ?\tilde{w} .\, s \rrbracket_k \;=\;& [\,k_1, p_1, cont\,](p.start?\langle |\tilde{w}| \rangle .\, (p.ack!\langle p_1 \rangle \mid \mathsf{Match}_{w_1}(p_1) \\ & \mid p_1 ?\langle \mathtt{false} \rangle .\, (\mathbf{kill}(k_1) \mid \lBrace \llbracket p ?\tilde{w} .\, s \rrbracket_k \rBrace) \\ & + p_1 ?\langle \mathtt{true} \rangle .\, (\mathbf{kill}(k) \mid \lBrace \mathsf{Receive}(p_1, cont) \\ & \qquad\qquad \mid cont?\langle \circ \rangle .\, (\mathsf{Subs}(p_{W_\alpha}, \dots p_{W_\omega}, cont) \mid cont?\langle \circ \rangle .\, (unlock!\langle \circ \rangle \mid \llbracket s \rrbracket)) \rBrace))) \end{aligned}$$

$$\begin{aligned} \llbracket u!\tilde{w} \rrbracket \;=\;& lock?\langle \circ \rangle .\, [\,c_u\,](p_u!\langle c_u \rangle \mid c_u?\langle 1 \rangle . \\ & \qquad\qquad [\,c_\alpha\,](p_{W_\alpha}!\langle c_\alpha \rangle \mid c_\alpha?\langle 1 \rangle .\;\; \dots \\ & [\,k, reset, result, go, countUp, countDown, countTotal, minAdd, minGet, minDone\,](\\ & \;\mathsf{Count}(0, countUp, countDown, countTotal, go, result) \\ & \;\mid \mathsf{WaitReqs}(u, |\tilde{w}|, countUp, countTotal, go, reset) \\ & \;\mid [\,startSending\,](\\ & \qquad \mathsf{ReqChooser}(result, startSending, countDown, minAdd, minGet, minDone, reset) \\ & \qquad \mid \mathsf{Send}(startSending, k)) \\ & \;\mid [\,c\,](\mathsf{Min}(\infty, c, minAdd, minGet, minDone) \mid c?\langle \mathtt{false} \rangle .\, \mathbf{0}) \\ & \;\mid reset?\langle \circ \rangle .\, (\mathbf{kill}(k) \mid \lBrace unlock!\langle \circ \rangle \mid \llbracket u!\tilde{w} \rrbracket \rBrace)) \\ & \qquad\qquad \dots \;+\; c_\alpha?\langle 2 \rangle .\, (unlock!\langle \circ \rangle \mid \llbracket u!\tilde{w} \rrbracket)) \\ & \qquad\qquad + c_u?\langle 2 \rangle .\, (unlock!\langle \circ \rangle \mid \llbracket u!\tilde{w} \rrbracket)) \end{aligned}$$

$$\begin{aligned} \mathsf{Subs}(p_1, \dots, p_n, cont) \;=\;& [\,c_1, c_2\,](p_1!\langle c_1 \rangle \\ & \quad \mid c_1?\langle 1 \rangle .\, \mathsf{Subs}(p_2, \dots, p_n, cont) \\ & \quad + c_1?\langle 2 \rangle .\, (p_1.stop!\langle c_2 \rangle \mid c_2?\langle \circ \rangle .\, (\mathsf{Known}(p_1) \mid \mathsf{Subs}(p_2, \dots, p_n, cont)))) \\ \mathsf{Subs}(cont) \;=\;& cont!\langle \circ \rangle \\[1ex] \mathsf{Known}(p) \;=\;& [\,C\,](p?\langle C \rangle .\, (\mathsf{Known}(p) \mid C!\langle 1 \rangle)) \\[1ex] \mathsf{Unknown}(p) \;=\;& [\,C\,](p?\langle C \rangle .\, (\mathsf{Unknown}(p) \mid C!\langle 2 \rangle) \\ & \quad + p.stop?\langle C \rangle .\, C!\langle \circ \rangle) \\[1ex] \mathsf{Lockit}() \;=\;& lock!\langle \circ \rangle \mid unlock?\langle \circ \rangle .\, \mathsf{Lockit}() \end{aligned}$$

Table A.3: Encoding functions from polyadic COWS to monadic COWS (part 1). $W_\alpha, \dots, W_\omega$ are all the variables in $\tilde{w}$, and $|\tilde{w}|$ is the number of elements in tuple $\tilde{w}$. Notice also that the two parts "$[\,c_u\,](p_u!\langle c_u \rangle \mid c_u?\langle 1 \rangle .$" and "$+\; c_u?\langle 2 \rangle .\, (unlock!\langle \circ \rangle \mid \llbracket u!\tilde{w} \rrbracket))$" are to be added to the encoding of $u!\tilde{w}$ only if u is a variable.

$$
\begin{array}{l}
\mathsf{Count}(n,countUp,countDown,countTotal,go,result) = \\
\hspace{2em}[\,C,P\,](\\
\hspace{4em}countUp.return?\langle C\rangle.\ countUp.channel?\langle P\rangle.\ (\\
\hspace{16em}\mathsf{Count}(n+1,\ldots)\\
\hspace{15em}\mid go?\langle\circ\rangle.\ [\,k,cUp,cTot\,](\mathsf{CountSubs}(0,cUp,cTot)\\
\hspace{26em}\mid \mathsf{CountMatches}(k,P,result,cUp,cTot))\\
\hspace{15em}\mid C!\langle\circ\rangle)\\
\hspace{4em}+\ countDown?\langle C\rangle.\ [\,p_1\,](p_1!\langle n > 1\rangle\\
\hspace{16em}\mid p_1?\langle\mathtt{true}\rangle.\ (\mathsf{Count}(n-1,\ldots)\mid C!\langle\mathtt{true}\rangle)\\
\hspace{16em}+\ p_1?\langle\mathtt{false}\rangle.\ C!\langle\mathtt{false}\rangle)\\
\hspace{4em}+\ countTotal?\langle C\rangle.\ (\mathsf{Count}(n,\ldots)\mid C!\langle n\rangle))
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{CountSubs}(n,cUp,cTot) & = & [\,P\,](cUp?\langle P\rangle.\ (\mathsf{CountSubs}(n+1,cUp,cTot)\mid P!\langle\circ\rangle)\\
 & & \hspace{1em}+\ cTot?\langle P\rangle.\ P!\langle n\rangle)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{WaitReqs}(u,nParams,countUp,countTotal,go,reset) = \\
\hspace{4em}[\,P_1\,](u.start!\langle nParams\rangle \mid u.ack?\langle P_1\rangle.\ [\,c\,](countUp.return!\langle c\rangle \mid countUp.channel!\langle P_1\rangle\\
\hspace{24em}\mid c?\langle\circ\rangle.\ \mathsf{WaitReqs}(\ldots))\\
\hspace{14em}+\ [\,X\,](u.start?\langle X\rangle.\ [\,c,N\,](countTotal!\langle c\rangle \mid c?\langle N\rangle.\ \mathsf{StartMatch}(go,N)\\
\hspace{33em}+\ c?\langle 0\rangle.\ reset!\langle\circ\rangle)))
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{StartMatch}(go,n) & = & [\,p\,](p!\langle n > 0\rangle\\
 & & \hspace{2em}\mid p?\langle\mathtt{true}\rangle.\ (go!\langle\circ\rangle \mid \mathsf{StartMatch}(go,n-1))\\
 & & \hspace{2em}+\ p?\langle\mathtt{false}\rangle.\ \mathbf{0})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{ReqChooser}(result,startSending,countDown,minAdd,minGet,minDone,reset) = \\
[\,c_1,C,TOT,CHAN\,](\\
\hspace{1em}result?\langle C\rangle.\ (C!\langle c_1\rangle \mid c_1.tot?\langle TOT\rangle.\ c_1.chan?\langle CHAN\rangle.\\
\hspace{2em}[\,c_2\,](countDown!\langle c_2\rangle\\
\hspace{6em}\mid c_2?\langle\mathtt{true}\rangle.\ (minAdd.tot!\langle TOT\rangle \mid minAdd.chan!\langle CHAN\rangle \mid minDone?\langle\circ\rangle.\ \mathsf{ReqChooser}(\ldots))\\
\hspace{5em}+\ c_2?\langle\mathtt{false}\rangle.\ (minAdd.tot!\langle TOT\rangle \mid minAdd.chan!\langle CHAN\rangle \mid minDone?\langle\circ\rangle.\\
\hspace{3em}[\,c_3,MIN,P\,](minGet!\langle c_3\rangle \mid c_3.min?\langle MIN\rangle.\ c_3.chan?\langle P\rangle.\ (P!\langle\mathtt{true}\rangle \mid startSending!\langle P\rangle)\\
\hspace{16em}+\ c_3.min?\langle\infty\rangle.\ c_3.chan?\langle P\rangle.\ (P!\langle\mathtt{false}\rangle \mid reset!\langle\circ\rangle))))))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Min}(m,c,minAdd,minGet,minDone) = \\
\hspace{2em}[\,N,C\,](\\
\hspace{4em}minAdd.tot?\langle\infty\rangle.\ minAdd.chan?\langle C\rangle.\ (minDone!\langle\circ\rangle \mid \mathsf{Min}(m,c,\ldots)\mid C!\langle\mathtt{false}\rangle)\\
\hspace{3em}+\ minAdd.tot?\langle N\rangle.\ minAdd.chan?\langle C\rangle.\\
\hspace{14em}[\,p\,](p!\langle N > m\rangle\\
\hspace{15em}\mid p?\langle\mathtt{true}\rangle.\ (minDone!\langle\circ\rangle \mid \mathsf{Min}(m,c,\ldots)\mid C!\langle\mathtt{false}\rangle)\\
\hspace{15em}+\ p?\langle\mathtt{false}\rangle.\\
\hspace{14em}[\,p_2\,](p_2!\langle N < m\rangle\\
\hspace{15em}\mid p_2?\langle\mathtt{true}\rangle.\ (minDone!\langle\circ\rangle \mid \mathsf{Min}(N,C,\ldots)\mid c!\langle\mathtt{false}\rangle)\\
\hspace{15em}+\ p_2?\langle\mathtt{false}\rangle.\\
\hspace{14em}[\,p_3,a\,](p_3!\langle a\rangle\\
\hspace{15em}\mid p_3?\langle a\rangle.\ (minDone!\langle\circ\rangle \mid \mathsf{Min}(N,C,\ldots)\mid c!\langle\mathtt{false}\rangle)\\
\hspace{15em}+\ p_3?\langle a\rangle.\ (minDone!\langle\circ\rangle \mid \mathsf{Min}(m,c,\ldots)\mid C!\langle\mathtt{false}\rangle)))))\\
\hspace{3em}+\ minGet?\langle C\rangle.\ (C.min!\langle m\rangle \mid C.chan!\langle c\rangle)
\end{array}
$$

Table A.4: Encoding functions from polyadic COWS to monadic COWS (part 2). Notice that we pass a parameter u to service WaitReqs, as the channel on which an invoke action can happen may also be a variable. Although the syntax does not allow to write $u?\tilde{w}$, we use it anyway, because the service WaitReqs will always be called passing a *name* as actual parameter for u.

$$\begin{array}{l}
\mathsf{HandleMatch}(\mathit{result,p,m,cont,k,cUp}) = \\
\quad [\,q\,](q!\langle m\rangle \mid q?\langle \mathit{1}\rangle.\,\mathit{cont}!\langle\circ\rangle \\
\qquad\qquad + q?\langle \mathit{2}\rangle.\,(\mathbf{kill}(k) \mid \{\![\,c_1, C_2\,](\mathit{result}!\langle c_1\rangle \mid c_1?\langle C_2\rangle.\,(C_2.\mathit{tot}!\langle\infty\rangle \mid C_2.\mathit{chan}!\langle p\rangle))\}\!\}\,) \\
\qquad\qquad + q?\langle \mathit{3}\rangle.\,[\,c\,](\mathit{cUp}!\langle c\rangle \mid c?\langle\circ\rangle.\,\mathit{cont}!\langle\circ\rangle))
\end{array}$$

$$\begin{array}{l}
\mathsf{CountMatches}(\mathit{k,p,result,cUp,cTot}) = \\
\quad [\,c\,](\\
\quad\quad [\,M_1, p_1\,](p.\mathit{name}!\langle w_1\rangle \mid p.\mathit{chan}!\langle p_1\rangle \mid p_1?\langle M_1\rangle.\,\mathsf{HandleMatch}(\mathit{result,p,}M_1\mathit{,c,k,cUp}) \mid c?\langle\circ\rangle. \\
\quad\quad [\,M_2, p_2\,](p.\mathit{name}!\langle w_2\rangle \mid p.\mathit{chan}!\langle p_2\rangle \mid p_2?\langle M_2\rangle.\,\mathsf{HandleMatch}(\mathit{result,p,}M_2\mathit{,c,k,cUp}) \mid c?\langle\circ\rangle. \\
\quad\quad \ldots \\
\quad\quad [\,M_j, p_j\,](p.\mathit{name}!\langle w_j\rangle \mid p.\mathit{chan}!\langle p_j\rangle \mid p_j?\langle M_j\rangle.\,\mathsf{HandleMatch}(\mathit{result,p,}M_j\mathit{,c,k,cUp}) \mid c?\langle\circ\rangle. \\
\quad\quad [\,c_2, \mathit{TOT}\,](\mathit{cTot}!\langle c_2\rangle \mid c_2?\langle \mathit{TOT}\rangle.\,[\,c_1, C_2\,](\mathit{result}!\langle c_1\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid c_1?\langle C_2\rangle.\,(C_2.\mathit{tot}!\langle \mathit{TOT}\rangle \mid C_2.\mathit{chan}!\langle p\rangle)))))\ldots)))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{Send}(\mathit{startSending,k}) & = & [\,P\,]\mathit{startSending}?\langle P\rangle.\,(\\
& & \quad P!\langle w_1\rangle \mid P.\mathit{ack}?\langle\circ\rangle.\,(\\
& & \quad P!\langle w_2\rangle \mid P.\mathit{ack}?\langle\circ\rangle.\,(\\
& & \quad \ldots \\
& & \quad P!\langle w_j\rangle \mid P.\mathit{ack}?\langle\circ\rangle.\,\mathbf{kill}(k))\ldots))
\end{array}$$

If $w_i \in \mathcal{V}$, then

$$\begin{array}{l}
\mathsf{Match}_{w_i}(p) = \\
\quad [\,Y, P_2\,](\\
\quad\quad p.\mathit{name}?\langle Y\rangle.\,p.\mathit{chan}?\langle P_2\rangle.\,[\,c_1\,](p_{w_i}!\langle c_1\rangle \mid c_1?\langle \mathit{1}\rangle.\,[\,q\,](q!\langle Y\rangle \mid q?\langle w_i\rangle.\,(P_2!\langle \mathit{1}\rangle \mid \mathsf{Match}_{w_{i+1}}(p)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad + [\,Z\,](q?\langle Z\rangle.\,P_2!\langle \mathit{2}\rangle)) \\
\qquad\qquad\qquad\qquad\qquad + c_1?\langle \mathit{2}\rangle.\,(P_2!\langle \mathit{3}\rangle \mid \mathsf{Match}_{w_{i+1}}(p))))
\end{array}$$

If $w_i \in \mathcal{N}$, then

$$\begin{array}{l}
\mathsf{Match}_{w_i}(p) = \\
\quad [\,Y, P_2\,](\\
\quad\quad p.\mathit{name}?\langle Y\rangle.\,p.\mathit{chan}?\langle P_2\rangle.\,[\,q\,](q!\langle Y\rangle \mid q?\langle w_i\rangle.\,(P_2!\langle \mathit{1}\rangle \mid \mathsf{Match}_{w_{i+1}}(p)) \\
\qquad\qquad\qquad\qquad\qquad + [\,Z\,](q?\langle Z\rangle.\,P_2!\langle \mathit{2}\rangle)))
\end{array}$$

$$\begin{array}{lcl}
\mathsf{Receive}(\mathit{p,cont}) & = & p?\langle w_1\rangle.\,(p.\mathit{ack}!\langle\circ\rangle \mid \\
& & \quad p?\langle w_2\rangle.\,(p.\mathit{ack}!\langle\circ\rangle \mid \\
& & \quad \ldots \\
& & \quad p?\langle w_j\rangle.\,(p.\mathit{ack}!\langle\circ\rangle \mid \mathit{cont}!\langle\circ\rangle)\ldots))
\end{array}$$

Table A.5: Encoding functions from polyadic COWS to monadic COWS (part 3). The definition of services CountMatches, Send, Match_{w_i} and Receive is based on the corresponding input/output tuple $\tilde{w} = \langle w_1, \ldots, w_j\rangle$.

In both cases, the lock is made available again through a synchronisation on channel *unlock*. In the second case, the service encoding the selected invoke action is also restarted.

The first thing the encoding of invoke action $u!\tilde{w}$ has to do is to check whether all variables in $\tilde{w}$ (and the variable possibly represented by u) have been substituted, in accordance with the semantics of COWS, which requires an invoke action to be able to execute only if all of its variables have been substituted. This is done by trying to contact a service of type Known for each variable. If the service exists, it responds to the request returning a "1", indicating that the corresponding variable has been substituted; in the case the service does not exist, the communication will be intercepted by a service of type Unknown: it will return a "2", meaning that the variable at hand has still to be substituted. In the first case, the protocol can continue (checking the next variable, or starting the actual encoding of the communication); in the second case, the encoding of the invoke action is restarted and the lock is handed back. Notice that it would have not been possible to acquire the lock after having checked for the substitution of all variables in the invoke action: if that would have been the case, there would have been concurrency issues when two or more invoke actions would have been activated at the same time (one could finish to check for the substitution of variables before another and get the lock, even if theoretically both have the same right to be executed).

When the check for the substitution of all variables in the invoke action is successful, a series of services is enabled: we will give a brief description of the task of each one here below.

- Count counter service which keeps track of the number of (unguarded) request actions on the same channel as the invoke action. For each new participant, a new service is spawned, which will count the substitutions generated by the communication with that participant. Service Count is also used as a reverse-counter in order to know when all request actions participating in the match counting service have given a response, and thus the best-matching one can be chosen.
- WaitReqs contacts all (unguarded) request actions on the same channel as the encoded invoke action and with the same number of parameters, counting them (through service Count). When no more request actions are available, the matching protocol is started through a number of synchronisations on channel *go* equal to the number of "registered" request actions. If no request actions have answered the call, the encoding for invoke action is restarted and the lock is returned.
- ReqChooser receives all matching results, and starts the actual communication with (one of) the best-matching request action(s).
- Send performs the actual sending of all parameters in tuple $\tilde{w}$ to the selected best-matching request action.
- Min calculates the minimum of a given series of numbers, each of which is stored in couple with a channel name. When requested, the name corresponding to the minimal number is returned. This service is used in order to select the best-matching request action.

- $reset?\langle\circ\rangle.\ \ldots$ handles the cases when the encoding of the invoke service has to be reset.

The service encoding a request action $p?\tilde{w}$ waits to be contacted by an invoke action, and then starts the protocol for assessing the match: services of the type Match_{w_i} (described below) are used for this purpose. When a negative response is obtained from the service encoding the invoke action, the encoding of current request action is restarted; whereas a positive response causes the actual reception of invoke parameters and the continuation with the encoding of the residual service. As request actions can guard services in a sum (non-deterministic multiple choice), if the service is selected for communication all other services in the sum are killed (cfr. the encoding for sum).

Services of type Match_{w_i} are defined differently according to the type of input entities they have to check: if w_i is a name, the service needs only to check w_i to be equal to the given input; if w_i is a variable, substitution has also to be taken into account. As explained before, the actual substitution state of a variable is obtained via services of type Known and Unknown: when the variable has been substituted, a Known service answers, while Unknown answers when no substitutions have been made for the variable at hand. Thus, there are three cases for the matching of a couple output/input entities, each associated by Match_{w_i} to a return value:

- the input entity is a name and matches with the output name: the matching service can continue (return value 1);
- the input entity is a name but is different from the output name: the matching service is cancelled and the request action is discarded by the invoke action protocol (return value 2);
- the input entity is a variable: the number of substitutions generated by the simulated communication is increased and the matching service continues (return value 3).

The actions associated to the different return values of a service of type Match_{w_i} are executed by service HandleMatch, which in turn relies on service CountSubs in order to keep count of the substitutions.

A.3.1 Service Min and scheduling fairness

Service Min needs a special treatment, as it can cause some scheduling problems. In fact, it can be noticed that when two numbers received by service Min through a communication on channel *minAdd.tot* are equal, the choice of which to keep as the minimum is delegated to a non-deterministic sum; i.e., the choice is left to the hypothetical runtime scheduler of the language. The exact semantics of the choice between more than one best-matching request actions is that *one among all of the actions* is chosen at once. With the proposed approach, however, the choice is repeated by choosing among only two of the actions at a time. This can change the probabilities for request actions to be selected.

For example, consider the following service

$$p!\langle a,b,c\rangle \mid \underbrace{p?\langle X,b,c\rangle}_{S_1}.\,\mathbf{0} \mid \underbrace{p?\langle a,X,c\rangle}_{S_2}.\,\mathbf{0} \mid \underbrace{p?\langle a,b,X\rangle}_{S_3}.\,\mathbf{0}$$

If the scheduler would assign the same probability to be chosen to all (best-matching) request actions S_1, S_2 and S_3, we would have that

$$P(S_1) = P(S_2) = P(S_3) = \frac{1}{3}$$

With the current approach, the order in which the request actions are serviced will make the difference. For example, if the encoding of invoke action $p!\langle a,b,c\rangle$ will service matching results in the order A, B, C, we will obtain that

$$P(S_1) = P(S_2) = \tfrac{1}{4}$$
$$P(S_3) = \tfrac{1}{2}$$

It is possible to compensate for this bias, but it would lead to have a much more complicated Min service. We have thus chosen to maintain the simpler version and concentrate on the actual work of the encoding.

A.4 Example

We propose an example of encoding of the following service:

$$[\,a\,][\,b\,][\,c\,][\,X\,][\,Y\,](p!\langle a,b,c\rangle \mid p?\langle a,X,Y\rangle.\,\mathbf{0} \mid p?\langle c,X,a\rangle.\,\mathbf{0})$$

In Tables A.6, A.7 and A.8 we show all definitions peculiar to the service at hand. Notice that, as explained in the previous sections, the communication which will eventually accomplish is the one between the encodings of $p!\langle a,b,c\rangle$ and $p?\langle a,X,Y\rangle.\,\mathbf{0}$.

$$\begin{array}{l}
\llbracket [\,a\,][\,b\,][\,c\,][\,X\,][\,Y\,](p!\langle a,b,c\rangle \mid p?\langle a,X,Y\rangle.\,\mathbf{0} \mid p?\langle c,X,a\rangle.\,\mathbf{0})\rrbracket = \\
\qquad [\,a\,][\,b\,][\,c\,][\,X\,][\,p_X\,][\,Y\,][\,p_Y\,](\llbracket p!\langle a,b,c\rangle\rrbracket \mid [\,k'\,](\llbracket p?\langle a,X,Y\rangle.\,\mathbf{0}\rrbracket_{k'}) \mid [\,k''\,](\llbracket p?\langle c,X,a\rangle.\,\mathbf{0}\rrbracket_{k''}) \mid \mathsf{Unknown}(p_X) \mid \mathsf{Unknown}(p_Y))
\end{array}$$

$$\begin{array}{lcl}
\llbracket p!\langle a,b,c\rangle\rrbracket & = & lock?\langle\circ\rangle.\,[\,k, reset, result, go, countUp, countDown, countTotal, minAdd, minGet, minDone\,](\\
 & & \mathsf{Count}(0,countUp,countDown,countTotal,go,result) \\
 & & \mid \mathsf{WaitReqs}(p,3,countUp,countTotal,go,reset) \\
 & & \mid [\,startSending\,](\\
 & & \qquad \mathsf{ReqChooser}(result,startSending,countDown,minAdd,minGet,minDone,reset) \\
 & & \qquad \mid \mathsf{Send}(startSending,k)) \\
 & & \mid [\,c\,](\mathsf{Min}(\infty,c,minAdd,minGet,minDone) \mid c?\langle\mathtt{false}\rangle.\,\mathbf{0}) \\
 & & \mid reset?\langle\circ\rangle.\,(\mathbf{kill}(k) \mid \{\!|unlock!\langle\circ\rangle \mid \llbracket p!\langle a,b,c\rangle\rrbracket\,|\!\}\,)) \\
 & & \\
\llbracket p?\langle a,X,Y\rangle.\,\mathbf{0}\rrbracket_{k'} & = & [\,k_1, p_1, cont\,](p.start?\langle 3\rangle.\,(p.ack!\langle p_1\rangle \mid \mathsf{Match}_{1,a}(p_1) \\
 & & \mid p_1?\langle\mathtt{false}\rangle.\,(\mathbf{kill}(k_1) \mid \{\!|\,\llbracket p?\langle a,X,Y\rangle.\,s\rrbracket_{k'}\,|\!\}\,) \\
 & & +\, p_1?\langle\mathtt{true}\rangle.\,(\mathbf{kill}(k') \mid \{\!|\mathsf{Receive}_1(p_1,cont) \\
 & & \qquad\qquad \mid cont?\langle\circ\rangle.\,(\mathsf{Subs}(p_X,p_Y,cont) \mid cont?\langle\circ\rangle.\,(unlock!\langle\circ\rangle \mid \mathbf{0}))|\!\}\,))) \\
 & & \\
\llbracket p?\langle c,X,a\rangle.\,\mathbf{0}\rrbracket_{k''} & = & [\,k_1, p_1, cont\,](p.start?\langle 3\rangle.\,(p.ack!\langle p_1\rangle \mid \mathsf{Match}_{2,c}(p_1) \\
 & & \mid p_1?\langle\mathtt{false}\rangle.\,(\mathbf{kill}(k_1) \mid \{\!|\,\llbracket p?\langle c,X,a\rangle.\,s\rrbracket_{k''}\,|\!\}\,) \\
 & & +\, p_1?\langle\mathtt{true}\rangle.\,(\mathbf{kill}(k'') \mid \{\!|\mathsf{Receive}_2(p_1,cont) \\
 & & \qquad\qquad \mid cont?\langle\circ\rangle.\,(\mathsf{Subs}(p_X,cont) \mid cont?\langle\circ\rangle.\,(unlock!\langle\circ\rangle \mid \mathbf{0}))|\!\}\,)))
\end{array}$$

$$\begin{array}{l}
\mathsf{CountMatches}(k,p,result,cUp,cTot) = \\
\qquad\qquad [\,c\,](\\
\qquad\qquad\quad [\,M_1, p_1\,](p.name!\langle a\rangle \mid p.chan!\langle p_1\rangle \mid p_1?\langle M_1\rangle.\,\mathsf{HandleMatch}(result,p,M_1,c,k,cUp) \mid c?\langle\circ\rangle. \\
\qquad\qquad\quad [\,M_2, p_2\,](p.name!\langle b\rangle \mid p.chan!\langle p_2\rangle \mid p_2?\langle M_2\rangle.\,\mathsf{HandleMatch}(result,p,M_2,c,k,cUp) \mid c?\langle\circ\rangle. \\
\qquad\qquad\quad [\,M_3, p_3\,](p.name!\langle c\rangle \mid p.chan!\langle p_3\rangle \mid p_3?\langle M_3\rangle.\,\mathsf{HandleMatch}(result,p,M_3,c,k,cUp) \mid c?\langle\circ\rangle. \\
\qquad\qquad\quad [\,c_2, TOT\,](cTot!\langle c_2\rangle \mid c_2?\langle TOT\rangle.\,[\,c_1, C_2\,](result!\langle c_1\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid c_1?\langle C_2\rangle.\,(C_2.tot!\langle TOT\rangle \mid C_2.chan!\langle p\rangle))))))))
\end{array}$$

Table A.6: Services resulting from the encoding of proposed example (Part 1).

$$
\begin{array}{lcl}
\mathsf{Send}(\mathit{startSending},k) & = & [\,P\,]\mathit{startSending}?\langle P\rangle.\,(P!\langle a\rangle \mid P.\mathit{ack}?\langle\circ\rangle.\,(\\
 & & \qquad\qquad\qquad P!\langle b\rangle \mid P.\mathit{ack}?\langle\circ\rangle.\,(\\
 & & \qquad\qquad\qquad P!\langle c\rangle \mid P.\mathit{ack}?\langle\circ\rangle.\,\mathbf{kill}(k)))) \\[1em]
\mathsf{Receive}_1(p,\mathit{cont}) & = & p?\langle a\rangle.\,(p.\mathit{ack}!\langle\circ\rangle \mid \\
 & & p?\langle X\rangle.\,(p.\mathit{ack}!\langle\circ\rangle \mid \\
 & & p?\langle Y\rangle.\,(p.\mathit{ack}!\langle\circ\rangle \mid \mathit{cont}!\langle\circ\rangle))) \\[1em]
\mathsf{Receive}_2(p,\mathit{cont}) & = & p?\langle c\rangle.\,(p.\mathit{ack}!\langle\circ\rangle \mid \\
 & & p?\langle X\rangle.\,(p.\mathit{ack}!\langle\circ\rangle \mid \\
 & & p?\langle a\rangle.\,(p.\mathit{ack}!\langle\circ\rangle \mid \mathit{cont}!\langle\circ\rangle))) \\[1em]
\mathsf{Subs}(p_X,p_Y,\mathit{cont}) & = & [\,c_1,c_2\,](p_X!\langle c_1\rangle \\
 & & \qquad \mid c_1?\langle 1\rangle.\,\mathsf{Subs}(p_Y,\mathit{cont}) \\
 & & \qquad + c_1?\langle 2\rangle.\,(p_1.\mathit{stop}!\langle c_2\rangle \mid c_2?\langle\circ\rangle.\,(\mathsf{Known}(p_X) \mid \mathsf{Subs}(p_Y,\mathit{cont})))) \\[1em]
\mathsf{Subs}(p_Y,\mathit{cont}) & = & [\,c_1,c_2\,](p_Y!\langle c_1\rangle \\
 & & \qquad \mid c_1?\langle 1\rangle.\,\mathsf{Subs}(\mathit{cont}) \\
 & & \qquad + c_1?\langle 2\rangle.\,(p_1.\mathit{stop}!\langle c_2\rangle \mid c_2?\langle\circ\rangle.\,(\mathsf{Known}(p_X) \mid \mathsf{Subs}(\mathit{cont})))) \\[1em]
\mathsf{Subs}(p_X,\mathit{cont}) & = & [\,c_1,c_2\,](p_X!\langle c_1\rangle \\
 & & \qquad \mid c_1?\langle 1\rangle.\,\mathsf{Subs}(\mathit{cont}) \\
 & & \qquad + c_1?\langle 2\rangle.\,(p_1.\mathit{stop}!\langle c_2\rangle \mid c_2?\langle\circ\rangle.\,(\mathsf{Known}(p_X) \mid \mathsf{Subs}(\mathit{cont}))))
\end{array}
$$

Table A.7: Services resulting from the encoding of proposed example (Part 2).

$$
\begin{aligned}
\mathsf{Match}_{1,a}(p) &= [\,V_1, P\,](p.name?\langle V_1\rangle.\,p.chan?\langle P\rangle.\,[\,q\,](q!\langle V_1\rangle \mid q?\langle a\rangle.\,(P!\langle 1\rangle \mid \mathsf{Match}_{1,X}(p)) \\
&\qquad\qquad + [\,V_2\,](q?\langle V_2\rangle.\,P!\langle 2\rangle))) \\[1em]
\mathsf{Match}_{1,X}(p) &= [\,V_1, P\,](p.name?\langle V_1\rangle.\,p.chan?\langle P\rangle.\,[\,c_1\,](p_X!\langle c_1\rangle \mid c_1?\langle 1\rangle.\,[\,q\,](q!\langle V_1\rangle \mid q?\langle X\rangle.\,(P!\langle 1\rangle \mid \mathsf{Match}_{1,Y}(p)) \mid [\,V_2\,](q?\langle V_2, 1\rangle.\,P!\langle 2, \circ\rangle) \\
&\qquad\qquad\qquad + [\,V_2\,](q?\langle V_2\rangle.\,P!\langle 2\rangle))) \\
&\qquad\qquad + c_1?\langle 2\rangle.\,(P!\langle 3\rangle \mid \mathsf{Match}_{1,Y}(p))))) \\[1em]
\mathsf{Match}_{1,Y}(p) &= [\,V_1, P\,](p.name?\langle V_1\rangle.\,p.chan?\langle P\rangle.\,[\,c_1\,](p_X!\langle c_1\rangle \mid c_1?\langle 1\rangle.\,[\,q\,](q!\langle V_1\rangle \mid q?\langle X\rangle.\,P!\langle 1\rangle \mid [\,V_2\,](q?\langle V_2, 1\rangle.\,P!\langle 2, \circ\rangle) \\
&\qquad\qquad\qquad + [\,V_2\,](q?\langle V_2\rangle.\,P!\langle 2\rangle))) \\
&\qquad\qquad + c_1?\langle 2\rangle.\,P!\langle 3\rangle))) \\[1em]
\mathsf{Match}_{2,c}(p) &= [\,V_1, P\,](p.name?\langle V_1\rangle.\,p.chan?\langle P\rangle.\,[\,q\,](q!\langle V_1\rangle \mid q?\langle a\rangle.\,(P!\langle 1\rangle \mid \mathsf{Match}_{2,X}(p)) \\
&\qquad\qquad + [\,V_2\,](q?\langle V_2\rangle.\,P!\langle 2\rangle))) \\[1em]
\mathsf{Match}_{2,X}(p) &= [\,V_1, P\,](p.name?\langle V_1\rangle.\,p.chan?\langle P\rangle.\,[\,c_1\,](p_X!\langle c_1\rangle \mid c_1?\langle 1\rangle.\,[\,q\,](q!\langle V_1\rangle \mid q?\langle X\rangle.\,(P!\langle 1\rangle \mid \mathsf{Match}_{2,a}(p)) \mid [\,V_2\,](q?\langle V_2, 1\rangle.\,P!\langle 2, \circ\rangle) \\
&\qquad\qquad\qquad + [\,V_2\,](q?\langle V_2\rangle.\,P!\langle 2\rangle))) \\
&\qquad\qquad + c_1?\langle 2\rangle.\,(P!\langle 3\rangle \mid \mathsf{Match}_{2,a}(p))))) \\[1em]
\mathsf{Match}_{2,a}(p) &= [\,V_1, P\,](p.name?\langle V_1\rangle.\,p.chan?\langle P\rangle.\,[\,q\,](q!\langle V_1\rangle \mid q?\langle a\rangle.\,P!\langle 1\rangle \\
&\qquad\qquad + [\,V_2\,](q?\langle V_2\rangle.\,P!\langle 2\rangle)))
\end{aligned}
$$

Table A.8: Services resulting from the encoding of proposed example (Part 3).

A.5 Properties of the encoding

In order to prove operational correspondence between polyadic and monadic versions of COWS (Theorems 2 and 3), we need an extended congruence relation $\hat{\equiv}$, which is defined as the minimal congruence relation satisfying laws for $\equiv$ plus the following law:

$$[\,p_X\,](s_1\{^n/_X\} \mid \mathsf{Unknown}(p_X) \mid \mathsf{Known}(p_X)) \mathrel{\hat{\equiv}} (s_1\{^n/_X\})\{^n/_X\}^S$$

where *service substitution* $\{^n/_X\}^S$ is defined on Monadic COWS by the following set of laws:

$$\mathbf{kill}(k)\{^n/_X\}^S = \mathbf{kill}(k) \qquad \mathbf{0}\{^n/_X\}^S = \mathbf{0} \qquad \{\!|s|\!\}\,\{^n/_X\}^S = \{\!|s\{^n/_X\}^S|\!\}$$

$$(s_1 \mid s_2)\{^n/_X\}^S = s_1\{^n/_X\}^S \mid s_2\{^n/_X\}^S \qquad (s_1 + s_2)\{^n/_X\}^S = s_1\{^n/_X\}^S + s_2\{^n/_X\}^S$$

$$\text{If } \mathsf{S}(n_1, \ldots, n_j) = s \text{ then } \mathsf{S}(m_1, \ldots, m_j)\{^n/_X\}^S = (s\{^{m_1,\ldots,m_j}/_{n_1,\ldots,n_j}\})\{^n/_X\}^S$$

$$p!\langle e\rangle\{^n/_X\}^S = p!\langle e\rangle \qquad [\,k\,]s\{^n/_X\}^S = [\,k\,](s\{^n/_X\}^S)$$

$$[\,m\,]s\{^n/_X\}^S = [\,m\,](s\{^n/_X\}^S) \text{ if } s{\not\downarrow}_{p_X}$$

$$(p?\langle w\rangle.\, s)\{^n/_X\}^S = p?\langle w\rangle.\,(s\{^n/_X\}^S)$$

$$[\,c\,](p_X!\langle c\rangle \mid c?\langle 1\rangle.\, s_1 + c?\langle 2\rangle.\, s_2)\{^n/_X\}^S = s_1\{^n/_X\}^S$$

where the last law is always applied with precedence w.r.t. all other laws.
$\{^n/_X\}^S$ will be used in order to transform a service in the form $[\![s]\!]\{^n/_X\}$ into $[\![s\{^n/_X\}]\!]$ by removing all pieces designed to manage the state of variable X.

Observation 1. *$s_1 \equiv s_2$ implies $s_1 \mathrel{\hat{\equiv}} s_2$.*

Proof. Comes directly from the definition of $\hat{\equiv}$. □

Notation 1. *The writing $s \overset{\alpha}{\Rightarrow} s'$ means that:*

- *either $s = s'$*
- *or $\alpha = \dagger$ and $s \xrightarrow{\dagger} s'$*
- *or $\alpha = p\ \lfloor\emptyset\rfloor\ \tilde{w}\ \tilde{v}$ and $\exists p', s_1, \ldots, s_t, i_1, \ldots, i_l, X, n$ such that*
  - $s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_t} s_t = s'$
  - *$l = |\tilde{w}| = |\tilde{v}|$ (the number of elements in the tuples)*

- $1 < i_1 < \cdots < i_l < t$
- $\alpha_{i_1} = p' \lfloor\emptyset\rfloor \langle w_1\rangle \langle v_1\rangle$,
  $\alpha_{i_2} = p' \lfloor\emptyset\rfloor \langle w_2\rangle \langle v_2\rangle$,
  $\ldots$,
  $\alpha_{i_l} = p' \lfloor\emptyset\rfloor \langle w_l\rangle \langle v_l\rangle$.

Theorem 2. *If* $s \xrightarrow{\alpha} s'$ *then* $[\![s]\!]^* \overset{\alpha}{\Rightarrow}\ \hat{=}\ [\![s']\!]^*$.

Proof. Taken service s, we suppose that $s \xrightarrow{\alpha} s'$. There are two possibilities: $\alpha = \dagger$ or $\alpha = p \lfloor\emptyset\rfloor\ \tilde{w}\ \tilde{v}$.

- $\alpha = \dagger$. In this case, we have that $s{\downarrow_{kill}}$. By construction of the encoding (in particular, for the treatment of killing labels, parallel composition and **kill** actions), we also have that $[\![s]\!]^*{\downarrow_{kill}}$. Then, $[\![s]\!]^* \xrightarrow{\dagger} s''$. Given that $[\![\_]\!]$ is an homomorphism for $|$ and $\{\!|\_|\!\}$, and that the scope of k (where k is the killer label involved in the action $\xrightarrow{\dagger}$) in $[\![s]\!]^*$ includes the same (encoded) services as in s (by the behaviour of the encoding regarding delimiters), and that $[\![\mathbf{kill}(k)]\!] = \mathbf{kill}(k)$ for every $k \in \mathcal{K}$, we can conclude that $s'' \equiv [\![s']\!]^*$. By Obs. 1, we have the thesis.

- $\alpha = p \lfloor\emptyset\rfloor\ \tilde{w}\ \tilde{v}$. In this case, rule (com) has been applied to derive the transition. Thus we know that there exist two unguarded sub-services of s, say s_1, s_2, and an index $j \in [1, r]$ such that $s_1 \equiv \sum_{i=1}^{r} p_i?\tilde{w}_i.\, s_i$, $s_2 \equiv p!\tilde{e}$, $p = p_j$, $\tilde{w} = \tilde{w}_j$ and $\mathcal{E}(\tilde{e}) = \tilde{v}$. A sequence of transitions which can be executed by $[\![s]\!]^*$ is the following (the transitions are numbered for convenience):

$$[\![s]\!]^* \xrightarrow[1]{lock\ \lfloor\emptyset\rfloor\ \langle\circ\rangle\ \langle\circ\rangle} \xrightarrow[(a)]{}^{*} \xrightarrow[2]{p.start\ \lfloor\emptyset\rfloor\ \langle|\tilde{w}|\rangle\ \langle|\tilde{v}|\rangle} \xrightarrow[3]{p.ack\ \lfloor\emptyset\rfloor\ \langle P_1\rangle\ \langle p_1\rangle}$$
$$\xrightarrow[4]{countUp.return\ \lfloor\emptyset\rfloor\ \langle C\rangle\ \langle c\rangle} \xrightarrow[5]{countUp.channel\ \lfloor\emptyset\rfloor\ \langle P\rangle\ \langle p_1\rangle} \xrightarrow[6]{c\ \lfloor\emptyset\rfloor\ \langle\circ\rangle\ \langle\circ\rangle} \xrightarrow[7]{p.start\ \lfloor\emptyset\rfloor\ \langle X\rangle\ \langle|\tilde{v}|\rangle}$$
$$\xrightarrow[8]{countTotal\ \lfloor\emptyset\rfloor\ \langle C\rangle\ \langle c\rangle} \xrightarrow[9]{c^0\ \lfloor\emptyset\rfloor\ \langle N\rangle\ \langle n\rangle} \xrightarrow[10]{p^0\ \lfloor\emptyset\rfloor\ \langle\mathtt{true}\rangle\ \langle\mathtt{true}\rangle} \xrightarrow[11]{go\ \lfloor\emptyset\rfloor\ \langle\circ\rangle\ \langle\circ\rangle}$$
$$\xrightarrow[12]{p^{00}\ \lfloor\emptyset\rfloor\ \langle\mathtt{false}\rangle\ \langle\mathtt{false}\rangle} \xrightarrow[(b)]{}^{*} \xrightarrow[13]{minGet\ \lfloor\emptyset\rfloor\ \langle C\rangle\ \langle c_3\rangle} \xrightarrow[14]{c_3.min\ \lfloor\emptyset\rfloor\ \langle MIN\rangle\ \langle m\rangle}$$
$$\xrightarrow[15]{c_3.chan\ \lfloor\emptyset\rfloor\ \langle P^0\rangle\ \langle p_1\rangle} \xrightarrow[16]{p_1\ \lfloor\emptyset\rfloor\ \langle\mathtt{true}\rangle\ \langle\mathtt{true}\rangle} \xrightarrow[17]{\dagger} \xrightarrow[18]{startSending\ \lfloor\emptyset\rfloor\ \langle P^{00}\rangle\ \langle p_1\rangle} \xrightarrow[(c)]{}^{*}$$
$$\xrightarrow[19]{continue\ \lfloor\emptyset\rfloor\ \langle\circ\rangle\ \langle\circ\rangle} \xrightarrow[20]{unlock\ \lfloor\emptyset\rfloor\ \langle\circ\rangle\ \langle\circ\rangle} s''$$

  Notice that the given list of transitions is not completely expanded. In particular, the following points have to be taken into account:

  - Transitions labelled by (a) are present if there is any variable in s_2, and represent the decision of the substitution state of each variable. These transitions are sequences in the form $\xrightarrow{p_W \lfloor\emptyset\rfloor \langle C\rangle \langle c_W\rangle} \xrightarrow{c \lfloor\emptyset\rfloor \langle 1\rangle \langle 1\rangle}$ for each variable W in s_2 (as we know, by the fact that operational semantics allows us to derive the transition $s \xrightarrow{\alpha} s'$, that they are all substituted).

- Transitions 2 - 6 are repeated for each unguarded request action $p?\tilde{w}$, generating private channels c and p_1 for internal communication.
- The value n in transition 9 is the total number of unguarded request actions (with total number of parameters equal to $|\tilde{v}|$) on p.
- Transitions 10 and 11 are repeated n times.
- Transitions labelled by (b) represent the check for matching in each request action and the selection of the minimal-substituting request action. Their form will be left out, but can be deduced from the corresponding parts in the encoding.
- Transition 16 represents the communication to the chosen request action $p_j?\tilde{w_j}.\ s_j$ (p_1 is the private channel for the communication with that request action), starting the actual substitution of $\tilde{w_j}$ by $\tilde{v}$.
- Transition 17 represents the termination of all non-chosen request actions in the sum possibly present in s_1.
- Transitions labelled by (c) represent the actual sending of all parameters in $\tilde{v}$ to the chosen request action and is in the form

$$\xrightarrow{p_1 \lfloor\emptyset\rfloor \langle w_{j_1}\rangle \langle v_1\rangle} \xrightarrow{p_1.ack^0 \lfloor\emptyset\rfloor \langle\circ\rangle \langle\circ\rangle}$$
$$\xrightarrow{p_1 \lfloor\emptyset\rfloor \langle w_{j_2}\rangle \langle v_2\rangle} \xrightarrow{p_1.ack^0 \lfloor\emptyset\rfloor \langle\circ\rangle \langle\circ\rangle}$$
$$\dots$$
$$\xrightarrow{p_1 \lfloor\emptyset\rfloor \langle w_{j_l}\rangle \langle v_l\rangle} \xrightarrow{p_1.ack^0 \lfloor\emptyset\rfloor \langle\circ\rangle \langle\circ\rangle}$$

where $l, j_i, \dots j_l$ are defined in Notation 1.

By the above sequence of transitions, we have that $[\![s]\!]^* \xRightarrow{p \lfloor\emptyset\rfloor\ \tilde{w_j}\ \tilde{v}} s''$.
From the previous observations on the structure of s, we have that s' contains service s_j and that all variables W_{j_i} in $\tilde{w_j}$ have been substituted in s' with the corresponding values v_i in $\tilde{v}$. The rest of the structure of s' is the same as in s. This means that in $[\![s']\!]^*$ there are no services $\mathsf{Known}(W_{j_i})$ nor $\mathsf{Unknown}(W_{j_i})$ for the variables in $\tilde{w_j}$, and the substitution state of these (already substituted) variables is checked nowhere in s'. As for s'', we know from the definition of the encoding that it contains a service $\mathsf{Known}(W_{j_i})$ for each variable W_{j_i} in $\tilde{w_j}$ and the substitution state of these variables is decided in the circumstances described by the encoding. The definition of the encoding allows us to state $[\![s']\!]^* \equiv s''\{^{\tilde{v}}/_{\tilde{w_j}}\}^S$, and thus to conclude $s'' \triangleq [\![s']\!]^*$ by definition of $\triangleq$.

□

Theorem 3. *If* $[\![s]\!]^* \xrightarrow{\alpha_1} s'$ *then there exist* α_2 *and* s'' *such that* $s' \overset{\alpha_2}{\Rightarrow} s''$ *and*

- *either* $s'' \equiv [\![s]\!]^*$
- *or* $s'' \mathrel{\hat{=}} [\![s_1]\!]^*$ *for some* s_1 *such that* $s \xrightarrow{\alpha_2} s_1$.

Proof. Given service s, we suppose that $[\![s]\!]^* \xrightarrow{\alpha_1} s'$. There are two cases.

- If $[\![s]\!]^* \downarrow_{kill}$, then $\alpha_1 = \dagger$ by semantics rule ($\mathsf{del}_{\mathsf{pass}}$). By construction of $[\![\_]\!]$, also $s \downarrow_{kill}$. Taken $\alpha_2 = \dagger$ and $s'' = s'$, we have that $s' \overset{\dagger}{\Rightarrow} s''$ and $s \overset{\dagger}{\longrightarrow} s_1$. The relation $s'' \equiv [\![s_1]\!]^*$ is shown the same way as in Theorem 2. By Obs. 1, we have the thesis.
- If $[\![s]\!]^* \not\downarrow_{kill}$, there are two cases. If s contains no unguarded invoke action, the service is deadlocked, and so is $[\![s]\!]^*$, thus the hypothesis does not apply. If, on the contrary, an invoke action exists unguarded in s, by construction of the encoding we know that $\alpha_1 = lock\ \lfloor\emptyset\rfloor\ \langle\circ\rangle\ \langle\circ\rangle$, as the communication on channel $lock$ is the only enabled interaction in $[\![s]\!]^*$. Let the service which has interacted with Lockit (the only one offering output $lock!\langle\circ\rangle$) be $[\![u!\tilde{e}]\!]$. There are three possible situations regarding $u!\tilde{e}$:

  1. not all variables in $u!\tilde{e}$ have been substituted;
  2. $u = p$, $\mathcal{E}(\tilde{e}) = \tilde{v}$ (i.e., all variables in $u!\tilde{e}$ have been substituted), but no unguarded service of the form $p?\tilde{w}.\,s_2$ such that $|\tilde{w}| = |\tilde{v}|$ exists in s;
  3. $u = p$, $\mathcal{E}(\tilde{e}) = \tilde{v}$ and there exists an unguarded service s_{req} in s such that $s_{\text{req}} \equiv p?\tilde{w}.\,s_2$, $|\tilde{w}| = |\tilde{v}|$ and $\tilde{w}$ is best-matching with respect to all other available request actions on p.

  In case (1), an interaction with a service of type $\mathsf{Unknown}$ makes the protocol to reset the encoding. Thus, setting $\alpha_2 = unlock\ \lfloor\emptyset\rfloor\ \langle\circ\rangle\ \langle\circ\rangle$, we have $s'' \equiv [\![s]\!]^*$ by construction of the encoding, and by Obs. 1 the thesis.
  In case (2), a communication labelled in the form $p.start\ \lfloor\emptyset\rfloor\ \langle X\rangle\ \langle|\ \tilde{v}\ |\rangle$ ends the search for available request actions (see definition of $\mathsf{WaitReqs}$). As the minimal number of substitutions will be returned to be ∞, the encoding of $u!\tilde{e}$ is reset also in this case (see $\mathsf{ReqChooser}$), resulting in the same situation as in case (1).
  The situation in (3) depicts the case in which a communication can happen in s between $u!\tilde{e}$ and s_{req}. We thus set $\alpha_2 = p\ \lfloor\emptyset\rfloor\ \tilde{w}\ \tilde{v}$, and s_1 to be the residual of such communication. Remembering that action $lock?\langle\circ\rangle$ guarding the rest of the service in $[\![p!\tilde{e}]\!]$ has already been consumed in s' and that no interaction between this subservice and any service of type $\mathsf{Unknown}$ can happen (all variables in $\tilde{e}$ have been substituted), s' will execute the protocol counting the matches with $[\![p?\tilde{w}.\,s_{\text{req}}]\!]_k$ on a specific private channel p_1. Once the protocol has finished the match computation for all available request actions on p, channel p_1 will be used to send all values in $\tilde{v}$ to the residual of $[\![p?\tilde{w}.\,s_{\text{req}}]\!]_k$, as it is the best-matching. The protocol terminates with the usual communication on channel $unlock$. Thus we can state $s' \overset{\alpha_2}{\Rightarrow} s''$, where s'' differs from $[\![s]\!]^*$ in the following points:

  – the residual of $[\![u!\tilde{e}]\!]$ is congruent to $\mathbf{0}$;

– the residual of $[\![p?\tilde{w}.\, s_{\text{req}}]\!]_k$ is congruent to $[\![s_{\text{req}}]\!]$;

– for each variable W_i in $\tilde{w}$: W_i has been substituted on its scope by the corresponding value v_i in $\tilde{v}$, and a new service of type $\mathsf{Known}(p_{W_i})$ has been activated, terminating at the same time the corresponding process of type $\mathsf{Unknown}$.

Noticing that s_1 differs from s in the following points:

– the residual of $u!\tilde{e}$ is congruent to $\mathbf{0}$;

– the residual of $p?\tilde{w}.\, s_{\text{req}}$ is congruent to s_{req};

– each variable W_i in $\tilde{w}$ has been substituted by the corresponding value v_i in $\tilde{v}$ on its scope,

we can state that $s''\{{}^{\tilde{v}'}\!/\!_{\tilde{w}'}\}^{\mathcal{S}} \equiv [\![s_1]\!]^*$, where $\tilde{w}'$ are all the variables in $\tilde{w}$ and $\tilde{v}'$ are the corresponding values in $\tilde{v}$. We can thus conclude $s'' \triangleq [\![s_1]\!]^*$ by definition of $\triangleq$.

□

Appendix B

Scows model of the SENSORIA finance case study

Here we present the Scows model of the SENSORIA Finance case study, which represents a service providing semi-automatic bank loan management. The scope of the service is to provide a customer with an automatic loan proposal if the balance data and loan amount are assessed favorably, and to require human intervention (as high in the bank hierarchy as the situation requires) only in the cases for which automatic response is not allowed (i.e., borderline cases and unfavorably evaluated financial situations). Table B.1 shows the services modelling the behaviour of the automatic process internal to the bank, while TableB.2 shows the default settings for the rates defined in a parametric way. Please notice that this model represents a point of view located inside the inside the bank, and thus all human action is emulated by the Portal service.

```
Finalize(id) =
 (portal#.goodbye#!<id>,1.0)
;

Accept(id,creditManagement,update,desired,result,ratingData) =
 (creditManagement#.generateOffer#!<id,ratingData>,1.0)
 | [agreementData]
  (creditReq#.generateOffer#?<id,agreementData>,1.0).
   (
    (portal#.offerToClient#!<id,agreementData>,1.0)
    | [accepted]
     (creditReq#.offerToClient#?<id,accepted>,1.0).
      [if#][then#][end#]
```

Table B.1: Scows model of the financial case study (cont.).

```
      (
       (if#.then#!<accepted>,1.0)
       | (if#.then#?<false#>,1.0).
          (update.desired!<false#>,1.0)
        + (if#.then#?<true#>,1.0).
          ( (creditManagement#.acceptOffer#!<id,accepted>,1.0)
           | (portal#.acceptOffer#?<id>,1.0).
             (update.desired!<false#>,1.0)
          )
      )
   )
;

Decline(id,creditManagement,update,desired,result,ratingData) =
 (creditManagement.generateDecline#!<id,ratingData>,1.0)
 | [declineData]
  (creditReq#.generateDecline#?<id,declineData>,1.0).
   (
    (portal#.declineToClient#!<id,declineData>,1.0)
    | [updateDesired]
     (creditReq#.declineToClient#?<id,updateDesired>,1.0).
      (update.desired!<updateDesired>,1.0)
   )
;

Approval(id,update,desired,result,ratingData,end,manualAcceptance) =
 [if_approval#][then_approval#][x]
 (
  (if_approval#.then_approval#!<result>,1.0)
  | ( (if_approval#.then_approval#?<bbb#>,1.0).
     (portal#.requestClerkApproval#!<id,ratingData>,1.0)
    + (if_approval#.then_approval#?<x>,1.0).
     (portal#.requestSupervisorApproval#!<id,ratingData>,1.0)
   )
  |
   [approvalData]
   (creditReq#.approvalresult#?<id,manualAcceptance,approvalData>,1.0).
    (approval#.end!<dummy#>,1.0)
```

Table B.1: Scows model of the financial case study (cont.).

```
 )
;

Decision(id,creditManagement,update,desired,result,ratingData) =
 [if_decision#][then_decision#][end#][undef#][manualAcceptance][x]
 (
  (if_decision#.then_decision#!<result>,1.0)
  | ( (
    (if_decision#.then_decision#?<aaa#>,1.0).
      (
     [var#][set#]
     (
      (var#.set#!<undef#>,1.0)
      | (var#.set#?<manualAcceptance>,1.0).
        (approval#.end#!<dummy#>,1.0)
     )
     )
     )
    + (if_decision#.then_decision#?<x>,1.0).
        Approval(id,update,desired,result,ratingData,end#,manualAcceptance)
   )
  | (approval#.end#?<dummy#>,1.0).
    [if_decision#][then_decision#]( (if_decision#.then_decision#!<result,manualAcceptance>,1.0)
     | [x1][x2][x3][x4]
      (
       (if_decision#.then_decision#?<aaa#,x1>,1.0).
          Accept(id,creditManagement,update,desired,result,ratingData)
      + (if_decision#.then_decision#?<x2,true#>,1.0).
          Accept(id,creditManagement,update,desired,result,ratingData)
      + (if_decision#.then_decision#?<x3,x4>,1.0).
          Decline(id,creditManagement,update,desired,result,ratingData)
      )
    )
 )
;

RatingCalculation(id,creditManagement,update,desired,loginname,firstname,lastname) =
 (
```

Table B.1: Scows model of the financial case study (cont.).

```
   (rating#.calculateRating#!<id,loginname,firstname,lastname>,1.0)
   | [result][ratingData]
    (creditReq#.calculateRating#?<id,result,ratingData>,1.0).
     Decision(id,creditManagement,update,desired,result,ratingData)
 )
;

HandleBalanceAndSecurityData(id,creditManagement,comp,update,desired,loginname,firstname,lastname) =
 [flow#][end#]
 (
  ( (portal#.enterBalanceData#!<id>,1.0)
   | [balancePackage]
    (creditReq#.enterBalanceData#?<id,balancePackage>,1.0).
    (
     (balance#.updatebalanceRating#!<id,loginname,firstname,lastname,balancePackage>,1.0)
     | (creditReq#.updatebalanceRating#?<id>,1.0). (flow#.end#!<dummy#>,1.0)
    )
  )
  |
  ( (portal#.enterSecurityData#!<id>,1.0)
   | [securityPackage]
    (creditReq#.enterSecurityData#?<id,securityPackage>,1.0).
    (
     (security#.updatesecurityRating#!<id,loginname,firstname,lastname,securityPackage>,1.0)
     | (creditReq#.updatesecurityRating#?<id>,1.0). (flow#.end#!<dummy#>,1.0)
    )
  )
  | (flow#.end#?<dummy#>,1.0).
    (flow#.end#?<dummy#>,1.0).
     ( RatingCalculation(id,creditManagement,update,desired,loginname,firstname,lastname)
     )
 )
;

Creation(id,creditManagement,comp,update,desired,loginname,firstname,lastname) =
 [customerid][creditAmount][creditType][monthlyInstalment]
 (creditReq#.createNewCreditRequest#?<id,customerid,creditAmount,creditType,monthlyInstalment>,1.0).
  (
```

Table B.1: Scows model of the financial case study (cont.).

```
    (creditManagement.initCreditData#!<id,customerid,creditAmount,creditType,monthlyInstalment>,1.0)
    | (creditReq#.initCreditData#?<id>,1.0).
     (
      [working#](portal#.createNewCreditRequest#!<id,working#>,1.0)
      | ( HandleBalanceAndSecurityData(id,creditManagement,comp,update,desired,loginname,firstname,lastname)
       )
     )
   )
;

Main_loop(id, creditManagement, comp, update, desired, loginname, firstname, lastname) =
    (repeat#.until#?<dummy#>,1.0).(
      ( Creation(id,creditManagement,comp,update,desired,loginname,firstname,lastname)
       | ((update.desired?<true#>,1.0).
          (
           (kill(k_loop),1.0) | { (repeat.until!<dummy#>,1.0) }
          )
        + (update.desired?<false#>,1.0). Finalize(id)
        )
      )
      | Main_loop(id, creditManagement, comp, update, desired, loginname, firstname, lastname)
    )
;

CompensateCreation()=
    [x1][x2] (comp#.creation#?<x1,x2>,1.0).(
        (comp#.x2!<dummy#>,1.0)
        | CompensateCreation()
    )
;

CompensateBSec()=
    [x3][x4] (comp#.handleBalanceAndSecurityData#?<x3,x4>,1.0). (
        (comp#.x4!<dummy#>,1.0)
        | CompensateBSec()
    )
;
```

Table B.1: Scows model of the financial case study (cont.).

```
Main(id,creditManagement,loginname,firstname,lastname) =
 ( [k]
  (
   [repeat#][until#]
   (
    (repeat#.until#!<dummy#>,1.0)
    |
    Main_loop(id, creditManagement, comp#, update#, desired#, loginname, firstname, lastname)
   )
   |
    (creditReq#.cancel#?<id>,1.0).
     ( {(kill(k),1.0) | (raise#.abort#!<dummy#>,1.0)} )
  )
  |
   (raise#.abort#?<dummy#>,1.0).
    (
     [end#]
     ( (comp#.creation#!<creation#,end#>,1.0)
      | (comp#.end#?<dummy#>,1.0).
       ( (comp#.handleBalanceAndSecurityData!<handleBalanceAndSecurityData,end#>,1.0)
        | (comp#.end#?<dummy#>,1.0).
          (portal#.abortProcess#!<id>,1.0)
       )
     )
    )
 )
;

CreditRequest(customerManagement,creditManagement) =
[k][raise#]
 (
  [id][name][password]
  (creditReq#.initialize#?<id,name,password>,1.0).
   (
    (customerManagement.checkUser#!<id,name,password>,1.0)
    | [userOK]
     (creditReq#.checkUser#?<id,userOK>,1.0).
      (
```

Table B.1: Scows model of the financial case study (cont.).

```
        {(portal#.initialize#!<id,userOK>,1.0)}
        | [if#][then#]
         ( (if#.then#!<userOK>,1.0)
          | (if#.then#?<false#>,1.0).
            ( {
              {(kill(k),1.0)}
              |
              (raise#.abort#!<dummy#>,1.0)
             }
            )
           + (if#.then#?<true#>,1.0).
            ( (customerManagement.getCustomerData#!<id,name,password>,1.0)
             | [loginname][firstname][lastname]
              (creditReq#.getCustomerData#?<id,loginname,firstname,lastname>,1.0).
               [update#][desired#] Main(id,creditManagement,loginname,firstname,lastname)
            )
         )
       )
       | {CreditRequest(customerManagement, creditManagement)}
    )
 )
;

Portal(k_clear) =
 [id#][name#][pwd#]
 (
  (creditReq#.initialize#!<id#,name#,pwd#>,1.0)
  | [userOK]
    (portal#.initialize#?<id#,userOK>,1.0).
    [if#][then#]
    ( (if#.then#!<userOK>,1.0)
      | (if#.then#?<false#>,1.0).
        (kill(k_clear),1)
      + (if#.then#?<true#>,1.0).
        [k]
        (
          [customerid#][amount#][mortgage#][instalment#](
           creditReq#.createNewCreditRequest#!<id#,customerid#,amount#,mortgage#,instalment#>,1.0)
```

Table B.1: Scows model of the financial case study (cont.).

```
| [working](portal#.createNewCreditRequest#?<id#,working>,1.0).
(
  (portal#.enterBalanceData#?<id#>,1.0).
  [balancePackage1#](creditReq#.enterBalanceData#!<id#,balancePackage1#>,1.0)
  |
  (portal#.enterSecurityData#?<id#>,1.0).
  [securityPackage1#](creditReq#.enterSecurityData#!<id#,securityPackage1#>,1.0)
  |
  [agreement][declineData]
  ( (portal#.offerToClient#?<id#,agreement>,1.0).
    [nonDet3#][choice#]
    (
      (nonDet3#.choice#!<dummy#>,1.0)
      | (nonDet3#.choice#?<dummy#>,1.0).
        (
          (kill(k),1.0)
          | { (creditReq#.offerToClient#!<id#,true#>,1.0)
            |
            (portal#.goodbye#?<id#>,1.0). (kill(k_clear),1)
          }
        )
      + (nonDet3#.choice#?<dummy#>,1.0).
        (
          (kill(k),1.0)
          | { (creditReq#.offerToClient#!<id#,false#>,1.0)
            |
            (portal#.goodbye#?<id#>,1.0). (kill(k_clear),1)
          }
        )
    )
  +
  (portal#.declineToClient#?<id#,declineData>,1.0).
    [nonDet4#][choice#]
    (
        (
          (kill(k),1.0)
          | { (creditReq#.declineToClient#!<id#,false#>,1.0)
            |
```

Table B.1: Scows model of the financial case study (cont.).

```
                      (portal#.goodbye#?<id#>,1.0). (kill(k_clear),1)
                    }
                  )
              )
            )
          )
        )
    )
 )
 |
 (
  Portal_for_clerks()
  |
  Portal_for_supervisors()
 )
;

Portal_for_clerks()=
[id][ratingData]
  (portal#.requestClerkApproval#?<id,ratingData>,1.0).(
    [nonDet5#][choice#]
    ( (nonDet5#.choice#!<dummy#>,clerkDecisionRate)
     | (nonDet5#.choice#?<dummy#>,clerkAcceptRate).
        [approvedDataByClerk#](creditReq#.approvalresult#!<id,true#,approvedDataByClerk#>,1.0)
      + (nonDet5#.choice#?<dummy#>,clerkRejectRate).
        [nonApprovedDataByClerk#](creditReq#.approvalresult#!<id,false#,nonApprovedDataByClerk#>,1.0)
    )
    | Portal_for_clerks()
  )
;

Portal_for_supervisors()=
  [id][ratingData]
   (portal#.requestSupervisorApproval#?<id,ratingData>,1.0).(
    [nonDet6#][choice#]
    ( (nonDet6#.choice#!<dummy#>,supDecisionRate)
     | (nonDet6#.choice#?<dummy#>,supAcceptRate).
     [approvedDataBySup#](creditReq#.approvalresult#!<id,true#,approvedDataBySup#>,1.0)
```

Table B.1: Scows model of the financial case study (cont.).

```
      + (nonDet6#.choice#?<dummy#>,supRejectRate).
     [nonApprovedDataBySup#](creditReq#.approvalresult#!<id,false#,nonApprovedDataBySup#>,1.0)
    )
    | Portal_for_supervisors()
  )
;

CustomerManagement(customerManagement) =
 [id][name][password]
  (customerManagement.checkUser#?<id,name,password>,1.0).
   [nonDet7#][choice#]
   ( (nonDet7#.choice#!<dummy#>,loginDecisionRate)
    | (nonDet7#.choice#?<dummy#>, loginFailRate).
        (creditReq#.checkUser#!<id,false#>,1.0)
     + (nonDet7#.choice#?<dummy#>, loginOkRate).
        ( (creditReq#.checkUser#!<id,true#>,1.0)
         |
          (customerManagement.getCustomerData#?<id,name,password>,1.0).
           [loginname#][firstname#][lastname#](
            creditReq#.getCustomerData#!<id,loginname#,firstname#,lastname#>,1.0)
        )
    | {CustomerManagement(customerManagement)}
   )
;

CreditManagement(creditManagement) =
 [id][customerid][creditAmount][creditType][monthlyInstalment]
  (creditManagement.initCreditData#?<id,customerid,creditAmount,creditType,monthlyInstalment>,1.0).
  [k]
  (
   (creditReq#.initCreditData#!<id>,1.0)
   | [ratingData]
    (
     (creditManagement.generateOffer#?<id,ratingData>,1.0).
           ( [agreementData#](creditReq#.generateOffer#!<id,agreementData#>,1.0)
            | [accepted]
             (creditManagement.acceptOffer#?<id,accepted>,1.0).
               (portal#.acceptOffer#!<id>,1.0)
```

Table B.1: Scows model of the financial case study (cont.).

```
            )
       +
       (creditManagement.generateDecline#?<id,ratingData>,1.0).
              [declineData#] (creditReq#.generateDecline#!<id,declineData#>,1.0)
      )
     | [customeridUpd][creditAmountUpd][creditTypeUpd][monthlyInstalmentUpd]
      (
       (creditManagement.removeData#?<id>,1.0).
        (
         (kill(k),1.0) | { (creditReq#.removeData#!<id>,1.0) }
        )
       +
       (creditManagement.initCreditData#?<id,customeridUpd,creditAmountUpd,
                        creditTypeUpd,monthlyInstalmentUpd>,1.0).
        (
         (kill(k),1.0)
         | { (creditManagement.initCreditData#!<id,customeridUpd,creditAmountUpd,
                        creditTypeUpd,monthlyInstalmentUpd>,1.0) }
        )
      )
      | {CreditManagement(creditManagement)}
    )
  ;

  Calculator(calculator) =
   [id][balanceRating][securityRating]
    (calculator.performRatingCalculation#?<id,balanceRating>,1.0).
     [nonDet8#][choice#]
     (
      (nonDet8#.choice#!<dummy#>,assessmentDecisionRate)
      | (nonDet8#.choice#?<dummy#>,assessmentArate).
        [overallatingDataAAA#](rating#.performRatingCalculation#!<id,aaa#,overallatingDataAAA#>,1.0)
       + (nonDet8#.choice#?<dummy#>,assessmentBrate).
        [overallatingDataBBB#](rating#.performRatingCalculation#!<id,bbb#,overallatingDataBBB#>,1.0)
       + (nonDet8#.choice#?<dummy#>,assessmentCrate).
        [overallatingDataCCC#](rating#.performRatingCalculation#!<id,ccc#,overallatingDataCCC#>,1.0)
      | {Calculator(calculator)}
     )
```

Table B.1: Scows model of the financial case study (cont.).

```
;

Rating(calculator) =
 [id][loginname][firstname][lastname]
  (rating#.calculateRating#?<id,loginname,firstname,lastname>,1.0).
   [balanceRating][securityRating]
   (
    (balance#.calculatebalanceRating#!<id,loginname,firstname,lastname>,1.0)
    | (rating#.calculatebalanceRating#?<id,balanceRating>,1.0). nil
    |
      (
       (calculator.performRatingCalculation#!<id,balanceRating>,1.0)
       | [result][overallRating]
        (rating#.performRatingCalculation#?<id,result,overallRating>,1.0).
         (creditReq#.calculateRating#!<id,result,overallRating>,1.0)
      )
    | {Rating(calculator)}
   )
;

BalAnalysisPersistent(id,loginname,firstname,lastname)=
    (balance#.calculatebalanceRating#?<id,loginname,firstname,lastname>,1.0).(
        [balanceRating#](rating#.calculatebalanceRating#!<id,balanceRating#>,1.0)
        | BalAnalysisPersistent(id,loginname,firstname,lastname)
    )
;

BalanceAnalysisProvider() =
 [id][loginname][firstname][lastname][balancePackage]
  (balance#.updatebalanceRating#?<id,loginname,firstname,lastname,balancePackage>,1.0).(
   [k](
     (creditReq#.updatebalanceRating#!<id>,1.0)
     |
      BalAnalysisPersistent(id,loginname,firstname,lastname)
     | [loginnameUpd][firstnameUpd][lastnameUpd][balancePackageUpd]
      (
       (balance#.clearData#?<id>,1.0).
        (
```

Table B.1: Scows model of the financial case study (cont.).

```
         (kill(k),1.0) | { (creditReq#.clearData#!<id,b>,1.0) }
        )
       +
       (balance#.updatebalanceRating#?<id,loginnameUpd,firstnameUpd,lastnameUpd,balancePackageUpd>,1.0).
        (
        (kill(k),1.0)
        | { (balance#.updatebalanceRating#!<id,loginnameUpd,firstnameUpd,lastnameUpd,balancePackageUpd>,1.0) }
        )
      )
    )
   | {BalanceAnalysisProvider()}
   )
;

SecAnalysisPersistent(id,loginname,firstname,lastname, securityRating) =
    (security#.calculatesecurityRating#?<id,loginname,firstname,lastname>,1.0).(
        (rating#.calculatesecurityRating#!<id,securityRating>,1.0)
        | SecAnalysisPersistent(id,loginname,firstname,lastname,securityRating)
        )
;

SecurityAnalysisProvider() =
 [id][loginname][firstname][lastname][securityPackage]
  (security#.updatesecurityRating#?<id,loginname,firstname,lastname,securityPackage>,1.0).(
   [k]
   (
     (creditReq#.updatesecurityRating#!<id>,1.0)
     |
      SecAnalysisPersistent(id,loginname,firstname,lastname, securityRating)
     | [loginnameUpd][firstnameUpd][lastnameUpd][securityPackageUpd]
      (
       (security#.clearData#?<id>,1.0).
        (
         (kill(k),1.0) | { (creditReq#.clearData#!<id,s>,1.0) }
        )
       +
       (security#.updatesecurityRating#?<id,loginnameUpd,firstnameUpd,
                          lastnameUpd,securityPackageUpd>,1.0).
```

Table B.1: Scows model of the financial case study (cont.).

```
        (
        (kill(k),1.0)
        | { (security#.updatesecurityRating#!<id,loginnameUpd,firstnameUpd,
                         lastnameUpd,securityPackageUpd>,1.0) }
        )
      )
    )
    | {SecurityAnalysisProvider()}
  )
;

$
[k_clear][raise#][comp#][dummy#][true#][false#]
[aaa#][bbb#][ccc#][creation#][balanceAndSecurityData#](
    [customerManagement#][creditManagement#]
    ( CreditRequest(customerManagement#,creditManagement#)
     | CustomerManagement(customerManagement#)
     | CreditManagement(creditManagement#) )
    |
    [calculator#] ({Rating(calculator#)} | {Calculator(calculator#)} )
    |
    {BalanceAnalysisProvider()} | {SecurityAnalysisProvider()}
    |
    Portal(k_clear)
)
$
finished : [0 .. 1];

$
portal#.goodbye# <id#> : finished < 2 : finished' = 1 ;
```

Table B.1: Scows model of the financial case study (last).

Rate	Value
`loginDecisionRate`	1.0
`loginOkRate`	1.5
`loginFailRate`	0.3
`assessmentDecisionRate`	1.0
`assessmentArate`	0.5
`assessmentBrate`	1.3
`assessmentCrate`	0.2
`clerkDecisionRate`	0.2
`clerkAcceptRate`	0.3
`clerkRejectRate`	0.4
`supDecisionRate`	0.25
`supAcceptRate`	0.2
`supRejectRate`	0.6

Table B.2: Initial rate settings for the bank case study.

Bibliography

[1] http://www.java.com.

[2] http://www.sensoria-ist.eu/.

[3] http://jflex.de.

[4] http://www.cs.princeton.edu/~appel/modern/java/CUP.

[5] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time markov chains. In Rajeev Alur and Thomas A. Henzinger, editors, *Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 269–276, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[6] A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Martin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. Technical report, WS-BPEL TC OASIS, August 2006.

[7] Ashok Argent-Katwala, Allan Clark, Howard Foster, Stephen Gilmore, Philip Mayer, and Mirco Tribastone. Safety and response-time analysis of an automotive accident assistance service. pages 191–205. 2009.

[8] Michele Boreale, Roberto Bruni, Luis Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, Antonio Ravara, Davide Sangiorgi, Vasco Vasconcelos, and Gianluigi Zavattaro. SCC: a Service Centered Calculus. In M. Bravetti and G. Zavattaro, editors, *Proceedings of WS-FM 2006, 3rd International Workshop on Web Services and Formal Methods*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer Verlag, 2006.

[9] Michele Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. Sessions and Pipelines for Structured Service Programming. *Formal Methods for Open Object-Based Distributed Systems (FMOODS2008)*, 5051:19–38, 2008.

[10] Mario Bravetti and Gianluigi Zavattaro. Service oriented computing from a process algebraic perspective. *Journal of Logic and Algebraic Programming*, 70(1):3–14, 2007.

[11] Igor Cappello. Scows_lts, `http://disi.unitn.it/~cappello/index.php?vis=dl`.

[12] Igor Cappello and Paola Quaglia. A Tool for Checking Probabilistic Properties of COWS Services. Submitted for publication, 2010.

[13] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[14] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

[15] M. Fujita, P.C. McGeer, and J.C.-Y. Yang. Multi-Terminal Binary Decision Diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10:149–169(21), 1997.

[16] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[17] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *Proceedings of ICSOC'06*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006.

[18] H. L. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2005.

[19] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In B. Steffen and G. Levi, editors, *Proc. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 2937, pages 307–329. Springer, 2004.

[20] H. Hermanns, Meyer J. Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In B. Plateau, W. Stewart, and M. Silva, editors, *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[21] Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274(1-2):43 – 87, 2002.

[22] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[23] Jane Hillston. *A compositional approach to performance modelling*. Cambridge University Press, New York, NY, USA, 1996.

[24] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[25] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[26] J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. Springer, 1976.

[27] Bartek Klin and Vladimiro Sassone. Structural operational semantics for stochastic process calculi. In *FoSSaCS*, pages 428–442, 2008.

[28] Peter Kratzer. Monte carlo and kinetic monte carlo methods. Apr 2009.

[29] G.W.M. Kuntz. *Symbolic Semantics and Verification of Stochastic Process Algebras*. PhD thesis, Friedrich-Alexander-Universitaet Erlangen-Nuernberg, Erlangen, March 2006.

[30] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In R. De Nicola, editor, *Proc. of 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science. Springer, 2006. To appear.

[31] A. Lapadula, R. Pugliese, and F. Tiezzi. A WSDL-based type system for WS-BPEL. In *Proc. of COORDINATION'06*, volume 4038 of *Lecture Notes in Computer Science*, pages 145–163. Springer, 2006.

[32] A. Lapadula, R. Pugliese, and F. Tiezzi. C⊕WS: A timed service-oriented calculus. In *Proc. of 4th International Colloquium on Theoretical Aspects of Computing (ICTAC'07)*, Lecture Notes in Computer Science. Springer, 2007. To appear.

[33] OMG BPMI notation working group. Business Process Modeling Notation Specification, 2006. `http://www.bpmn.org`.

[34] Davide Prandi, Corrado Priami, and Paola Quaglia. Communicating by compatibility. *Journal of Logic and Algebraic Programming*, 75(2):167 – 181, 2008.

[35] Davide Prandi and Paola Quaglia. Stochastic COWS. In *Proc. 5th International Conference on Service Oriented Computing, ICSOC '07*, volume 4749 of *LNCS*, 2007.

[36] Corrado Priami. Stochastic pi-calculus. *Comput. J.*, 38(7):578–589, 1995.

[37] J. Recker, M. Indulska, M. Rosemann, and P. Green. How good is BPMN really? Insights from theory and practice. In J. Ljungberg and M. Andersson, editors, *Proceedings 14th European Conference on Information Systems*, 2006.

[38] Jan Recker and Jan Mendling. On the translation between BPMN and BPEL: Conceptual mismatch between process modeling languages, 2006.

[39] A. Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 1945.

[40] H. L. Younes. Ymer: A statistical model checker. In *Computer Aided Verification*, pages 429–433. 2005.