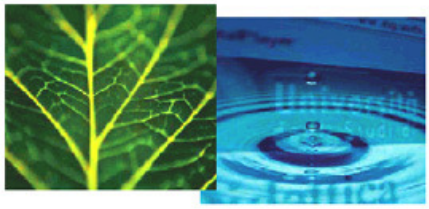
PhD Dissertation



International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Communication Priorities and Stochastic Measures for Web Services Modeling

Igor Cappello

February 2012

Abstract

Service Oriented Architecture is an important trend in the development of services composed of loosely coupled, heterogeneous and interacting components. We first consider COWS, which is a language tailored to model the behavioural aspects of these systems. We analyse the peculiarities of the communication mechanism of the language, a key ingredient in its modelling capabilities, presenting a separation result between a fragment of CCS equipped with global priorities and the fragment of COWS relevant for its communication mechanism. We then consider a stochastic approach to model the quantitative aspects of Web Services through SCOWS, a stochastic extension of COWS. We present a prototype tool, named SCOWS_lts, for the derivation of the complete representation of the behaviour of a SCOWS model as a Continuous Time Markov Chain. In order to validate the approach, a number of case studies are modelled in SCOWS considering both the SOA and the concurrency literature. Using PRISM as model checker, the results of the simulation phase are analysed using properties written to check the probabilistic behaviours of the considered systems.

Keywords
Process Calculi, Priorities, Model Checking, Operational Semantics, Language Encoding

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The integration of loosely coupled heterogeneous systems, known as Web Services, represents an issue that both the academic and the industrial communities have faced in recent years with increasing interest and efforts. The orchestration of Web Services in a Service Oriented Architecture, i.e. the automatic arrangement and coordination of executable services, is described using languages as WS-BPEL (Business Process Execution Language) [4], which is an executable description of the activities and interactions that can take place in a system. Since BPEL representations (given in XML format) of even small business processes are hardly readable and understandable by humans, typically a graphical representation of BPEL processes is given using BPMN (Business Process Model and Notation) [2] which helps human modeling and reasoning on large business processes. There is, however, a matching problem between BPMN and BPEL representations of business processes [30], which raises the problem of lack of a human-readable and formal way to represent business processes. The COWS [20] process algebra has been built considering these facts, providing a language formally defined both in its syntax and semantics.

1.1 Objective of the Thesis

The aim of this work is to investigate the peculiarities of COWS and its potentials in formal quantitative reasoning to build a platform to analyse concurrent systems in general, not only derived from the Service Oriented Architecture paradigm. The presented assessment is composed of two parts.

In the first part, the communication mechanism of COWS is analysed relatively to the expressing power of FAP [32], a process calculus that can be considered as a common ground for a comparison of prioritized capabilities among process calculi. The second part studies the feasibility of a practical approach to formally reason on COWS models considering quantitative measures.

1.2 Related Works

Given the scope of the thesis, we can divide the discussion of related works in two parts: the first is relative to COWS and to tools that support process algebras for modelling concurrent systems, while the second is relative to the assessment of the expressive power of the language itself.

The Calculus for the Orchestration of Web Services language (COWS) has been first introduced in [20]. With respect to that version of the language, we refer to a dialect in which agent identifiers are used instead of replication to express recursive behaviour. Stochastic extensions of the language have been presented in [27] for a monadic version of the language and in [31] for its polyadic version. In both cases, labelled semantics using scope opening/closing of identifier delimiters have been introduced. We refer mainly to the polyadic version presented in [31] and define a different approach to compute the rate associated to transitions of a SCOWS term, resulting in a less involved formalization of the transition rate computation.

The development of the presented tool, named SCOWS_lts, has its roots in

the work presented in [9], from which major performance and usability updates have been developed. The original work on SCOWS_lts was compared to a related project presented in [29], which describes a statistical approach, i.e. based on repetitions of an experiment to get statistically significant results, for the model checking phase of SCOWS models.

The tool we present can be also partially compared to CMC [13], an on-the-fly model checker supporting the verification of *qualitative* properties of COWS services. The approach used in CMC is slightly different from the one we adopt, since in CMC only fragments of the state space are generated as required by the verification of the considered property. In our approach, an optimized representation of the entire state space is generated, and the resulting structure can be reused to check properties using different quantitative configurations, since SCOWS_lts supports the use of parametric rates. The main purpose for the derivation of the whole Labelled Transition System (LTS) of a SCOWS service is the possibility of performing quantitative model checking on the CTMC derived from the LTS discarding transition labels.

In the field of the comparison of the expressive power of process calculi, two main approaches have been adopted. One is based on the comparison of the *absolute* expressive power of calculi: given a problem and a modelling language, the existence (or lack) of a solution satisfying some given properties is shown. If a language is able to solve the problem under these conditions and another language cannot, then a separation result can be derived. A second approach to the problem of comparing the expressive power of different process calculi is the study of encoding functions from one language to another. In this case, the *relative* expressive power of the language is studied, focussing on the properties that the used encoding function satisfies.

The *absolute* approach has been used in [7], where the Leader Election Problem is used to compare three dialects of CSP and in [33, 25], where the expressiveness of CCS [22], π-calculus [23] and Mobile Ambients [10] dialects are

compared using the same problem as a test base. Another example of this approach is presented in [32], where the Leader Election Problem is used to prove separation results between calculi equipped with different priority enforcing mechanisms. In [25, 32, 33] both behavioural and syntactic properties of encoding functions are taken into account. In particular, an encoding is defined *uniform* if it is distribution preserving, i.e. homomorphic w.r.t. parallel composition, permutation preserving, i.e. name invariant, and observation respecting, i.e. when considering maximal computations, observables are maintained.

The *relative* approach has been used, for example, in [14], where various combinations of features of communication primitives (synchronous vs asynchronous, monadic vs polyadic, message passing vs tuple spaces, pattern matching) are compared, showing the existence of an encoding (or lack thereof) of one variant into the other, resulting in a hierarchy of languages. In this work, an encoding is *reasonable* if it enjoys compositionality, name invariance, faithfulness (i.e. the semantics of the original term must not be modified by the encoding) and operational correspondence.

Another example of the second approach has been recently proposed in [5], where the criterion of replacement freeness is used to categorize process algebras. Replacement freeness is a property based on the sensitivity of a given process algebra, in terms of behaviour inhibition, to the replacement of a generic process with an invisible one. A weaker and a stronger version of the criterion are defined, inducing the creation of three categories: strong replacement-free calculi (insensitive to substitution of processes), weak replacement-free calculi (sensitive to substitution of non closed invisible processes), non replacement-free calculi (sensitive even to substitution of closed invisible processes). It is then shown that it is not possible to write an encoding from a more sensitive (e.g. CCS with global priorities, COWS) to a less sensitive process algebra (e.g. π-calculus, CCS), when the encoding function has to respect some properties. In particular, compositionality, interaction sensitiveness (based on barbs)

and independence preservation (no communication links must be introduced by the encoding) are taken into consideration.

1.3 Structure of the Thesis

Chapter 2 presents the formalization of the COWS language that we will refer to throughout the first part of the thesis. COWS syntax and operational semantics are presented and commented. Chapter 3 presents an assessment of the COWS prioritized communication mechanism, which is based on pattern matching and correlation sets. In particular, this assessment is based on a separation result between COWS_KF, the kill and protection free fragment of COWS, and FAP, the finite fragment of CCS equipped with global priorities. Chapter 4 presents SCOWS, first introduced in [31], a polyadic stochastic extension of COWS in which exponentially distributed delays are associated to basic actions. Differences with the original formulation, such as the different transition rate computation algorithm and the decoration system used in the operational semantics, are presented. SCOWS_lts, a tool for the derivation and compact representation of the behaviour of SCOWS services, is described in terms of implementation choices and optimizations. In Chapter 5 SCOWS_lts is applied to two case studies, the former inspired from a BPMN scenario and the latter derived from the comparison of a number of well-known algorithms solving the Mutual Exclusion problem. Finally, Chapter 6 draws conclusions from the presented work. Appendix A presents the proofs omitted from Chapter 3. Appendix B presents the code for the SCOWS_lts models mentioned in Chapter 5.

Chapter 2

COWS

In this chapter we describe COWS, in terms of the syntax and the operational semantics which formally define the behaviour of COWS terms. With respect to [20], here we present a version of the calculus which uses agent identifiers in place of replication for expressing iterative/recursive behaviour.

2.1 Syntax and Semantics of COWS

To define the syntax of COWS, we identify three countable and pairwise disjoint sets: $\mathcal{N}$ set of *names* $\mathcal{N}$ (ranged over by $m, n, o, p, m', n', o', p'$), $\mathcal{V}$ set of *write-once variables* (ranged over by v, v', x, y, x', y') and $\mathcal{K}$ set of *killer labels* (ranged over by k, k').

In the following, we will use d, d' to range over $\mathcal{N} \cup \mathcal{V} \cup \mathcal{K}$ and $u, u', \ldots$ to range over $\mathcal{N} \cup \mathcal{V}$. We use the notation $\widetilde{\_}$ for tuples, so, e.g., $\widetilde{n}$ stays for a tuple of elements in $\mathcal{N}$ and $\widetilde{u}$ stays for a tuple of elements in $\mathcal{N} \cup \mathcal{V}$.

With a slight abuse of notation, we sometimes read tuples as sets so, for example, we will write $u \in \widetilde{u}$ to mean that the element u occurs in $\widetilde{u}$.

We use $\sigma, \sigma', \ldots$ to denote substitutions of names for variables. Substitutions are functions from variables in $\mathcal{V}$ to elements of $\mathcal{N}$, which we write as finite collections of pairs in the form $\{n_1/x_1, \ldots, n_j/x_j\}$, where $\{x_1, \ldots, x_j\}$ is the domain of the substitution, and $\{n_1, \ldots, n_j\}$ is the cosupport of the substitution,

denoted by csp(σ). Given $\sigma = \{n_1/v_1, \ldots, n_k/v_k\}$, we define $|\sigma| = k$.

The empty substitution is denoted by ε, and given the two substitutions σ_1 and σ_2, we write $\sigma_1 \uplus \sigma_2$ to denote the union of σ_1 and of σ_2 which is defined only if the two substitutions have disjoint domains.

A communication endpoint is denoted as $u.u'$, where u represents the *partner* and u' the *operation* involved in communications modelled in COWS.

The syntax of COWS is presented in Table 2.1 in BNF form. As can be seen, nondeterministic choice has to be input-guarded. Endpoints for input (request) operations are of the form $p.o$, meaning that they cannot include COWS variables. This fact will have a crucial role in the results presented in Chapter 3. As in the asynchronous version of π-calculus [16, 6], we do not have the output (invoke) prefix operator.

$$\begin{array}{lcl} s & ::= & \mathbf{kill}(k) \mid u.u' \,!\widetilde{u} \mid g \mid s \mid s \mid \{\!|s|\!\} \mid [\,d\,]s \mid S(d_1, \ldots, d_j) \\ g & ::= & \mathbf{0} \mid p.o\,?\widetilde{u}.\, s \mid g + g \end{array}$$

Table 2.1: Syntax of COWS

COWS presents two peculiar operators: kill activities **kill**(k), defined on kill labels k, are used to terminate (kill) services. The execution of kill activities is prioritized with respect to communications. The second peculiar operator, which is the dual of kill activities, is represented by the protection operator $\{\!|s|\!\}$, which defines a scope of inhibition for the effects of kill activities executed in parallel. These two operators, used in conjunction, can be used to model compensation procedures which have to take place in order to recover from erroneous configurations reached at runtime. This need stems from the fact that Web Services are loosely-coupled by definition, and so runtime errors have to be taken into account when modeling them. In Chapter 3 and, for the stochastic extension of COWS, in Chapter 5, we will see that the usefulness of these operators is not limited to compensation activities: we will take advantage of the

prioritized nature of kill activities over communication actions.

The delimitation operator $[\,d\,]s$ binds the identifier d inside s. This operator is the only binding operator in the language. Given $\widetilde{d} = d_1, \ldots, d_n$, we will often use the notation $[\,\widetilde{d}\,]s$ or, alternatively, $[\,d_1, \ldots, d_n\,]s$ to represent service $[\,d_1\,] \ldots [\,d_n\,]s$.

In contrast to the original definition of COWS [20], where replication was introduced, we make use of service definitions in order to model persistent and iterative behaviour. For this reason, for each service identifier $S(d_1, \ldots, d_j)$, we assume the presence of a definition $S(d'_1, \ldots, d'_j) = s$ in an environment of execution.

As we said before, we allow the presence of nondeterministic choice only when it appears guarded by an input (request, in COWS nomenclature). A request $p.o\,?\widetilde{u}.\ s$ can play a role in a communication over endpoint $p.o$, involving the identifiers in $\widetilde{u}$; $s\sigma$, where σ is a (possibly empty) sequence of substitutions induced by the communication operation, represents the residual of the request activity and specifies the subsequent behaviour of the service. Note that we require that each request activity specifies a static endpoint (being p and o ground names), whereas invoke activities can specify dynamic endpoints (being u and u' write-once variables). This fact means that, similarly to what happens in [21] for local π-calculus, it is not possible, for a COWS service, to perform a request activity over an endpoint containing an identifier received by a previously executed request activity.

We denote the set of COWS services respecting the syntax in Table 2.1 as $\mathcal{L}_{\mathrm{COWS}}$.

An identifier d is free in s if it appears outside the scope of a delimiter $[\,d\,]$. Function $fid : \mathcal{L}_{\mathrm{COWS}} \mapsto (\mathcal{N} \cup \mathcal{V} \cup \mathcal{K})$, which returns the set of all free identifiers in a COWS service, is defined in Table 2.2. When considering an agent identifier $S(d_1, \ldots, d_n)$, where $S(d'_1, \ldots, d'_n) = s$ is a given service definition, we require the instantiation of the body $s\{d_1, \ldots, d_n/d'_1, \ldots, d'_n\}$ to be closed mod-

ulo actual parameters $d_1, \ldots, d_n$, which compose the set of free identifiers of the service invocation itself.

$$
\begin{array}{lcl}
fid(\mathbf{0}) & = & \emptyset \\
fid(\mathbf{kill}(k)) & = & \{k\} \\
fid(s_1 \mid s_2) & = & fid(s_1) \cup fid(s_2) \\
fid(g_1 + g_2) & = & fid(g_1) \cup fid(g_2) \\
fid(\{\!|s|\!\}) & = & fid(s) \\
fid([\,d\,]s) & = & fid(s) \setminus \{d\} \\
fid(p.o\,?\widetilde{u}.s) & = & fid(s) \cup \{p, o, u_i \mid u_i \in \widetilde{u}\} \\
fid(u.u'\,!\,\widetilde{u}) & = & \{u, u', u_i \mid u_i \in \widetilde{u}\} \\
fid(S\,(d_1, \ldots, d_n)) & = & \{d_1, \ldots, d_n\}
\end{array}
$$

Table 2.2: Definition of function $fid(.)$

We define also auxiliary functions $fn : \mathcal{L}_{\mathrm{COWS}} \mapsto \mathcal{N}$, which returns the set of free names inside service s as $fn(s) = fid(s) \cap \mathcal{N}$, $fv : \mathcal{L}_{\mathrm{COWS}} \mapsto \mathcal{V}$, which returns the set of free variables inside service s as $fv(s) = fid(s) \cap \mathcal{V}$ and $fkl : \mathcal{L}_{\mathrm{COWS}} \mapsto \mathcal{K}$, which returns the set of free kill labels inside service s as $fkl(s) = fid(s) \cap \mathcal{K}$. Function $bid : \mathcal{L}_{\mathrm{COWS}} \mapsto (\mathcal{N} \cup \mathcal{V} \cup \mathcal{K})$ is the function which computes the set of bound identifiers defined inside a given service s. Its formal definition can be seen in Table 2.3. As before, we define also auxiliary functions $bn : \mathcal{L}_{\mathrm{COWS}} \mapsto \mathcal{N}$, which returns the set of bound names inside service s as $bn(s) = bid(s) \cap \mathcal{N}$, $bv : \mathcal{L}_{\mathrm{COWS}} \mapsto \mathcal{V}$, which returns the set of bound variables inside service s as $bv(s) = bid(s) \cap \mathcal{V}$ and $bkl : \mathcal{L}_{\mathrm{COWS}} \mapsto \mathcal{K}$, which returns the set of bound kill labels inside service s as $bkl(s) = bid(s) \cap \mathcal{K}$.

In this setting, we identify COWS services up to structural congruence, denoted as $\equiv$, which we define as the least relation satisfying the properties reported in Table 2.4. Nondeterministic choice and parallel composition are commutative, associative, and have $\mathbf{0}$ as identity element. The protection of $\mathbf{0}$ is equivalent to $\mathbf{0}$ itself, while the multiple protection of a service s is structurally congruent to the single protection of s. Renaming of bound identifiers

$$
\begin{array}{lcl}
bid(\mathbf{kill}(k)) = bid(\mathbf{0}) & = & \emptyset \\
bid(s_1 \mid s_2) & = & bid(s_1) \cup bid(s_2) \\
bid(g_1 + g_2) & = & bid(g_1) \cup bid(g_2) \\
bid(\{\!|s|\!\}) & = & bid(s) \\
bid([\,d\,]s) & = & \begin{cases} \{d\} \cup bid(s) & \text{if } d \in fid(s) \\ bid(s) & \text{otherwise} \end{cases} \\
bid(p.o\,?\widetilde{u}.s) & = & bid(s) \\
bid(u.u'\,!\,\widetilde{u}) & = & \emptyset \\
bid(S\,(d_1,\ldots,d_n)) & = & bid(s\{d_1,\ldots,d_n/d'_1,\ldots,d'_n\}) \\
 & & \text{where } S\,(d'_1,\ldots,d'_n) = s \text{ is a given service definition}
\end{array}
$$

Table 2.3: Definition of function $bid(.)$

(α-equivalence) preserves structural congruence; Finally, agent identifiers for which a service definition is contained in the execution environment are recognized structurally congruent to their instantiation.

$s_1 \equiv s_2$ if $s_1 =_\alpha s_2$

$[\,d\,]\mathbf{0} \equiv \mathbf{0}s$	$s_1 \mid [\,d\,]s_2 \equiv [\,d\,](s_1 \mid s_2)$	if $d \notin fid(s_1) \cup fkl(s_2)$
$[\,d_1\,][\,d_2\,]s \equiv [\,d_2\,][\,d_1\,]$	$\{\!|[\,d\,]s|\!\} \equiv [\,d\,]\{\!|s|\!\}$	
$s_1 \mid (s_2 \mid s_3) \equiv (s_1 \mid s_2) \mid s_3$	$s_1 \mid s_2 \equiv s_2 \mid s_1$	$s \mid \mathbf{0} \equiv s$
$(g_1 + g_2) + g_3 \equiv g_1 + (g_2 + g_3)$	$g_1 + g_2 \equiv g_2 + g_1$	$(p.o\,?\widetilde{u}.\,s, \gamma) + \mathbf{0} \equiv (p.o\,?\widetilde{u}.\,s, \gamma)$
$\{\!|\mathbf{0}|\!\} \equiv \mathbf{0}$	$\{\!|\{\!|s|\!\}|\!\} \equiv \{\!|s|\!\}$	
$S\,(d'_1,\ldots,d'_n) \equiv s\{d'_1/d_1,\ldots,d'_n/d_n\}$	if $S\,(d_1,\ldots,d_n) = s$	

Table 2.4: Structural Congruence rules for COWS

Before presenting the operational semantics of the language, we need to de-

fine further auxiliary functions. When dealing with the execution of kill actions, we take advantage of function $halt : \mathcal{L}_{\mathrm{COWS}} \mapsto \mathcal{L}_{\mathrm{COWS}}$, which definition can be seen in Table 2.5. This function is used to define how the execution of a kill activity affects a service s. Note that a protected service is not affected by the $halt(.)$ function, and that halting a nondeterministic choice $g_1 + g_2$ results in the $\mathbf{0}$ service, since g_1 and g_2 are input-guarded by definition.

$$
\begin{array}{lcl}
halt(\mathbf{0}) & = & \mathbf{0} \\
halt(u.u' \,!\, \widetilde{u}) & = & \mathbf{0} \\
halt(p.o \,?\, \widetilde{u}.s) & = & \mathbf{0} \\
halt(g_1 + g_2) & = & \mathbf{0} \\
halt(S\,(d_1, \ldots, d_n)) & = & \mathbf{0} \\
halt(s_1 \mid s_2) & = & halt(s_1) \mid halt(s_2) \\
halt(\{\!|s|\!\}) & = & \{\!|s|\!\}
\end{array}
$$

Table 2.5: Definition of function $halt(.)$

The operational semantics of COWS generates transitions of the form $\xrightarrow{\alpha}$; the syntax for labels α is presented in Table 2.6. Transitions of the form $\xrightarrow{\dagger}$ and $\xrightarrow{p.o \cdot \varepsilon \cdot \widetilde{n} \cdot \widetilde{u}}$ for some $p, o, \widetilde{n}, \widetilde{u}$, are considered as pseudo-τ transitions and indicate proper steps of execution for a COWS service s. We will write $s \hookrightarrow s'$ if $s \xrightarrow{\dagger} s'$ or $s \xrightarrow{p.o \cdot \varepsilon \cdot \widetilde{n} \cdot \widetilde{u}} s'$ for some $p, o, \widetilde{n}$.

Moreover, we will write $s \overset{\alpha_i}{\Rightarrow}{}^{(k)} s'$ to denote the sequence of k transitions such that $s \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_k} s'$.

$$\alpha \;::=\; \dagger k \mid \dagger \mid p.o \,?\, \widetilde{u} \mid p.o \,!\, \widetilde{n} \mid p.o \cdot \sigma \cdot \widetilde{n} \cdot \widetilde{u}$$

Table 2.6: Grammar for COWS labels

Function $d(\alpha)$ is used to determine identifiers inside label α. In particular, the definition of function $d(.)$ can be seen in Table 2.7. Note the last two cases: labels corresponding to completed kill activities and to communication actions with no pending substitutions contain no identifier.

$$
\begin{array}{rcl}
d(\dagger k) & = & \{k\} \\
d(p.o\,?\,\widetilde{u}) & = & \{p, o, u_i \mid u_i \in \widetilde{u}\} \\
d(p.o\,!\,\widetilde{n}) & = & \{p, o, n_i \mid n_i \in \widetilde{n}\} \\
d(p.o \cdot \{n/v\} \uplus \sigma \cdot \widetilde{n} \cdot \widetilde{u}) & = & \{n, v\} \cup d(p.o \cdot \sigma \cdot \widetilde{n} \cdot \widetilde{u}) \\
d(\dagger) & = & \emptyset \\
d(p.o \cdot \emptyset \cdot \widetilde{n} \cdot \widetilde{u}) & = & \emptyset
\end{array}
$$

Table 2.7: Definition of function $d(.)$

In order to check whether a service s can perform a kill action on identifier d, we use the predicate $s \downarrow_d$; this predicate is true if and only if some unguarded subterm of s has the shape $\mathbf{kill}(k)$. We also write $s \downarrow_{p.o}^{\widetilde{n} \cdot j}$ if s can perform a request activity on endpoint $p.o$ with some $\widetilde{u}$ as the tuple of parameters, where the matching between $\widetilde{n}$ and $\widetilde{u}$ generates strictly than j substitutions.

We now comment the rules composing the operational semantics of COWS.

Rule (*kill*) states that a kill activity $\mathbf{kill}(k)$, when executed, reduces to $\mathbf{0}$ with a transition labelled by $\dagger k$.

$$\frac{-}{\mathbf{kill}(k) \xrightarrow{\dagger k} \mathbf{0}}\ (kill)$$

Rule (*inv*) states that an invoke activity on endpoint $p.o$, with $\widetilde{n}$ as parameter, can be executed, reducing to $\mathbf{0}$ and generating the label $p.o\,!\,\widetilde{n}$. Note that $p, o, \widetilde{n} \in \mathcal{N}$, i.e. an invoke activity can be executed only if all parameters and components of the endpoint have been instantiated to ground names.

$$\frac{-}{p.o\,!\,\widetilde{n} \xrightarrow{p.o\,!\,\widetilde{n}} \mathbf{0}}\ (inv)$$

Rule (*req*) states that a request activity over endpoint $p.o$, with $\widetilde{u}$ as parameter and with s as residual service, can be executed reducing to s generating the label $p.o\,?\,\widetilde{u}$. Note that $\widetilde{u}$ can contain variables.

$$\frac{-}{p.o\,?\widetilde{u}.\,s \xrightarrow{p.o\,?\widetilde{u}} s}\ (req)$$

Rule (*choice*) states that a nondeterministic choice presents the same behaviour as its composing services g_1 and g_2, but once one of the two performs an action, the other component is removed from the residual service.

$$\frac{g_1 \xrightarrow{\alpha} s}{g_1 + g_2 \xrightarrow{\alpha} s}\ (choice)$$

Rule (*del_sub*) defines the behaviour of a service $[\,x\,]s$ when the label generated by the premiss of the rule is a communication involving a substitution of x with a name m. The delimiter of the variable is deleted and the only needed action is the substitution of x with m in the residual s'.

$$\frac{s \xrightarrow{p.o\cdot\sigma\uplus\{m/x\}\cdot\widetilde{u}\cdot\widetilde{n}} s'}{[\,x\,]s \xrightarrow{p.o\cdot\sigma\cdot\widetilde{u}\cdot\widetilde{n}} s'\{m/x\}}\ (del\_sub)$$

Rule (*del_k*) defines the behaviour of a service $[\,x\,]s$ when k is involved in the execution of a kill action; in particular, since the whole scope on which the kill label is defined has been reached, the effects of the execution of the kill action must not be propagated any further. This effect is accomplished by propagating the label † in place of $\dagger k$.

$$\frac{s \xrightarrow{\dagger k} s'}{[\,k\,]s \xrightarrow{\dagger} [\,k\,]s'}\ (del\_k)$$

Rule (*del_p*) can be applied when, given a delimiter $[\,d\,]$ and a label α, it happens that d is not contained in the identifiers in α and, if service s can perform

a kill activity, then α must be a completed kill action or an active kill action on a label $k \neq d$. This last condition makes the rule enforce the priority of kill actions over other types of actions. Rule (*prot*) states that the protection of a service s behaves exactly as s.

$$\frac{s \xrightarrow{\alpha} s' \quad d \notin \mathrm{d}(\alpha) \quad s \downarrow_d \Rightarrow (\alpha = \dagger \text{ or } \alpha = \dagger k)}{[\,d\,]s \xrightarrow{\alpha} [\,d\,]s'}\ (del\_p)$$

Using rule (*par_k*), the effects of an executed (and still active) kill action are propagated to services in parallel with the one originating the kill execution. Here we make use of the *halt*(.) function defined in Table 2.5.

$$\frac{s_1 \xrightarrow{\dagger k} s_1'}{s_1 \mid s_2 \xrightarrow{\dagger k} s_1' \mid \mathrm{halt}(s_2)}\ (par\_k)$$

As said before, function *halt*(.) has no effect on protected services $\{\!|s|\!\}$. As stated by rule (*prot*), if a protected service s performs a transition labelled by α reducing to s', then the protection of s can perform the same transition, labelled in the same way, and reduce to the protection of s'.

$$\frac{s \xrightarrow{\alpha} s'}{\{\!|s|\!\} \xrightarrow{\alpha} \{\!|s'|\!\}}\ (prot)$$

Rule (*par_p*) states that, considering a parallel composition $s_1 \mid s_2$, the execution of a request, invoke or concluded kill activity by one of the components represents, in an interleaved fashion, an execution step of the parallel composition.

$$\frac{s_1 \xrightarrow{\alpha} s_1' \quad \alpha \neq p.o \cdot \sigma \cdot \widetilde{u} \cdot \widetilde{n} \quad \alpha \neq \dagger k}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2}\ (par\_p)$$

Rule (*com*) is the one formalizing the communication paradigm of COWS.

$$\frac{s_1 \xrightarrow{p.o\,!\,\widetilde{n}} s_1' \quad s_2 \xrightarrow{p.o\,?\,\widetilde{u}} s_2' \quad \mathcal{M}(\widetilde{u},\widetilde{n}) = \sigma \quad \neg(s_1 \mid s_2)\downarrow_{p.o}^{\widetilde{n}\cdot|\sigma|}}{s_1 \mid s_2 \xrightarrow{p.o\cdot\sigma\cdot\widetilde{u}\cdot\widetilde{n}} s_1' \mid s_2'}\ (com)$$

Considering a service s_1 that reduces to s_1' by executing an invoke action $p.o\,!\,\widetilde{n}$ and a service s_2 reducing to s_2' by executing a request action $p.o\,?\,\widetilde{u}$, we first have to compute the list of substitution induced by matching tuple $\widetilde{n}$ with tuple $\widetilde{u}$. For this purpose, we use function $\mathcal{M}(.,.)$, which is defined in functional style in Table 2.8. This function is used to determine the list of needed substitutions when trying to match a tuple $\widetilde{n}$, composed of names and a tuple $\widetilde{u}$, composed of names and variables. The first case defined for function $\mathcal{M}(.,.)$ considers matching a name with itself, which induces no substitution. The following case considers the match of a name n with a variable u. Matching lists of identifiers is considered in the third case.

If we have $\mathcal{M}(\widetilde{u},\widetilde{n}) = \sigma$, then σ represents the list of substitutions induced by matching $\widetilde{n}$ with $\widetilde{u}$. Given $\mathcal{M}(\widetilde{u},\widetilde{n}) = \sigma$, we denote the cardinality of $\mathcal{M}(\widetilde{u},\widetilde{n})$ as the size of the substitution set σ: $|\mathcal{M}(\widetilde{u},\widetilde{n})| = |\sigma|$.

$$\begin{array}{rcl}
\mathcal{M}(\widetilde{u},\widetilde{n}) & = & \texttt{match}\ \widetilde{u}\,,\,\widetilde{n}\ \texttt{with} \\
n\,,\,n & = & \varepsilon \\
u\,,\,n & = & \{n/u\} \\
u\widetilde{u}_1\,,\,n\widetilde{n}_1 & = & \mathcal{M}(u,n) \uplus \mathcal{M}(\widetilde{u}_1,\widetilde{n}_1) \\
\textbf{default} & = & \bot \\
\texttt{end_match} & &
\end{array}$$

Table 2.8: COWS: definition of function $\mathcal{M}(.,.)$

Predicate $\neg(s_1 \mid s_2)\downarrow_{p.o}^{\widetilde{n}\cdot j}$ is used to enforce the *best matching* mechanism. Even if $\widetilde{n}$ and $\widetilde{u}$ match, this is not enough for the communication to take place: $\widetilde{u}$ must be the best matching tuple when considering $\widetilde{n}$, i.e. there must be no request activity on endpoint $p.o$ with $\widetilde{u_0}$ as parameter tuple for which $|\mathcal{M}(\widetilde{u_0}, \widetilde{n})| < j$. This condition is enforced also by rule (*par_c*) when considering a parallel composition where one of the components evolves executing a communication where the other service plays no role.

$$\frac{s_1 \xrightarrow{p.o\cdot\sigma\cdot\widetilde{u}\cdot\widetilde{n}} s_1' \quad \neg s_2\downarrow_{p.o}^{\widetilde{n}\cdot|\mathcal{M}(\widetilde{u},\widetilde{n})|}}{s_1 \mid s_2 \xrightarrow{p.o\cdot\sigma\cdot\widetilde{u}\cdot\widetilde{n}} s_1' \mid s_2}\ (par\_c)$$

Rule (*struct*) is applied when a rearrangement of the structure of a service s, according to the notion of structural congruence among COWS services, is needed in order to obtain the derivation tree for a transition.

$$\frac{s \equiv s_1 \quad s_1 \xrightarrow{\alpha} s_2 \quad s_2 \equiv s'}{s \xrightarrow{\alpha} s'}\ (struct)$$

2.2 COWS Derivation Example

We now present an example that will help in understanding the peculiarities of COWS. The example is aimed at showing how pattern matching works and how restriction is the only binding operator in the language: consider a COWS service $[\,p, o, m, x, y\,](r)$ where

- $r = s_1 \mid s_2 \mid s_3 \mid s_4 \mid s_5$
- $s_1 = [\,n\,]p.o\,!\,\langle n, m\rangle$
- $s_2 = p.o\,?\,\langle x, m\rangle.\, s_2'$

- $s_3 = r.m\,?\,x$
- $s_4 = p.o\,?\,\langle m, m\rangle$
- $s_5 = p.o\,?\,\langle x, y\rangle$

Invocation $p.o\,!\,\langle n, m\rangle$ in s_1 can synchronize only with request $p.o\,?\,\langle x, m\rangle$ in s_2, with the substitution of x with the private name n. The scope of this substitution contains s_3 and s_5. Request $p.o\,?\,\langle m, m\rangle$ in s_4 cannot synchronize with $p.o\,!\,\langle n, m\rangle$, since $\mathcal{M}(\langle n, m\rangle, \langle m, m\rangle) = \bot$, while the synchronization involving request s_5 and invocation s_1 is inhibited, since $\langle x, y\rangle$ is not the best matching tuple for $\langle n, m\rangle$: $|\mathcal{M}(\langle x, m\rangle, \langle n, m\rangle)| < |\mathcal{M}(\langle x, y\rangle, \langle n, m\rangle)|$.

We now comment on the derivation tree for the execution of the communication action between s_1 and s_2, which is broken down for clarity and spacing reasons.

In the first part of the derivation tree, axioms (*inv*) and (*req*) along with rule (*com*) are applied.

$$\dfrac{\dfrac{}{s_1 \xrightarrow{p.o\,!\,\langle n,m\rangle} \mathbf{0}}\,(inv) \qquad \dfrac{}{s_2 \xrightarrow{p.o\,?\,\langle x,m\rangle} s_2'}\,(req) \qquad \mathcal{M}(\langle x, m\rangle, \langle n, m\rangle) = \{n/x\} \quad \neg(s_1 \mid s_2)\downarrow_{p.o}^{\langle n,m\rangle \cdot 1}}{s_1 \mid s_2 \xrightarrow{\alpha = p.o \cdot \{n/x\} \cdot \langle x,m\rangle \cdot \langle n,m\rangle} \mathbf{0} \mid s_2'}\,(com)$$

The communication label $\alpha = p.o \cdot \{n/x\} \cdot \langle x, m\rangle \cdot \langle n, m\rangle$ is formed. As can be seen, the substitution $\{n/x\}$ induced by the communication action is delayed.

The second block composing the derivation tree presents the repeated application of rule (*par_c*).

$$\dfrac{\dfrac{\dfrac{s_1 \mid s_2 \xrightarrow{\alpha} \mathbf{0} \mid s_2' \quad \neg\left(\prod_{i\leq 3} s_i\right)\downarrow_{p.o}^{\langle n,m\rangle\cdot 1}}{\prod_{i\leq 3} s_i \xrightarrow{\alpha} \mathbf{0} \mid s_2' \mid s_3}(par\_c) \quad \neg\left(\prod_{i\leq 4} s_i\right)\downarrow_{p.o}^{\langle n,m\rangle\cdot 1}}{\prod_{i\leq 4} s_i \xrightarrow{\alpha} \mathbf{0} \mid s_2' \mid s_3 \mid s_4}(par\_c) \quad \neg r\downarrow_{p.o}^{\langle n,m\rangle\cdot 1}}{r \xrightarrow{\alpha} \mathbf{0} \mid s_2' \mid s_3 \mid s_4 \mid s_5}(par\_c)$$

The context composed of parallel services s_3, s_4 and s_5 is gathered and no conflicting action, according to predicate $.\downarrow_{p.o}^{\langle n,m\rangle\cdot 1}$ is found.

The third block is composed of the application of rule (*del_p*), applied when the delimiter for variable y has to be considered, and of rule (*del_sub*), which applies the substitution $\{n/x\}$ to its residual.

$$\dfrac{\dfrac{r \xrightarrow{\alpha} \mathbf{0} \mid s_2' \mid s_3 \mid s_4 \mid s_5 \quad y \notin d(\alpha) \quad [\,y\,]r \not\downarrow y}{[\,y\,]r \xrightarrow{\alpha} [\,y\,](\mathbf{0} \mid s_2' \mid s_3 \mid s_4 \mid s_5)}(del\_p)}{[\,x,y\,]r \xrightarrow{\beta=p.o\cdot\varepsilon\cdot\langle x,m\rangle\cdot\langle n,m\rangle} r' = [\,y\,](\mathbf{0} \mid s_2'\{n/x\} \mid r.m\,?\,n \mid s_4 \mid p.o\,?\,\langle n,y\rangle)}(del\_sub)$$

As a consequence of the application of rule (*del_sub*), the action label $\beta = p.o \cdot \varepsilon \cdot \langle x,m\rangle \cdot \langle n,m\rangle$ takes the place of α in the conclusion of the rule, since the information about the pending substitution is no more needed.

The fourth, and final, block composing the derivation tree presents the repeated application of rules (*del_p*), some of which are omitted and replaced by $\left(\vdots\right)$ when considering the delimiters for names p, o, m. Note that $n \notin d(\beta)$, since β is a silent action. In the derivation tree we used the definitions $[\,\widetilde{u}_1\,] = [\,n,x,y\,]$, $[\,\widetilde{u}_2\,] = [\,m,n,x,y\,]$, $[\,\widetilde{u}_3\,] = [\,p,o,m,n,y\,]$ and $[\,\widetilde{u}_4\,] = [\,p,o,m,n,x,y\,]$.

Finally, the application of rule (*struct*) is needed in order to rearrange the scope of delimiters (in this example no renaming of bound names is necessary) as presented in service s.

$$\dfrac{\dfrac{\dfrac{\dfrac{[\,x,y\,]r \xrightarrow{\beta} r' \quad n \notin d(\beta) \quad [\,\widetilde{u}_1\,]r \nmid n}{[\,\widetilde{u}_1\,]r \xrightarrow{\beta} [\,n,y\,]r'}(del\_p) \qquad m \notin d(\beta) \quad [\,\widetilde{u}_2\,] = r \nmid m}{\left(\vdots\right)}(del\_p)}{s \equiv [\,u_4\,]r \qquad [\,u_4\,]r \xrightarrow{\beta} [\,\widetilde{u}_3\,](\mathbf{0} \mid s_2'\{n/x\} \mid r.m\,?\,n \mid s_4 \mid p.o\,?\,\langle n,y\rangle)}(del\_p)}{s \xrightarrow{\beta} [\,\widetilde{u}_3\,](\mathbf{0} \mid s_2'\{n/x\} \mid r.m\,?\,n \mid s_4 \mid p.o\,?\,\langle n,y\rangle)}(struct)$$

Chapter 3

COWS Communications

In this chapter we will analyse the expressive power of the prioritized communication mechanism that stems from the use of correlation sets in the semantics of COWS. In particular, we provide a separation result between a fragment of CCS with global priority (FAP) and the kill and protection free fragment of COWS, named COWS_KF. Relaxing the properties used to prove the separation result, we are able to define two encoding functions. The first, represented by $[\![.]\!]$, encodes FAP processes in COWS while the second, represented by $\langle . \rangle$, encodes FAP processes in COWS_KF. Some of these results have been tackled in [8].

3.1 Preliminaries

FAP [32] is a finite fragment of asynchronous CCS enriched adding global priorities to the language. Given $\mathcal{N}_F$ a set of names, which is ranged over over by $x, x_1, x', \ldots$, the syntax of FAP is expressed as

$$P ::= \mathbf{0} \mid x.P \mid \overline{x} \mid \underline{\overline{x}} \mid P \mid P$$

FAP processes are ranged over by $P, P', Q, \ldots$. Note the presence of two versions of the output action. The first, $\overline{x}$, is the usual output action defined for

CCS, while $\underline{\overline{x}}$ is the prioritized output action. The operational semantics, defined in reduction style, makes use of two different relations, one for prioritized actions ($\twoheadrightarrow$) and one for standard ones ($\mapsto$). The whole set of rules can be seen in Table 3.1. The definition of structural congruence for FAP processes, used in the reduction semantics of the language, is given in Definition 1.

Definition 1. *Structural congruence $\equiv_F$ on FAP processes is defined as the least relation generated by the following rules*

$$P \mid \mathbf{0} \equiv_F P \qquad P \mid Q \equiv_F Q \mid P \qquad P \mid (Q \mid R) \equiv_F (P \mid Q) \mid R$$

The semantic rule denoted by $(*)$ is the one enforcing the priority of $\twoheadrightarrow$ derivations against $\mapsto$ derivations. We say that $P \rightarrow P'$ (P reduces to P') if and only if either $P \twoheadrightarrow P'$ or $P \mapsto P'$.

$$\frac{}{x.P \mid \overline{x} \mapsto P} \qquad \frac{}{x.P \mid \underline{\overline{x}} \twoheadrightarrow P}$$

$$\frac{P \twoheadrightarrow P'}{P \mid Q \twoheadrightarrow P' \mid Q} \qquad \frac{P \mapsto P' \quad P \mid Q \not\twoheadrightarrow R}{P \mid Q \mapsto P' \mid Q}\ (*)$$

$$\frac{P \equiv_F Q \quad P \mapsto P' \quad P' \equiv_F Q'}{Q \mapsto Q'} \qquad \frac{P \equiv_F Q \quad P \twoheadrightarrow P' \quad P' \equiv_F Q'}{Q \twoheadrightarrow Q'}$$

Table 3.1: Operational Semantics for FAP

The function $fn_F(.)$ returns the set of free names of a FAP process, is defined in Table 3.2.

$$\begin{array}{ll} fn_F(\mathbf{0}) = \{\} & fn_F(x.P) = \{x\} \cup fn_F(P) \\ fn_F(\overline{x}) = fn_F(\underline{\overline{x}}) = \{x\} & fn_F(P \mid Q) = fn_F(P) \cup fn_F(Q) \end{array}$$

Table 3.2: Definition of function $fn_F(.)$

Before presenting the details of the obtained separation result, we recall some needed definitions from [32], and introduce corresponding adapted definitions for COWS_KF.

Definition 2. *A* computation *C of a FAP process P is a sequence* $C : P \to P_1 \to \ldots \to P_n$.

A FAP process cannot have infinite computations, since the language has no iterative/recursive capability.

Definition 3. *Given* $C : P \to \ldots \to P_n$ *a FAP computation,* C' *extends* C *(written* $C \prec C'$*) iff there exists a computation* $C'' : P_n \to \ldots \to P_{(n+m)}$ *with* $m \geqslant 1$ *such that* $C' = CC''$.

Definition 4. *A computation of a FAP process* $C : P \to \ldots \to P_n$ *is* maximal *if it cannot be extended further, i.e. there exists no* C' *such that* $C \prec C'$.

Definition 5. *A computation C of a COWS_KF service s is a sequence (finite or infinite)* $s \xrightarrow{\alpha} s_1 \xrightarrow{\alpha_1} s_2 \to \ldots$.

Definition 6. *Given* $C : s_1 \to \ldots \to s_n$ *a COWS_KF computation,* C' *extends* C *(written* $C \prec C'$*) iff there exists a computation* $C'' : s_n \to \ldots \to s_{(n+m)}$ *with* $m \geqslant 1$ *or* $C'' : s_n \to \ldots$ *such that* $C' = CC''$.

Definition 7. . *A computation C of a COWS_KF service* $s \to \ldots \to s' \to \ldots$ *is* maximal *if it cannot be extended further, i.e. there exists no* C' *such that* $C \prec C'$.

Note that an infinite computation C cannot be extended, so C is a maximal computation according to Definition 7.

We now focus our attention on *observables*, also called observable actions. These special actions are denoted by $\omega, \omega_1, \ldots$. Observables are not in $\mathcal{N}_F$ nor in $\mathcal{N} \cup \mathcal{V} \cup \mathcal{K}$. They are explicitly added to the syntax of both FAP and COWS_KF, and are treated as asynchronous outputs. Note, however, that the operational semantics of both FAP and COWS_KF remain unchanged.

To denote the fact that a FAP process contains an unguarded observable action ω, we rely on the notion of observable barbs for FAP processes and COWS_KF services.

Definition 8. *A FAP process P exhibits the observable barb ω, written $P \downarrow_\omega$, if $P \equiv_F \omega \mid R$ for some R.*

Definition 9. *A COWS_KF service s exhibits the observable barb ω, written $s \downarrow_\omega$ if $s \equiv [\widetilde{u}](\omega \mid s_1)$ for some $\widetilde{u}$ and s_1.*

Our interest is directed towards the sets of observables that appear unguarded along maximal computations.

Definition 10. *Let $P_0, \ldots, P_i$ be FAP processes. Let C be a computation $P_0 \rightarrow \ldots \rightarrow P_i \ldots$. Given a set $Ob = \{\omega_i\}$ of intended observables, the observables of C are $Obs(C) = \{x \in Ob : \exists i.P_i \downarrow_x\}$.*

Definition 11. *Let $s_0, \ldots, s_i$ be COWS_KF services. Let C be a computation $s_0 \xrightarrow{\alpha} \ldots \xrightarrow{\alpha_i} s_i \ldots$. Given a set $Ob = \{\omega_i\}$ of intended observables, the observables of C are $Obs(C) = \{x \in Ob : \exists i.s_i \downarrow_x\}$.*

We now recall the definitions of independent FAP processes and introduce similar definitions for COWS_KF services.

Definition 12. *Two FAP processes P_1, P_2 are independent if they do not share free names, i.e. $fn(P_1) \cap fn(P_2) = \emptyset$.*

Definition 13. *Two COWS_KF services s_1 and s_2 are* variable independent *if $fv(s_1) \cap fv(s_2) = \emptyset$.*

Definition 14. *Two COWS_KF services s_1, s_2 are* independent *if they do not share free names or free variables i.e. $fn(s_1) \cap fn(s_2) = fv(s_1) \cap fv(s_2) = \emptyset$.*

Definition 15. *Function $res(.) : \mathcal{L}_{COWS\_KF} \mapsto \{p.o$ such that $p, o \in \mathcal{N}\}$ (re-*

quest endpoint set) is defined as

$$res(s) = \begin{cases} \{p.o\} \cup res(r) & \text{if } s = p.o\,?\widetilde{u}.r \\ res(s_1) \cup res(s_2) & \text{if } s = s_1 \mid s_2 \\ res(g_1) \cup res(g_2) & \text{if } s = g_1 + g_2 \\ res(s') \setminus \{p.o \text{ such that } p \neq u \wedge o \neq u\} & \text{if } s = [\,u\,]s' \\ res(s'\{u_1/u'_1, \ldots, u_n/u'_n\}) & \text{if } s = S(u_1, \ldots, u_n) \\ & \text{and } s' = S(u'_1, \ldots, u'_n) \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 16. *Two COWS_KF services s_1 and s_2 are* independent requesting *if* $res(s_1) \cap res(s_2) = \emptyset$

Note that if s_1, s_2 are two independent COWS_KF services, then they are both independent requesting and variable independent.

Given two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ we characterize an encoding function $\lfloor . \rfloor$ from $\mathcal{L}_1$ to $\mathcal{L}_2$ as

1. *Observation respecting* if, for all $P \in \mathcal{L}_1$:
   - for every maximal computation C of $P \in \mathcal{L}_1$ there exists a maximal computation C' of $\lfloor P \rfloor$ such that $Obs(C) = Obs(C')$.
   - for every maximal computation C of $\lfloor P \rfloor$ there exists a maximal computation C' of P such that $Obs(C) = Obs(C')$.
2. *Distribution preserving* if, $\forall P_1, P_2 \in \mathcal{L}_1$, we have $\lfloor P_1 \mid P_2 \rfloor = \lfloor P_1 \rfloor \mid \lfloor P_2 \rfloor$
3. *Independence preserving* if, given $P_1, P_2 \in \mathcal{L}_1$ such that P_1, P_2 are independent, then $\lfloor P_1 \rfloor$ and $\lfloor P_2 \rfloor$ are independent.
4. *Renaming preserving* if, for any permutation π of the identifiers in $\mathcal{L}_1$ there exists a permutation π' in $\mathcal{L}_2$ such that $\lfloor \pi(P) \rfloor = \pi'(\lfloor P \rfloor)$ and $\pi(\omega_i) = \pi'(\omega_i)$ for all i, i.e. π, π' do not affect the special identifiers ω_i reserved as observables.

An observation respecting encoding is required to preserve and not add observables expressed along maximal computations. A distribution preserving encoding has to maintain the parallel structure of encoded processes. An independent-preserving encoding must not create new communication capabilities at encoding time; the renaming preserving property, on the other hand, enforces an encoding to be independent of the actual identifiers used in encoded processes.

3.2 The Leader Election Problem

The Leader Election Problem (LEP) has been used in the expression and formalization of separation results [25, 32]. In an electoral system, it is required that all participants eventually reach an agreement on a winner, which becomes the new leader. In the model presented in [32], this fact is represented by the presence of a barb ω_j, if P_j is the winner participant. Once a barb ω_j is observable, no other barb $\omega_i, i \neq j$ can become observable. We recall the definition of the FAP process $NET = P_1 \mid \ldots \mid P_k$ from [32]: NET is defined as a system of k participants where each P_i is defined as

$$\begin{aligned} P_i = \overline{m_i} \mid \underline{s_i} \mid m_i.s_i.\left(\omega_i \mid \underline{\overline{d_{i1}}} \mid \ldots \underline{\overline{d_{iz_i}}}\right) \\ \mid d_{i1}.\left(s_i \mid \underline{\overline{d_{i1}}} \mid \ldots \underline{\overline{d_{iz_i}}}\right) \\ \vdots \\ \mid d_{iz_n}.\left(s_i \mid \underline{\overline{d_{i1}}} \mid \ldots \underline{\overline{d_{iz_i}}}\right) \end{aligned} \tag{3.1}$$

NET is an electoral system [32]: once a participant process P_i exhibits the observable ω_i, prioritized synchronizations on channels d_{nm} enable inputs over s_t and inhibit synchronizations over channels m_t; when these are enabled, the prioritized output over corresponding channels s_t have already been consumed, so residuals of processes $P_j, j \neq i$ enter a deadlock state.

In the model, z_i represents the number of neighbours for participant P_i, where two participants P_i and P_j are neighbours if they share a communication link,

represented in NET by channel d_{ij}. In general $z_i \leq k$, meaning that NET is not needed to be fully connected for the model to work; in fact it is sufficient that NET is connected, i.e. if some participants P_r, P_s are independent (sharing no free name), then there exists a list of participants $Nbs = [P_1, \ldots, P_k]$ such that P_r, P_1 are neighbours as well as P_k, P_s, and for each $1 \leq i < k$, we have that $P_i, P_{i+1} \in Nbs$ are neighbours.

Note that each P_i, if considered in isolation, is an electoral system on its own.

3.3 Separation Result

Using the Leader Election Problem example, we will show that FAP cannot be encoded into COWS_KF by an observation respecting, independence preserving, distribution preserving and renaming preserving encoding function. First, we want to underline a peculiarity of the communication priority mechanism of COWS_KF, which will be used when deriving the main result.

Lemma 1. *Let* $s = [\widetilde{u}_0]([n_1]s_1 \mid (g_1 + g_2))$ *such that* $s_1 \xrightarrow{p.o\,!\widetilde{n}} s_1'$, $g_1 + g_2 \xrightarrow{p.o\,?\widetilde{u}}$ s_2' *and* $s \xrightarrow{p.o\cdot\varepsilon\cdot\widetilde{n}\cdot\widetilde{u}} s_1' \mid s_2'\sigma$ *where* $\sigma = \{n_1/v_1\} \cup \sigma'$ *for some* $v_1 \in \widetilde{u}$*; then* $s_2'\sigma$ *cannot perform a request action on an endpoint using* n_1 *as either partner or operation name.*

Proof. By the definition of the syntax of COWS_KF, all endpoints of request actions are composed of names in $\mathcal{N}$, so no substitution induced by a communication action can alter their partner or operation components.

□

The implications of the inherent prioritization of communications given by the matching mechanism used in COWS_KF can be subtle. For example, a communication inhibition can happen only on a common endpoint. In other words, service s_1 can prevent a communication involving s_2 only if s_1 and s_2 share at

least two free names (forming a communication endpoint): communications on different endpoints are executed without taking into account the different number of substitutions induced by the matching function, which is the mechanism used to introduce the concept of priority in the semantics of the language.

Moreover, for Lemma 1, the creation of new communication links at execution time between previously independent services, given by the name-passing capabilities of the language, is not as powerful as the static definition of communication capabilities at design time: while the latter can involve a two-way use of endpoints (both for request and invoke activities), the former can create only one-way communication endpoints (only for invoke activities): if the scope of a name n is extruded to incorporate service s_2 as result of a substitution induced by the execution of a communication with s_1, then s_2 and its residuals cannot use n as part of the endpoint of a request action.

We now consider a FAP process NET_4 defined as in Definition 17.

Definition 17. *NET_4 is the process defined as*

$$P_0 \mid P_1 \mid P_2 \mid P_3$$

where each P_i is defined as in Equation (3.1) *and, for each pair i, j with $j = (i + 1) \bmod 4$, P_i and P_j share a unique name $d_{ij} = d_{ji}$. For clarity, process P_i is associated to observable ω_i.*

Given the structure of processes P_i and the fact that NET_4 is connected, then it is a FAP electoral system [32]. We will show that, when considering an observation respecting, distribution preserving, independence preserving and renaming preserving encoding $\lfloor . \rfloor$ from FAP to COWS_KF, $\lfloor NET_4 \rfloor$ is not an electoral system.

First of all, to understand the dynamics of the electoral system NET_4 and of its encoded version $\lfloor NET_4 \rfloor$, it can be useful to consider a graphical visualization of their interaction capabilities. Using as a basis the concept of hypergraph

associated to a network [26, 25], we can represent NET_4 and $\lfloor NET_4 \rfloor$ as connected graphs, presented in Figure 3.1. This figure presents also the structure of $[\widetilde{u}] \prod_{i\leq 3} s_i$, a generic evolution of $\lfloor NET_4 \rfloor$ after k steps such that

$$\lfloor NET_4 \rfloor \stackrel{\alpha_i\ ((k))}{\Rightarrow} \equiv [\widetilde{u}] \prod_{i\leq 3} s_i$$

Each solid edge in Figure 3.1 represents an interaction capability among subservices on shared endpoints introduced at encoding time, i.e. two nodes connected by a solid edge can use the shared endpoints for performing both invoke and request actions. We denote such endpoints as *bidirectional*. Dotted edges represent runtime-created interaction capabilities that, for Lemma 1, cannot involve bidirectional endpoints. This difference in the nature of communication endpoints has a pivotal role in the separation result that is presented in Theorem 2.

The structure of $\lfloor NET_4 \rfloor$ can be derived reasoning on the properties of function $\lfloor . \rfloor$ starting from the definition of NET_4: $\lfloor . \rfloor$ is distribution preserving, so we can identify a service $\lfloor P_i \rfloor$ for each P_i composing NET_4; $\lfloor . \rfloor$ is independence preserving, so no new interaction capability is introduced in $\lfloor NET_4 \rfloor$ with respect to the ones already present in NET_4.

Intuitively, the separation result between FAP and COWS_KF is based on the fact that the symmetry in the COWS_KF service $\lfloor NET_4 \rfloor$, for some computation, cannot be broken, as shown in Theorem 1.

When considering $NET_4 = P_0 \mid P_1 \mid P_2 \mid P_3$ as defined in Definition 17, the concept of symmetry can be expressed identifying a particular substitution σ_1 such that

$$P_i\sigma_1 \equiv_F P_{(i+2) \bmod 4} \text{ for } i \in \{0, 1, 2, 3\}$$

Given the definition of NET_4, in which processes P_i are equal modulo renaming, such a substitution σ_1 exists. Since we are considering a distribution preserving and renaming preserving encoding function $\lfloor . \rfloor$ from FAP to

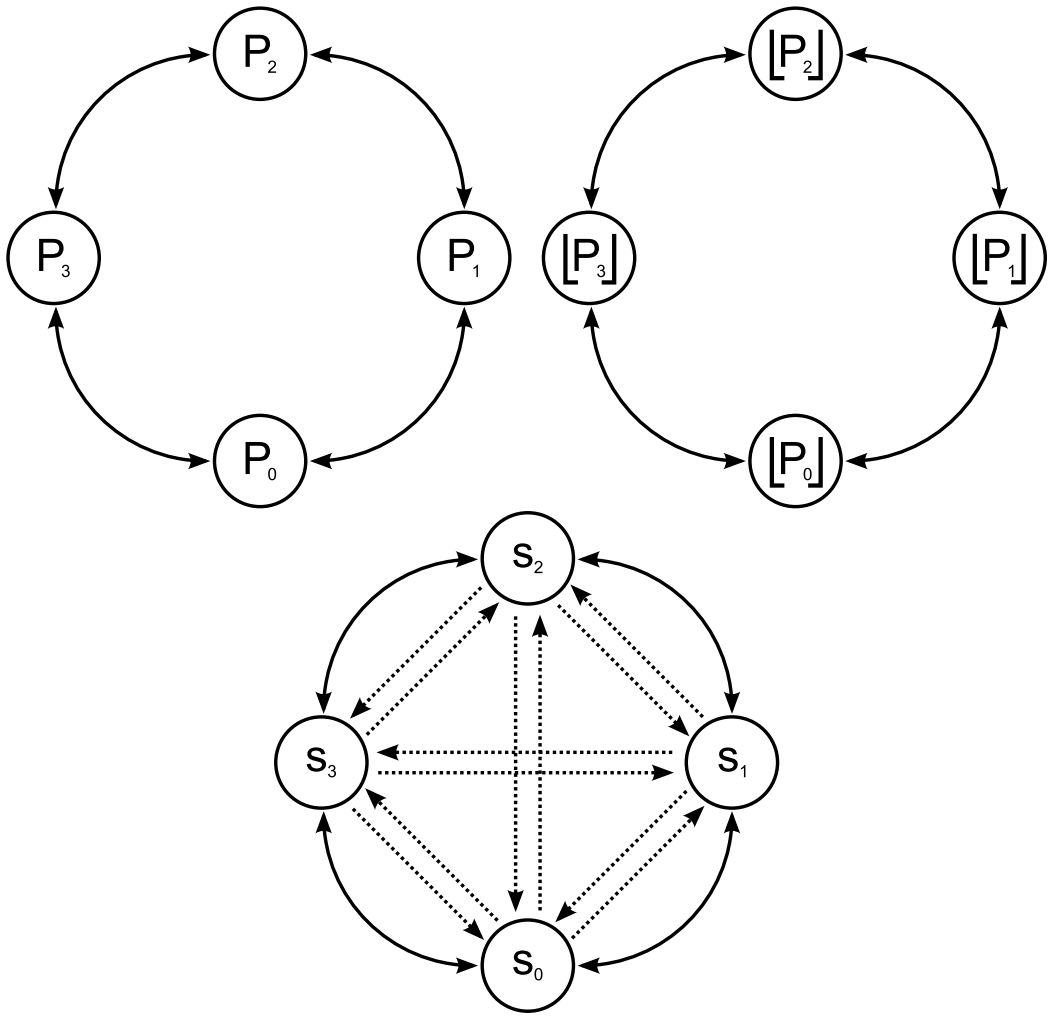


Figure 3.1: Graph representations of NET_4 (top left), $\lfloor NET_4 \rfloor$ (top right) and a residual of $\lfloor NET_4 \rfloor$ with runtime-created communication capabilities (bottom): nodes represent processes/services, solid arcs represent communication capabilities defined at design/encoding time, dotted arcs represent communication capabilities obtained at runtime.

COWS_KF, a similar property holds also for $\lfloor NET_4 \rfloor = \lfloor P_0 \rfloor \mid \lfloor P_1 \rfloor \mid \lfloor P_2 \rfloor \mid \lfloor P_3 \rfloor$: there exists a substitution σ_2 such that

$$\lfloor P_i \rfloor \sigma_2 \equiv \lfloor P_{(i+2) \bmod 4} \rfloor \text{ for } i \in \{0, 1, 2, 3\}$$

We will use the notation $\langle\langle r \rangle\rangle$ to denote any service $s \equiv [\widetilde{u}]r$, to focus on the relevant structure of services. We will also use function $\varphi(.)$, defined as

$$\varphi(n) = (n + 2) \bmod 4$$

as a shorthand to denote the indexes of symmetric COWS_KF services in $\lfloor NET_4 \rfloor$.

Theorem 1. *Let $\lfloor . \rfloor$ be a observation respecting, distribution preserving, independence preserving and renaming preserving encoding of FAP into COWS_KF. Then $\lfloor NET_4 \rfloor \overset{\alpha_h}{\Rightarrow}{}^{(2k)} s_{sym} \equiv \langle\langle r_0 \mid r_1 \mid r_2 \mid r_3 \rangle\rangle$ where $r_i\theta_1 \equiv r_{\varphi(i)}$ and such that $r_i, r_{\varphi(i)}$ are independent requesting and r_i, r_j, with $i, j \in \{0, 1, 2, 3\}$ and $i \neq j$, are variable-independent.*

Proof. By induction on the length of the transition sequence $\overset{\alpha_h}{\Rightarrow}{}^{(2k)}$. Without loss of generality, we will assume that $s_i = s_0$ is the service performing the invoke action involved in the transition $s \equiv \langle\langle s_0 \mid s_1 \mid s_2 \mid s_3 \rangle\rangle \overset{\alpha}{\rightarrow} s'$ with $\alpha = p.o \cdot \varepsilon \cdot \widetilde{n} \cdot \widetilde{u}$, so we have that $s_0 \xrightarrow{p.o\,!\,\widetilde{n}} s_0'$.

- $k = 1$

  We have that $\lfloor NET_4 \rfloor \equiv s \overset{\alpha}{\rightarrow} s'$ with $\alpha = p.o \cdot \varepsilon \cdot \widetilde{n} \cdot \widetilde{u}$ for some $p, o, \widetilde{n}, \widetilde{u}$. Since $s \equiv [\widetilde{u}](\prod s_i)$, and since s is closed, we can consider the derivation tree for transition $\overset{\alpha}{\rightarrow}$ as the one concluded with a unique application of rule (*struct*) and with applications of rules (*del_sub*) and (*del_p*) involving all names and variables in $\widetilde{u}$ at the bottom, so that we can focus on the relevant subservice $s_0 \mid s_1 \mid s_2 \mid s_3$ and its behaviour.

  In the base of the induction we have to distinguish two cases, based on the relationship between the two services s_0 and s_j involved in the synchronization.

1. (s_0, internal synchronization). We have $s_0 \xrightarrow{p.o \cdot \sigma_0 \cdot \widetilde{n} \cdot \widetilde{u}} s_0'$. By hypothesis services s_1, s_2, s_3 cannot inhibit this transition by offering a best matching request action on endpoint $p.o$. The list of substitutions σ_0 induced by the communication cannot affect s_1 and s_3, since services s_i are variable independent. By the symmetry existing between s_0 and s_2, service s_2 can perform its own internal synchronization

$$s_2 \xrightarrow{p'.o' \cdot \sigma_2 \cdot \widetilde{n'} \cdot \widetilde{u'}} s_2'$$

with $|\mathcal{M}(\widetilde{n}, \widetilde{u})| = |\mathcal{M}(\widetilde{n'}, \widetilde{u'})|$. By symmetry this transition cannot be inhibited by s_1, s_3. The same holds for $s_0'\sigma_0$, since s_0 and s_2 are independent services, so they are both variable independent and request independent. We get that

$$s \xrightarrow{p.o \cdot \varepsilon \cdot \widetilde{n} \cdot \widetilde{u}} s' = \left\langle\left\langle s_0'\sigma_0 \mid s_1 \mid s_2 \mid s_3 \right\rangle\right\rangle$$

and that

$$\begin{aligned} s' \xrightarrow{p'.o' \cdot \varepsilon \cdot \widetilde{n'} \cdot \widetilde{u'}} s'' &= \left\langle\left\langle s_0'\sigma_0 \mid s_1 \mid s_2'\sigma_2 \mid s_3 \right\rangle\right\rangle \\ &= \langle\langle (r_0 \mid r_1 \mid r_2 \mid r_3) \rangle\rangle \end{aligned}$$

Starting from θ_1 and taking into consideration new substitutions in σ_0 and σ_2 to define θ_2, we can state that

$$r_i\theta_2 \equiv r_{\varphi(i)}$$

Service s'' is still symmetric and is such that pairs (r_0, r_2) and (r_1, r_3) are independent requesting and variable-independent.

2. ($s_0, s_j, j \in \{1, 3\}$, i.e. neighbour services and $p.o$ bidirectional between s_0, s_j). The request action is performed by either s_1 or s_3; let consider s_1 (the other case is specular) and let σ_1 be the list of substitutions induced by the executed communication, whose scope is limited to s_1', since services s_i are all variable independent. We have that

$s_1 \xrightarrow{p.o\,?\widetilde{u}_1} s'_1$. By the symmetry hypothesis, we have that $s_2 \xrightarrow{p'.o'\,!\widetilde{n}'} s'_2$, with s'_0, s'_2 symmetric and $s_3 \xrightarrow{p'.o'\,?\widetilde{u}'} s'_3$ with s'_1, s'_3 symmetric. Note that s_1, s_3 are independent services, and by Lemma 1 also $s'_1\sigma_1$, s'_3 are independent requesting and, moreover, they are also variable independent. For these reasons, no request in s'_0 or $s'_1\sigma_1$ can inhibit the synchronization between s_2 and s_3. Moreover, a match between $\widetilde{u}'$ and $\widetilde{n}'$ such that $|\mathcal{M}(\widetilde{n}, \widetilde{u})| = |\mathcal{M}(\widetilde{n}', \widetilde{u}')|$ has to be defined for the symmetry condition. We finally get that

$$
\begin{aligned}
s \xrightarrow{p.o\cdot\varepsilon\cdot\widetilde{n}\cdot\widetilde{u}} s' \xrightarrow{p.o\cdot\varepsilon\cdot\widetilde{n}'\cdot\widetilde{u}'} \equiv s'' &= \left\langle\left\langle (s'_0 \mid s'_1\sigma_1 \mid s'_2 \mid s'_3\sigma_3)\right\rangle\right\rangle \\
&= \langle\langle (r_0 \mid r_1 \mid r_2 \mid r_3)\rangle\rangle
\end{aligned}
$$

where σ_3 is the list of substitutions induced by the second communication action, whose scope is limited to s'_3. Starting from θ_1 and taking into consideration new substitutions in σ_1 and σ_2 to define θ_2, we can state that

$$r_i\theta_2 \equiv r_{\varphi(i)}$$

We obtain that s'' is symmetric and is such that pairs (r_0, r_2) and (r_1, r_3) are independent requesting and variable-independent.

Note that, after the application of the substitution lists σ_1 and σ_3 to respectively s'_1 and s'_3, new communication capabilities could have been formed between these residual services. As shown in Lemma 1 these substitutions cannot involve any component of a request endpoint and, therefore, cannot involve bidirectional endpoints. These communication capabilities created at runtime are represented by dotted arcs in Figure 3.2.

- $k > 1$

As before, we can consider the derivation tree for transition $s \xrightarrow{\alpha} s'$, with $\alpha = p.o \cdot \varepsilon \cdot \widetilde{n} \cdot \widetilde{u}$, as the one concluded with an unique application of rule (*struct*) and with applications of rules (*del_sub*) and (*del_p*) involving all names and variables in $\widetilde{u}$, at the bottom.

In the proof for the step of the induction on the length of $\stackrel{\alpha}{\Rightarrow}^{(2k)}$ we have to consider two additional cases with respect to the base of the induction. These additional cases are given by the possibility of a communication happening on runtime-created channels, represented in Figure 3.2 with dotted arcs, between s_0 and its symmetric service s_2 or with one of its neighbours s_1, s_3.

1. (s_0, internal synchronization). The proof for this case is identical to the corresponding case presented for the base of the induction.
2. ($s_0, s_j, j \in \{1, 3\}$ and $p.o$ bidirectional between s_0, s_j). The proof for this case is identical to the corresponding case presented for the base of the induction.
3. ($s_0, s_j, j \in \{1, 3\}$ and $p.o$ not bidirectional between s_0, s_j). Note that the proof for the previous point, which refers to the corresponding case of the base of the induction, does not depend on the fact that $p.o$ is bidirectional, and so holds also for this case.
4. (s_0, s_2, i.e. symmetric services). In this case, we consider a communication happening on an endpoint whose components have been shared at runtime between independent requesting services. By assumption we have $s_0 \xrightarrow{p.o\,!\widetilde{n}} s_0'$ and $s_2 \xrightarrow{p.o\,?\widetilde{u}_2} s_2'$. Given that s_0 and s_2 are symmetric, we have

$$s_0 \equiv p.o\,!\widetilde{n} \mid p'.o'\,?\widetilde{u}_0.r_0 + g_0$$

$$s_2 \equiv p'.o'\,!\widetilde{n}' \mid p.o\,?\widetilde{u}_2.t_2 + g_2$$

We can then write that $s'_0 = p'.o' \,?\widetilde{u}_0.t_0 + g_0$ and $t_2 = p'.o' \,!\widetilde{n}' \mid s'_2$. We get that

$$s \xrightarrow{p.o \cdot \varepsilon \cdot \widetilde{n} \cdot \widetilde{u}} \equiv s' = \langle\langle (p'.o' \,?\widetilde{u}_0.t_0 + g_0 \mid s_1 \mid p'.o' \,!\widetilde{n}' \mid t_2 \mid s_3)\sigma_2 \rangle\rangle$$

for some substitution list σ_2 induced by the communication. Since s_0, s_2 are independent requesting and variable independent, so are services s'_0 and s'_2, even when considering substitution σ_2. Since function $\lfloor . \rfloor$ is distribution preserving, services s_i do not share variables, so substitution σ_2 does not affect s'_0, s_1, s_3.

Services s_1 and s_3 are symmetric and since they did not interfere with the communication on endpoint $p.o$, they do not offer a request on endpoint $p'.o'$ with $\mathcal{M}(\widetilde{n}', \widetilde{u}_0) = (\_, \_, j', \_, \_)$ with $j' < j$, even when considering substitution σ_2 so we can conclude that

$$\begin{aligned} s' \xrightarrow{p'.o' \cdot \varepsilon \cdot \widetilde{n}' \cdot \widetilde{u}_0} \equiv s'' &= \langle\langle (t_0\sigma_0 \mid s_1 \mid t_2\sigma_2 \mid s_3) \rangle\rangle \\ &= \langle\langle (r_0 \mid r_1 \mid r_2 \mid r_3) \rangle\rangle \end{aligned}$$

where $|\mathcal{M}(\widetilde{n}, \widetilde{u})| = |\mathcal{M}(\widetilde{n}', \widetilde{u}_0)|$. Starting from θ_1 and taking into consideration new substitutions in σ_0 and σ_2 to define θ_2, we can state that

$$r_i\theta_2 \equiv r_{\varphi(i)}$$

Service s'' is still symmetric and is such that pairs (r_0, r_2) and (r_1, r_3) are independent requesting and variable-independent.

□

Given the result presented in Theorem 1, $\lfloor NET_4 \rfloor$ has at least one computation that does not break the initial symmetry. Theorem 2 uses this fact to obtain the separation result between FAP and COWS_KF.

Theorem 2. *There is no observation respecting, distribution preserving, independence preserving and renaming preserving encoding of the FAP process NET_4 in COWS_KF.*

Proof. Let assume $\lfloor . \rfloor$ is a observation respecting, distribution preserving, independence preserving and renaming preserving encoding of FAP into COWS_KF. Applying Theorem 1 to $\lfloor NET_4 \rfloor$, we obtain that, since there exists at least one maximal computation C of $\lfloor NET_4 \rfloor$ along which the symmetry cannot be broken, one of the following holds

1. $\{\omega_i \text{ such that } i = 0, 1, 2, 3\} \cap Obs(C) = \emptyset$
2. $\{\omega_i \text{ such that } i = 0, 1, 2, 3\} \cap Obs(C) = \left\{\omega_i, \omega_j \text{ such that } i \neq j\right\}$
3. $\{\omega_i \text{ such that } i = 0, 1, 2, 3\} = Obs(C)$

In either case, $\lfloor . \rfloor$ is not observation respecting and $\lfloor NET_4 \rfloor$ is not an electoral system, which leads to a contradiction. □

Theorem 2 underlines the difference between the global priority mechanism of FAP and the communication mechanism based on correlation sets and best matching peculiar to COWS. The distribution preserving and independence preserving properties, in particular, are at the basis of the proof of the non-existence of the encoding of FAP into COWS_KF: enforcing these properties means limiting the interaction and interference capabilities of COWS_KF services to such an extent that a global prioritization of actions cannot take place.

3.4 Global Priorities in COWS

Even if a separation result between FAP and COWS_KF exists, this does not mean that COWS and, more interestingly, COWS_KF cannot encode FAP. Of course we will need to lift some of the requirements on the encoding function. In particular, first we will show that FAP can be encoded in COWS using an

observation respecting encoding function, whose definition will take advantage of the full syntax/semantics of COWS introducing also *kill* activities. Subsequently, we will present an observation respecting encoding function from FAP to COWS_KF.

To better understand the rationale of the encoding functions, it can be useful to start abstracting from the concurrent paradigm and focus on the coarse-grained steps that the protocols introduced by the encoding functions perform. For this reason, we start the presentation of the encoding functions by commenting on a high-level imperative view of the protocol, represented in Figure 3.2 and summarized in Pseudocode 3.1. Given a FAP process $P \equiv_F \prod_{i\leqslant n} x_i.P_{i'} \mid \prod_{j\leqslant m} \underline{\overline{x_j}} \mid \prod_{k\leqslant o} \overline{x_k}$, encoded unguarded input actions $x_i.P_{i'}$ are taken into account one at a time (line 2), and a corresponding encoded high-priority output action $\underline{\overline{x_i}}$ is searched (line 3). If one is found, then the synchronization takes place, the encoding of the residual $P_{i'}$ is instantiated and a subsequent cleaning phase is executed before restarting the loop (lines 4-7). If no high-priority match could be found for any encoded input, then the protocol proceeds performing a similar search for a low-priority match (lines 10-17). If no match is found, then lines 12-15 are never executed, and the algorithm terminates reaching a deadlock state (lines 18 and 20), meaning that $P \not\twoheadrightarrow$. In the presented encoding, if statements are rendered by nondeterministic choices composed of competing request activities presenting different numbers of variables as parameters, thus having different priorities, given the communication paradigm of COWS. Cleanup phases are modelled using combinations of kill activities and protections, while for operators and *goto* operators are modelled using agent identifiers.

3.4.1 Encoding Function $[\![.]\!]$

We now present the formalization of $[\![.]\!] : \mathcal{L}_F \to \mathcal{L}_{\text{COWS}}$. This function is not distribution preserving nor independence preserving. We make use of a sup-

Pseudocode 3.1 Algorithmic representation of the protocol introduced by the encoding functions

Input: encoded process $P \equiv_F \prod_i x_i.P_{i'} \mid \prod_j \underline{\overline{x_j}} \mid \prod_k \overline{x_k}$

```
1:  loop
2:    for all input x_i.P_i' do
3:      if ∃ x̲_i then
4:        consume x_i, x̲_i
5:        instantiate P_i';
6:        cleanup;
7:        goto 1;
8:      end if
9:    end for
10:   for all input x_i.P_i' do
11:     if ∃ x̄_i then
12:       consume x_i, x̄_i
13:       instantiate P_i';
14:       cleanup;
15:       goto 1;
16:     end if
17:   end for
18:   goto 20;
19: end loop
20: deadlock;
```

porting function $(\!|.|\!)^{h\,l}_{r} \mathcal{L}_F \rightarrow \mathcal{L}_{\mathrm{COWS}}$. Table 3.3 presents the definition of the encoding function $[\![.]\!]$. In the encoding, h, l, r are bound names introduced by the encoding, while $\widetilde{m}$ contains all the names in the process P. Observables are not in $\widetilde{m}$. COWS variables introduced by the encoding are denoted by v_i. Other bound identifiers introduced by the encoding are to be considered COWS names. Pseudocode 3.2, Pseudocode 3.3 and Pseudocode 3.4 present the definitions of the COWS agents used in the encoding procedure.

The search protocol introduced by function $[\![.]\!]$ is driven by the encoding of input actions which, once activated, try to synchronize with a matching output function. The activation part of the protocol is performed using a token,

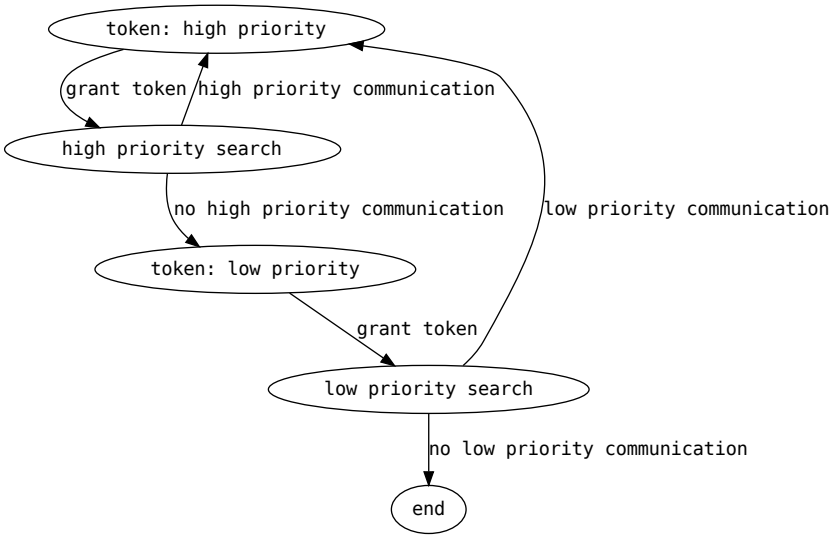


Figure 3.2: Representation of the protocol introduced by the encoding functions

which is initialized by a *Monitor* agent and is passed along *Input* agents. When considering the evolution of the search protocol, various scenarios can arise, depending on the presence of a matching *Output* agent and depending on its priority: as expected, higher priority output actions are probed before low priority ones. After each successful communication, blocked and unexecutable actions are removed and the protocol is reset. If no successful communication could be performed, the protocol is halted: no further repetition of the search will find an executable communication so, to avoid livelocks and to reflect the deadlocked state of the encoded FAP process, agent *Monitor* is removed from the system along with the token.

The use of a single token allows only one *Input* agent to be active at any given time. Each activated *Input* agent performs a search for a matching high priority output. If the search is successful, then the synchronization is performed. If this search ends without finding a suitable high priority output, then another input action gets the token and is allowed to perform a similar search.

$$
\begin{array}{lcl}
[\![P]\!] & = & [\,h, l, r, \widetilde{m}\,]\left(Mon2(h,l,r) \mid (\!|P|\!)_r^{h\,l}\right) \\
(\!|P \mid Q|\!)_r^{h\,l} & = & (\!|P|\!)_r^{h\,l} \mid (\!|Q|\!)_r^{h\,l} \\
(\!|x.Q|\!)_r^{h\,l} & = & \{\!|Input_{x.Q}(x,h,l,r)|\!\} \\
(\!|\overline{x}|\!)_r^{h\,l} & = & \{\!|Output(x,l)|\!\} \\
(\!|\underline{x}|\!)_r^{h\,l} & = & \{\!|Output(x,h)|\!\} \\
(\!|\mathbf{0}|\!)_r^{h\,l} & = & \mathbf{0}
\end{array}
$$

Table 3.3: Definition of the function $[\![.]\!]$, encoding FAP into COWS

If no input action finds a suitable high priority output, all input actions, once granted the token, perform a search for a suitable low priority output. As before, if a suitable encoded output is found, then the synchronization is performed and the encoded residual of the input action is instantiated. Encoded input actions perform their searches in a serialized way, so that there is at most one input action active at any given time.

After a successful communication, the residuals of the *Input* agents involved in the search, but not in the communication, trigger a *kill* action each; these actions clean up their states, removing any stale part and leaving a protected instance of the input actions.

We now give a finer description of agents involved in the encoding. Agent $Monitor(h, l, r)$, whose definition is presented in Table 3.2, has the primary role of instantiating the token, performing the first invocation on channel *search.h*. An encoded input action can perform the corresponding request activity (with the same parameter, thus involving no substitution) and, then, search for an encoded high priority output process or regenerate the invocation over *search.h*, which activates another encoded input. After all *Input* agents have performed the search for a high priority output, the last activated *Input* action has a pending invoke action on endpoint *search.h*. This is removed by agent *Monitor*, which performs a request activity on the same endpoint involving two substitutions.

If the search performed by an active *Input* agent is concluded positively, that

agent (Table 3.3) synchronizes over $comm.finish$ with the found encoding of a high priority output, with h (for *high*) as parameter. $Monitor$ then forces a reset of the state of all invocations not involved in the executed communication, as well as its state. On the other hand, if the search is concluded without finding a suitable encoded high priority output, then the last invocation over *search.h* can only synchronize with the request defined in agent $Monitor$; this communication involves one substitution. The residual of agent $Monitor$ generates then a signal triggering the search for a low priority output action, with an invocation over *search.l*. At this point, if the search is concluded positively, then the corresponding $Input$ agent performs an invocation on $comm.finish$ with l (for *low*) as parameter; otherwise, the last $Input$ agent involved synchronizes with $Monitor$ over *search.l* with a communication involving one substitution. In both cases, a reset procedure for the $Input$ agents involved in the search is initiated. In the first case, a fresh instance of agent $Monitor$ is ready to restart the search procedure. In the second case agent $Monitor$ is not instantiated again, since a new search would lead to the same (negative) result: this means that the residual of the encoded process enters a deadlock, since no service has an unguarded invocation on endpoint *search.h*.

The signalling of the reset procedure is performed through channel *hard.r*. As before, agent $Monitor$ performs the first invoke activity while each $Input$ agent, which has been considered while searching for a feasible synchronization, first performs the corresponding request activity and subsequently is ready to perform another invoke activity on channel *hard.r*. The last $Input$ agent can only synchronize over *hard.r* with agent $Monitor$, which can perform a request activity on endpoint *hard.r* involving a substitution.

After the synchronization over endpoint *search.h*, an $Input$ agent, encoding an input on x, searches for a corresponding encoded high priority output activity, trying to synchronize over endpoint *x.h*. If such an agent exists, then a communication involving no substitution can happen and, subsequently, a syn-

Pseudocode 3.2 Definition of the COWS agent $Monitor(x, l, r)$, introduced by function $[\![.]\!]$

```
 1: Monitor(h, l, r) =
 2:     (search.h ! ⟨h⟩
 3:         | ([ v1 ]search.h ? ⟨v1⟩ .(
 4:               search.l ! ⟨l⟩
 5:               | [ v2 ]search.l ? ⟨v2⟩
 6:                   .CBlock(r, kcl)
 7:               +comm.finish ? ⟨l⟩
 8:                   .CReset(h, l, r, kcl)
 9:               )
10:           +comm.finish ? ⟨h⟩
11:               .CReset(h, l, r, kcl)
12:           )
13:     );
14: CReset(h, l, r, kcl) =
15:     hard.r ! ⟨r⟩
16:     | [ v3 ]hard.r ? ⟨v3⟩ .(
17:         Monitor(h, l, r)
18:     );
19: CBlock(h, l, r, kcl) =
20:     hard.r ! ⟨r⟩
21:     | [ v4 ]hard.r ? ⟨v4⟩ .0;
```

chronization over $x.com$ is the only possible outcome. Conversely, the *Input* agent performs an internal communication on endpoint $x.h$ involving one substitution (thus having lower priority); the residual of the *Input* agent gives up its turn, performing an invocation activity over $search.h$. The remainder of the behaviour consists of a request activity over $search.l$; after this request activity is executed, it means the *Input* agent can search for a low priority output over x. The behaviour is very similar to the high priority search.

In parallel to this subprocess, each *Input* agent presents a subprocess (a request activity over $hard.r$) involved in resetting the state of the encoded input to its initial state, as well as a subprocess (a request activity over $soft.r$) which

Pseudocode 3.3 Definition of $(\!|\, . \,|\!)_r^{h\,l}$ for a FAP input process

```
1:  Input_{x.Q}(x, h, l, r) =
2:      [ kin ][ myid ](search.h ? ⟨h⟩ .(
3:          x.h ! ⟨x, h⟩
4:          |(x.com ? ⟨x, h⟩
5:              .(soft.r ! ⟨myid, h⟩)
6:          +[ v6 ]x.h ? ⟨x, v6⟩ .(
7:              search.h ! ⟨h⟩
8:              |search.l ? ⟨l⟩ .(
9:                  x.l ! ⟨x, l⟩
10:                 |(x.com ? ⟨x, l⟩
11:                     .(soft.r ! ⟨myid, l⟩)
12:                 +[ v7 ]x.l ? ⟨x, v7⟩ .(search.l ! ⟨l⟩)
13:                     ))))
14:         |(hard.r ? ⟨r⟩ .(
15:             (kill(kin))
16:             |{|hard.r ! ⟨r⟩ |}
17:             |(|x.Q|)_r^{h l})
18:         +[ prio ]soft.r ? ⟨myid, prio⟩ .(
19:             |(|Q|)_r^{h l}
20:             |comm.finish ! ⟨prio⟩
21:         )))
22:     );
```

is triggered if and only if a suitable *Output* agent has been found and the corresponding communication has been performed. Service $(\!|R|\!)_r^{h\,l}$, encoding the residual R is then instantiated. The invoke activities over channel $comm.finish$ are used to signal to *Monitor* that a communication has been successfully performed, and that a hard reset is needed.

Pseudocode 3.4 Definition of the COWS agent $Output(x, prio)$, introduced by function $[\![\,.\,]\!]$

```
1:  Output(x, prio) = x.prio ? ⟨x, prio⟩ .x.com ! ⟨x, prio⟩ ;
```

The encoding of a FAP output (in Pseudocode 3.4), both for high and low priority, is rather simple when compared to the ones already presented. The

burden of the organization of the prioritized search is in fact divided between agents *Monitor* and *Input*, so that an *Output* agent consists only of a request activity on channel *x.prio*, where *prio* can be either *h* or *l*, and of a subsequent invoke activity on channel *x.com*, which signals to the current activated *Input* agent that the search has successfully concluded.

3.4.2 Encoding Function $\langle . \rangle$

Encoding $\langle . \rangle$ maps FAP processes to a subset of COWS, named COWS_KF, which is obtained removing kill actions and protection operators from the language. Using properly nondeterministic choices and the communication paradigm of COWS, it is possible to represent the protocol in Pseudocode 3.1 without the use of *kill* activities and protection operators: the cleanup phase is modelled by communications over endpoints $notify.r$ and $retry.r$, which are used to notify the completion of the simulation of a FAP synchronization and to signal to activated input processes, not involved in the successful operation, to reset their state. We present the definition of the encoding function $\langle . \rangle$ from FAP processes to COWS_KF services in Table 3.4. The definitions of the agents used in Table 3.4 are given in Pseudocode 3.5, Pseudocode 3.6 and Pseudocode 3.4 (FAP output processes are encoded in the same way in both encoding functions).

$$
\begin{array}{lcl}
\langle P \rangle & = & [\, h, l, r, \widetilde{m} \,] \left(Mon2(h, l, r) \mid \langle |P| \rangle_r^{h\,l} \right) \\
\langle |P \mid Q| \rangle_r^{h\,l} & = & \langle |P| \rangle_r^{h\,l} \mid \langle |Q| \rangle_r^{h\,l} \\
\langle |x.Q| \rangle_r^{h\,l} & = & In2_{x.Q}(x, h, l, r) \\
\langle |\overline{x}| \rangle_r^{h\,l} & = & Output(x, l) \\
\langle |\underline{x}| \rangle_r^{h\,l} & = & Output(x, h) \\
\langle |\mathbf{0}| \rangle_r^{h\,l} & = & \mathbf{0}
\end{array}
$$

Table 3.4: Definition of the function $\langle . \rangle$, encoding FAP into COWS

As can be seen comparing the definitions of homologous agents in $[\![.]\!]$ and $\langle . \rangle$, the different approach in modeling the cleanup/reset phase causes only a

Pseudocode 3.5 Definition of $Mon2(h, l, r)$

```
1:  Mon2(h, l, r) =
2:    search.h ! ⟨h⟩
3:    | ([ x0 ]search.h ? ⟨x0⟩ .(
4:          search.l ! ⟨l⟩
5:          | ([ x1 ]search.l ? ⟨x1⟩ .0
6:             +notify.r ? ⟨r⟩ .(
7:                retry.r ! ⟨r⟩
8:                | [ x2 ]retry.r ? ⟨x2⟩ .Mon2(h, l, r)
9:             )
10:         )
11:      )
12:      +notify.r ? ⟨r⟩ .(
13:         retry.r ! ⟨r⟩
14:         | [ x3 ]retry.r ? ⟨x3⟩ .Mon2(h, l, r)
15:      )
16:   )
```

slight modification of the protocol: the major changes regard how service deal with their internal cleanup, rather than the way in which cleanup notifications are propagated between different services. Note that the definition of the encoding for a FAP output process is identical in $[\![.]\!]$ and $\langle . \rangle$.

3.4.3 Properties of encoding functions $[\![.]\!]$ and $\langle . \rangle$

Since the properties for the two encoding functions are identical and the proof strategies are very similar, we report here the proofs regarding the properties of function $\langle . \rangle$. This encoding function can be seen as more interesting, since it maps FAP processes to the kill and protection-free fragment of COWS. Proofs regarding the properties of function $[\![.]\!]$ are reported in Appendix A.

Theorem 3 states that all executions of a COWS_KF service $\langle P \rangle$ represent the first transition of an execution trace which, with a finite number of steps, reaches a configuration congruent to the encoding of a residual of P itself or, if

Pseudocode 3.6 Definition of agent $In2_{x.Q}(h, l, r)$

```
1:  In2_{x.Q}(h, l, r) =
2:  search.h ? ⟨h⟩ .(
3:      x.h ! ⟨h⟩
4:      | (x.com ? ⟨h⟩ .(
5:              notify.r ! ⟨r⟩
6:              | ⟨|Q|⟩_r^{h l}
7:          )
8:          +[ y0 ]x.h ? ⟨y0⟩ .(
9:              search.h ! ⟨h⟩
10:             | (
11:                 search.l ? ⟨l⟩ .(
12:                     x.l ! ⟨l⟩
13:                     | (
14:                         x.com ? ⟨l⟩ .(
15:                             notify.r ! ⟨r⟩
16:                             | ⟨|Q|⟩_r^{h l}
17:                         )
18:                         +[ y1 ]x.l ? ⟨y1⟩ .(
19:                             search.l ! ⟨l⟩
20:                             | ⟨|x.Q|⟩_r^{h l}
21:                         )))
22:                 +retry.r ? ⟨r⟩ .(
23:                     retry.r ! ⟨r⟩
24:                     | ⟨|x.Q|⟩_r^{h l}
25:                 )
26:             )
27:     ))
```

P cannot execute any action, a deadlocked configuration where the $Mon2$ agent is no more present.

Theorem 3. *If $\langle P \rangle \rightarrow s$ then there exists s' such that $s \rightarrow^* s'$ and either $s' \equiv \langle P' \rangle$ for some P' such that $P \rightarrow P'$, or $s' \equiv [\, h, l, r, \widetilde{m}\,]\langle |P| \rangle_r^{h\,l}$ with $\widetilde{m}$ as defined above and P such that $P \nrightarrow$.*

Proof. By definition of the encoding, we have

$$\langle P \rangle = [\, h, l, r, \widetilde{m}\,] \left(Mon2(h, l, r) \mid \langle |P| \rangle_r^{h\,l} \right)$$

We also know that the first action that service $\langle P \rangle$ can perform is a communication over endpoint *search.h*.

Branch 1 The structure of the derivation tree of the transition $\langle P \rangle \xrightarrow{search.h \cdot \sigma \cdot h \cdot u} s$, where we explicited the label , can have either the structure presented in Figure 3.3, in which case $u = x_0$ or the structure presented in Figure 3.4, in which case $u = h$. Given the semantic rules of COWS, the possibility of executing one of the two is mutually exclusive.

$$\dfrac{\dfrac{\vdots}{[\, l, r, \widetilde{m}\,]\langle |P| \rangle_r^{h\,l} \xrightarrow{search.h \cdot \varepsilon \cdot h \cdot x_0} s_1' \qquad h \notin \mathrm{d}(\alpha) \qquad s \downarrow_h}\,(del\_p)}{s = [\, h, l, r, \widetilde{m}\,]\langle |P| \rangle_r^{h\,l} \xrightarrow{search.h \cdot \varepsilon \cdot h \cdot x_0} [\, h\,]s_1' = s'}$$

Figure 3.3: Structure of the derivation tree for the transition $\langle P \rangle \xrightarrow{search.h \cdot \varepsilon \cdot h \cdot x_0} s$

In the former case, depicted in Figure 3.3, there is no $In2$ agent introduced by the encoding, otherwise a communication presenting a best matching tuple, thus having higher priority, could take place. Service s' can evolve only by performing a synchronization on endpoint *search.l* internal to agent $Mon2$ (lines 4,5 in Pseudocode 3.5) involving one substitution; the reached residual service, s'', is in a deadlock state. We have that the sequence of reduc-

tions $\langle P \rangle = [\, h, l, r, \widetilde{m} \,]\left(Mon2(h,l,r) \mid \langle |P| \rangle_r^{h\,l}\right) \rightarrow^* s''$ involves only actions in agent $Mon2$, which leaves no residual in s''. We can then conclude that $s'' \equiv [\, h, l, r, \widetilde{m} \,]\langle |P| \rangle_r^{h\,l}$.

For the analysis of the remaining cases, we note that process P can be generally written as $P \equiv_F \prod_{i \leqslant n} x_i.P_i \mid \prod_{j \leqslant m} \overline{x_j} \mid \prod_{k \leqslant o} \overline{x_k}$: P presents $n \geqslant 0$ unguarded input actions, $m \geqslant 0$ unguarded prioritized outputs and $o \geqslant 0$ unguarded unprioritized outputs.

$$\dfrac{\dfrac{\vdots}{[\, l, r, \widetilde{m} \,]\langle |P| \rangle_r^{h\,l} \xrightarrow{search.h \cdot \varepsilon \cdot h \cdot h} s_1' \qquad h \notin \mathrm{d}(\alpha) \qquad s \downarrow_h}{[\, h, l, r, \widetilde{m} \,]\langle |P| \rangle_r^{h\,l} \xrightarrow{search.h \cdot \varepsilon \cdot h \cdot h} [\, h \,]s_1' = s'}\,(del\_p)$$

Figure 3.4: Structure of the derivation tree for the transition $\langle P \rangle \xrightarrow{search.h \cdot \varepsilon \cdot h \cdot h} s$

High priority search If the latter case of Branch 1, depicted in Figure 3.4, takes place, i.e. if the initial communication over endpoint *search.h* is not internal to agent $Mon2$, by definition of encoding function $\langle . \rangle$ the only other possibility is that an $In2$ agent obtains the token for searching for a matching high-priority output action, executing the request action over endpoint *search.h* at line 2 in Pseudocode 3.6. Note that this communication involves no substitution. Without loss of generality, let suppose that the activated agent has been introduced by the encoding when considering a FAP process $x_{i'}.Q_{i'}$. In this case, the $In2$ agent has been instantiated as $In2_{x.Q_{i'}}(x_{i'}, h, l, r)$.

Branch 2 After the acquisition of the token, there are only two possible continuations: either the activated input agent performs an internal synchronization on endpoint $x_{i'}.h$ (lines 3,8 in Pseudocode 3.6) involving one substitution, or a communication over $x_{i'}.h$ involving no substitution between the activated $In2$

agent (line 3 in Pseudocode 3.6) and an agent $Output(x_{i'}, h)$ happens (line 1 in Pseudocode 3.4).

In the former case, the encoding function has not introduced any service $Output(x_{i'}, h)$, meaning that $P \neq Q_1 \mid \underline{\overline{x_{i'}}} \mid Q_2$ for any FAP processes Q_1, Q_2. As a consequence of the synchronization, the residual of the activated $In2$ agent is ready to release the high priority token, providing an invoke action on endpoint *search.h* (line 9 of Pseudocode 3.6) and waiting for the low priority token (line 11) or for a reset signal (line 22). If the token is acquired by another $In2$ agent, then the evolution is described reconsidering Branch 2.

High priority synchronization In the latter case, the only possible continuation is a synchronization between the residual of the activated *Input* agent and the residual of the found $Output(x_{j'}, h)$ agent over endpoint $x_{i'}.com$, for $x_{j'} = x_{i'}$ (lines 4 in Pseudocode 3.6 and line 1 in Pseudocode 3.4). The only possible continuation is a communication over endpoint $notify.r$ with r as parameter (line 12 in Pseudocode 3.5 and line 5 in Pseudocode 3.6). There is only one possible continuation: a synchronization over endpoint *retry.r*: all residuals of *Input* agents which unsuccessfully took part in the search for a matching high priority output can perform a request activity over *retry.r* (line 22 in Pseudocode 3.6) which equally matches the invoke activity at line 13 in Pseudocode 3.5. For each of these *Input* residuals, the residual of the *Input* agent is composed of an invoke action propagating the reset signal (line 23 in Pseudocode 3.6) and of a new instance of the encoding of the input action. The last synchronization over endpoint *retry.r* takes place between the last *Input* residual needing a reset (line 23 in Pseudocode 3.6)and the residual of agent *Monitor* (line 14 in Pseudocode 3.5). After this synchronization, the residual of the system, identified by s_2, has the form presented in Pseudocode 3.7.

Note that s_2 is congruent to the encoding of the FAP process P', reachable from P after a high priority synchronization over channel $x_{i'} = x_{j'}$.

Pseudocode 3.7 Encoding of a FAP process after a successful low priority communication

1: $s_2 \equiv [\, h, l, r, \widetilde{m}\,](Monitor(h, l, r)$
2: $\quad | \langle |Q_{i'}| \rangle_r^{h\,l}$
3: $\quad | \prod_{i \leqslant n, i \neq i'} \langle |x_i.Q_i| \rangle_r^{h\,l}$
4: $\quad | \prod_{j \leqslant m, j \neq j'} \langle |\overline{x_j}| \rangle_r^{h\,l}$
5: $\quad | \prod_{k \leqslant o} \langle |\overline{x_k}| \rangle_r^{\overline{h\,l}}$
6: $\quad)$

Low priority search If each activated $In2_{x_i.Q_i}(x_i, h, l, r)$ agent has performed the internal synchronization over endpoint $x_i.h$, meaning that no corresponding high priority output encoding service was found, the residual of service s is a service s_1 which we can identify, by construction of the encoding, as the service presented in Pseudocode 3.8 If the w-th activated $In2$ agent is instantiated as $In2_{x_w.Q_w}(x_w, h, l, r)$ and one of the m unguarded prioritized outputs in P, indexed by p is instantiated as $Output(x_p, h)$ where $x_w = x_p$, then the invoke action on endpoint $x_w.h$ at line 3 in Pseudocode 3.6 best matches with the request action at line 1 in Pseudocode 3.4. After this synchronization, the only possible continuation is a synchronization over endpoint $x_w.com$, following by an internal synchronization in agent $In2$ over endpoint $soft.r$ (lines 5,18 in Pseudocode 3.6). The encoding of the residual of the FAP input process Q is exposed, and the information about the executed synchronization is forwarded to the residual of the $Mon2$ service over endpoint $comm.finish$. (line 20 in Pseudocode 3.6 and line 10 in Pseudocode 3.5). The only possible continuation is the repetition, $w - 1$ times, of a synchronization over endpoint $hard.r$ (line 15 in Pseudocode 3.5 and line 16 in Pseudocode 3.6 synchronizing with line 14 in Pseudocode 3.6). This synchronization triggers the cleanup phase, first carried out by the residual of the w-th activated $In2$ agent, which performs the kill activity over kill label kin. In this way each of the $w-1$ input agents is reset to its initial state $\langle |x_i.Q_i| \rangle_r^{h\,l}$. After this sequence, the only possible synchronization is again on endpoint $hard.r$ and involves the invoke action at line 16 of the $w - 1$-th considered $In2$ agent

and the residual of agent $Mon2$ (at line 16 in Pseudocode 3.5) or, if $w = 0$ this synchronization is internal to agent $Mon2$. In either case this step resets the monitor to its initial state $Mon2(h, l, r)$. At the end we obtain a service

$$\begin{aligned} s'' \equiv [\,h, l, r, \widetilde{m}\,]\Big(\quad & Mon2(h, l, r) \\ & \mid \textstyle\prod_{i \leq n, i \neq w} \langle |x_i . Q_i| \rangle_r^{h\,l} \\ & \mid \langle |Q_w| \rangle_r^{h\,l} \\ & \mid \textstyle\prod_{j \leqslant m, j \neq p} \langle |\overline{x_j}| \rangle_r^{h\,l} \\ & \mid \textstyle\prod_k \langle |\overline{x_k}| \rangle_r^{h\,l} \quad \Big) \end{aligned}$$

Service s'' is structurally congruent to the residual of process P after a high priority synchronization over name x.

Pseudocode 3.8 Encoding of a FAP process after an unsuccessful high priority search

```
1: s1 ≡ [h, l, r, m̃](search.l ! ⟨l⟩
2:     | ([v2]search.l ? ⟨v2⟩ .0
3:             +notify.r ? ⟨r⟩ .(...))
4:     | ∏_{i⩽n} search.l ? ⟨l⟩ .(x_i.l ! ⟨x_i, l⟩ | ...)
5:     | ∏_{j⩽m} Output(x_j, h)
6:     | ∏_{k⩽o} Output(x_k, l)
7:     )
```

In case of an unsuccessful high priority search, the presented behaviour is repeated, considering l in place of h as name for the identification of the priority. In particular, we have that at least one synchronization happens on endpoint $search.l$. Either it is an internal synchronization to the residual of agent $Mon2$ (lines 1-2 in Pseudocode 3.8) or, for some i', it involves the request at line 5 in Pseudocode 3.6 for the i'-th considered $In2$ agent.

Branch 3 In the first case, the encoding phase did not introduce any $In2$ agent. This situation has already been discussed in Branch 1. In the second case, similarly to what happened before, the residual of an $In2$ agent, indexed as i' and thus instantiated as $In2_{x_{i'}.Q_{i'}}(x_{i'}, h, l, r)$ gets the token to perform the search for

an unguarded low priority output action indexed as k', and thus encoded as $Output(x_{k'}, l)$ with $x_{i'} = x_{k'}$.

Branch 4 If such an *Output* agent does not exist, the only possible executable action is an internal synchronization over endpoint $x_{i'}.l$ (lines 11 and 18 in Pseudocode 3.6), after which the token is again available. Another $In2$ agent can then obtain the low priority token, and the protocol continues from the second choice of Branch 3.

Low priority synchronization If, on the other hand, such an *Output* agent exists, the only possible executable action is a synchronization over endpoint $x_{i'}.l$ involving no substitution (line 11 in Pseudocode 3.6 and line 1 in Pseudocode 3.4); this communication can be followed only by a synchronization over endpoint $x_{i'}.com$ (line 14 in Pseudocode 3.6 and line 1 in Pseudocode 3.4). At this point the encoding of the residual $Q_{i'}$ of action $x_{i'}.Q_{i'}$ is instantiated, alongside with the invoke activity over endpoint $notify.r$ among agent $In2$ (line 15 in Pseudocode 3.6) and agent $Mon2$ (line 6 in Pseudocode 3.5). This synchronization triggers $n-1$ repetitions of a synchronization involving all $In2$ actions which participated in the search, but not in the successful communication. These synchronizations happen over endpoint $retry.r$ (line 7 in Pseudocode 3.5, lines 22 and 23 in Pseudocode 3.6). After this sequence, the last considered $In2$ action synchronizes over endpoint $retry.r$ with the residual of agent $Mon2$ (line 23 in Pseudocode 3.6 and line 8 Pseudocode 3.5) involving one substitution. After this synchronization, a fresh instance of the $Mon2$ agent is ready to restart the execution of the communication protocol. We obtain a service s_2 which is of the form presented in Pseudocode 3.9.

Note that service s_2 has the form of the encoding of a FAP process

$$P' \equiv_F Q_{i'} \mid \prod_{i \leqslant n, i \neq i'} x_i.P_i \mid \prod_{j \leqslant m} \overline{x_j} \mid \prod_{k \leqslant o, k \neq k'} \overline{x_k}$$

Pseudocode 3.9 Encoding, according to $\langle . \rangle$, of a FAP process after a successful low priority communication

1: $s_2 \equiv [\, h, l, r, \widetilde{m}\,](Mon2(h, l, r)$
2: $\quad | \, \langle |Q_{i'}| \rangle_r^{h\,l}$
3: $\quad | \, \prod_{i \leqslant n, i \neq i'} \langle |x_i.Q_i| \rangle_r^{h\,l}$
4: $\quad | \, \prod_{j \leqslant m} \langle |\overline{x_j}| \rangle_r^{h\,l}$
5: $\quad | \, \prod_{k \leqslant o, k \neq k'} \langle |\overline{x_k}| \rangle_r^{h\,l}$
6: $\quad)$

which represents the residual of process P after the execution of a low priority synchronization over name $x_{i'} = x_{k'}$ between process $x_{i'}.Q_{i'}$ and process $\overline{x_{k'}}$.

The last possibility that we have to consider is the one where all $In2$ agents perform unsuccessfully a low priority search, i.e. no synchronization is possible in the encoded FAP process P. In this case, the last synchronization happening over endpoint *search.l* involves the n-th $In2$ agent and the residual of the $Mon2$ agent (line 5 in Pseudocode 3.5 and line 19 in Pseudocode 3.6). Note that the obtained residual service s_3 is congruent to the service presented in Pseudocode 3.10, which can perform no action.

Pseudocode 3.10 Encoding after a totally performed unsuccessful search

1: $s_3 \equiv [\, h, l, r, \widetilde{m}\,]($
2: $\quad | \, \prod_{i \leqslant n} \langle |x_i.Q_i| \rangle_r^{h\,l}$
3: $\quad | \, \prod_{j \leqslant m} \langle |\overline{x_j}| \rangle_r^{h\,l}$
4: $\quad | \, \prod_{k \leqslant o} \langle |\overline{x_k}| \rangle_r^{h\,l}$
5: $\quad)$

Given the distributive property of function $\langle |. | \rangle_r^{h\,l}$, service s_3 is congruent to $[\, h, l, r\,] \langle |P| \rangle_r^{h\,l}$.

□

Theorem 4 states that each transition a FAP process P can execute can be simulated by a number of transitions by the COWS_KF service obtained encoding P using function $\langle . \rangle$.

Theorem 4. *If $P \to P'$ then $s = \langle P \rangle \to^* \equiv \langle P' \rangle$*

Proof. Given the semantics of FAP, a process P can perform a reduction $P \to P'$ in two cases: either it performs a high-priority synchronization $P \twoheadrightarrow P'$ or it performs a low-priority synchronization $P \mapsto P'$. By rule $(*)$ in Table 3.1, these two cases are mutually exclusive.

1. $P \twoheadrightarrow P'$. There exists a proof tree for this derivation of the form

$$\frac{P \equiv_F Q_1 \qquad \dfrac{\dfrac{}{x.R_1 \mid \overline{\underline{x}} \twoheadrightarrow R_1}}{Q_1 = x.R_1 \mid \overline{\underline{x}} \mid R_2 \twoheadrightarrow R_1 \mid R_2 = Q_2} \qquad Q_2 \equiv_F P'}{P \twoheadrightarrow P'}$$

Given this structure, we have $P \equiv_F x.R_1 \mid \overline{\underline{x}} \mid R_2$ and $P' \equiv_F R_1 \mid R_2$. We can then derive the encoding of P according to $s = \langle . \rangle$ to be

$$\begin{aligned} s = \langle P \rangle \;=\; & [\, h, l, r\widetilde{m} \,](\\ & \quad Mon2(h, l, r) \\ & \quad \mid In2_{x.R_1}(x, h, l, r) \\ & \quad \mid Output(x, h) \\ & \quad \mid \langle |R_2| \rangle_r^{h\,l} \\ &) \end{aligned}$$

Given this structure and the definition of agents $Mon2$, $In2$ and $Output$, we can describe a possible evolution of $\langle P \rangle$ that will reach a configuration congruent to $\langle P' \rangle$, as needed.

We have that s can perform a communication over endpoint *search.h* involving no substitution (line 2 in Pseudocode 3.5 and line 2 in Pseudocode 3.6). Given the definition of agent $Output(x, h)$ introduced by the encoding, the only possible continuation is a synchronization over endpoint $x.h$ involving no substitution (line 3 in Pseudocode 3.6 and line 1 in Pseudocode 3.4) followed by a synchronization over endpoint *x.com* (line 1 in Pseudocode 3.4

and line 4 in Pseudocode 3.6, which consumes the residual of agent $Output(x, h)$. The only possible continuation for the residual service is represented by a synchronization over endpoint $notify.r$ between the activated $In2$ agent (line 5 in Pseudocode 3.6) and the residual of the agent $Mon2$ (line 6 in Pseudocode 3.5. At this point the residual of agent $In2$ is composed of the instantiation of the encoding of Q, which is the residual of the FAP input action $x.Q$ (line 6 in Pseudocode 3.6). The residual of agent $Mon2$ performs an internal synchronization over endpoint $retry.r$ (no other $In2$ agent was activated, so the internal synchronization is the best matching, even if it involves one substitution). A new instance of the $Mon2$ agent is instantiated. After this action, the residual service is $[\,h, l, r, \widetilde{m}\,](Mon2(h, l, r) \mid \langle |R_1| \rangle_r^{h\,l} \mid \langle |R_2| \rangle_r^{h\,l})$, which is congruent to the service obtained applying the encoding function to P'.

2. $P \mapsto P'$. Given the semantics of FAP, we have that $P \equiv_F Q_1 \not\twoheadrightarrow$, otherwise rule $(*)$ in Table 3.1 could not have been applied. We have that the derivation tree for $P \mapsto P'$ is

$$\dfrac{P \equiv_F Q_1 \qquad \dfrac{\dfrac{}{x.R_1 \mid \overline{x} \mapsto R_1} \qquad x.R_1 \mid \overline{x} \mid R_2 \not\twoheadrightarrow}{Q_1 = x.R_1 \mid \overline{x} \mid R_2 \mapsto R_1 \mid R_2 = Q_2} \qquad Q_2 \equiv_F P'}{P \mapsto P'}$$

Process P can be generally written as $\prod_{i \leqslant n} x_i.P_i \mid \prod_{j \leqslant m} \overline{x_j} \mid \prod_{k \leqslant o} \overline{x_k}$. Given the reduction tree, no prioritized synchronization can happen in P, so we can state that for all i, j such that $P \equiv_F x_i.P_i \mid \overline{x_j} \mid R$, we have $x_i \neq x_j$. In the following, we will identify $x.R_1$ as the $\hat{i}$-th input process, i.e. $x_{\hat{i}}.R_{\hat{i}}$, and $\overline{x}$ as the $\hat{k}$-th output process, i.e. $\overline{x_{\hat{k}}}$.

We now consider the COWSservice $s = \langle P \rangle$. We can state that the number of FAP input processes, and subsequently the number of $In2$ agents introduced by the encoding, is $n > 0$; for this reason, s can evolve performing

a transition on endpoint *search.h* with no substitution involved between agent $Mon2$ (line 2 in Pseudocode 3.5) and one of the $In2$ agents (line 2 in Pseudocode 3.6). At this point, the residual of the activated $In2$ agent can only perform an internal synchronization on endpoint $x.h$ involving one substitution (lines 3 and 8 in Pseudocode 3.6): no matching service $Output(x, h)$, which could provide a better-matching request, has been introduced by the encoding function. The residual of the $In2$ agent releases the token, providing an invoke action on endpoint *search.h*. The sequence of transitions is repeated for all the $In2$ agents introduced by the encoding, where the last activated $In2$ agent releases the token by synchronizing with the residual of the $Mon2$ agent on endpoint *search.h*) with one substitution (line 9 in Pseudocode 3.6 and line 3 in Pseudocode 3.5).

At this point, the residual of the $Mon2$ agent provides an invoke action on endpoint *search.l* representing the low priority token, which can be acquired nondeterministically by one of the n $In2$ residuals. By simplicity, let the residual of the encoding introduced by considering the FAP input action $x_{\hat{i}}.R_{\hat{i}}$ be the first one to acquire the token by synchronizing on endpoint *search.l* with the residual of agent $Mon2$. By reasoning on the hypotheses, we derived the fact that a matching low priority output action is present in P, so an agent of the form $Output(x_{\hat{i}}, l)$ has been introduced by the encoding. For this reason, a synchronization on endpoint $x_{\hat{i}}.l$, involving no substitution, is possible (line 12 in Pseudocode 3.6 and line 1 in Pseudocode 3.4). After a subsequent synchronization on endpoint $x_{\hat{i}}.com$ (line 14 in Pseudocode 3.6 and line 1 in Pseudocode 3.4), the encoding of the residual $P_{\hat{i}}$ is instantiated. The only possible continuation is a synchronization on endpoint $notify.r$ (line 15 in Pseudocode 3.5 and line 6 in Pseudocode 3.5). This action signals to $Mon2$ that a low synchronization has been performed, and that a reset signal has to be sent out on endpoint *retry.r* to all the $In2$ agents who unsuccessfully searched for

a matching partner. The first signal is sent by the $Mon2$ agent itself (line 7 in Pseudocode 3.5); it matches with the request activity at line 22 in Pseudocode 3.6. After this synchronization, the reset signal is replicated, in parallel with a clean instance of the $In2$ agent. This sequence is repeated $n - 1$ times; the last invoke action on endpoint *retry.r* (line 23 in Pseudocode 3.6) is intercepted by the residual of agent $Mon2$ (line 8 in Pseudocode 3.5), whose residual is a fresh instance of the $Mon2$ agent.

After this sequence of transitions, we get a service

$$\begin{aligned} s' \equiv\ & [\,h, l, r\widetilde{m}\,](\\ & \quad Mon2(h,l,r)\\ & \quad \mid \langle|R_{\hat{i}}|\rangle_r^{h\,l}\\ & \quad \mid \textstyle\prod_{i\leqslant n, i\neq\hat{i}} \langle|x_i.R_i|\rangle_r^{h\,l}\\ & \quad \mid \textstyle\prod_{j\leqslant m} \langle|\underline{\overline{x_j}}|\rangle_r^{h\,l}\\ & \quad \mid \textstyle\prod_{k\leqslant o, k\neq\hat{k}} \langle|\overline{x_k}|\rangle_r^{h\,l}\\ &)\end{aligned}$$

Service s' is congruent to service $\langle P' \rangle$, as required.

□

Theorem 5 states that encoding $\langle . \rangle$ is livelock-free.

Theorem 5. *If $P \nrightarrow$ then $s = \langle P \rangle \rightarrow^* \equiv [\,h, l, r\widetilde{m}\,]\langle|P|\rangle_r^{h\,l}$*

Proof. Given that $P \nrightarrow$, we have that a proper derivation tree for a transition of P could not be built. Given the set of semantic rules for FAP, this means that neither of the two axioms in Table 3.1, namely rule (*com*) and (*pr_com*), could be applied. In other words, if we write $P \equiv_F \prod_{i\leqslant n} x_i.P_i \mid \prod_{j\leqslant m} \underline{\overline{x_j}} \mid \prod_{k\leqslant o} \overline{x_k}$, $x_i \neq x_j$ and $x_i \neq x_k$ for all values $0 < i \leqslant n$, $0 < j \leqslant m$ and $0 < k \leqslant o$. Applying the encoding function $\langle . \rangle$ to P, we obtain a service $s = \{\!|hlr|\!\}(Mon2(h,l,r) \mid \prod_{i\leqslant n}\langle|x_i.P_i|\rangle_r^{h\,l} \mid \prod_{j\leqslant m}\langle|\underline{\overline{x_j}}|\rangle_r^{h\,l} \mid \prod_{k\leqslant o}\langle|\overline{x_k}|\rangle_r^{h\,l})$. Based on

the value of n, i.e. the number of $In2$ agent instantiated by the encoding, we have two possible scenarios:

1. $n = 0$: the evolution of service s is the one presented for choice 1 in Branch 1 for Theorem 6: service s can evolve only by performing a synchronization on endpoint *search.h* internal to agent $Mon2$ (lines 2,3 in Pseudocode 3.2) involving one substitution; after another internal synchronization on endpoint *search.l* (lines 4,5 in Pseudocode 3.2), the residual service, let call it be s', is in a deadlock state. We have that the sequence of reductions $\langle P \rangle = [\, h, l, r, \widetilde{m} \,] \left(Mon2(h, l, r) \mid \langle |P| \rangle_r^{h\,l} \right) \rightarrow^* s'$ involves only actions in agent $Mon2$, which is no more present in s', i.e. $s' \equiv [\, h, l, r \,] \langle |P| \rangle_r^{h\,l}$.

2. $n > 0$: the evolution of service s has been presented in Theorem 6 when considering unsuccessful searches for both high and low priority outputs. We have that in P there is at least one unguarded input FAP process ($n > 0$); given the definition of the encoding function, this means that at least one unguarded $In2$ service is in the encoding $\langle P \rangle$. We also know that in P, for all input processes, there is no unguarded matching output, otherwise P could perform a transition, contradicting the hypothesis of the theorem.

   Given the definition of the encoding function applied to P under these conditions, all $In2$ agents, instantiated as $In2_{x_i.Q_i}(x_i, h, l, r)$, are activated (one at a time) through a communication labelled as $search.h \cdot \varepsilon \cdot h \cdot h$ (line 2 in Pseudocode 3.5 and line 2 in Pseudocode 3.6) and declare the search as unsuccessful by performing an internal synchronization labelled as $x_i.h \cdot \varepsilon \cdot h \cdot y_0$ (lines 3,8 in Pseudocode 3.6). The last considered $In2$ agent synchronizes with agent $Mon2$ with a communication labelled as $search.h \cdot \varepsilon \cdot h \cdot x_0$ (line 3 in Pseudocode 3.5 and line 9 in Pseudocode 3.6). This removes the high priority token and instantiates the low priority one. All residuals of $In2$ agents, which are at this point instantiated as a service congruent to

$s_i^* =\equiv (search.l\,?\ \langle l\rangle\,.(\ldots).$

The residual of service $\langle P\rangle$, at this point, can only continue by performing n times the sequence composed of a communication on endpoint $search.l$ (line 4 in Pseudocode 3.5 and lines 11,19 in Pseudocode 3.6), labelled as $search.l \cdot \varepsilon \cdot l \cdot l$, which activates one of the $In2$ residuals, followed by a synchronization internal to the activated $In2$ agent on endpoint $x_i.l$ labelled as $x_i.l \cdot \varepsilon \cdot l \cdot y_1$ (lines 12, 18 in Pseudocode 3.6). As before, the possibility of having a communication on endpoint $x_i.l$ involving no substitution (which would take precedence) is denied by the lack of any matching output action in the encoded process. After each synchronization over $x_i.l$, a fresh instance of the i-th $In2$ agent involved in the last synchronization is instantiated. The last synchronization on endpoint $search.l$ is between the last activated $In2$ agent and the residual of agent $Mon2$ (line 5 in Pseudocode 3.5 and line 19 in Pseudocode 3.6). After this the residual of agent $Mon2$ disappears. The obtained residual is congruent to service $[\,h, l, r, \widetilde{m}\,]\langle|P|\rangle_r^{h\,l}$. Since agent $Mon2$ is not present, the high priority token (in the form of the unguarded invoke action on endpoint $search.h$) is not reinstantiated, so the process is in a deadlock state.

□

We now present the theoretical results, homologous to the ones presented for function $\langle\,.\,\rangle$, that formalize the operational correspondence between a FAP process and the COWS_KF service encoding it according to function $[\![\,.\,]\!]$. Theorem 3 states that all executions of a COWS_KF service $[\![P]\!]$ represent the first transition of an execution trace which, with a finite number of steps, reaches a configuration congruent to the encoding of a residual of P itself or, if P cannot execute any action, a deadlocked configuration where the *Monitor* agent is no more present. Theorem 7 is the counterpart to Theorem 6, as it states that each transition a FAP process P can execute can be simulated by a number of transi-

tions by the COWS_KF service obtained encoding P using function $[\![.]\!]$. Finally, Theorem 8 states that if P cannot perform any transition, then its encoding $[\![P]\!]$, after a sequence of transitions, reaches a deadlock state as well, avoiding the presence of livelock loops.

Theorem 6. *If $[\![P]\!] \rightarrow s$ then there exists s' such that $s \rightarrow^* s'$ and either $s' \equiv [\![P']\!]$ for some P' such that $P \rightarrow P'$, or $s' \equiv [\, h, l, r, \widetilde{m}\,](\!|P|\!)^{h\,l}_{r}$ with P such that $P \nrightarrow$.*

Proof. The proof, which can be found in Appendix A.1, is very similar to the one given for Theorem 6. □

Theorem 7 states that if a FAP process P is able to perform an action reducing to P', then the corresponding COWS service $[\![P]\!]$ can, after a number of reductions, reach a residual that is congruent to $[\![P']\!]$. The number of steps required to reach this residual depends on the priority of the reduction that process P performs.

Theorem 7. *If $P \rightarrow P'$ then $s = [\![P]\!] \rightarrow^* \equiv [\![P']\!]$*

Proof. The proof of Theorem 7, given in Appendix A.2, is very similar to the proof of Theorem 4.

□

Theorem 8. *If $P \nrightarrow$ then $s = [\![P]\!] \rightarrow^* \equiv [\, h, l, r, \widetilde{m}\,](\!|P|\!)^{h\,l}_{r}$*

Proof. The proof of Theorem 8, given in Appendix A.3, is very similar to the proof of Theorem 5. □

3.4.4 Comments on Encoding Functions $[\![.]\!]$ and $\langle . \rangle$

As we said in Chapter 1, there are two approaches to compare the expressive power of process calculi: the absolute one, which regards the capabilities of solving a given problem using a process calculus, and the relative one, which studies of encoding functions from one language to another.

Whether one approach or the other is chosen, one has to specify which are the properties that define a *good* encoding function, depending on the context and on the task at hand. Given this fact, it is not surprising that a general agreement has not been found on which properties have to be considered. For example, in [25, 32] the concept of network symmetry plays a crucial role. For this reason, an encoding is deemed as *good* if it is homomorphic w.r.t parallel composition: a *good* encoding should not introduce a central synchronization process, which would break the symmetry. In [24], however, it is argued that this requirement is too strict and that, for practical purposes, a compositional encoding function (which is allowed to introduce a central controller) is a more suitable choice; the same property is required in [5] for basic encodings. Another example of difference in the treatment of encoding properties is represented by name independence: while in [5] and [32] independence preservation plays a crucial role, it is considered in [33] only for the solvability of the Leader Election Problem for ring network topologies. Another point of discussion is represented by divergence: observation respecting encodings, such as those considered in [33], cannot introduce divergence, while for some authors (e.g. [24]) to some extent divergence introduction can be tolerated, if certain fairness or scheduling assumptions are made; in some works (e.g. [5]), the requirement on divergence introduction is dropped altogether.

Following the characterization of the COWS process algebra presented in [5], where the language is proved to be non replacement free by analysing the **kill**(.) prioritized operator, we focused our work on the communication paradigm of COWS. This paradigm is based on pattern matching and introduces a priority mechanism built on top of a best matching policy. We compared the relevant fragment of the language (named COWS_KF) to a fragment of CCS with global priorities named FAP, presented in [32]. We concluded that there is no observation respecting, independence preserving and distribution preserving encoding of FAP into COWS_KF. We then presented a compositional (but not

distribution preserving) encoding of FAP into COWS, making use of both the communication paradigm and the **kill**(.) operator. We showed that there exists an operational correspondence between the original and the encoded process. Moreover, the communication protocol introduced by the encoding allows a COWS service, obtained encoding a FAP process, to detect a deadlock state in the original process in a finite number of internal synchronizations, i.e. without the introduction of live-locks.

Chapter 4

A Stochastic Extension

In this chapter we present a stochastic extension of COWS, named SCOWS, revisiting the parts of the syntax and of the semantics presented in Chapter 2 which are affected by the extension. SCOWS was first introduced in [31]. With respect to that work, the stochastic extension to COWS presented here differs mainly in the definition of the computation of transition rates, which is now less involved. Moreover, we present the implementation of a simulator which is able to derive the whole Labelled Transition Systems of SCOWS terms and producing the associated Continuous Time Markov Chains. We describe optimizations and implementation choices needed to consider non-trivial models.

4.1 Basic Notions

We now revise some notions about Continuous Time Markov Chains (CTMCs) and quantitative properties expressed using CSL [3].

4.1.1 Continuous Time Markov Chains

CTMCs (Continuous Time Markov Chains) are one of the most common representation for stochastic transition systems. In this formalism, a model C is represented by a structure $C =< S, \mathbf{Q} >$ where S is a countable set of states

and $\mathbf{Q} : \left[q_{ij}\right], \quad i, j \in \{1, \ldots, |S|\}$ is the transition rate matrix associating a non-negative real number to the pair of states (s_i, s_j), if $i \neq j$; this means that the system, when in s_i, performs a transition to s_j with a propensity or, equivalently, with a delay which is exponentially distributed according to rate $q_{ij} \in \mathbb{R}_{\geqslant 0}$. Given the nature of $\mathbf{Q}$, we have that $q_{ii} = -\sum_{j \neq i} q_{ij} \leqslant 0$.

Given $C =< S, \mathbf{Q} >$ and $s_i \in S$, the exit rate of s_i is given by

$$E(s_i) = \sum_{s_j \in S, i \neq j} q_{ij}$$

Using the exit rate of a state s_i, it is possible to compute the probability that a model C in state s_i at a given time will, at the next step, perform a transition to state s_j as

$$P_C[s_i \rightarrow s_j] = \frac{q_{ij}}{E(s_i)}$$

Each rate in matrix $\mathbf{Q}$ is associated to a negative exponential distribution, whose probability density function is defined as $f(\gamma, x) = \gamma e^{-\gamma x}$ with $\gamma, x \in \mathbb{R}_{>0}$ and cumulative density function is defined as $F(\gamma, x) = 1 - e^{-\gamma x}$. Historically, the choice of exponential distributions for modelling time has been preferred for two reasons: this is the only continuous distribution satisfying the memoryless property and it is described by a single parameter, the stochastic rate γ.

4.1.2 Expressing Quantitative Properties

If we want to get quantitative information, e.g. regarding time, about a formal model, we have to be able to express in a formal way the property we want to check. In this work we consider CSL [3] as the language to express such properties. In particular, here we present a version of CSL supported by the PRISM model checker [17]. This tool supports properties specified to inquire the transient behaviour of models P $\bowtie b[pathprop]$) or their steady-state behaviour (S$\bowtie b[pathprop]$), where *pathprop* is a path property evaluated in the current state and $\bowtie b$ is a probability bound or =?. In the former case, checking

the property gives a boolean result, while in the latter case the result is the probability of verifying *pathprop*. In the following, we will write $s_1, s_2, \ldots, s_n$ to indicate a path composed of the sequence of transitions $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$ assuming that this sequence of transitions is defined for the considered CTMC model. Path properties are expressed using propositions on local variables, boolean operators and path operators **X** , **U** , **F** , whose meaning can be summarized as:

1. next: **X** *pathprop* is true if the next state verifies *pathprop*

2. until: $pathprop_1$ **U** $pathprop_2$ is true for a path $s_1, \ldots, s_n$ if $pathprop_2$ is true in a state s_i with $1 \leq i \leq n$ and in all preceding states s_j, where $1 \leq j < i$, we have that $pathprop_1$ is true

3. eventually: **F** *pathprop* is true if *true* **U** *pathprop* is true

Using these properties as building blocks, we can derive other useful operators, such as the *always* operator **G** *pathprop*, which can be used to express the fact that *pathprop* is always true, the *weak until* operator ($pathprop_1$ **W** $pathprop_2$), which differs from the *until* operator in that it does not require $pathprop_2$ to ever become true if $pathprop_1$ remains true along the considered path, an the *release* operator $pathprop_1$ **R** $pathprop_2$ which expresses the fact that $pathprop_2$ is true until $pathprop_1$ becomes true, or $pathprop_2$ is true forever.

Formally, these derived operators can be defined as

1. always: **G** *pathprop* is true if ¬ **F** ¬*pathprop* is true

2. weak until: $pathprop_1$ **W** $pathprop_2$ is true if $pathprop_1$ **U** $pathprop_2$ is true or **G** $pathprop_1$ is true

3. release: $pathprop_1$ **R** $pathprop_2$ is true if ¬(¬$pathprop_1$ **U** ¬$pathprop_2$) is true

4.2 SCOWS

Following an approach similar to the ones used in [28], [15] and [27], the stochastic extension of the considered language is based on defining a labelled operational semantics and in associating each action to a stochastic rate, denoted by $\gamma, \gamma', \delta, \lambda, \ldots \in \mathbb{R}_{>0}$. Rates model the time needed for actions to be executed, or, alternatively, the delay associated to the triggering of actions. Execution times and delays are modelled according to exponential distributions. The syntax of SCOWS, derived from the one presented in Chapter 2 adding rates, is presented in Table 4.1.

$$
\begin{array}{lcl}
s & ::= & (\mathbf{kill}(k), \lambda) \mid (u.u' \,!\, \widetilde{u}, \gamma) \mid g \mid s \mid s \mid \{\!|s|\!\} \mid [\,d\,]s \mid S(d_1, \ldots, d_j) \\
g & ::= & \mathbf{0} \mid (p.o \,?\, \widetilde{u}.\, s, \delta) \mid g + g
\end{array}
$$

Table 4.1: Syntax of SCOWS

As stated in Chapter 2, we will use d, d' to range over $\mathcal{N} \cup \mathcal{V} \cup \mathcal{K}$ and $u, u', \ldots$ to range over $\mathcal{N} \cup \mathcal{V}$. Given that scope extrusion is handled using opening and closing of delimiters, the operational semantics of SCOWS is given as a labelled system. For this reason, we let labels of the operational semantics contain names and variables enclosed in parenthesis, like, e.g., (n) and (x), where (n) denotes a name n whose scope has been opened. The set of names enclosed in parenthesis is collectively referred to as $(\mathcal{N})$, and the set of variables enclosed in parenthesis is referred to as $(\mathcal{V})$. We will use $a, a', \ldots$ to range over $\mathcal{N} \cup (\mathcal{N})$ and $w, w', \ldots$ to range over $\mathcal{N} \cup \mathcal{V} \cup (\mathcal{V})$. We extend the use of notation $\widetilde{\_}$ for tuples so that $\widetilde{a}$ stays for a non-empty tuple of elements in $\mathcal{N} \cup (\mathcal{N})$ and $\widetilde{w}$ stays for a non-empty tuple of elements in $\mathcal{N} \cup \mathcal{V} \cup (\mathcal{V})$. Functions $fid(.), fn(.), fv(.), fkl(.)$ and $bid(.), bn(.), bv(.), bkl(.)$, defined in Chapter 2 for COWS services, are intuitively extended to consider SCOWS services.

As for COWS services, given $\widetilde{d} = d_1, \ldots, d_n$, we will often use the notation $[\,\widetilde{d}\,]s$ or, alternatively, $[\,d_1, \ldots, d_n\,]s$ to represent service $[\,d_1\,] \ldots [\,d_n\,]s$.

We use $\sigma, \sigma', \ldots$ to denote substitutions of names for variables. The usual notation for substitutions is slightly abused, and in the labels of the operational semantics we actually use substitutions to also carry information about scope opening: for example, $\{(n)/v\}$ denotes the substitution of a variable v with a name n, when the scope of n has been opened. We also extend the definition of the cosupport of a substitution: given a set of substitutions $\{a_1/x_1, \ldots, a_j/x_j\}$, its cosupport $\mathrm{csp}(\{a_1/x_1, \ldots, a_j/x_j\})$ is defined to be $\{a_1, \ldots, a_j\}$. Given a set of k substitutions $\sigma = \{a_1/w_1, \ldots, a_k/u_k\}$, we define $|\sigma| = k$.

Each derivation step of SCOWS has to carry the information needed to compute the stochastic delay associated to the executed action. If a SCOWS service is able to perform a transition labelled by α, executing an action whose associated rate is ρ and reducing to s', we write $s \xrightarrow{\alpha}_{\rho\ q} s'$. Set $q = \{d_1, \ldots, d_n\}$ is a subset of identifiers in s, which is used in the decoration system introduced by the operational semantics. The rate ρ is used to compute the transition rate, and its form depends on α: if $\alpha = p.o \cdot \sigma \cdot \widetilde{n} \cdot \widetilde{u}$, then $\rho = [\gamma, \delta]$, where γ is the rate of the invocation and δ is the rate of the request activity involved in the communication. If α has one of the other forms, then ρ is the rate associated to the single activity whose execution generated the transition. Set q will be omitted when empty.

We now comment on the approach adopted for defining the operational semantics of SCOWS and on the difference with respect to the one presented for COWS. The most visible difference is the lack of a rule (*struct*) for SCOWS. In a stochastic setting, such a rule would introduce the possibility of deriving infinite transitions (labelled in a different way), from a term, thus breaking the subsequent quantitative reasoning on the obtained transition system. For this reason, we use the mechanism of scope opening and closing for names and variables, along with a decoration system for agent instantiations.

We require that in SCOWS terms there must be non-homonymy between bound and free identifiers, and also that bound identifiers must be all distinct.

These two conditions are necessary for the correct functioning of the open/close mechanism used to modify the scope of a binder as consequence of the execution of a communication. These conditions are also necessary for the correct behaviour of instantiated agent identifiers. For this reason, we adopt the mechanism of *decoration* in order to deal with the instantiation of agent identifiers, whose body can contain bound identifiers. Note that, for each agent definition $S(d'_1, \ldots, d'_j) = s$, we assume that service s respects the non-homonymy condition. The decoration mechanism allows us to enforce the non-homonymy condition when dealing, for example, with agents defined using recursion. We denote with $d^{(j)}$ the fact that identifier d is associated with decoration $j \in \mathbb{N}$. If, in a service s, d appears associated with two different decorations j_1 and j_2, then we consider $d^{(j_1)}$ and $d^{(j_2)}$ to be different.

The decoration mechanism is only a syntactic convenience, since $d^{(j)}$ can be read as identifier $d\,j$, where j is integrated with the identifier itself. For this reason, the decoration mechanism can be thought as a background machinery; when not needed, we will avoid to make it explicit, in order to keep the notation as clean as possible. For instance, when dealing with rule (s_id), the decoration mechanism plays a central role through the application of function $fdec(.,.,.,.,.)$, which is defined in Table 4.2. This function creates a list of decorated identifiers which will substitute bound identifiers in the body of the instantiated agent. The non-homonymy condition is maintained, since the fresh decorations are created taking into account the set q, which contains the identifiers used in the whole service under consideration, the list $\widetilde{b}$ of bound identifiers defined in the body of the agent and the lists of formal/actual parameters $d_1, \ldots, d_n$ and $d'_1, \ldots, d'_n$ for the agent under consideration.

When an identifier u is opened by the application of one of the semantic rules, we record this fact by putting its occurrences in parentheses in the transition label α. This procedure is formalized by function $open(.,.)$, presented in Table 4.3.

$$
\begin{array}{lcl}
fdec([\,], q, \widetilde{d}, \widetilde{d'}) & = & [\,] \\
fdec(d :: \widetilde{b'}, q, \widetilde{d}, \widetilde{d'}) & = & d^{(j)} :: fdec(\widetilde{b'}, q, \widetilde{d}, \widetilde{d'}) \\
 & & \text{where } j = \max\left(\left\{k \mid d^{(k)} \in (q \cup \widetilde{b'} \cup \widetilde{d} \cup \widetilde{d'})\right\} \cup \{0\}\right) + 1
\end{array}
$$

Table 4.2: Definition of function $fdec(.,.,.,.,.)$

$$
open(n, \alpha) = \begin{cases}
p.o\,?\,\widetilde{w}\{(n)/n\} & \text{if } \alpha = p.o\,?\,\widetilde{w} \\
p.o\,!\,\widetilde{a}\{(n)/n\} & \text{if } \alpha = p.o\,!\,\widetilde{a} \\
p.o \cdot \sigma\{(n)/x\} \cdot \widetilde{n} \cdot \widetilde{u} & \text{if } \alpha = p.o \cdot \sigma \cdot \widetilde{n} \cdot \widetilde{u} \\
\alpha & \text{otherwise}
\end{cases}
$$

Table 4.3: Definition of function $open(.,.)$

We report the remarkable differences in the operational semantics rules of SCOWS with respect to the ones introduced for COWS. The whole set of rules for SCOWS can be found in Table 4.6 and Table 4.7. Given that we cannot have a homologous of rule (*struct*) for SCOWS, the rule system uses the mechanism of scope opening and closing of names and variables. This mechanism is formalized by rules (*o_req*), (*o_inv*), (*o_tau*) and (*del_c*).

Rule (*o_req*) opens the scope of a variable x which appears in the list of parameters $\widetilde{w}$ of a request action.

$$
\frac{s \xrightarrow[\delta]{p.o\,?\,\widetilde{w}}_{q \cup \{x\}} s' \qquad x \in \widetilde{w}}{[\,x\,]s \xrightarrow[\delta]{\text{open}(x\,,\,p.o\,?\,\widetilde{w})}_{q} s'} \; (o\_req)
$$

Rule (*o_inv*) opens the scope of a name n appearing in the list of parameters $\widetilde{a}$ of an invoke action.

$$
\frac{s \xrightarrow[\gamma]{p.o\,!\,\widetilde{a}}_{q \cup \{n\}} s' \qquad n \in \widetilde{a} \qquad n \notin \{p, o\}}{[\,n\,]s \xrightarrow[\gamma]{\text{open}(n\,,\,p.o\,!\,\widetilde{a})}_{q} s'} \; (o\_inv)
$$

When considering a transition labelled by $\alpha = p.o \cdot \sigma \cdot \widetilde{n} \cdot \widetilde{w}$ for a service $[\,n\,]s$, rule (*o_tau*) opens the scope of name n appearing in the cosupport of σ.

$$\frac{s \xrightarrow[\rho]{p.o\cdot\sigma\cdot\widetilde{n}\cdot\widetilde{w}} {}_{q\cup\{n\}}\ s' \qquad n \in \mathrm{csp}(\sigma)}{[\,n\,]s \xrightarrow[\rho]{\mathrm{open}(n\,,\,p.o\cdot\sigma\cdot\widetilde{n}\cdot\widetilde{w})} {}_{q}\ s'}\ (o\_tau)$$

When considering a transition labelled by $\alpha = p.o \cdot \{(n)/x\} \uplus \sigma \cdot \widetilde{n} \cdot \widetilde{w}$ for a service $[\,x\,]s$, rule (del_c) performs the delayed substitution $\{n/x\}$ on the residual service s', while removing the delimiter for x and substituting it with the delimiter for n.

$$\frac{s \xrightarrow[\rho]{p.o\cdot\{(n)/x\}\uplus\sigma\cdot\widetilde{n}\cdot\widetilde{w}} {}_{q\cup\{x\}}\ s'}{[\,x\,]s \xrightarrow[\rho]{p.o\cdot\sigma\cdot\widetilde{n}\cdot\widetilde{w}} {}_{q}\ [\,n\,]s'\{n/x\}}\ (del\_c)$$

Agent instantiation is treated explicitly by rule (s_id), which is based on the decoration machinery described before: when instantiating the service body s, $\widetilde{f}$ is the list containing fresh identifiers that can be used in place of the ones appearing bound in s, in order to avoid name clashes with the identifiers already present in the whole service. These are recorded in the set q.

$$\frac{s\{d'_1/d_1,\ldots,d'_j/d_j\}\{f_1/b_1,\ldots,f_k/b_k\} \xrightarrow[\rho]{\alpha} {}_{q}\ s' \quad S(d_1,\ldots,d_j) = s \quad \widetilde{b} = bid(s) \quad \widetilde{f} = fdec(\widetilde{b}, q, \widetilde{d}, \widetilde{d'})}{S(d'_1,\ldots,d'_j) \xrightarrow[\rho]{\alpha} {}_{q}\ s'}\ (s\_id)$$

Other differences regard the definition of the matching function $\mathcal{M}(.\,,.\,)$, whose definition is revised in Table 4.5 to treat the cases in which names and variables have been opened by one of the rules (o_req) and (o_inv). In the definition, operator @ represents the concatenation of two lists.

If we have $\mathcal{M}(.\,,.\,) = (\sigma, n, j, L, \sigma')$, then σ represents the list of delayed substitutions, σ' is the list of substitutions to be applied to the residual of the receiving service, named s'_2 in rule (com), L is the list of identifiers for which a binder with scope $s'_1 \mid s'_2$ must be created, j is the total number of substitutions induced by matching $\widetilde{n}$ with $\widetilde{u}$, i.e. $j = |\sigma| + |\sigma'|$.

$$\text{tau\_of}(s'_1, s'_2, L, \sigma') = \begin{cases} s'_1 \mid s'_2\sigma' & \text{if } L = [\,] \\ [\,n_1, \ldots, n_h\,]\left(s'_1 \mid s'_2\sigma'\right) & \text{if } L = [n_1, \ldots, n_h] \end{cases}$$

Table 4.4: Definition of function tau_of(. , . , . , .)

$$\frac{s_1 \xrightarrow[\gamma]{p.o\,!\,\widetilde{a}} q \cup id(s_2)\ s'_1 \quad s_2 \xrightarrow[\delta]{p.o\,?\,\widetilde{w}} q \cup id(s_1) \cup id(s'_1) \cup \{\widetilde{n}\}\ s'_2 \quad \mathcal{M}(\widetilde{a}, \widetilde{w}) = (\sigma, \widetilde{n}, j, L, \sigma') \quad \neg(s_1 \mid s_2)\downarrow_{p.o}^{\widetilde{n}\cdot j}}{s_1 \mid s_2 \xrightarrow[[\gamma,\delta]]{p.o\cdot\sigma\cdot\widetilde{n}\cdot\widetilde{w}} q\ \text{tau\_of}(s'_1, s'_2, L, \sigma')}\ (com)$$

The conclusion of rule (*com*) is based on the definition of tau_of(. , . , . , .). The purpose of this function, defined in Table 4.4, is to check if the name list L, computed by function $\mathcal{M}(\widetilde{a}, \widetilde{w})$, is not empty, in which case for each element of L a binder upstream of the parallel composition $s'_1 \mid s'_2$ is introduced. Names in L are those opened names for which a substitution with an opened variable has been induced by the matching mechanism.

$$\begin{array}{rcl}
\mathcal{M}(\widetilde{w}, \widetilde{a}) \;=\; \texttt{match}\ \widetilde{w}\,,\,\widetilde{a}\ \texttt{with} & & \\
n\,,\,n & = & (\varepsilon, n, 0, [\,], \varepsilon) \\
x\,,\,n & = & (\{n/x\}, n, 1, [\,], \varepsilon) \\
x\,,\,(n) & = & (\{(n)/x\}, n, 1, [\,], \varepsilon) \\
(x)\,,\,n & = & (\varepsilon, n, 1, [\,], \{n/x\}) \\
(x)\,,\,(n) & = & (\varepsilon, n, 1, [n]\,, \{n/x\}) \\
w\widetilde{w}_1\,,\,a\widetilde{a}_1 & = & (\sigma \uplus \sigma_1, n\widetilde{n}_1, j + j_1, L \mathbin{@} L_1, \sigma' \uplus \sigma'_1) \\
 & & \text{if } \mathcal{M}(w, a) = (\sigma, n, j, L, \sigma') \\
 & & \text{and } \mathcal{M}(\widetilde{w}_1, \widetilde{a}_1) = (\sigma_1, n_1, j_1, L_1, \sigma'_1) \\
\mathcal{M}(w, n) \uplus \mathcal{M}(\widetilde{w}_1, \widetilde{n}_1) & & \\
\textbf{default} & = & \bot \\
\texttt{end_match} & &
\end{array}$$

Table 4.5: SCOWS: definition of function $\mathcal{M}(. , .)$

$$\frac{-}{(\mathbf{kill}(k), \lambda) \xrightarrow[\lambda]{\dagger k}_q \mathbf{0}}\ (kill) \qquad \frac{-}{(p.o\,?\widetilde{u}.\ s, \delta) \xrightarrow[\delta]{p.o\,?\widetilde{u}}_q s}\ (req) \qquad \frac{s \xrightarrow[\rho]{\alpha}_q s'}{\{\!|s|\!\} \xrightarrow[\rho]{\alpha}_q \{\!|s'|\!\}}\ (prot)$$

$$\frac{-}{(p.o\,!\widetilde{n}, \gamma) \xrightarrow[\gamma]{p.o\,!\widetilde{n}}_q \mathbf{0}}\ (inv) \qquad \frac{g_1 \xrightarrow[\rho]{\alpha}_{q \cup id(g_2)} s}{g_1 + g_2 \xrightarrow[\rho]{\alpha}_q s}\ (choice) \qquad \frac{s \xrightarrow[\rho]{p.o \cdot \{n/x\} \uplus \sigma \cdot \widetilde{n} \cdot \widetilde{u}}_{q \cup \{p,o,\widetilde{n},x\}} s'}{[\,x\,]s \xrightarrow[\rho]{p.o \cdot \sigma \cdot \widetilde{n} \cdot \widetilde{u}}_q s'\{n/x\}}\ (del\_sub)$$

$$\frac{s \xrightarrow[\rho]{\dagger k}_{q \cup \{k\}} s'}{[\,k\,]s \xrightarrow[\rho]{\dagger}_q [\,k\,]s'}\ (del\_k) \qquad \frac{s \xrightarrow[\rho]{\alpha}_{q \cup \{d\}} s' \qquad d \notin \mathrm{d}(\alpha) \qquad s \downarrow_d \Rightarrow (\alpha = \dagger \text{ or } \alpha = \dagger k)}{[\,d\,]s \xrightarrow[\rho]{\alpha}_q [\,d\,]s'}\ (del\_p)$$

$$\frac{s_1 \xrightarrow[\gamma]{p.o\,!\widetilde{a}}_{q \cup id(s_2)} s_1' \qquad s_2 \xrightarrow[\delta]{p.o\,?\widetilde{w}}_{q \cup id(s_1) \cup id(s_1') \cup \{\widetilde{n}\}} s_2' \qquad \mathcal{M}(\widetilde{a}, \widetilde{w}) = (\sigma, \widetilde{n}, j, L, \sigma') \qquad \neg(s_1 \mid s_2) \downarrow_{p.o}^{\widetilde{n} \cdot j}}{s_1 \mid s_2 \xrightarrow[[\gamma,\delta]]{p.o \cdot \sigma \cdot \widetilde{n} \cdot \widetilde{w}}_q \mathrm{tau\_of}(s_1', s_2', L, \sigma')}\ (com)$$

Table 4.6: Operational Semantics of SCOWS, part 1

4.3 Transition Rates Computation

We define the computation of transition rates for pseudo-τ transitions when considering a SCOWS service s, namely $s \xrightarrow[\rho]{\dagger} s'$ and $s \xrightarrow[\rho]{p.o \cdot \varepsilon \cdot \widetilde{n} \cdot \widetilde{w}} s'$. Following the approach presented in [15], we make use of *apparent rates*. Informally, the apparent rate of an action of type t is determined by the rate at which an external observer witnesses a transition of type t, without being able to distinguish between actions of the same type, i.e. the apparent rate of an action is the sum of the rates of all executable actions that, according to an external observer, are not different from the considered one. For this reason, the apparent rate depends not only on the composed rate ρ associated to a transition, but also on all the actions that the considered SCOWS service s can perform. We now formally define the computation of pseudo-τ transition rates.

When we consider transitions of the form $s \xrightarrow[\rho]{\dagger} s'$, the apparent rate is the sum of the rates of all the kill activities that can be executed in s, which are all

$$\frac{s_1 \xrightarrow[\rho]{p.o\cdot\sigma\cdot\widetilde{n}\cdot\widetilde{w}}_{q\cup id(s_2)} s_1' \qquad \neg s_2 \downarrow^{\widetilde{n}\cdot j}_{p.o} \qquad \mathcal{M}(\widetilde{w},\widetilde{n}) = (\sigma,\widetilde{n},j,L,\sigma')}{s_1 \mid s_2 \xrightarrow[\rho]{p.o\cdot\sigma\cdot\widetilde{n}\cdot\widetilde{w}}_{q} s_1' \mid s_2}\ (par\_c)$$

$$\frac{s_1 \xrightarrow[\rho]{\dagger k}_{q\cup id(s_2)} s_1'}{s_1 \mid s_2 \xrightarrow[\rho]{\dagger k}_{q} s_1' \mid \mathrm{halt}(s_2)}\ (par\_k) \qquad \frac{s_1 \xrightarrow[\rho]{\alpha}_{q\cup id(s_2)} s_1' \qquad \alpha \neq p.o\cdot\sigma\cdot\widetilde{n}\cdot\widetilde{w} \qquad \alpha \neq \dagger k}{s_1 \mid s_2 \xrightarrow[\rho]{\alpha}_{q} s_1' \mid s_2}\ (par\_p)$$

$$\frac{s\{d_1'/d_1,\ldots,d_j'/d_j\}\{f_1/b_1,\ldots,f_k/b_k\} \xrightarrow[\rho]{\alpha}_q s' \quad S(d_1,\ldots,d_j) = s \quad \widetilde{b} = bid(s) \quad \widetilde{f} = fdec(\widetilde{b},q,\widetilde{d},\widetilde{d'})}{S(d_1',\ldots,d_j') \xrightarrow[\rho]{\alpha}_q s'}\ (s\_id)$$

$$\frac{s \xrightarrow[\rho]{p.o\cdot\{(n)/x\}\uplus\sigma\cdot\widetilde{n}\cdot\widetilde{w}}_{q\cup\{x\}} s'}{[\,x\,]s \xrightarrow[\rho]{p.o\cdot\sigma\cdot\widetilde{n}\cdot\widetilde{w}}_{q} [\,n\,]s'\{n/x\}}\ (del\_c) \qquad \frac{s \xrightarrow[\delta]{p.o\,?\,\widetilde{w}}_{q\cup\{x\}} s' \qquad x \in \widetilde{w}}{[\,x\,]s \xrightarrow[\delta]{\mathrm{open}(x\,,\,p.o\,?\,\widetilde{w})}_{q} s'}\ (o\_req)$$

$$\frac{s \xrightarrow[\gamma]{p.o\,!\,\widetilde{a}}_{q\cup\{n\}} s' \qquad n \in \widetilde{a} \qquad n \notin \{p,o\}}{[\,n\,]s \xrightarrow[\gamma]{\mathrm{open}(n\,,\,p.o\,!\,\widetilde{a})}_{q} s'}\ (o\_inv) \qquad \frac{s \xrightarrow[\rho]{p.o\cdot\sigma\cdot\widetilde{n}\cdot\widetilde{w}}_{q\cup\{n\}} s' \qquad n \in \mathrm{csp}(\sigma)}{[\,n\,]s \xrightarrow[\rho]{\mathrm{open}(n\,,\,p.o\cdot\sigma\cdot\widetilde{n}\cdot\widetilde{w})}_{q} s'}\ (o\_tau)$$

Table 4.7: Operational Semantics of SCOWS, part 2

the unguarded kill activities on a closed kill label. Formally, this apparent rate is computed using function $ark(s, s)$, defined in Table 4.8.

$$\begin{aligned}
ark(s_1 \mid s_2) &= ark(s_1) + ark(s_2) \\
ark((\mathbf{kill}(k), \lambda), s) &= \begin{cases} \lambda & \text{if } k \in bkl(s) \\ 0 & \text{otherwise} \end{cases} \\
ark(\{\!|s_1|\!\}, s) = ark([\,d\,]s_1, s) &= ark(s_1, s) \\
ark((p.o\,?\,\widetilde{u}, \delta).s_1, s) = ark(g_1 + g_2, s) &= 0 \\
ark((p.o\,!\,\widetilde{u}, \gamma), s) = ark(\mathbf{0}, s) &= 0
\end{aligned}$$

Table 4.8: Definition of function $ark(.,.)$

In a service s, an unguarded kill action with rate λ is executed with a probability computed as the ratio between λ and the sum of the rates of all executable unguarded kill actions in s. Given a transition $s \xrightarrow[\rho=\lambda]{\dagger} s'$, the rate of the transition is then computed as in Equation (4.1). As can be seen, the approach of computing transition rates using apparent rates results in a simplification that leaves as result the rate of the executed kill action.

$$rate\left(s \xrightarrow[\rho=\lambda]{\dagger} s'\right) = \frac{\lambda}{ark(s, s)} ark(s, s) = \lambda \tag{4.1}$$

In the case of a transition $s \xrightarrow[\rho=[\gamma,\delta]]{p.o \cdot \varepsilon \cdot \widetilde{n} \cdot \widetilde{w}} s'$, which is triggered by the execution of a communication, an unguarded invocation and an unguarded request activity perform a synchronization over a common endpoint. Therefore, to compute the transition rate we have to combine their basic rates γ, δ. Applying the same methodology presented in [15] and already used for kill actions, we first have to compute the probability of having a particular invoke action and a particular request action involved in a synchronization. For the sake of the clarity of the explanation, in the following informal descriptions we will refer explicitly to the participants of the communication. This will not be the case for the formalization of the computation, since we are dealing with pseudo-τ transitions, through which we cannot explicitly identify the actual participants in the

communication.

Given the best matching policy implemented in COWS and SCOWS, a synchronization is driven by the invoke activity involved in it. The request activities that can take part in a communication depend on the particular invoke activity that we consider. For this reason, given a service $s = s_1 \mid \left(g' + \sum_j g_j\right) \mid s_3$, where $s_1 = (p.o\,!\,\widetilde{n}, \gamma)$ and $g' = (p.o\,?\,\widetilde{u}.s_2, \delta)$, the probability of having s_1 and g' involved in a communication, denoted by $P_1(\gamma \blacktriangleright \delta, p.o, \widetilde{n}, s)$, involves the computation of a conditional probability, as can be seen in Equation 4.2.

$$P_1(\gamma \blacktriangleright \delta, p.o, \widetilde{n}, s) = P_2(\gamma, p.o, s) \cdot P_3(\delta \text{ given } \gamma, p.o, \widetilde{n}, s) \tag{4.2}$$

Given a service s and an unguarded invoke activity $s_1 = (p.o\,!\,\widetilde{n}, \gamma)$ able to take part in a communication, the probability of having s_1 involved in a communication is defined as the ratio between the basic rate of s_1 and the sum of all the invoke actions in s over endpoint $p.o$ which can take part in a communication. We denote this as $P_2(\gamma, p.o, s) = \frac{\gamma}{inv(p.o,s,s)}$ where $inv(.\,,.\,,.\,)$ is defined in Table 4.9. Notably, $inv(p.o, s, s)$ is also the apparent rate of an invocation which can take part in a communication over endpoint $p.o$ when considering a service s.

$$\begin{array}{rcl}
inv(p.o, (\mathbf{kill}(k), \lambda)\,, s) & = & 0 \\
inv(p.o, \mathbf{0}, s) = inv(p.o, g_1 + g_2, s) & = & 0 \\
inv(p.o, (p.o\,?\,\widetilde{u}, \delta)\,.s', s) & = & 0 \\
inv(p.o, s_1 \mid s_2, s) & = & inv(p.o, s_1, s) + inv(p.o, s_2, s) \\
inv(p.o, \{\!|s_1|\!\}, s) = inv(p.o, [\,d\,]s_1, s) & = & inv(p.o, s_1, s) \\
inv(p.o, (p.o\,!\,\widetilde{n}, \gamma)\,, s) & = & \begin{cases} \gamma & \text{if } s = (p.o\,?\,\widetilde{u}, \delta)\,.s' + \sum_j g_j \mid s_2 \\ & \text{and } \mathcal{M}(\widetilde{n}, \widetilde{u}) = (\sigma, \widetilde{n}, j, L, \sigma') \\ & \text{and } \neg s \downarrow_{p.o}^{\widetilde{n} \cdot j} \\ 0 & \text{otherwise} \end{cases}
\end{array}$$

Table 4.9: Definition of function $inv(.\,,.\,,.\,)$

The conditional probability $P_3(\delta \text{ given } \gamma, p.o, \widetilde{n}, s)$ can be computed consid-

ering the basic rate δ and the sum of all the request activities in s that compete to communicate on endpoint $p.o$ matching with $\widetilde{n}$. This can be formalized as $P_3(\delta \text{ given } \gamma, p.o, \widetilde{n}, s) = \frac{\delta}{req(p.o,\widetilde{n},s,s)}$, where $req(.,.,.,.,.)$ is defined in Table 4.10. Function $req(.,.,.,.,.)$ is also used to compute the apparent rate of a request activity. This is the main difference in the stochastic rate computation with respect to the approach adopted in [31], where a more involved computation of the apparent rate of a request activity was presented.

$$
\begin{array}{rcl}
req(p.o, \widetilde{n}, (\mathbf{kill}(k), \lambda), s) & = & 0 \\
req(p.o, \widetilde{n}, \mathbf{0}, s) & = & 0 \\
req(p.o, \widetilde{n}, (p.o\,!\,\widetilde{n}, \gamma), s) & = & 0 \\
req(p.o, \widetilde{n}, g_1 + g_2, s) & = & req(p.o, \widetilde{n}, g_1, s) + req(p.o, \widetilde{n}, g_2, s) \\
req(p.o, \widetilde{n}, s_1 \mid s_2, s) & = & req(p.o, \widetilde{n}, s_1, s) + req(p.o, \widetilde{n}, s_2, s) \\
req(p.o, \widetilde{n}, \{\!|s_1|\!\}, s) = req(p.o, \widetilde{n}, [\,d\,]s_1, s) & = & req(p.o, \widetilde{n}, s_1, s) \\
req(p.o, \widetilde{n}, (p.o\,?\,\widetilde{u}, \delta).s', s) & = & \begin{cases} \delta & \text{if } \mathcal{M}(\widetilde{n}, \widetilde{u}) = (\sigma, \widetilde{n}, j, L, \sigma') \\ & \text{and } \neg s \downarrow_{p.o}^{\widetilde{n} \cdot j} \\ 0 & \text{otherwise} \end{cases}
\end{array}
$$

Table 4.10: Definition of function $req(.,.,.,.,.)$

Using the same approach presented in [15], we model the apparent rate of a communication using the minimum of the apparent rates of the participants. We can now define the computation of the rate of a transition step triggered by a communication in $s = s_1 \mid \left(g' + \sum_j g_j\right) \mid s_3$ with $s_1 = (p.o\,!\,\widetilde{n}, \gamma)$ and $g' = (p.o\,?\,\widetilde{u}, \delta).s_2$. Equation 4.3 defines the function $R_1(\gamma \blacktriangleright \delta, p.o, \widetilde{n}, s)$, which formalizes this computation.

$$
R_1(\gamma \blacktriangleright \delta, p.o, \widetilde{n}, s) = P_1(\gamma \blacktriangleright \delta, p.o, \widetilde{n}, s) \cdot min\left(inv(p.o, s, s), req(p.o, \widetilde{n}, s, s)\right) \tag{4.3}
$$

Finally, we can formalize the computation of a transition step triggered by the execution of a pseudo-τ communication. The result is presented in Equation 4.4.

$$rate\left(s \xrightarrow[[\gamma,\delta]]{p.o \cdot \varepsilon \cdot \widetilde{n} \cdot \widetilde{w}} s'\right) = R_1\left(\gamma \blacktriangleright \delta, p.o, \widetilde{n}, s\right) \tag{4.4}$$

4.3.1 Transition Rate Computation Example

To better understand how the transition rate computation works, we present now an example. Let s be the service defined as

$$\begin{aligned} s = [\,n, m, x\,](\; & (p.o\,!\,n, 3) \\ & \mid (p.o\,?\,n, 5) \\ & \mid (p.o\,?\,n, 7) \\ & \mid (p.o\,!\,m, 11) \\ & \mid (p.o\,?\,x, 13) \;\;) \end{aligned}$$

Service s can perform three communications, the first involving invoke action $(p.o\,!\,n, 3)$ and request action $(p.o\,?\,n, 5)$, the second involving invoke action $(p.o\,!\,n, 3)$ and request action $(p.o\,?\,n, 7)$ and the third involving invoke action $(p.o\,!\,m, 11)$ and request action $(p.o\,?\,x, 13)$. We focus on the first one and compute its rate.

We start by computing the apparent rate of an invoke action over $p.o$ in s, defined as $inv(p.o, s, s)$. Since both invoke actions in s can take part in a communication over $p.o$, the apparent rate is equal to the sum of their rates:

$$inv(p.o, s, s) = 3 + 11 = 14$$

We proceed by computing the apparent rate of a request activity in s, over endpoint $p.o$ and with respect to n, which is the list of parameters of the considered invoke action. Computing $req(p.o, n, s, s)$ we obtain:

$$req(p.o, n, s, s) = 5 + 7 = 12$$

The other request activity in s does not take part in the computation, since is not a best matching request action with respect to $(p.o\,!\,n, 3)$.

We now compute $P_2(3, p.o, s)$, which is defined as

$$P_2(3, p.o, s) = \frac{3}{inv(p.o, s, s)} = \frac{3}{14}$$

The next step is represented by the computation of $P_3(5 \text{ given } 3, p.o, n, s)$. We get

$$P_3(5 \text{ given } 3, p.o, n, s) = \frac{5}{req(p.o, n, s, s)} = \frac{5}{12}$$

We can now compute $P_1(3 \blacktriangleright 5, p.o, n, s)$ using Equation 4.2. We obtain:

$$P_1(3 \blacktriangleright 5, p.o, n, s) = \frac{3}{14} \cdot \frac{5}{12} = \frac{15}{168}$$

We can then compute $R_1(3 \blacktriangleright 5, p.o, \widetilde{n}, s)$, according to Equation 4.3:

$$R_1(3 \blacktriangleright 5, p.o, n, s) = \frac{15}{168} \cdot min(12, 14) = \frac{15}{168} \cdot 12 = \frac{15}{14}$$

Finally, for Equation 4.4, we have that the rate of the considered transition is

$$rate\left(s \xrightarrow[[3,5]]{p.o \cdot \varepsilon \cdot n \cdot n} s'\right) = R_1(3 \blacktriangleright 5, p.o, n, s) = \frac{15}{14}$$

We have defined a stochastic extension of COWS and we have formalized how to associate exponential delays to transitions. We now present SCOWS_lts, a simulator able to derive the whole Transition Graph of a SCOWS term and produce the associated Continuous Time Markov Chain.

4.4 SCOWS_lts: a tool for the analysis of SCOWS models

SCOWS_lts is a tool which is able to derive the whole Labelled Transition System (LTS) of a SCOWS term, if the LTS can be represented using a finite number of states, and can build a Continuous Time Markov Chain (CTMC) model based on the derived LTS. The derived CTMC can then be imported in a model checker such as PRISM to perform a quantitative analysis of the model by checking CSL properties. In order to mitigate the effects of the State Space Explosion Problem, the tool implements a notion of structural congruence among SCOWS terms. We describe the implementation of the peculiarities of SCOWS_lts, along with other performance optimizations. The latest available version of the software, which is written in Java, is available at [1] packaged as a Jar executable file, along with a number of examples. We presented an earlier version of the tool in [9]. The version presented here, along with many bug fixes, shows a much more performant search criterion for structural congruent terms.

4.4.1 PRISM overview

PRISM [17] is an open-source model checker developed in Java. The main features of the tool include the support for different probabilistic models (e.g. Discrete and Continuous Time Markov Chains, Markov Decision Processes) and temporal logics (e.g. LTL, PCTL, CSL); moreover, PRISM implements various model checking engines, involving both symbolic and explicit representations of states. PRISM can be used using a graphical user interface, through which it is possible to plot the results of the model checking phase, or through a command line, which is more useful when dealing with an automated verification process.

These features, along with the fact that it is actively developed, make PRISM one of the reference model checkers for quantitative verification in different

scenarios, ranging from verification of the energy consumption in networks to systems biology.

4.4.2 SCOWS identifiers in SCOWS_lts

In SCOWS_lts models, identifiers consist of strings composed using numbers, letters and the underscore symbol "_". Each identifier must begin with a letter, and different classes of identifiers are recognized following these rules: kill label identifiers must start with the letter *k*; names are recognized by the presence of a trailing hash symbol #, agent identifiers must start with an uppercase letter. Identifiers not following these conventions are treated as variables, with the exception of reserved keywords *true*, $false$, $fixed$. Each SCOWS_lts model is composed of four main sections, divided by the $ symbol. The first section contains the definitions of agent identifiers in the form $Id_i(param_j) = S_i;$. The second section specifies the definition of the initial service. The third section contains PRISM variable definitions that will be inserted in the resulting PRISM model; the fourth section contains the list of SCOWS action abstractions, which link the execution of particular labelled SCOWS actions to PRISM transitions.

4.4.3 SCOWS_lts: the main loop

The basic structure of SCOWS_lts is an adaptation of a Breadth First Search algorithm for visiting a graph. In the specific case, visiting a node *n* means also adding it to the Transition Graph and computing all its derivatives, determined applying to the SCOWS term associated to *n* the operational semantics defined in Section 4.2. For the sake of readability, the shown algorithm does not contain all the implementation details. In the presentation, we retain the Java notation of *generics*, using angled parentheses <> to specify the type of elements in a collection.

The representation of a LTS is given by an object of type Graph, which is

defined as State $\mapsto$ List < Transition >, where type State is a representation of a SCOWS service and Transition is a type defined as $(State \times Label) \mapsto State$.

The pseudocode for the main loop of SCOWS_lts is presented in Pseudocode 4.1. The tool takes as input a file containing the specification of a SCOWS service, represented by State s. Another parameter, which we left implicitly specified, is represented by the global execution environment e, containing, among other structures, the definitions of SCOWS agents that appear in s. The result given by SCOWS_lts is a CTMC c; the current version of SCOWS_lts is able to specify the CTMC according to the format used by the PRISM model checker [17], which is available at `http://www.prismmodelchecker.org` and also according to the format used by the GraphViz tool available at `http://www.graphviz.org`, which is useful if a graphical representation of the Transition System is needed.

Queue q (line 3 in Pseudocode 4.1) is a list of states which implements the FIFO policy: states are added at the end of the queue and are removed from its head. While q is not empty, the main loop of the algorithm is executed: the first element in the queue is removed (line 6) and its transitions are computed (line 7). This operation is performed according to the operational semantics defined in Section 4.2, using the agent definitions contained in the execution environment e. In SCOWS_lts, each type of SCOWS service is implemented as a subclass of State, so that the Object-Oriented paradigm of *method overloading* can take place. The obtained transitions, stored in list tr, have to be elaborated to be added to graph g (line 8): a transition tr_i in tr is defined as $h \times \alpha_i \mapsto s_i$. If s_i is structurally congruent to a state s_j already in g, we have to:

1. rearrange tr_i, substituting s_i with s_j and possibly modifying α_i
2. flag s_i as a discarded state, avoiding to add it to queue q

The second action is reflected in the presence of list *toBeAdded*: this list contains a subset of the states generated when computing the transitions for

state s, namely it contains the states for which a structurally congruent state inside g could not be found. This list is added at the end of queue q (line 9).

Pseudocode 4.1 Main loop for SCOWS_lts

Input: State s, global Env e

Output: Ctmc c

```
1:  Graph g = new Graph();
2:  g.setFirstState(s);
3:  Queue < State > q = new Queue();
4:  q.add(s);
5:  while q.notEmpty() do
6:      State h = q.removeHead();
7:      List < Transition > tr = h.getTransitions();
8:      List < State > toBeAdded = g.addTransitions(tr);
9:      q.addAll(toBeAdded);
10: end while
11: Ctmc c = g.getCtmc();
```

A basic implementation of method *addTransitions*(), used in line 8, can be seen in Pseudocode 4.2. Each residual service is compared to the states already in the LTS using function *congruence*(. , . , . , . , .), described in the following section and defined in Pseudocode 4.3.

An algorithm such as the one presented in Pseudocode 4.1 terminates if the LTS originated from the SCOWS service associated to s can be represented using a finite number of states. Implementing structural congruence, we can also consider services with a cyclic, i.e. infinitely looping, behaviour. A weaker result could be obtained using only structural equivalence but, being this a stricter relation, services differing only for syntactic reasons would be recognized as different.

Pseudocode 4.2 Definition of function *addTransitions*( )

Input: List < Transition > *transitions*

Output: List < State > *result*

```
1:  List < State > result = new List < State > ();
2:  for all Transition t ∈ transitions do
3:     for all State s ∈ getGraphStates() do
4:        Map < Id, Id > χ = new Map < Id, Id >()
5:        if !congruence(t.getDestination(), s, |, +, χ) then
6:           addTransition(t.getSource(), t.getLabel(), t.getDestination())
7:           result.add(t.getDestination());
8:        else
9:           addTransition(t.getSource(), t.getLabel(), s)
10:       end if
11:    end for
12: end for
13: return result;
```

4.4.4 Structural Congruence for SCOWS

When deriving the LTS of a SCOWS term, it is quite common to generate terms that differ only for syntactic reasons. For example, consider a service

$$s = [\, n\, x\,]\,((p.o\,!\,n, \gamma) \mid (p.o\,?\,x.s', \delta) \mid (p.o\,?\,x.s', \delta))$$

Two different communication actions can be triggered, reaching states

$$s_1 = [\, n\,]\,(\mathbf{0} \mid ((p.o\,?\,x.s', \delta)) \mid s')\,\{n/x\}$$

and

$$s_2 = [\, n\,]\,(\mathbf{0} \mid s' \mid ((p.o\,?\,x.s', \delta)))\,\{n/x\}$$

As can be seen, s_1 and s_2 differ only for the order in which subservices are arranged through parallel compositions. In order to mitigate the problem of State Space Explosion, we decided to implement a procedure to recognize when two SCOWS services can be considered structurally congruent. The check performed by this procedure is carried out when inserting new residuals in the LTS.

Using this procedure, each state in the LTS represents a congruence class among SCOWS services, rather than a single SCOWS service. The definition of structural congruence we implemented is the least congruence relation induced by the rules in Table 4.11. With respect to the definition given in Table 2.4, we removed the rule regarding agent identifiers, which are treated explicitly by the semantic rule (*s_id*). The description of the implementation of the procedure for checking whether two SCOWS services are congruent is given in Section 4.4.5.

$s_1 \equiv s_2$ if $s_1 =_\alpha s_2$

$[\,d\,]\mathbf{0} \equiv \mathbf{0}$ $\quad s_1 \mid [\,d\,]s_2 \equiv [\,d\,](s_1 \mid s_2)$ if $d \notin fid(s_1) \cup fkl(s_2)$

$[\,d_1\,][\,d_2\,]s \equiv [\,d_2\,][\,d_1\,]s$

$s_1 \mid (s_2 \mid s_3) \equiv (s_1 \mid s_2) \mid s_3$ $\quad s_1 \mid s_2 \equiv s_2 \mid s_1$ $\quad s \mid \mathbf{0} \equiv s$

$(g_1 + g_2) + g_3 \equiv g_1 + (g_2 + g_3)$ $\quad g_1 + g_2 \equiv g_2 + g_1$ $\quad (p.o\,?\widetilde{u}.\,s, \gamma) + \mathbf{0} \equiv (p.o\,?\widetilde{u}.\,s, \gamma)$

$\{\!|\mathbf{0}|\!\} \equiv \mathbf{0}$ $\quad \{\!|\{\!|s|\!\}|\!\} \equiv \{\!|s|\!\}$

Table 4.11: Structural Congruence rules for SCOWS

4.4.5 Structural Congruence: an implementation

The rules presented in 4.11 can be divided in three categories. The first category is represented by rules involving **0** services and redundant protection operators, which can be implemented using a recursive minimization function *min*(.) presented in Table 4.12, where $\star \in \{|, +\}$: when checking the congruence of two services s_1 and s_2, we actually consider their minimized versions. In the following, we assume that the minimization procedure has already been applied to all mentioned services.

$$
\begin{array}{rcl}
min(\mathbf{0}) & = & \mathbf{0} \\
min((\mathbf{kill}(k), \lambda)) & = & (\mathbf{kill}(k), \lambda) \\
min((p.o\,!\,\widetilde{u}, \gamma)) & = & (p.o\,!\,\widetilde{u}, \gamma) \\
min((p.o\,?\,\widetilde{u}.s', \delta)) & = & (p.o\,?\,\widetilde{u}.min(s'), \delta) \\
min(\{\!|s|\!\}) & = & \begin{cases} \min(s) & \text{if } s = \{\!|s'|\!\} \\ \{\!|min(s)|\!\} & \text{if } s \neq \{\!|s'|\!\} \text{ and } min(s) \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise} \end{cases} \\
min(s_1 \star s_2) & = & \begin{cases} min(s_1) \star min(s_2) & \text{if } min(s_1), min(s_2) \neq \mathbf{0} \\ min(s_1) & \text{if } min(s_2) = \mathbf{0} \text{ and } min(s_1) \neq \mathbf{0} \\ min(s_2) & \text{if } min(s_1) = \mathbf{0} \text{ and } min(s_2) \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise} \end{cases} \\
min([\,d\,]s) & = & \begin{cases} [\,d\,](min(s_1 \star s_2)) & \text{if } s = s_1 \star s_2 \\ & \text{and } d \in fid(s_1) \\ & \text{and } d \in fid(s_2) \\ min([\,d\,]s_1 \star s_2) & \text{if } s = s_1 \star s_2 \\ & \text{and } d \in fid(s_1) \\ & \text{and } d \notin fid(s_2) \\ min(s_1 \star [\,d\,]s_2) & \text{if } s = s_1 \star s_2 \\ & \text{and } d \notin fid(s_1) \\ & \text{and } d \in fid(s_2) \\ [\,d\,]min(s) & \text{if } s \neq s_1 \star s_2 \\ & \text{and } d \in fid(s) \\ min(s) & \text{otherwise} \end{cases}
\end{array}
$$

Table 4.12: Minimization function $min(.)$ for SCOWS services, $\star \in \{|, +\}$

The second category of rules for structural congruence is represented by the distributive and commutative properties of parallel composition and nondeterministic choice. The implementation of these rules produces a flattened view of a service, in which its syntactic structure is recursively broken into its basic subservices, which are then collected in a multiset (i.e. repetitions count). We will underline the use of multisets by adding a subscript m to operators: $\{e_i\}_m$ is the multiset composed by elements e_i (possibly with repetitions), $\cup_m$, $\cap_m$, $\setminus_m$ are respectively the multiset union, multiset intersection and multiset subtraction operators. Parallel composition and nondeterministic choice must be considered in separate instances of the flattening procedure, otherwise the procedure would recognize as congruent services $g_1 + g_2$ and $g_1 \mid g_2$. For this reason, a nondeterministic choice is treated as a basic subservice when flattening with respect to parallel composition, and vice versa. This will be clarified when describing Pseudocode 4.3 and Pseudocode 4.4.

The delimiter construct is the subject of the third category of rules for structural congruence; we consider it in the flattening procedure for service s, presented in Table 4.13: given $\star, \diamond \in \{|, +\}$ and $\diamond \neq \star$, flattening $s_1 \star s_2$ means obtaining recursively the basic subservices of both s_1 and s_2; flattening a service $[\,u\,]s_1$ means collecting the subservices obtained by a recursive call on s_1. In this way, we forget about the actual position of the delimiter in the service; all we need to know is whether an identifier is bound or free in s: since the non-homonymy condition holds, each name and variable, bound or free, is univocally identified. Given the particular operational semantics of kill actions, a service $[\,k\,]s$ is treated as a basic service. The same is true for protected services and for compositions $s_1 \diamond s_2$. These three cases will be treated recursively by the procedure presented in Pseudocode 4.4. Figure 4.1 presents a graphical representation of the flattening function applied to services $[\,n\,]((s_1 \mid s_2) \mid (s_3 \mid s_1))$ and $(s_1 \mid [\,n\,](s_2 \mid (s_1 \mid s_3))))$, in which we assume n is shared between services s_2 and s_3. The procedure produces two equal multisets.

The flattening function presented in Table 4.13, however, is still incomplete: we need to take into account also the fourth class of rules for structural congruence, which is represented by the rule for α-equivalence, according to which two SCOWS services which are equal modulo renaming of bound names/variables/kill labels have to be recognized as congruent.

$$
\begin{array}{lcl}
flt_{\star}(s_1 \star s_2) & = & \{t_i \text{ such that } t_i \in flt_{\star}(s_1) \cup_m flt_{\star}(s_2)\}_m \\
flt_{\star}([\,u\,]s_1) & = & \{t_i \text{ such that } t_i \in flt_{\star}(s_1)\}_m \\
flt_{\star}(s) & = & \{s\}_m \text{ if } s \neq s_1 \star s_2 \text{ and } s \neq [\,u\,]s_1
\end{array}
$$

Table 4.13: Flattening function $flt_{\star}(.\,)$, for $\star \in \{|, +\}$

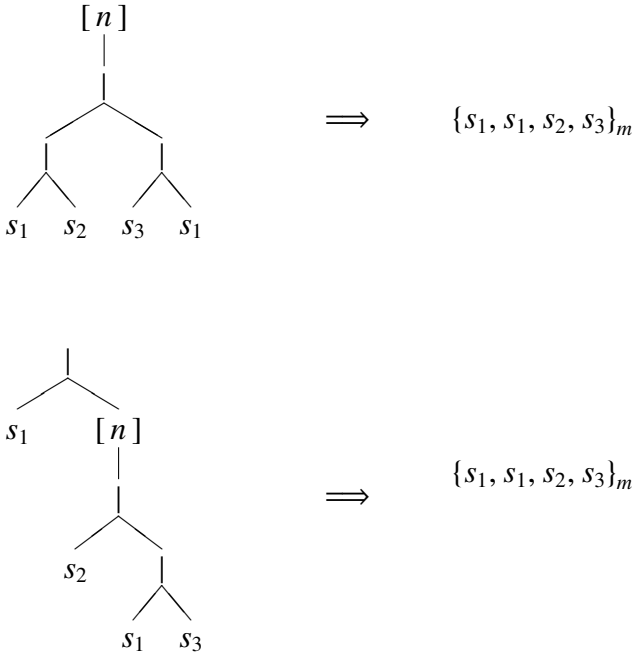


Figure 4.1: An example of the flattening function $flt_{|}(.\,)$, defined in Table 4.13, applied to services $[\,n\,]((s_1 \mid s_2) \mid (s_3 \mid s_1))$ and $(s_1 \mid [\,n\,](s_2 \mid (s_1 \mid s_3))))$

Our solution to the problem of deciding if two services can be obtained by renaming bound identifiers stems from the observation that, if two services s_1 and s_2 are α-equivalent, then there exists a bijective function $\chi : (\mathcal{N} \cup \mathcal{V} \cup$

$\mathcal{K}) \mapsto (\mathcal{N} \cup \mathcal{V} \cup \mathcal{K})$ mapping each bound identifier in s_1 to a bound identifier in s_2. We will write $dom(\chi) = \{d \text{ such that } (d \mapsto d') \in \chi\}$, $cod(\chi) = \{d' \text{ such that } (d \mapsto d') \in \chi\}$, $\chi(d) = d'$ if $(d \mapsto d') \in \chi$ and $\chi(d) = \bot$ if $d \notin dom(\chi)$.

Deciding whether s_1 and s_2 are α-equivalent is equivalent to deciding if such a function χ exists. In order to do so, while considering also structural rules for delimiters, we use an enriched flattening function $flt_{|_\alpha}(.)$, which maps each basic service identified by $flt_|(.)$ to the set of identifiers bound upstream in the considered service. Since $flt_|(.)$ produces a multiset, we have to keep track of the number of identical basic services produced by the flattening function, otherwise services $s_1 \mid s_1 \mid s_2$ and $s_1 \mid s_2 \mid s_2$ would be recognized as congruent.

We formalize the definition of $flt_{|_\alpha}(.)$ in Table 4.14: when applied to a service s, the function produces a mapping from the basic subservices in s, obtained by using function $flt_|(s)$, to a pair (n_i, b_i) composed of n_i, the number of instances of a particular subservice and b_i, the set of names, variables, and kill labels appearing in the subservice and bound upstream. We make use of auxiliary functions $instances(.,.)$, whose purpose is counting how many instances of an element appear in a multiset and $unique(.)$, whose purpose is producing a set containing all elements in the multiset passed as parameter.

$$
\begin{array}{rcl}
flt_{|_\alpha}(s) & = & \Big\{ s_i \mapsto (n_i, b_i) \text{ such that } s_i \in unique(flt_|(s)) \\
& & \qquad \text{and } n_i = instances(s_i, flt_|(s)) \\
& & \qquad \text{and } b_i = fid(s_i) \cap bid(s) \Big\} \\
instances(e, A_m) & = & \begin{cases} 1 + instances(e, A'_m) & \text{if } A_m = \{e\}_m \cup_m A'_m \\ 0 & \text{if } e \notin_m A_m \end{cases} \\
unique(A_m) & = & \begin{cases} \{e\} \cup unique(A'_m) & \text{if } A_m = \{e\}_m \cup_m A'_m \\ \emptyset & \text{if } A_m = \emptyset \end{cases}
\end{array}
$$

Table 4.14: Definition of function $flt_{|_\alpha}(.)$

Given $M_s = flt_{|_\alpha}(s)$ and $s_i \in unique(flt_|(s))$, we write $M_s(s_i) = (n_i, b_i)$ if $(s_i \mapsto (n_i, b_i)) \in M_s$ and $M_s(s_i) = \bot$ otherwise. M_s represents the normal-

ized view of service s. We now describe the algorithm implementing the given notion of structural congruence, which is based on the normalized view of services.

As said before, in order to decide if two given services s_1 and s_2 are congruent, we need to find a bijective function χ mapping each name/variable/kill label of service s_1 into one of s_2. The search is carried out by obtaining first the normalized views M_{s_1} and M_{s_2}; two ordered lists of subservices L_1 and L_2 are then built from $unique(flt_|(s_1))$ and $unique(flt_|(s_2))$, respectively. By matching corresponding elements $l_{1i} = L_1[i]$ and $l_{2i} = L_2[i]$, function χ can be built, possibly with the use of recursion, considering how identifiers are used in l_{1i} and l_{2i}. If a match is not possible or a conflict in the definition arises, i.e. χ is not a bijection, then a new permutation of the elements in L_1 is considered. If no permutation of the elements in L_1 induces the construction of a bijection χ, then s_1 and s_2 are not structurally congruent. If, on the other hand, there exists a permutation of the elements in L_1 such that a bijection χ is found, then s_1 and s_2 are structurally congruent.

The pseudocode for $congruence(.,.,.,.,.,.)$, the procedure implementing this algorithm, is presented in Pseudocode 4.3. Given two services s_1 and s_2, operators $\star$ and $\diamond$ and a partially built map χ, the algorithm first computes the flattened views M_1 and M_2 and the corresponding lists of basic subservicees L_1 and L_2. A permuter for list L_1 is then instantiated (line 5). The core of the algorithm is the *while* cycle at line 6: while there is another permutation P of list L_1 to be considered, the algorithm checks whether P and L_2 induce a bijection χ among bound identifiers. In order to do so, for each i-th subservice in P and L_2, the algorithm checks (line 12) the number of identical copies of $P[i]$ and $L_2[i]$ inside s_1 and s_2; moreover, the number of upstream bound identifiers is checked as well. Using function $compare(.,.,.,.,.,.)$, defined in Pseudocode 4.4, the algorithm then checks (lines 15-16) whether $P[i]$ and $L_2[i]$ can be compared and if the mapping induced by their use of identifiers is compatible with the tem-

porary mapping χ_l. Function $compare(.,.,.,.,.,.)$ also triggers the recursive calls to function $congruence(.,.,.,.,.,.)$; the recursive cases are represented by $\diamond$-compositions of services $s_1 \diamond s_2$, protected services $\{\!|s|\!\}$ and delimiters for kill labels $[\,k\,]s$.

Pseudocode 4.1 considers two services s_1 and s_2 as congruent, representing them with the same state in the LTS, if $congruence(s_1, s_2, |, +, \emptyset)$ gives *true* as result.

Pseudocode 4.3 Implementation of the relation $congruence(.,.,.,.,.,.)$ for SCOWS services

Input: Service s_1, s_2 Operator$\star, \diamond$ Map$\& \chi$
Output: bool b

```
1:  View M1 = flt_{⋆α}(s1);
2:  View M2 = flt_{⋆α}(s2);
3:  List < Service > L1 = new List(unique(flt_⋆(s1));
4:  List < Service > L2 = new List(unique(flt_⋆(s2));
5:  Permuter perm = new Permuter(L1);
6:  while perm.hasNextPermutation() do
7:     Map χt = χ;
8:     List < Service > P = perm.getNext();
9:     for i = 1 → size(p) do
10:       Pair < int, List < Id >>(n1, b1) = M1(P[i]);
11:       Pair < int, List < Id >>(n2, b2) = M2(L2[i]);
12:       if n1 ≠ n2 or b1.size() ≠ b2.size() then
13:          goto 6;
14:       end if
15:       bool res = compare(P[i], L2[i], ⋆, ⋄, χt);
16:       if !res then
17:          goto 6;
18:       end if
19:    end for
20:    χ.merge(χl);
21:    return true;
22: end while
23: return false;
```

Function $compare(.,.,.,.,.,.)$, presented in Pseudocode 4.4, compares the types of two services s_1 and s_2 given as parameters. This control considers the nature of services s_1 and s_2 and, when present, the number of parameters. Note that if s_1 and s_2 are of different types or if the number of parameters does not match, none of the expression checked in the if statements is true, so the procedure returns $false$. If the types of s_1 and s_2 match, the algorithm tries to extend χ with a new mapping for each used identifier. This role is carried out by function $extendMap(.,.,.)$, which is defined in Pseudocode 4.5. If no error arises, then the algorithm in Pseudocode 4.4 returns *true* as result and, as a side effect, modifies the mapping χ passed as parameter.

The function defined in Pseudocode 4.4 is also responsible for performing recursive calls of the congruence relation. The most interesting case is the one considered in lines 25-26: s_1 and s_2 are service compositions defined using the $\diamond$ operator, which is the nondeterministic choice (parallel composition) if the flattening was performed on parallel compositions (nondeterministic choices). Note that the recursive call swaps $\diamond$ and $\star$. Other cases which need a special treatment are those considered at lines 20 and 23: since the scopes of application of kill label delimiters and protection operators cannot be modified by congruence rules, but the services representing these scopes can be rearranged using these rules, when $s_1 = [\,k_1\,]s_1'$ and $s_2 = [\,k_2\,]s_2'$, or when $s_1 = \{\!|s_1'|\!\}$ and $s_2 = \{\!|s_2'|\!\}$ we have to recursively apply function $congruence(.,.,.,.,.,.)$. In these cases, we force $\star = |$ and $\diamond = +$.

Function $extendMap(d_1, d_2, \chi)$, defined in Pseudocode 4.5, implements the check performed when trying to extend a bijective map between identifiers. First, the function controls the types of the involved identifiers d_1 and d_2: names can be mapped only into names, and the same holds for variables and kill labels. If the mapping $d_1 \mapsto d_2$ is already in χ, then no conflict arises (lines 3-4). If there is no mapping in χ from d_1 to a generic $d \neq d_2$ and χ does not map a generic $d \neq d_1$ to d_2, then $d_1 \mapsto d_2$ can be added to χ, obtaining again a

Pseudocode 4.4 Implementation of auxiliary function $compare(.,.,.,.,.,.)$

Input: Service s_1, s_2 Operator $\star$ Operator $\diamond$ Map& χ

Output: bool res

1: bool $res = false$;
2: **if** $s_1 = (\mathbf{kill}(k_1), \lambda)$ and $s_2 = (\mathbf{kill}(k_2), \lambda)$ **then**
3: $res = extendMap(k_1, k_2, \chi)$;
4: **else if** $s_1 = \left(p_1.o_1\ ?\widetilde{u}_{11},.,u_{1n}.s'_1, \delta\right)$ and $s_2 = \left(p_2.o_2\ ?\widetilde{u}_{21},.,u_{2n}.s'_2, \delta\right)$ **then**
5: $res = extendMap(p_1, p_2, \chi)$ and $extendMap(o_1, o_2, \chi)$;
6: **for** $i = 1 \rightarrow n$ **do**
7: $res = res$ and $extendMap(u_{1i}, u_{2i}, \chi)$;
8: **end for**
9: $res = res$ and $compare(s'_1, s'_2, \star.\diamond, \chi)$;
10: **else if** $s_1 = (p_1.o_1\ !\widetilde{u}_{11},.,u_{1n}, \gamma)$ and $s_2 = (p_2.o_2\ !\widetilde{u}_{21},.,u_{2n}, \gamma)$ **then**
11: $res = extendMap(p_1, p_2, \chi)$ and $extendMap(o_1, o_2, \chi)$;
12: **for** $i = 1 \rightarrow n$ **do**
13: $res = res$ and $extendMap(u_{1i}, u_{2i}, \chi)$;
14: **end for**
15: **else if** $s_1 = S(d_{11},.,d_{1n})$ and $s_2 = S(d_{21},.,d_{2n})$ **then**
16: $res = true$;
17: **for** $i = 1 \rightarrow n$ **do**
18: $res = res$ and $extendMap(d_{1i}, d_{2i}, \chi)$;
19: **end for**
20: **else if** $s_1 = [\,k_1\,]s'_1$ and $s_2 = [\,k_2\,]s'_2$ **then**
21: $res = extendMap(k_1, k_2, \chi)$;
22: $res = res$ and $congruence(s'_1, s'_2, |, +, \chi)$;
23: **else if** $s_1 = \{\!|s'_1|\!\}$ and $s_2 = \{\!|s'_2|\!\}$ **then**
24: $res = congruence(s'_1, s'_2, |, +, \chi)$;
25: **else if** $s_1 = s'_1 \diamond s'_2$ and $s_2 = s'_3 \diamond s'_4$ **then**
26: $res = congruence(s_1, s_2, \diamond, \star, \chi)$;
27: **end if**
28: **return** res;

bijective function.

Pseudocode 4.5 Implementation of function $extendMap(.,.,.)$

Input: Identifier d_1, d_2 Map& χ

Output: bool $result$

```
1: if type(d1) ≠ type(d2) then
2:     return false;
3: else if χ = {d1 ↦ d2} ∪ χ' then
4:     return true;
5: else if d1 ∉ dom(χ) and d2 ∉ cod(χ) then
6:     χ.add(d1 ↦ d2);
7:     return true;
8: else
9:     return false;
10: end if
```

4.4.6 Complexity Analysis and Performance Optimizations

We described a procedure implementing the given definition of structural congruence for SCOWS services. Analysing Pseudocode 4.3, it is possible to see that the complexity of such an algorithm, when applied to services s_1 and s_2, depends on the number of permutations of a list of subservices of s_1. Since this number is equal to the factorial of the length of the list under consideration, if we denote by $length(s)$ the number of parallel compositions, nondeterministic choices, kill label delimiters and protections inside s, we can identify the worst case scenario as the one requiring $O(length(s)!)$ cycles to decide $s_1 \stackrel{?}{\equiv} s_2$: other operations, such as flattening and comparing services to build χ are linear in the length of the considered services. This complexity seems to deny any effectiveness of the presented procedure. Applying a simple reasoning, however, we can mitigate this problem: if we partition the list of basic subservices of s_1 and s_2 with respect to the service type, dividing e.g. request activities from invoke activities, we can avoid to consider some permutations that would certainly not

lead to a positive conclusion. The worst case scenario, however, represented by the case in which all subservices have the same type, has again time complexity $O(length(s)!)$. For instance, dividing a list of three request services and three invoke services in two sublists, we reduce the number of permutations to consider in the *while* cycle in Pseudocode 4.3 from $6! = 720$ to $3! \times 3! = 36$.

Another important optimization that can be easily introduced regards the search for a congruent service inside the LTS (line 8, Pseudocode 4.1): without optimizations, the search has to be performed considering all the states, and their associated SCOWS services, inside the LTS. If the graph g modelling the LTS is $g =< St, Tr >$ and s_{new} is the state to be inserted in g, we can denote the time complexity of a search for a state congruent to s_{new} as $O(length(s_{new})! \cdot |St|)$. However, we notice that the time-consuming check for α-equivalence is only needed in a subset of the cases: to identify these, we abstract from bound names, variables and kill labels. In order to do this, we fix a particular name $nh_{\mathcal{N}}$, a particular variable $vh_{\mathcal{V}}$ and a particular kill label $kh_{\mathcal{K}}$, which are fresh in all states in g and cannot be obtained decorating any identifier occurring in any state in g. We refer to these special identifiers as *holes*. We then apply function $holed(.)$, defined in Table 4.15, which performs the substitutions that allow us to abstract from the bound identifiers used in a service s. Note that in holed versions of services the non-homonymy condition is not guaranteed to hold. Holed services are used solely to compare the syntactic structure of services, and not to derive their behaviour, so this fact does not break our working assumptions.

If we apply the flattening function $flt_{|_\alpha}$ to $holed(s)$, we obtain a *flattened holed view* of s. If two services have different flattened holed views, then they cannot be structurally congruent. The states in the LTS can then be partitioned using this over-approximation; when a new state s has to be inserted, the algorithm checks if a subset of the states in the LTS shares the same flattened holed view of s, and performs the full congruence check only on these states. The worst case scenario, in which all states share the same flattened holed view,

$$
\begin{array}{lcl}
holed(\mathbf{0}) & = & \mathbf{0} \\
holed((\mathbf{kill}(k_1), \lambda)) & = & (\mathbf{kill}(k_1), \lambda) \\
holed((p_1.o_1\,?\widetilde{u}_{11},..,u_{1n}.s', \delta)) & = & (p_1.o_1\,?\widetilde{u}_{11},..,u_{1n}.holed(s'), \delta) \\
holed((p_1.o_1\,!\widetilde{u}_{11},..,u_{1n}, \gamma)) & = & (p_1.o_1\,!\widetilde{u}_{11},..,u_{1n}, \gamma) \\
holed(s_1 \mid s_2) & = & holed(s_1) \mid holed(s_2) \\
holed(g_1 + g_2) & = & holed(g_1) + holed(g_2) \\
holed([\,n\,]s) & = & [\,nh_{\mathcal{N}}\,]holed(s)\{nh_{\mathcal{N}}/n\} \\
holed([\,v\,]s) & = & [\,vh_{\mathcal{V}}\,]holed(s)\{vh_{\mathcal{V}}/v\} \\
holed([\,k\,]s) & = & [\,kh_{\mathcal{K}}\,]holed(s)\{kh_{\mathcal{K}}/k\}
\end{array}
$$

Table 4.15: Definition of function $holed(.)$

presents the same time complexity, namely $O(length(s)! \cdot |St|)$, but for practical situations the obtained gain is relevant. For instance, when running the BPMN Mail example presented in Chapter 5, SCOWS_lts is able to compute the whole LTS in $185s$ when the optimization is active, while it requires $124 \times 10^2 s$ to compute the same LTS, which is composed of 1453 states, without this optimization. The tests have been carried out on the same machine, under similar conditions of workload. This improvement is obtained caching the flattened holed views of services and, more importantly, using them as keys in a Java HashMap < Map, List < State >>. The optimized version of Pseudocode 4.2 can be seen in Pseudocode 4.6.

4.4.7 Other features of SCOWS_lts

The main purpose for the derivation of the whole LTS of a SCOWS service is the possibility of performing quantitative model checking on the CTMC derived from the LTS discarding transition labels. Because of this, SCOWS_lts presents two features aimed at the model checking phase. In particular, SCOWS_lts is able to operate with parametric rates: one can define a model containing actions of the type $(p.o\,!\widetilde{u}, \gamma)$, $(p.o\,?\widetilde{u}.s', \delta)$, $(\mathbf{kill}(k), \lambda)$ without specifying the actual values for rates γ, δ, λ. These values will be specified when performing

Pseudocode 4.6 Definition of function *addTransitions*(.), optimized

Input: List < Transition > *transitions*
Output: List < State > *result*

```
1: List < State > result = new List < State > ();
2: for all Transition t ∈ transitions do
3:   for all State s ∈ getGraphStates() such that flt|α(holed(s)) = flt|α(holed(t.getDestination())) do
4:     Map < Id, Id > χ = new Map < Id, Id >()
5:     if !congruence(t.getDestination(), s, |, +, χ) then
6:       addTransition(t.getSource(), t.getLabel(), t.getDestination())
7:       result.add(t.getDestination());
8:     else
9:       addTransition(t.getSource(), t.getLabel(), s)
10:    end if
11:  end for
12: end for
13: return result;
```

the model checking phase itself. In this way, a model can be derived once using SCOWS_lts and then be used to verify different configurations using a supported model checker (at the time of writing, PRISM is the only supported model checker). This feature is implemented introducing the use of arithmetical expressions in the computation of transition rates. If a parametric rate is present in the expression of a transition rate, then the latter is only partially evaluated, leaving the burden of completing its evaluation phase to the model checker, in which the actual values for parametric rates are defined.

Another important feature implemented by SCOWS_lts is the presence of *abstraction rules*: a particular SCOWS action, e.g. a communication on a given channel, can trigger a particular action in the model checker, e.g. the assignment of a value to a variable. SCOWS_lts implements this feature visiting the derived LTS and comparing each transition label to a list of abstraction rules defined in the input model. In this way we can introduce in the model flags

and counters, which will be used when specifying the properties to be verified in the model checking phase. For instance, $stop.votes < * >: stopc < 100 : stopc' = (stopc + 1)$; is an abstraction rule defined in the BPMN Mail example presented in Chapter 5. When this rule is applied, each transition involving a communication on endpoint *stop.votes*, whichever the parameters ($*$ operates as a *catch-all*), triggers the modification of a PRISM variable named *stopc*, which is incremented, as long as its value is below the threshold 100. The last requirement is needed by the semantics of PRISM: a transition can be triggered only if its boolean guard, checking the values of variables, is evaluated to true. In this case, the increment of variable *stopc* can be triggered as long as the value of the variable is below 100.

4.4.8 Usage Example

After downloading the software and the needed libraries, following the instructions available at [1], we can launch SCOWS_lts using a command similar to the one reported in Pseudocode 4.7.

Pseudocode 4.7 SCOWS_lts Usage Example

```
java -cp lib/java-cup-11a-runtime.jar:lib/log4j.jar\
:TOOL_LTS_0.3a.jar:. scows.Main input=model.scows
```

If the file *model.scows* exists and is a valid SCOWS_lts model, then the tool starts building the whole LTS describing the behaviour of the SCOWS agent specified inside *model.scows*. If the LTS can be represented using a finite number of states, i.e. if SCOWS_lts terminates its execution, the output of the tool consists of two files. The first file, in the example named *model.scows.dot*, is a representation of the LTS that can be used to produce a graphical representation of the transition graph using the dot tool, available at `http://www.graphviz.org`. The second file, named *model.scows.sm*, is the translation of the LTS in a CTMC model that can be opened using the PRISM model checker.

Chapter 5

Case Studies

In this chapter we apply SCOWS_lts to two case studies: the first one is derived from the BPMN Mail example, available at `http://www.omg.org/spec/BPMN/2.0/examples/PDF`, while the other case study is a comparison of three historically relevant algorithms solving the Mutual Exclusion Problem, in which a number of running processes have to compete for the exclusive access to a resource, using only shared variables. In particular we analyse Dekker's algorithm, attributed to Dekker by Dijkstra in [12], Dijkstra's algorithm, presented in [11] and one algorithm developed by Lamport, named the Bakery algorithm [18].

5.1 PRISM Notation

Here we briefly introduce the syntax of PRISM models, focussing our attention to the case of Continuous Time Markov Chain (CTMC) models. Each PRISM model is a composition of *modules* which can interact with each other. The state of each module is described by the state of its local variables, and the state of the whole model is represented by the states of the modules composing it. The behaviour of each module, i.e. the possible changes of the states of its local variables, is described by a command of the form $[\mathtt{l_i}]\mathtt{g_i} -> \mathtt{r_1} : \mathtt{u_1} + \ldots + \mathtt{r_n} : \mathtt{u_n}$; where r_i are rates associated to local state updates u_i, l_i is the label on which

parallel modules synchronize, using multi-way synchronization in the style of PEPA [15] and g_i is a predicate over the variables of the model. If g_i is evaluated to *true*, then $u_1, \ldots, u_n$ represent possible evolutions of the model.

PRISM models generated using SCOWS_lts are composed of one module, with at least one local variable named $prIdx$. Each state change in the value of $prIdx$ reflects a transition in the LTS of the SCOWS term given as input to SCOWS_lts. As we will see in the case studies, each variable appearing in an applied abstraction rule is represented by a different local variable in the PRISM model.

5.2 Case Study: BPMN Mail

This Case Study is based on the example that can be found at `http://www.omg.org/spec/BPMN/2.0/examples/PDF`. It describes the activities carried out by the manager of a mailing list for a voting board. Subscribers to the list (external participants, in terms of the BPMN specification) can discuss issues and, when the manager opens a voting session, send their vote. Other duties of the manager are represented by moderating the discussion, alert members of incoming deadlines, count votes and present the results of the voting procedure. When needed, the manager restarts the voting session or reset the discussion procedure. These actions are taken when no clear majority has arisen from a previous voting session.

We present the main SCOWS agents composing the model, which can be found in its entirety at `http://disi.unitn.it/~cappello/mail_bpmn.scows`.

Agent $Voting(.,.)$, presented as Pseudocode 5.1, represents the activities carried out by the manager when a new vote is instantiated. When the issue list is prepared and received (executing the request activity on channel *start#.vote#*, the sub-activities are activated in parallel. Parameters $firsttime$ and *alreadywarned* are identifiers associated with boolean values. These pa-

rameters are later used to identify if the voting procedure has already been resetted and if the mailing list members have already received a voting warning. Kill label *klvoting* is used to remove from the system activities associated with terminated voting instances.

Pseudocode 5.1 SCOWS agent for the voting procedure

```
1:  Voting(firsttime, alreadywarned) =
2:  [klvoting][issueList]
3:  (start#.vote#? < issueList >, 1.0).(
4:      CheckCalendar()
5:      |ModerateMailDiscussion()
6:      |DeadlineWarning()
7:      |CollectVote()
8:      |UpdateVotes()
9:      |ElabResults(firsttime, alreadywarned, klvoting)
10: );
```

Agent *ElabResults*(., ., .) (Pseudocode 5.2) models the behaviour of the mailing list manager when the voting procedure approaches its termination, i.e. when the discussion has been moderated and a deadline warning has been issued. After these activities the voting procedure stops, then the results are prepared (note the use of the parametric rate *modResposiveness* for example at lines 4,5). The SCOWS model proceeds checking for the number of collected votes. If this number is not enough and the situation happens for the first time, then the voting procedure is reset and a new one is issued; on the other hand, if this case has already happened in the past or if enough votes have been collected, then the system proceeds computing the relative majority of the votes (this behaviour is modeled by another agent, not depicted here). Note that the third parameter of *ElabResults*() is the kill label used by the kill activity triggered when the voting procedure has to be reset. The delimiter defining the scope of this kill label is defined in Pseudocode 5.2 at line 22.

Pseudocode 5.3 represents the code for the agent modeling the behaviour of

Pseudocode 5.2 SCOWS agent for the elaboration of votes

```
1:  ElabResults(firsttime, alreadywarned, klvoting) =
2:  (moderate#.cc#? <>, 1.0).(moderate#.maildisc#? <>, 1.0).
3:  [warn](dispatch#.deadline.warning#? < warn >, 1.0).(
4:     [result#](prepare#.results#! < result# >, modResponsiveness)
5:     |[res](prepare#.results#? < res >, modResponsiveness).(
6:        ...
7:     )
8:     |[vt#]((stop#.votes#! < vt# >, 1.0)
9:        |... .(
10:          [nbvotes#](enough#.votes#! < nbvotes# >, 1.0)
11:          |(//enough?no
12:             [nv1](enough#.votes#? < nv1 >, 1.0).(
13:                (members#.warned#! < alreadywarned >, modResponsiveness)
14:                |(
15:                   (members#.warned#? < true >, modResponsiveness).(
16:                      (reduce#.votes#! <>, 1.0)
17:                      |(reduce#.votes#? <>, 1.0).(...)
18:                   )
19:                   +(members#.warned#? < false >, modResponsiveness).(
20:                      [wmsg#](issue#.warning#! < wmsg# >, 1.0)
21:                      |[msg](issue#.warning#? < msg >, 1.0).(
22:                         (kill(klvoting), 1.0)
23:                         |{...})))
24:             )
25:             //enough?yes
26:             +[nv2](enough#.votes#? < nv2 >, 1.0).(...)
27:          )))));
```

the list manager when enough votes have been collected. The agent first checks if a majority on the vote has been reached. If it is not the case (lines 4-13), then the model restarts only the voting procedure (if it is the first time this situation happens, line 7) or the whole procedure (if a majority could not be found in preceding votes, lines 8-11). If a majority has been found (line 15), then no further action has to be taken, and the execution of the model stops.

Pseudocode 5.3 SCOWS agent for the elaboration of votes

```
 1: EnoughVotes(firsttime, alreadywarned, klvoting) =
 2: (reached#.majority#! <>, 1.0)
 3: |(   //majority?no
 4:    (reached#.majority#? <>, minorityRate).(
 5:       (outer#.loop#! < firsttime >, 1.0)
 6:       |(
 7:          (outer#.loop#? < true >, 1.0).(...)
 8:          +(outer#.loop#? < false >, 1.0).(
 9:             (kill(klvoting), 1.0)
10:             |...
11:          )
12:       )
13:    )
14:    //majority?yes
15:    +(reached#.majority#? <>, majorityRate).nil
16: );
```

A graphical representation of the transition system (without transition labels), generated by SCOWS_lts applied to the considered model, is available at `http://disi.unitn.it/~cappello/mail_unlabelled.svg`. Given the size of the representation, we do not insert it here.

In order to characterize the behaviour of the modelled system, we associate a counter (named *redisc*) to specific SCOWS actions. The SCOWS_lts code for this abstraction rule can be seen in Pseudocode 5.4. This counter is used to collect information on how many times the discussion procedure is restarted. In the

model, additional counters are defined, which collect information on how many times a voting procedure is stopped (*stopc*), on how many times a warning is issued to the list members (*warnc*), and on how many times a voting procedure is restarted (*revotec*).

Pseudocode 5.4 Example of an abstraction rule in the BPMN Mail Case Study

1: $outer\#.loop\# < false >: redisc < 15 : redisc' = (redisc + 1);$

First we analyse the steady-state probability of having x resets of the discussion procedure, with all rates set to 1.0. The PRISM property we consider is Property 5.1.

$$\mathrm{S} =?[redisc = x] \tag{5.1}$$

We limit our analysis to a maximum of 10 loops. Figure 5.1 presents the plot of the results, from which it it is possible to see that the probability of reaching the end of the model after a given number of loops decreases exponentially with the number of the considered loops.

Another analysis we present regards the sensitivity of the model to the propensity of not gaining a clear majority during the voting phase. We consider Property 5.2.

$$\mathrm{P} =?[\ \mathbf{F}\ \leq T(redisc = 1)] \tag{5.2}$$

The change in the propensity of not gaining a clear majority is obtained changing the value of the parametric rate *minorityRate* in agent $EnoughVotes$ (see line 3 in Pseudocode 5.3). The different values for *minorityRate* determine the propensities of executing one of the two available communications; the value for $majorityRate$ (see line 15 in Pseudocode 5.3) is fixed and equal to 1. Figure 5.2 reports the results obtained checking for Property 5.2 with $minorityRate \in \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19\}$, between 0 and 100 time units. The same data is also presented in Figure 5.3, where it is clearer that the impact

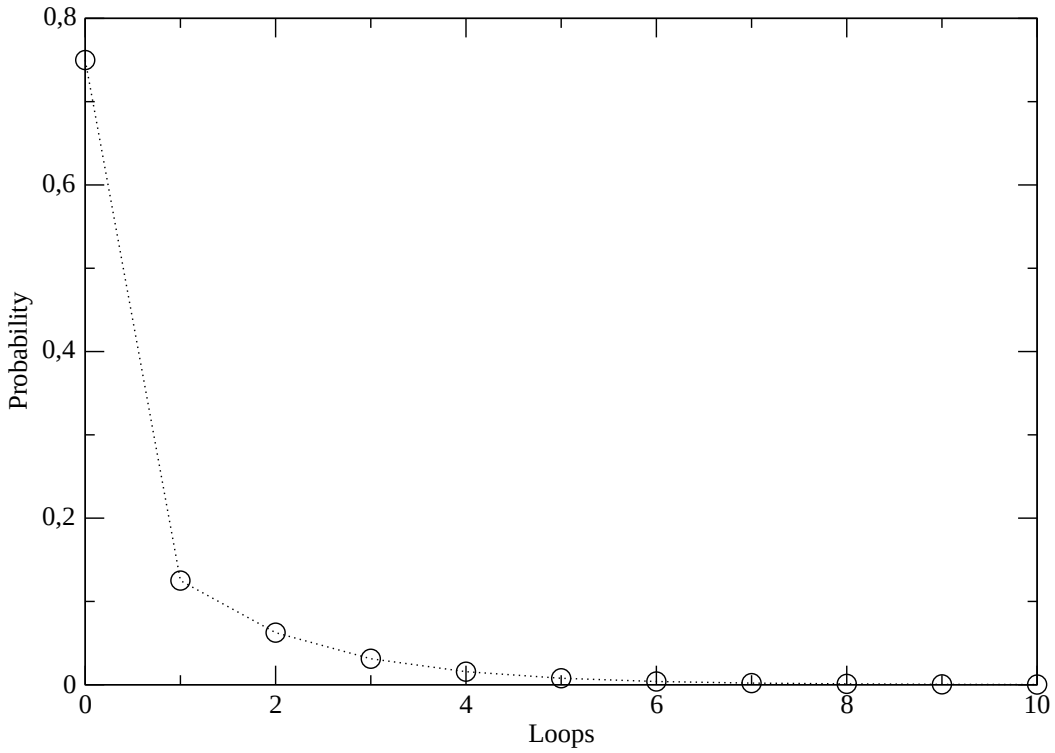


Figure 5.1: Plot of the steady-state probability of having x discussion loops

on the result of a change in the value of *minorityRate* decreases with the increase of the absolute value of the parameter itself: the 3D graph, in fact, shows that the slope of the surface becomes smoother when minorityRate has higher values.

5.3 Case Study: Mutual Exclusion

In this section we analyse several algorithms developed to solve the Mutual Exclusion problem, in which two or more processes compete to access a resource in an exclusive way. When a process is granted this access, it is said

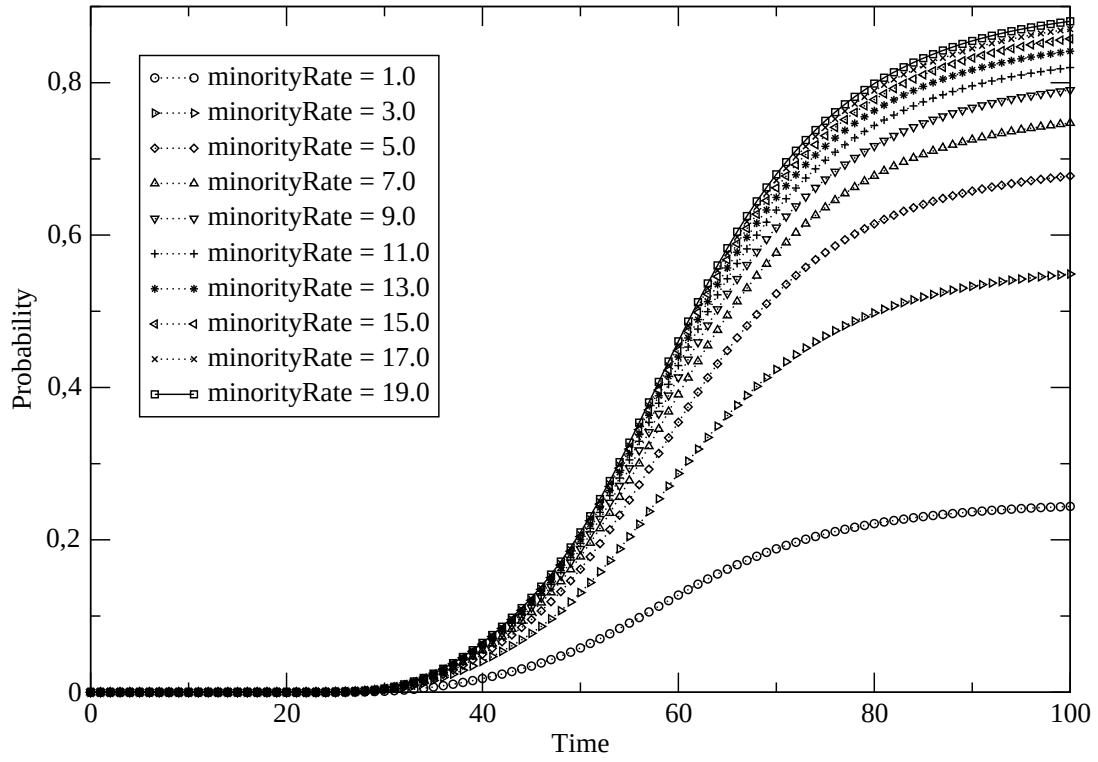


Figure 5.2: Probability of having to perform one reset of the discussion, given the propensity of not having a clear majority after a vote (series representation)

to be in its *critical section*. These algorithms have been developed so that, using only shared variables, at any time at most one of the competing processes is in its critical section. We model the considered algorithms in SCOWS and use SCOWS_lts to obtain, for each model, the LTS which will be imported and studied in PRISM. In particular, we consider three algorithms: one attributed to Dekker [12], one presented by Dijkstra [11] and one presented by Lamport [18]. Dekker's algorithm is limited to the case of two competitors, while the others algorithms can solve the problem for N concurrent processes.

Each variable defined in the algorithms is modelled in SCOWS using a ser-

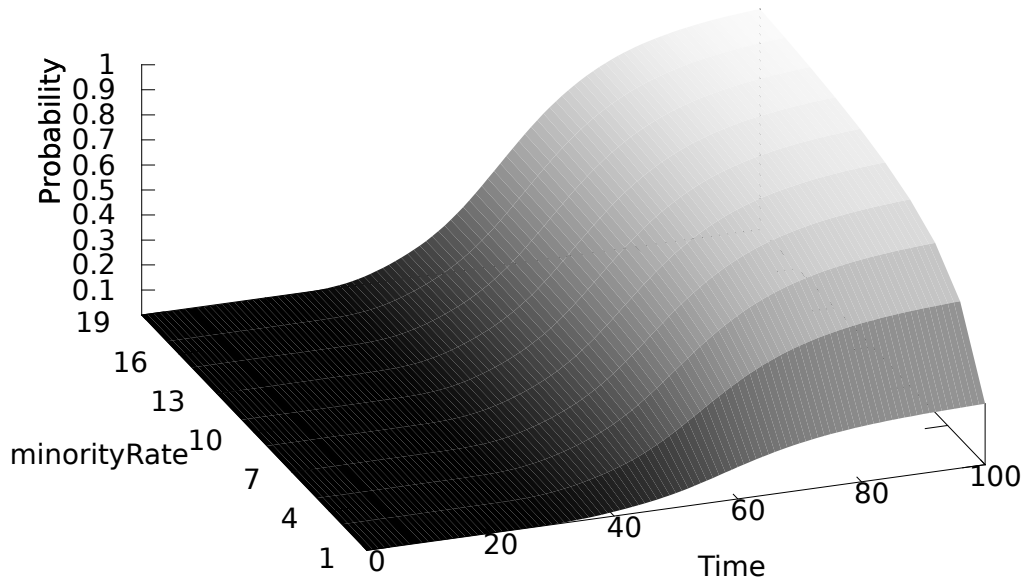


Figure 5.3: Probability of having to perform one reset of the discussion, given the propensity of not having a clear majority after a vote (3D representation)

vice of the form presented in Pseudocode 5.5, which models a variable named *n#*. A modelled process S_P willing to change the value associated to variable *n#* has to interact with agent $CSv(n\#,k)$ over endpoint *set#.n#*. After the new value has been received, the residual of agent $CSv(n\#,k)$ proceeds to kill parallel instances of $CGv(n\#,oldvalue)$. It then signals to the residual of process S_P that the new value has been received and instantiated. Service $CGv(n\#,value)$, on the other hand, interacts over endpoint *need#.get#* with a modelled process S_P willing to perform a read operation of the variable *n#*. In this case, the service has a simple persistent request-response structure. Note, however, that after S_P requests the value of *n#* any subsequent interaction with $CSv(n\#,k)$ with another process S_Q will not change the value obtained by S_P, since the invoke action at line 19 in Pseudocode 5.5 is protected and thus cannot be killed. This

can be seen also in Figure 5.4, which represents the LTS for the SCOWS service obtained putting in parallel an agent of the form represented in Pseudocode 5.5 for a variable named *n#* initially equal to 0, a service trying to read the value of *n#* and a service trying to set the value of *n#* to 1. As can be seen from the sequence of highlighted labelled actions, if S_P first, then eventually it reads the value 0, while if the write service executes first, then the read service eventually gets the value 1. In other words, the presented models verify *sequential consistency*, defined in [19] as the condition in which *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. A multiprocessor satisfying this condition will be called sequentially consistent*. Starting from this modeling technique for single variables, we model a vector *vec* of size N as a set of independent N variables named $vec0, \ldots, vecN$, so for each vector cell there is an instance of agent CSv and one of agent CGv.

Modelling a read operation of the value associated to a variable named v is done through a service of the form $(need\#.get\#! < v, pid >, 1.0)|[myv](get\#.v? < myv, pid >, 1.0).(\ldots)$, where *pid* is the unique identifier for the process, which interacts with a service $CGv(v, val)$ put in parallel. In the same way, an update of the value associated to variable v with a new value *val* is performed by a service of the form $(set\#.v! < val, pid >, 1.0)|(ok\#.v? < val, pid >, 1.0).(\ldots)$. The evaluation of arithmetic and boolean expressions, needed to model the behaviour of conditional statements and cycles, is performed first reading the values of the involved variables, then putting in parallel an invoke service having as parameter the expression to check and a sequence of nondeterministic choices between request services, each having as parameter a relevant result for the evaluation of the expression. For example, the construct $if(exp)then \ldots else \ldots$ can be modelled by a service of the form $(check\#.local\#! < exp >, 1.0)|((check\#.local\#? < true >, 1.0).(\ldots) + (check\#.local\#? < false >, 1.0).(\ldots))$. Iterative constructs

can be modelled in a similar way by recursive agents.

To express PRISM properties controlling the behaviour of models with respect to mutual exclusion, we defined two abstraction rules which can be seen in Table 5.1. These rules state that, when the i-th process enters the critical section, the PRISM variable i_crit is set to 1 (initially it is 0). The second rule states that when the i-th process leaves the critical section, the PRISM variable i_crit is set to 2.

is#.crit#	$< @1 >$	$: @1\_crit < 2$	$: (@1\_crit' = 1);$
is#.noncrit#	$< @1 >$	$: @1\_crit < 2$	$: (@1\_crit' = 2);$

Table 5.1: Abstraction rules for Mutual Exclusion models

The whole SCOWS_lts models for the presented algorithms can be seen in Appendix B.

We now present the pseudocode for the considered algorithms.

Pseudocode 5.6 presents the Dekker's algorithm for the process indexed by i, while the only antagonist is indexed by j. The algorithm is based on the use of two boolean variables b_i and b_j, initially set to false, and of a shared variable k, whose initial value can be either i or j. If b_i (b_j) is set to *true*, it means that process i (j) is actively trying to enter the critical section. Variable k contains the index of the next process that will be able to enter the critical section. When exiting the critical section, process i (j) flips the value of k and sets b_i (b_j) to $false$.

Dijkstra's algorithm, presented in Pseudocode 5.7 solves the problem of mutual exclusion for N contending problems. The set of shared variables is composed of two boolean vectors b, c of size N. Each i-th element of these vectors, initially equal to *true*, is set by the process indexed as i and is read by the other processes. Shared variable $flag$ is read and set by all processes, and its initial value can be any value in $\{1, \ldots, N\}$.

The Bakery Algorithm is presented in Pseudocode 5.8. This algorithm uses

Pseudocode 5.5 SCOWS agent modelling a variable named *n#*

```
1:  [k](
2:      CS v(n#, k)
3:      |CGv(n#, zero)
4:  )
5:  CS v(v, k_v) =
6:      [pr][val](set#.v? < val, pr >, 1.0).(
7:          (kill(k_v), 1.0)
8:          | {
9:              (ok#.v! < val, pr >, 1.0)
10:             | [k_new_v](
11:                 CS v(v, k_new_v)
12:                 CGv(v, val)
13:             )
14:         }
15:     )
16: ;
17: CGv(v, val) =
18:     [lpr](need#.get#? < v, lpr >, 1.0).(
19:         {(get#.v! < val, lpr >, 1.0)}
20:         | CGv(v, val)
21:     )
22: ;
```

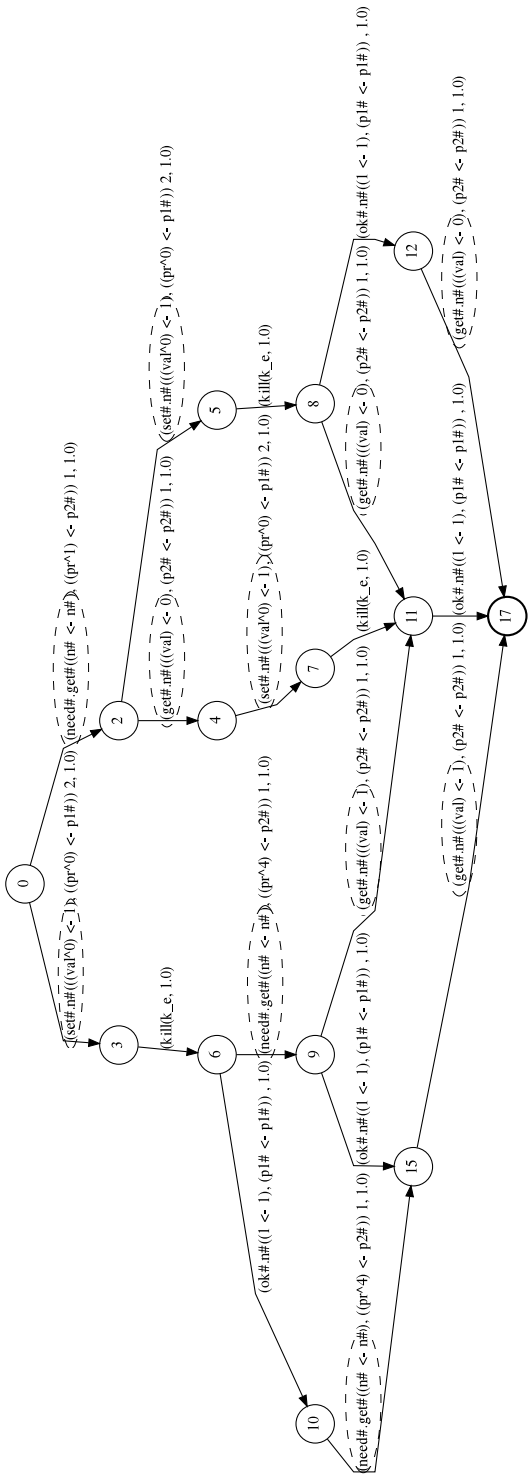


Figure 5.4: LTS derived interleaving a write (set) and a read (get) of a variable *n#*

Pseudocode 5.6 Dekker's Algorithm

```
 1: while true do
 2:    b_i := true;
 3:    while b_j do
 4:       if k ≠ i then
 5:          b_i := false;
 6:          while k ≠ i do
 7:             skip;
 8:          end while
 9:          b_i := true;
10:       end if
11:    end while
12:    {beginning critical section}
13:    {ending critical section}
14:    k := j;
15:    b_i := false;
16: end while
```

a boolean vector *choosing* of size N, initialized to $false$ and an integer vector *nmb* of size N, initialized to 0. This algorithm captures the behaviour of a set of customers waiting for their turn at a store: a process i gets an index, which is stored in $nmb[i]$. The construct $(a, b) < (c, d)$ used in the condition at line 9 is a shortcut for $(a < c) \vee ((a = c) \wedge (b < d))$. Notably, this algorithm works even when the condition of sequential consistency is dropped.

We studied the presented algorithms when considering two processes contending for entering the critical section. In the models, we introduced parametric rates for SCOWS actions controlling the *busy waiting* loops in Dekker's and Lamport's algorithm, and for SCOWS actions involved in the control of the loop at line 10 of Dijkstra's algorithm. In this way, we can provide temporal information on the performances of the algorithms varying one parameter.

In order to check mutual exclusion, we checked Property 5.3. Variables $proc0\_crit$ and $proc1\_crit$ are the ones modified by the first abstraction rule in

Pseudocode 5.7 Dijkstra's Algorithm for the i-th process

```
1:  b[i] := false;
2:  if flag ≠ i then
3:      c[i] := true;
4:      if b[flag] then
5:          flag := i;
6:      end if
7:      goto 2;
8:  else
9:      c[i] := false;
10:     for j := 1 → N do
11:         if j ≠ i ∧ ¬c[j] then
12:             goto 2;
13:         end if
14:     end for
15: end if
16: c[i] := true;
17: b[i] := true;
18: {beginning critical section}
19: {ending critical section}
20: goto 1;
```

Pseudocode 5.8 Lamport's Bakery algorithm for process i

```
1:  while true do
2:      choosing[i] := true;
3:      nmb[i] := 1 + max(nmb[1], ..., nmb[N]);
4:      choosing[i] := false;
5:      for j := 1 → N do
6:          while choosing[j] do
7:              skip;
8:          end while
9:          while (nmb[j] ≠ 0) ∧ (nmb[j], j) < (nmb[i], i) do
10:             skip;
11:         end while
12:     end for
13:     {beginning critical section}
14:     {ending critical section}
15:     nmb[i] := 0;
16: end while
```

Table 5.1, which sets as 1 the variable corresponding to the process inside the critical section. If both variables are equal to 1 at the same time, it means that the mutual exclusion property does not hold. All three models, when checked with PRISM, give *true* as response, meaning that the mutual exclusion property holds.

$$\mathrm{P} = 0[(trueU((proc1\_crit = 1)\&(proc2\_crit = 1)))] \tag{5.3}$$

Given that mutual exclusion holds, we can use Property 5.4 to infer quantitative information about the performance of the modelled algorithms. Property 5.4, in particular, considers the time needed for one of the contending processes to be granted access to the critical section. We checked this property considering three values for the parametric rate, namely $1.0, 50, 0, 99.0$, setting all other rates to 1.0. In this way we can study the impact of busy waiting (and of the for cycle for Dijkstra's algorithm) on the overall performances of the sys-

tem. By considering the results, we can see that Dekker's solution is the fastest algorithm to grant one of the processes the access to the critical section. Since it is the only solution built to solve the problem for exactly two contenders, this is not surprising: the other algorithms introduce additional variables and controls which delay the access to the critical section. The results obtained checking Property 5.4 are presented in Figure 5.5 for Dekker's algorithm, in Figure 5.6 for Djikstra's algorithm and in Figure 5.7 for Lamport's algorithm.

$$\mathrm{P} =?[(\mathit{true}\ \mathbf{U}\ <= T((\mathit{proc}1\_\mathit{crit} = 1)|(\mathit{proc}2\_\mathit{crit} = 1)))] \tag{5.4}$$

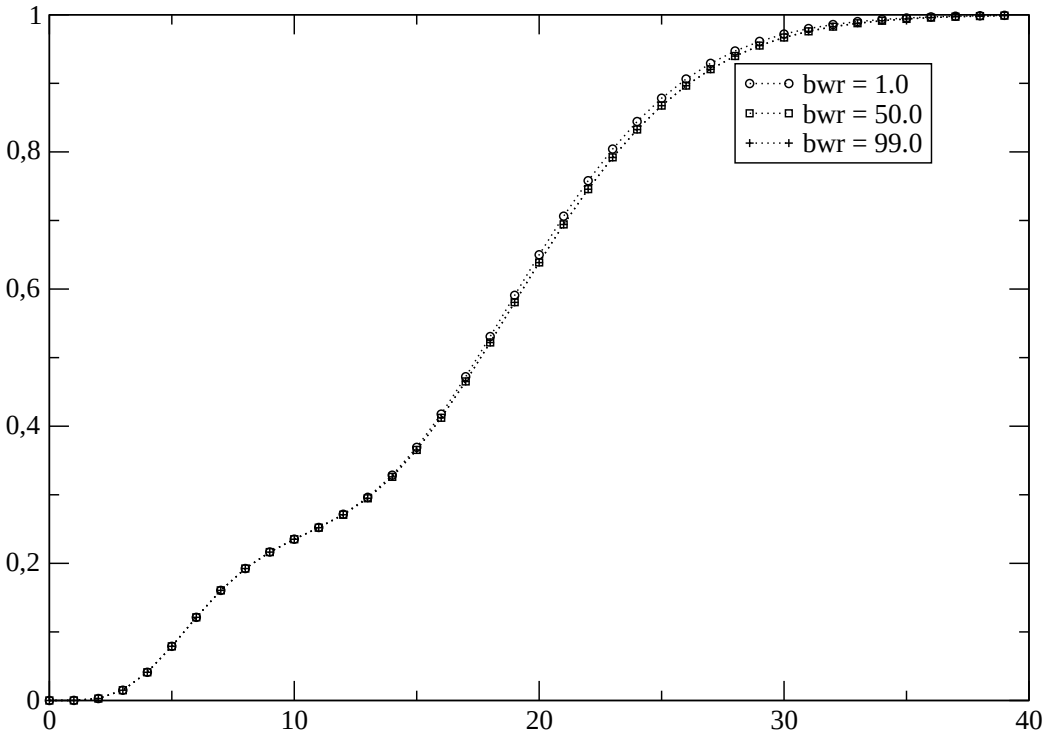


Figure 5.5: Results obtained checking Property 5.4 for Dekker's algorithm

The results for Dekker's algorithm, presented in Figure 5.6, show that the algorithm is not very sensitive to the propensity of the actions involved in the busy waiting loop at lines 6-7 in Pseudocode 5.6. The slight differences between the three cases can be seen between 15 and 30 time units.

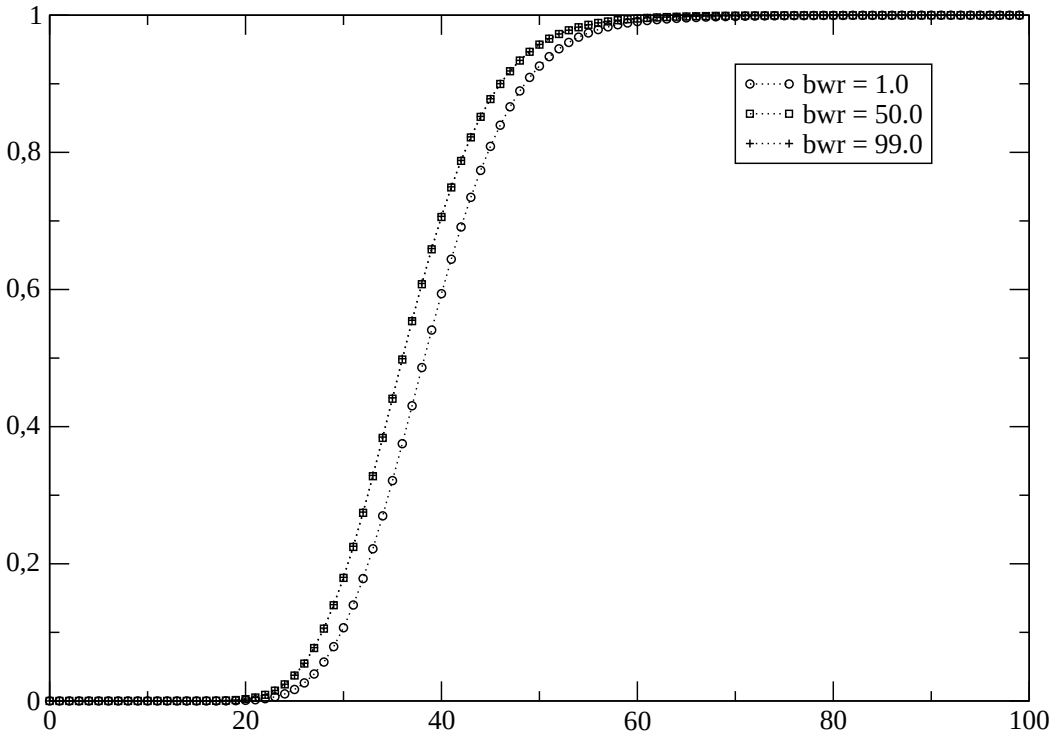


Figure 5.6: Results obtained checking Property 5.4 for Djikstra's algorithm

When comparing the results for the model of Djikstra's algorithm, we can see that the behavioural difference between the cases in which $bwr = 50.0$ and $bwr = 99.0$ is negligible, while the transient behaviour of the model is significantly slower when $bwr = 1.0$.

The results obtained checking Property 5.4 for the model of Lamport's algo-

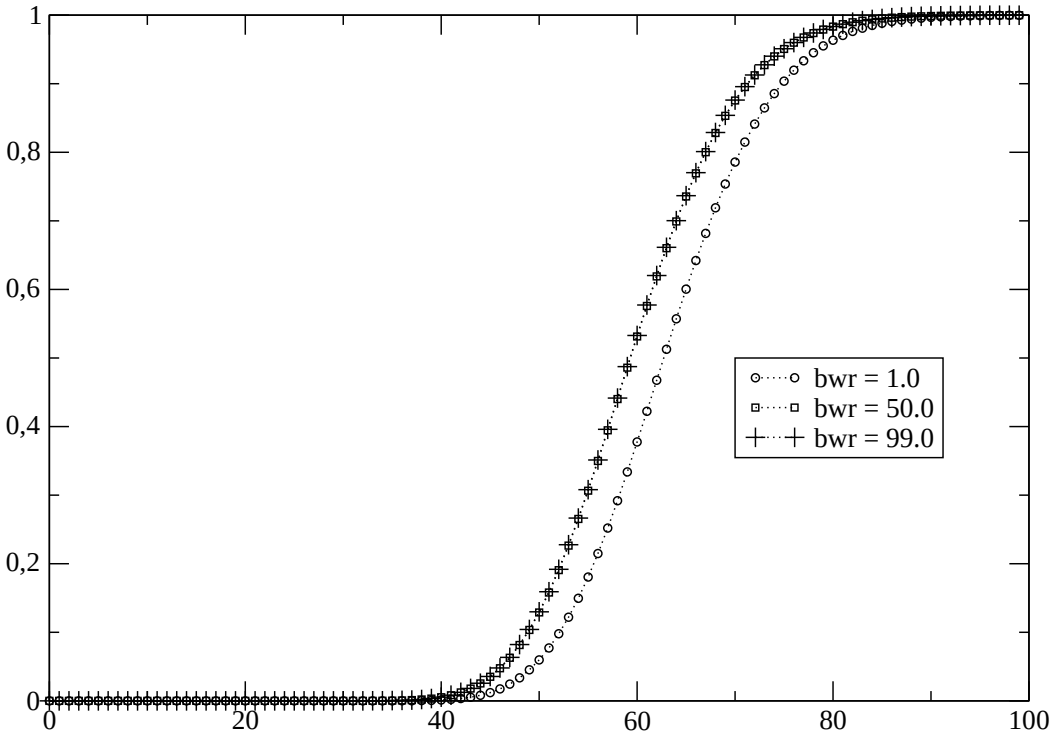


Figure 5.7: Results obtained checking Property 5.4 for Lamport's algorithm

rithm reveal that the transient behaviour of the system is slower when compared with the model of Djikstra's algorithm. This can be explained considering that Djikstra's algorithm is based on the assumption of having atomic read and write operations, while Lamport's algorithm does not operate under this assumption, has to introduce additional operations to compensate. Note however that the read/write operations in the SCOWS models presented in Appendix B respect the *sequential consistency* property.

To give a quantitative measure of the comparison of the transient evolution of the models, we report in Table 5.2 the time for which Property 5.4 is verified with probability $p \geqslant 0.5$ when $bwr = 50.0$.

Algorithm	Time Unit	Probability
Dekker	18	~ 0.52
Djikstra	37	~ 0.55
Lamport	60	~ 0.53

Table 5.2: Time for which Property 5.4 is verified with probability $p \geqslant 0.5$ for $bwr = 50.0$

Chapter 6

Conclusions

Among the various languages built to describe in a formal way Web Services, COWS is an interesting representative given its communication mechanism, based on correlation sets and priority levels determined by best matching, and the presence of peculiar operators such as kill activities and protection delimiters. We gave an assessment of the prioritized communication mechanism induced by COWS semantics comparing, by means of a separation result and encoding functions, a fragment of CCS with global priorities (FAP) using the Leader Election Problem as base. We determined that, under some requirements regarding encoding functions from FAP to COWS, the priority mechanism of FAP cannot be replicated into COWS. Relaxing these requirements, however, we were able to define an encoding function from FAP to COWS and proving a notion of behavioural correspondence, without introducing livelocks. From an operative point of view, then, we can say that the presented encoding function is reasonable.

We then presented a refined version of SCOWS, a stochastic dialect of COWS, one of the calculi developed in the European project SENSORIA `http://www.sensoria-ist.eu/` in the context of modelling and analysing in a formal way Web Services for both qualitative and quantitative properties. Starting from SCOWS syntax and semantics, we developed a Java tool, named SCOWS_lts

and available at [1], to derive the whole Transition System of SCOWS terms. In order to minimise the effect of the State Space Explosion Problem, we introduced in the implementation a notion of structural congruence between SCOWS services, allowing SCOWS_lts to reduce the actual number of states needed to represent the behaviour of a SCOWS term using congruence classes. The result of the simulation phase, in the form of a LTS representation, can be imported in PRISM [17] to be analysed. Apart from an optimized implementation of structural congruence, we described other features of SCOWS_lts, such as the support for parametric rates associated to basic actions and rules of abstraction from SCOWS actions to PRISM operations.

We presented the features of SCOWS_lts and the feasibility of the approach considering two main cases studies: the first is a manual translation of one of the examples given in the BPMN guide, the primary target language for the modelling ambitions of the COWS family of languages, while the second is composed of the analysis of various mutual exclusion algorithms, relevant mainly for historical reasons. We presented the results of model checking for quantitative properties on the presented examples using PRISM as the model checker of choice.

The work presented here could be the basis on which both theoretical and practical approaches to formal modelling, especially of Web Services, can be further advanced: assessing characteristics and peculiarities of modelling languages is fundamental to characterize their proper scope of application and specialization. On the practical side, the development of tools to promote and expand the use of these formal techniques, especially in non-academic communities, is a long-term goal that has to be pursued. In this sense the SENSORIA European Project, whose results are presented in [34], has shown that there is a great potential in this field which could be further investigated.

Bibliography

[1] disi.unitn.it/~cappello.

[2] Business process model and notation. http://www.omg.org/spec/BPMN/.

[3] Vigyan Singhal Adnan Aziz, Kumud Sanwal and Robert Brayton. Verifying continuous time Markov chains. In R. Alur and T. Henzinger, editors, *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 269–276. Springer, 1996.

[4] BEA Alexandre Alves, Intalio Assaf Arkin, Individual Sid Askary, Adobe Systems Charlton Barreto, Systinet Ben Bloch, IBM Francisco Curbera, Inc. Mark Ford, Active Endpoints, BEA Yaron Goland, Inc. Alejandro Guzar, JBoss, Sterling Commerce Neelakantan Kartha, SAP Canyang Kevin Liu, IBM Rania Khalaf, IBM Dieter Knig, formerly FileNet Corporation Mike Marin, IBM, Deloitte Vinkesh Mehta, Microsoft Satish Thatte, TIBCO Software Danny van der Rijn, webMethods Prasad Yendluri, and Oracle Alex Yiu. Web services business process execution language version 2.0. Technical report.

[5] Federico Banti, Rosario Pugliese, and Francesco Tiezzi. A criterion for separating process calculi. In *EXPRESS'10*, pages 16–30, 2010.

[6] Gérard Boudol. Asynchrony and the pi-calculus, 1992.

[7] Luc Bougé. On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. *Acta Inf.*, 25(2):179–201, 1988.

[8] Igor Cappello and Paola Quaglia. Expressing global priorities by best matching. In *SAC'12, to appear*.

[9] Igor Cappello and Paola Quaglia. A tool for checking probabilistic properties of cows services. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Trustworthly Global Computing*, volume 6084 of *Lecture Notes in Computer Science*, pages 364–378. Springer Berlin / Heidelberg, 2010.

[10] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[11] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[12] Edsger W. Dijkstra. Cooperating sequential processes. 1968.

[13] Alessandro Fantechi, Stefania Gnesi, Alessandro Lapadula, Franco Mazzanti, Roberto Pugliese, and Francesco Tiezzi. A model checking approach for verifying COWS specifications. In *Proc. of Fundamental Approaches to Software Engineering (FASE'08)*, volume 4961 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2008.

[14] Daniele Gorla. Comparing communication primitives via their relative expressive power. *Information and Computation*, 206(8):931–952, 2008.

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

[16] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP*, pages 133–147. Springer-Verlag, 1991.

[17] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):40–45, 2009.

[18] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[19] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[20] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A Calculus for Orchestration of Web Services. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[21] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 856–867. Springer, 1998.

[22] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[23] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I and II. *Information and Computation*, 100(1):1–77, 1992.

[24] Uwe Nestmann. What is a "good" encoding of guarded choice? *Inf. Comput.*, 156(1-2):287–319, 2000.

[25] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *POPL*, pages 256–265, 1997.

[26] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[27] Davide Prandi and Paola Quaglia. Stochastic cows. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 245–256. Springer, 2007.

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

[29] Paola Quaglia and Stefano Schivo. Approximate model checking of stochastic cows. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *TGC*, volume 6084 of *Lecture Notes in Computer Science*, pages 335–347. Springer, 2010.

[30] Jan Recker, Marta Indulska, Michael Rosemann, and Peter Green. How good is bpmn really? insights from theory and practice. *Proceedings of the 14th European Conference on Information Systems*, pages 1582 – 1593, 2006.

[31] Stefano Schivo. *Statistical model checking of Web Services.* PhD thesis, Int. Doctorate School in Information and Communication Technologies, University of Trento, 2010.

[32] Cristian Versari, Nadia Busi, and Roberto Gorrieri. An expressiveness study of priority in process calculi. *Mathematical Structures in Computer Science*, 19(6):1161–1189, 2009.

[33] Maria Grazia Vigliotti, Iain Phillips, and Catuscia Palamidessi. Tutorial on separation results in process calculi via leader election problems. *Theor. Comput. Sci.*, 388(1-3):267–289, 2007.

[34] Martin Wirsing and Matthias M. Hölzl, editors. *Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, volume 6582 of *Lecture Notes in Computer Science*. Springer, 2011.

Appendix A

Operational Correspondence for $[\![.]\!]$

A.1 Theorem 6

Theorem. *If* $[\![P]\!] \rightarrow s$ *then there exists* s' *such that* $s \rightarrow^* s'$ *and either* $s' \equiv [\![P']\!]$ *for some* P' *such that* $P \rightarrow P'$*, or* $s' \equiv [\, h, l, r, \widetilde{m}\,](\!|P|\!)_r^{h\,l}$ *with* P *such that* $P \not\rightarrow\!\!\!\!\rightarrow$.

Proof. By definition of the encoding, we have

$$[\![P]\!] = [\, h, l, r, \widetilde{m}\,]\left(Monitor(h, l, r) \mid (\!|P|\!)_r^{h\,l}\right)$$

and also we know that the first action that service $[\![P]\!]$ can perform is a communication over endpoint *search.h*.

Branch 1 The structure of the derivation tree of the transition $[\![P]\!] \xrightarrow{search.h \cdot \sigma \cdot h \cdot u} s$, where we explicited the label, can have either the structure presented in Figure A.1, in which $u = x_0$, or the structure presented in Figure A.2, in which $u = h$. Note that the only difference between the two is the presence of one substitution in the communication label. Given the semantic rules of COWS, the former represents an acceptable COWS transition only if the latter cannot be executed.

In the former case, depicted in Figure A.1, there is no *Input* agent introduced by the encoding, otherwise a communication presenting a best matching tuple

$$\dfrac{\dfrac{\vdots}{[\,l,r,\widetilde{m}\,](\!|P|\!)_r^{h\,l} \xrightarrow{search.h\cdot\varepsilon\cdot h\cdot x_0} s_1' \qquad h\notin \mathrm{d}(\alpha) \qquad s\downarrow_h}(del\_p)}{s=[\,h,l,r,\widetilde{m}\,](\!|P|\!)_r^{h\,l} \xrightarrow{search.h\cdot\varepsilon\cdot h\cdot x_0} [\,h\,]s_1' = s'}$$

Figure A.1: Structure of the derivation tree for the transition $[\![P]\!] \xrightarrow{search.h\cdot\varepsilon\cdot h\cdot x_0} s$

could take place. Service s' can evolve only by performing a synchronization on endpoint *search.l* internal to agent *Monitor* (lines 4,5 in Pseudocode 3.2) involving one substitution; with another internal synchronization on endpoint *hard.r*, the reached residual service, s'', is in a deadlock state. We have that the sequence of reductions $[\![P]\!] = [\,h,l,r,\widetilde{m}\,]\left(Monitor(h,l,r) \mid (\!|P|\!)_r^{h\,l}\right) \rightarrow^* s''$ involves only actions in agent *Monitor*, which leaves no residual in s''. We can then conclude that $s'' \equiv [\,h,l,r,\widetilde{m}\,](\!|P|\!)_r^{h\,l}$.

Process P can be generally written as $P \equiv_F \prod_{i\leqslant n} x_i.P_i \mid \prod_{j\leqslant m} \overline{x_j} \mid \prod_{k\leqslant o} \overline{x_k}$ be the encoded FAP process: P presents $n \geqslant 0$ unguarded input actions, $m \geqslant 0$ unguarded prioritized outputs and $o \geqslant 0$ unguarded unprioritized outputs.

$$\dfrac{\dfrac{\vdots}{[\,l,r,\widetilde{m}\,](\!|P|\!)_r^{h\,l} \xrightarrow{search.h\cdot\varepsilon\cdot h\cdot h} s_1' \qquad h\notin \mathrm{d}(\alpha) \qquad s\downarrow_h}(del\_p)}{[\,h,l,r,\widetilde{m}\,](\!|P|\!)_r^{h\,l} \xrightarrow{search.h\cdot\varepsilon\cdot h\cdot h} [\,h\,]s_1' = s'}$$

Figure A.2: Structure of the derivation tree for the transition $[\![P]\!] \xrightarrow{search.h\cdot\varepsilon\cdot h\cdot h} s$

High priority search If the latter case of Branch 1, depicted in Figure A.2, takes place, i.e. if the initial communication over endpoint *search.h* is not internal to agent *Monitor*, by definition of encoding function $[\![.]\!]$ the only other possibility is that an *Input* agent obtains the token for searching for a matching high-priority output action, executing the request action over endpoint *search.h* at line 2 in Pseudocode 3.3. Note that this communication involves no substi-

tution. Without loss of generality, let suppose that the activated agent has been introduced by the encoding when considering a FAP process $x_{i'}.Q_{i'}$. In this case, the *Input* agent has been instantiated as $Input_{x.Q_{i'}}(x_{i'}, h, l, r)$.

Branch 2 After the acquisition of the token, there are only two possible continuations: either the activated input agent performs an internal synchronization on endpoint $x_{i'}.h$ (lines 3,6 in Pseudocode 3.3) involving one substitution, or a communication over $x_{i'}.h$ involving no substitution between the activated *Input* agent (line 3 in Pseudocode 3.3) and an agent $Output(x_{i'}, h)$ happens (line 1 in Pseudocode 3.4).

In the former case, the encoding function has not introduced any service $Output(x_{i'}, h)$, meaning that $P \neq Q_1 \mid \overline{x_{i'}} \mid Q_2$ for any FAP processes Q_1, Q_2. As a consequence of the synchronization, the residual of the activated *Input* agent is ready to release the high priority token, providing an invoke action on endpoint *search.h* (line 7 of Pseudocode 3.3) and waiting for the low priority token (line 8) or for a reset signal (lines 14 and 18). If the token is acquired by another *Input* agent, then the evolution is described reconsidering Branch 2.

High priority synchronization In the latter case, the only possible continuation is a synchronization between the residual of the activated *Input* agent and the residual of the found $Output(x_{j'}, h)$ agent over endpoint $x_{i'}.com$, for $x_{j'} = x_{i'}$ (lines 4 in Pseudocode 3.3 and line 1 in Pseudocode 3.4). The activated *Input* agent performs then an internal synchronization over endpoint $soft.r$ (lines 5 and 18 in Pseudocode 3.3). The only possible continuation is a communication over endpoint $comm.finish$ with h as parameter (line 10 in Pseudocode 3.2 and line 20 in Pseudocode 3.3). At this point the residual of agent *Monitor* is represented by agent *CReset*. Again, there is only one possible continuation: a synchronization over endpoint *hard.r*: all residuals of *Input* agents which unsuccessfully took part in the search for a matching high priority output

can perform a request activity over *hard.r* (line 14 in Pseudocode 3.3) which equally matches the invoke activity at line 15 in Pseudocode 3.2. For each of these *Input* residuals, after the synchronization over endpoint *hard.r* a prioritized local kill activity over label *kin* is performed; after this action, the residual of the *Input* agent is composed of a protected invoke action propagating the reset signal (line 16 in Pseudocode 3.3) and of a new instance of the encoding of the input action. The last synchronization over endpoint *hard.r* takes place between the last *Input* residual needing a reset (line 16 in Pseudocode 3.3)and the residual of agent *Monitor* (line 16 in Pseudocode 3.2). After this synchronization, the residual of the system, identified by s_2, has the form presented in Pseudocode A.1.

Pseudocode A.1 Encoding of a FAP process after a successful low priority communication

1: $s_2 \equiv [\, h, l, r, \widetilde{m}\,](Monitor(h, l, r)$
2: $\quad | \; (\!|Q_{i'}|\!)_r^{h\,l}$
3: $\quad | \; \prod_{i \leqslant n, i \neq i'} (\!|x_i.Q_i|\!)_r^{h\,l}$
4: $\quad | \; \prod_{j \leqslant m, j \neq j'} (\!|\overline{x_j}|\!)_r^{h\,l}$
5: $\quad | \; \prod_{k \leqslant o} (\!|\overline{x_k}|\!)_r^{h\,\overline{l}}$
6: $\quad)$

Note that s_2 is congruent to the encoding of the FAP process P', reachable from P after a high priority synchronization over channel $x_{i'} = x_{j'}$.

Low priority search If each activated $Input_{x_i.Q_i}(x_i, h, l, r)$ agent has performed the internal synchronization over endpoint $x_i.h$, meaning that no corresponding high priority output encoding service was found, the residual of service s is a service s_1 which we can identify, by construction of the encoding, as the service presented in Pseudocode A.2 If the w-th activated *Input* agent is instantiated as $Input_{x_w.Q_w}(x_w, h, l, r)$ and one of the m unguarded prioritized outputs in P, indexed by p is instantiated as $Output(x_p, h)$ where $x_w = x_p$, then the invoke action on endpoint $x_w.h$ at line 3 in Pseudocode 3.3 best matches with the re-

quest action at line 1 in Pseudocode 3.4. After this synchronization, the only possible continuation is a synchronization over endpoint $x_w.com$, following by an internal synchronization in agent $Input$ over endpoint $soft.r$ (lines 5,18 in Pseudocode 3.3). The encoding of the residual of the FAP input process Q is exposed, and the information about the executed synchronization is forwarded to the residual of the $Monitor$ service over endpoint $comm.finish$. (line 20 in Pseudocode 3.3 and line 10 in Pseudocode 3.2). The only possible continuation is the repetition, $w-1$ times, of a synchronization over endpoint $hard.r$ (line 15 in Pseudocode 3.2 and line 16 in Pseudocode 3.3 synchronizing with line 14 in Pseudocode 3.3). This synchronization triggers the cleanup phase, first carried out by the residual of the w-th activated $Input$ agent, which performs the kill activity over kill label kin. In this way each of the $w-1$ input agents is reset to its initial state $(\!|x_i.Q_i|\!)^{h\,l}_r$. After this sequence, the only possible synchronization is again on endpoint $hard.r$ and involves the invoke action at line 16 of the $w-1$-th considered $Input$ agent and the residual of agent $Monitor$ (at line 16 in Pseudocode 3.2) or, if $w=0$ this synchronization is internal to agent $Monitor$. In either case this step resets the monitor to its initial state $Monitor(h,l,r)$. At the end we obtain a service

$$
\begin{aligned}
s'' \equiv [\,h,l,r,\widetilde{m}\,]\Big(\quad & Monitor(h,l,r) \\
& \mid \textstyle\prod_{i\leq n, i\neq w} (\!|x_i.Q_i|\!)^{h\,l}_r \\
& \mid (\!|Q_w|\!)^{h\,l}_r \\
& \mid \textstyle\prod_{j\leqslant m, j\neq p} (\!|\overline{x_j}|\!)^{h\,l}_r \\
& \mid \textstyle\prod_{k} (\!|\overline{x_k}|\!)^{h\,l}_r \Big)
\end{aligned}
$$

Service s'' is structurally congruent to the residual of process P after a high priority synchronization over name x.

In case of an unsuccessful high priority search, the presented behaviour is repeated, considering l in place of h as name for the identification of the priority. In particular, we have that at least one synchronization happens on endpoint

Pseudocode A.2 Encoding of a FAP process after an unsuccessful high priority search

```
1: s1 ≡ [ h, l, r, m̃ ](⦃[ kcl ](search.l ! ⟨l⟩
2:       | ([ v2 ]search.l ? ⟨v2⟩ .CBlock(r, kcl)
3:                 +comm.finish ? ⟨l⟩ .CReset(h, l, r, kcl)))⦄
4:       | ∏_{i⩽n} ⦃search.l ? ⟨l⟩ .(x_i.l ! ⟨x_i, l⟩ | ...)⦄
5:       | ∏_{j⩽m} Output(x_j, h)
6:       | ∏_{k⩽o} Output(x_k, l)
7:       )
```

search.l. Either it is an internal synchronization to the residual of agent *Monitor* (lines 1-2 in Pseudocode A.2) or, for some i', it involves the request at line 4 in Pseudocode 3.3 for the i'-th considered *Input* agent.

Branch 3 In the first case, the encoding phase did not introduce any *Input* agent. This situation has already been discussed in Branch 1. In the second case, similarly to what happened before, the residual of an *Input* agent, indexed as i' and thus instantiated as $Input_{x_{i'}.Q_{i'}}(x_{i'}, h, l, r)$ gets the token to perform the search for an unguarded low priority output action indexed as k', and thus encoded as $Output(x_{k'}, l)$ with $x_{i'} = x_{k'}$.

Branch 4 If such an *Output* agent does not exist, the only possible executable action is an internal synchronization over endpoint $x_{i'}.l$ (lines 9 and 12 in Pseudocode 3.3), after which the token is again available. Another *Input* agent can then obtain the low priority token, and the protocol continues from the second choice of Branch 3.

Low priority synchronization If, on the other hand, such an *Output* agent exists, the only possible executable action is a synchronization over endpoint $x_{i'}.l$ involving no substitution (line 9 in Pseudocode 3.3 and line 1 in Pseudocode 3.4); this communication can be followed only by a synchronization over endpoint $x_{i'}.com$ (line 10 in Pseudocode 3.3 and line 1 in Pseudocode 3.4). Again, the

only possible continuation of the protocol is represented by an internal synchronization of agent *Input* over endpoint $soft.r$ (lines 11 and 18 Pseudocode 3.3). At this point the encoding of the residual $Q_{i'}$ of action $x_{i'}.Q_{i'}$ is instantiated, alongside with the protected communication over endpoint $comm.finish$ among agent *Input* (line 21 in Pseudocode 3.3) and agent *Monitor* (line 7 in Pseudocode 3.2). This synchronization triggers $n-1$ repetitions of a synchronization involving all *Input* actions which participated in the search, but not in the successful communication. These synchronizations happen over endpoint $hard.r$ (line 15 in Pseudocode 3.2, lines 14 and 16 in Pseudocode 3.3), and after them the prioritized kill action over *kin* takes place (line 15 in Pseudocode 3.3), which cleans the state of the *Input* agent under consideration. After this sequence, the last considered *Input* action synchronizes over endpoint $hard.r$ with the residual of agent *Monitor* (line 16 in Pseudocode 3.3 and line 16 Pseudocode 3.2) involving one substitution. After this synchronization, a fresh instance of the *Monitor* agent is ready to restart the execution of the communication protocol. We obtain a service s_2 which is of the form presented in Pseudocode A.3.

Pseudocode A.3 Encoding of a FAP process after a successful low priority communication

```
1: s2 ≡ [ h, l, r, m̃ ](Monitor(h, l, r)
2:      | (|Q_{i'}|)^{hl}_r
3:      | ∏_{i⩽n, i≠i'} (|x_i.Q_i|)^{hl}_r
4:      | ∏_{j⩽m} (|x̄_j|)^{hl}_r
5:      | ∏_{k⩽o, k≠k'} (|x̄_k|)^{hl}_r
6:      )
```

Note that service s_2 has the form of the encoding of a FAP process

$$P' \equiv_F Q_{i'} \mid \prod_{i\leqslant n, i\neq i'} x_i.P_i \mid \prod_{j\leqslant m} \overline{x_j} \mid \prod_{k\leqslant o, k\neq k'} \overline{x_k}$$

Process P' represents the residual of process P after the execution of a low priority synchronization over name $x_{i'} = x_{k'}$ between process $x_{i'}.Q_{i'}$ and process $\overline{x_{k'}}$.

The last possibility that we have to consider is the one where all *Input* agents perform unsuccessfully a low priority search, i.e. no synchronization is possible in the encoded FAP process P. In this case, the last synchronization happening over endpoint *search.l* involves the n-th *Input* agent and the residual of the *Monitor* agent (line 5 in Pseudocode 3.2 and line 12 in Pseudocode 3.3), after which the cleanup phase for *Input* residuals, consisting of n repetitions of a synchronization over endpoint *hard.r* takes place. The last synchronization over endpoint *hard.r* involves the last *Input* residual (line 16 in Pseudocode 3.3) and agent *Monitor* (line 22 in Pseudocode 3.2). Note that the obtained residual service s_3 is congruent to the service presented in Pseudocode A.4, which can perform no action.

Pseudocode A.4 Encoding after a totally performed unsuccessful search

```
1: s3 ≡ [ h, l, r, m̃ ](
2:       | ∏_{i≤n} (| x_i.Q_i |)_r^{h l}
3:       | ∏_{j≤m} (| x̄_j |)_r^{h l}
4:       | ∏_{k≤o} (| x̄_k |)_r^{h l}
5:       )
```

Given the distributive property of function $(\!|.|\!)_r^{h\,l}$, service s_3 is congruent to $[\,h, l, r\,](\!|P|\!)_r^{h\,l}$. □

A.2 Theorem 7

Theorem. *If* $P \rightarrow P'$ *then* $s = [\![P]\!] \rightarrow^* \equiv [\![P']\!]$

Proof. Given the semantics of FAP, a process P can perform a reduction $P \rightarrow P'$ in two cases: either it performs a high-priority synchronization $P \twoheadrightarrow P'$ or it performs a low-priority synchronization $P \mapsto P'$. By rule $(*)$ in Table 3.1, these two cases are mutually exclusive.

1. $P \twoheadrightarrow P'$. The proof tree producing this derivation is of the form

$$\dfrac{P \equiv_F Q_1 \qquad \dfrac{\dfrac{-}{x.R_1 \mid \overline{\underline{x}} \twoheadrightarrow R_1}}{Q_1 = x.R_1 \mid \overline{\underline{x}} \mid R_2 \twoheadrightarrow R_1 \mid R_2 = Q_2} \qquad Q_2 \equiv_F P'}{P \twoheadrightarrow P'}$$

Given this structure, we have $P \equiv_F x.R_1 \mid \overline{\underline{x}} \mid R_2$ and $P' \equiv_F R_1 \mid R_2$. We can then derive the encoding of P according to $s = [\![.]\!]$ to be

$$\begin{aligned} s = [\![P]\!] \;=\; & [\, h, l, r, \widetilde{m}\,](\\ & \quad Monitor(h, l, r)\\ & \quad \mid Input_{x.R_1}(x, h, l, r)\\ & \quad \mid Output(x, h)\\ & \quad \mid (\!|R_2|\!)^{h\,l}_{r}\\ &) \end{aligned}$$

Given this structure and the definition of agents *Monitor*, *Input* and *Output*, we can describe a possible evolution of $[\![P]\!]$ that will reach a configuration congruent to $[\![P']\!]$, as needed.

We have that s can perform a communication over endpoint *search.h* involving no substitution (line 2 in Pseudocode 3.2 and line 2 in Pseudocode 3.3). Given the definition of agent $Output(x, h)$ introduced by the encoding, the only possible continuation is a synchronization over endpoint $x.h$ involving no substitution (line 3 in Pseudocode 3.3 and line 1 in Pseudocode 3.4) followed by a synchronization over endpoint *x.com* (line 1 in Pseudocode 3.4 and line 4 in Pseudocode 3.3, which consumes the residual of agent $Output(x, h)$. The only possible contiunation for the residual service is represented by an internal synchronization performed by the residual of the activated input over endpoint $soft.r$. At this point the residual of agent *Input* is composed of the instantiation of the encoding of Q, the residual of the FAP input action $x.Q$ (line 19 in Pseudocode 3.3) and an invoke action on endpoint

comm.finish (line 20 in Pseudocode 3.3). This matches with the request action on the same endpoint defined in the *Monitor* agent (line 10 in Pseudocode 3.2). The communication on endpoint *comm.finish* involves one substitution; after it is executed, a new instance of the *Monitor* agent is instantiated. After this action, the residual service is $[\, h, l, r, \widetilde{m}\,](Monitor(h, l, r) \mid (\!|R_1|\!)_r^{h\,l} \mid (\!|R_2|\!)_r^{h\,l})$, which is congruent to the service obtained applying the encoding function to P'.

2. $P \mapsto P'$. Given the semantics of FAP, we have that $P \equiv_F Q_1 \not\twoheadrightarrow$, otherwise rule $(*)$ in Table 3.1 could not have been applied. We have that the derivation tree for $P \mapsto P'$ is

$$\dfrac{P \equiv_F Q_1 \qquad \dfrac{\dfrac{-}{x.R_1 \mid \overline{x} \mapsto R_1} \qquad x.R_1 \mid \overline{x} \mid R_2 \not\twoheadrightarrow}{Q_1 = x.R_1 \mid \overline{x} \mid R_2 \mapsto R_1 \mid R_2 = Q_2} \qquad Q_2 \equiv_F P'}{P \mapsto P'}$$

Process P can be generally written as $\prod_{i \leqslant n} x_i.P_i \mid \prod_{j \leqslant m} \underline{\overline{x_j}} \mid \prod_{k \leqslant o} \overline{x_k}$. Given the reduction tree, no prioritized synchronization can happen in P, so we can state that for all i, j such that $P \equiv_F x_i.P_i \mid \underline{\overline{x_j}} \mid R$, we have $x_i \neq x_j$. In the following, we will identify $x.R_1$ as the $\hat{i}$-th input process, i.e. $x_{\hat{i}}.R_{\hat{i}}$, and $\overline{x}$ as the $\hat{k}$-th output process, i.e. $\overline{x_{\hat{k}}}$.

We now consider the COWS service $s = [\![P]\!]$. We can state that the number of FAP input processes, and subsequently the number of *Input* agents introduced by the encoding, is $n > 0$; for this reason, s can evolve performing a transition on endpoint *search.h* with no substitution involved between agent *Monitor* (lin 2 in Pseudocode 3.2) and one of the *Input* agents (line 2 in Pseudocode 3.3). At this point, the residual of the activated *Input* agent can only perform an internal synchronization on enpoint *x.h* involving one substitution (lines 3 and 6 in Pseudocode 3.3): no matching service $Output(x, h)$, which could provide a better-matching re-

quest, has been introduced by the encoding function. The residual of the *Input* agent releases the token, providing an invoke action on endpoint *search.h*. The sequence of transitions is repeated for all the *Input* agents introduced by the encoding, where the last activated *Input* agent releases the token by synchronizing with the residual of the *Monitor* agent on endpoint *search.h*) with one substitution (line 3 in Pseudocode 3.3 and line 3 in Pseudocode 3.2).

At this point, the residual of the *Monitor* agent provides an invoke action on endpoint *search.l* representing the low priority token, which can be acquired nondeterministically by one of the *n Input* residuals. By simplicity, let the residual of the encoding introduced by considering the FAP input action $x_{\hat{i}}.R_{\hat{i}}$ be the first one to acquire the token by syncronizing on endpoint *search.l* with the residual of agent *Monitor*. By reasoning on the hypotheses, we derived the fact that a matching low priority output action is present in P, so an agent of the form $Output(x_{\hat{i}}, l)$ has been introduced by the encoding. For this reason, a synchronization on endpoint $x_{\hat{i}}.l$, involving no substitution, is possible (line 9 in Pseudocode 3.3 and line 1 in Pseudocode 3.4). A subsequent synchronization on endpoint $x_{\hat{i}}.com$ triggers the second phase of the protocol, which consists in initializing the encoding of the residual $P_{\hat{i}}$ and resetting all the agents involved in the search to their initial state. The first task is performed with a local communication on endpoint $x_{\hat{i}}.com$ by the residual of the agent encoding the input action involved in the FAP synchronization (lines 11 and 18 in Pseudocode 3.3). The residual of this synchronization and the residual of the *Monitor* agent can synchronize on endpoint $comm.finish$. This action signals to *Monitor* that a low synchronization has been performed, and that a reset signal has to be sent out on endpoint $hard.reset$ to all the *Input* agents who unsuccessfully searched for a matching partner. The first signal is sent by the *Monitor* agent itself (line 15 in Pseudocode 3.2); it matches with the re-

quest activity at line 14 in Pseudocode 3.3. After this synchronization, a prioritized kill activity is performed by the residual of *Input* on label *kin*. The reset signal is then replicated, in parallel with a clean instance of the *Input* agent. This sequence is repeated $n-1$ times; the last invoke action on endpoint *hard.reset* (line 16 in Pseudocode 3.3) is intercepted by the residual of agent *Monitor* (line 16 in Pseudocode 3.2), whose residual is a fresh instance of the *Monitor* agent.

After this sequence of transitions, we get a service

$$\begin{aligned} s' \equiv\ & [\, h, l, r, \widetilde{m}\,](\\ & \quad Monitor(h,l,r)\\ & \quad \mid (\!|R_{\hat{i}}|\!)^{h\,l}_{r}\\ & \quad \mid \textstyle\prod_{i \leqslant n, i \neq \hat{i}} (\!|x_i.R_i|\!)^{h\,l}_{r}\\ & \quad \mid \textstyle\prod_{j \leqslant m} (\!|\overline{x_j}|\!)^{h\,l}_{r}\\ & \quad \mid \textstyle\prod_{k \leqslant o, k \neq \hat{k}} (\!|\overline{x_k}|\!)^{h\,l}_{r}\\ &)\end{aligned}$$

Service s' is congruent to service $[\![P']\!]$, as required.

□

A.3 Theorem 8

Theorem. *If* $P \nrightarrow$ *then* $s = [\![P]\!] \rightarrow^{*} \equiv [\, h, l, r, \widetilde{m}\,](\!|P|\!)^{h\,l}_{r}$

Proof. Given that $P \nrightarrow$, we have that a proper derivation tree for a transition of P could not be built. Given the set of semantic rules for FAP, this means that neither of the two axioms in Table 3.1, namely rule (*com*) and (*pr_com*), could be applied. In other words, if we write $P \equiv_F \prod_{i \leqslant n} x_i.P_i \mid \prod_{j \leqslant m} \overline{x_j} \mid \prod_{k \leqslant o} \overline{x_k}$, $x_i \neq x_j$ and $x_i \neq x_k$ for all values $0 < i \leqslant n$, $0 < j \leqslant m$ and $0 < k \leqslant o$. Applying the encoding function $[\![.]\!]$ to P, we obtain a service

$$s = \{\!|hlr|\!\}(Monitor(h,l,r) \mid \prod_{i \leqslant n} (\!|x_i.P_i|\!)_r^{h\,l} \mid \prod_{j \leqslant m} (\!|\overline{x_j}|\!)_r^{h\,l} \mid \prod_{k \leqslant o} (\!|\overline{x_k}|\!)_r^{h\,l})$$

Based on the value of n, i.e. the number of *Input* agent instantiated by the encoding, we have two possible scenarios:

1. $n = 0$: the evolution of service s is the one presented for choice 1 in Branch 1 for Theorem 6: service s can evolve only by performing a synchronization on endpoint *search.h* internal to agent *Monitor* (lines 2,3 in Pseudocode 3.2) involving one substitution; after another internal synchronization on endpoint (lines 4,5 in Pseudocode 3.2), the only possible continuation is a synchronization on endpoint *search.h hard.r* (lines 21,22 in Pseudocode 3.2). The residual service, let call it be s', is in a deadlock state. We have that the sequence of reductions

   $$[\![P]\!] = [\, h, l, r, \widetilde{m} \,]\left(Monitor(h,l,r) \mid (\!|P|\!)_r^{h\,l}\right) \rightarrow^* s'$$

   involves only actions in agent *Monitor*, which is no more present in s', i.e. $s' \equiv [\, h, l, r \,](\!|P|\!)_r^{h\,l}$.

2. $n > 0$: the evolution of service s has been presented in Theorem 6 when considering unsuccessful searches for both high and low priority outputs. We have that in P there is at least one unguarded input FAP process ($n > 0$); given the definition of the encoding function, this means that at least one unguarded *Input* service is in the encoding $[\![P]\!]$. We also know that in P, for all input processes, there is no unguarded matching output, otherwise P could perform a transition, contradicting the hypothesis of the theorem.

   Given the definition of the encoding function applied to P under these conditions, all *Input* agents, instantiated as $Input_{x_i.Q_i}(x_i, h, l, r)$, are activated (one at a time) through a communication labelled as $search.h \cdot \varepsilon \cdot h \cdot h$ (line 2 in Pseudocode 3.2 and line 2 in Pseudocode 3.3); each activated *Input*

agent declares the search as unsuccessful by performing an internal synchronization labelled as $x_i.h \cdot \varepsilon \cdot x_i, h \cdot x_i, v_6$ (lines 3,6 in Pseudocode 3.3). The last considered *Input* agent synchronizes with agent *Monitor* with a communication labelled as $search.h \cdot \varepsilon \cdot h \cdot v_1$ (line 3 in Pseudocode 3.2 and line 7 in Pseudocode 3.3). This removes the high priority token and instantiates the low priority one. All residuals of *Input* agents, which are at this point instantiated as a service congruent to s_i^*, where

$$\begin{aligned} s_i^* \equiv\ & [\,kin\,][\,myid\,](search.l\,?\,\langle l\rangle\,.(\ldots) \\ & |(hard.r\,?\,\langle r\rangle\,.(\ldots) \\ & +[\,prio\,]soft.r\,?\,\langle myid, prio\rangle\,.(\ldots))); \end{aligned}$$

The residual of service $[\![P]\!]$, at this point, can only continue by performing n times the sequence composed of a communication on endpoint $search.l$ (line 4 in Pseudocode 3.2 and lines 8, 12 in Pseudocode 3.3), labelled as $search.l \cdot \varepsilon \cdot l \cdot l$, which activates one of the *Input* residuals, followed by a synchronization internal to the activated *Input* agent on endpoint $x_i.l$ labelled as $x_i.l \cdot \varepsilon \cdot x_i, l \cdot x_i, v_7$ (lines 9, 12 in Pseudocode 3.3). As before, the possibility of having a communication on endpoint $x_i.l$ involving no substitution (which would take precedence) is denied by the lack of any matching output action in the encoded process.

The last synchronization on endpoint $search.l$ is between the last activated *Input* agent and the residual of agent *Monitor* (line 5 in Pseudocode 3.2 and line 12 in Pseudocode 3.3). After this, the residual of agent *Monitor* is composed by the instantiation of agent *CBlock*. If we call s_a the residual of service $[\![P]\!]$ after the last possible synchronization on endpoint $search.l$, we have that s_a can only evolve performing n times a sequence of synchronizations over endpoint $hard.r$ (line 21 in Pseudocode 3.2 and lines 14, 16 in Pseudocode 3.3). After each synchronization, an *Input* agent is reset to its initial state. The last synchronization on endpoint $hard.r$ is performed

between the last resetted *Input* agent and the residual of agent *Monitor*, involving one substitution (line 16 in Pseudocode 3.2 and line 16 in Pseudocode 3.3). After it is performed, the obtained residual is congruent to service $[\,h, l, r\,](\!|P|\!)_r^{h\,l}$. Since agent *Monitor* is not present, the high priority token (in the form of the unguarded invoke action on endpoint *search.h*) is not reinstantiated, so the process is in a deadlock state.

□

Appendix B

SCOWS_lts Models

B.1 Mail BPMN Case Study

```
Discussion(firsttime, alreadywarned) =
[issueList]
[k_disc]
  (identify#.issues#?<issueList>,1.0).(
    (announce#.issues#!<issueList>,modResponsiveness)
    | (announce#.issues#?<issueList>,modResponsiveness).(
      (moderate#.discussion#!<>,modResponsiveness)
      |(moderate#.discussion#?<>,modResponsiveness).(
        (discussion#.token#!<>,1.0)
      )
      |(warn#.deadline#!<issueList>,modResponsiveness)
      |(warn#.deadline#?<issueList>,modResponsiveness).(
        (deadline#.token#!<>,1.0)
      )
      |[date#](check#.confCall#!<date#>,1.0)
      |[week](
        (check#.confCall#?<week>,1.0).(
          //yes
```

```
        (moderate#.confCall#!<week>,modResponsiveness)
        |(moderate#.confCall#?<week>,modResponsiveness).(
          (confCall#.token#!<>,1.0)
        )
      )
      +(check#.confCall#?<week>,1.0).(
        //no
        (confCall#.token#!<>,1.0)
      )
    )
  )
  |(deadline#.token#?<>,1.0).(confCall#.token#?<>,1.0)
   .(discussion#.token#?<>,1.0).(
    (eval#.discussion#!<issueList>,modResponsiveness)
    |(
      (eval#.discussion#?<issueList>,modResponsiveness).(
        //ready
        (announce#.vote#!<issueList>,1.0)
        | BetweenDiscAndVote(firsttime,alreadywarned)
      )
      +(eval#.discussion#?<issueList>,modResponsiveness).(
        //not ready
        (kill(k_disc),1.0)
        |
        {
          (identify#.issues#!<issueList>,1.0)
          | Discussion(firsttime, alreadywarned)
        }
      )
```

```
      )
    )
  )
;

BetweenDiscAndVote(firsttime, alreadywarned)=
[iL](announce#.vote#?<iL>,1.0).(
  (start#.vote#!<iL>,1.0)
  |Voting(firsttime, alreadywarned)
)
;

CollectVote()=
nil
;

UpdateVotes()=
nil
;

CheckCalendar()=
[wk#](check#.week#!<wk#>,1.0)
|(
  [w1](check#.week#?<w1>,1.0).(
    //cc in voting week? NO
    CheckCalendar()
  )
  +[w2](check#.week#?<w2>,1.0).(
    //cc in voting week? YES
```

```
    (moderate#.cc#!<>,1.0)
  )
)
;

ModerateMailDiscussion()=
(moderate#.maildisc#!<>,1.0)
;

DeadlineWarning()=
[warningtext#]
(dispatch#.deadlinewarning#!<warningtext#>,1.0)
;

EnoughVotes(firsttime, alreadywarned, k_voting)=
(reached#.majority#!<>,1.0)
|(
  //majority? no
  (reached#.majority#?<>,minorityRate).(
    (outer#.loop#!<firsttime>,1.0)
    |(
      (outer#.loop#?<true>,1.0).(
        [newsolutions#](reduce#.solutions#!<newsolutions#>
                        ,modResponsiveness)
        | [sol](reduce#.solutions#?<sol>,modResponsiveness).(
          (kill(k_voting),1.0)
          |
          {
            (mail#.voters#!<sol>,1.0)
```

```
            | BetweenDiscAndVote(false, alreadywarned)
            | [newissues#](announce#.vote#!<newissues#>,1.0)
          }
        )
      )
      +(outer#.loop#?<false>,1.0).(
        //restart discussion
        (kill(k_voting),1.0)
        |
        {
          [new_issues#](identify#.issues#!<new_issues#>,1.0)
          |Discussion(firsttime, alreadywarned)
        }
      )
    )
  )
  //majority? yes
  + (reached#.majority#?<>,majorityRate).nil
)
;

ElabResults(firsttime, alreadywarned, k_voting)=
(moderate#.cc#?<>,1.0).
(moderate#.maildisc#?<>,1.0).
[warn](dispatch#.deadlinewarning#?<warn>,1.0).
(
  [result#](prepare#.results#!<result#>,modResponsiveness)
  | [res](prepare#.results#?<res>,modResponsiveness).(
    (post#.results#!<res>,modResponsiveness)
```

```
    |(post#.results#?<res>,modResponsiveness).
      (eval#.votes1#!<>,1.0)
    |(mail#.results#!<res>,modResponsiveness)
    |(mail#.results#?<res>,modResponsiveness).
      (eval#.votes2#!<>,1.0)
  )
  |[vt#](
    (stop#.votes#!<vt#>,1.0)
    |(stopped#.vt#?<>,1.0).(eval#.votes1#?<>,1.0)
     .(eval#.votes2#?<>,1.0).(
      [nbvotes#](enough#.votes#!<nbvotes#>,1.0)
      |
      (
        //enough? no
        [nv1](enough#.votes#?<nv1>,1.0).(
          (members#.warned#!<alreadywarned>,modResponsiveness)
          |(
            (members#.warned#?<true>,modResponsiveness).(
              (reduce#.votes#!<>,1.0)
              | (reduce#.votes#?<>,1.0).
                EnoughVotes(firsttime, true, k_voting)
            )
            +(members#.warned#?<false>,modResponsiveness).(
              [wmsg#](issue#.warning#!<wmsg#>,1.0)
              |[msg](issue#.warning#?<msg>,1.0).(
                (kill(k_voting),1.0)
                |
                {
                  [adjustedissues#](
```

```
                    (start#.vote#!<adjustedissues#>,1.0))
                    | Voting(firsttime, true)
                  }
                )
              )
            )
          )
          //enough? yes
          +
          [nv2](enough#.votes#?<nv2>,1.0).(
            EnoughVotes(firsttime, alreadywarned, k_voting)
          )
        )
      )
    )
)
;

Voter()=
  [vote](stop#.votes#?<vote>,1.0).(
    (stopped#.vote!<>,1.0)
    | Voter()
  )
  + [solutions](mail#.voters#?<solutions>,1.0).Voter()
;

Voting(firsttime, alreadywarned) =
[k_voting]
[issueList]
```

```
(start#.vote#?<issueList>,1.0).(
  CheckCalendar()
  |ModerateMailDiscussion()
  |DeadlineWarning()
  |CollectVote()
  |UpdateVotes()
  |ElabResults(firsttime, alreadywarned, k_voting)
)
;

$

fixed;
[list#](
(review#.issues#!<list#>,1.0)
| (
  //go to Discussion Cycle
  (review#.issues#?<list#>,1.0).(
    Discussion(true, false)
    | (identify#.issues#!<list#>,1.0)
  )
)
| Voter()
)

$
$

//stopping vote
```

```
stop#.votes# <*> : stopc < 100 : stopc'=(stopc+1);

//warning users about the need for their vote.
issue#.warning# <*> : warnc < 15 : warnc'=(warnc+1);

//reissuing voting.
outer#.loop#<true> : revotec < 15 : revotec'=(revotec+1);

//reissuing discussion.
outer#.loop#<false> : redisc < 15 : redisc'=(redisc+1);
```

B.2 Dekker's Algorithm

```
CGv(v,val)=
  [pr](need#.get#?<v, pr>,1.0).(
    {(get#.v!<val, pr>,1.0)}
    |  CGv(v,val)
  )
;

CSv(v, k_v) =
  [pr][val](set#.v?<val, pr>,1.0).(
    (kill(k_v),1.0)
    | {
      (ok#.v!<val, pr>,1.0)
      | [k_new_v](
        CSv(v,k_new_v)
        | CGv(v, val)
```

```
      )
    }
  )
;

Controller(f0, f1, t, zero, one, token) =
  [k_f0](
    CSv(f0, k_f0)
    | CGv(f0, zero)
  )
  | [k_f1](
    CSv(f1, k_f1)
    | CGv(f1, zero)
  )
  | [k_t](
    CSv(t, k_t)
    | CGv(t, zero)
  )
;

ProcessBusyWait(zero, one, x, notx, turn, token, proceed) =
  [check#]
  (
    (need#.get#!<turn, x>,1.0)
    | [lt](get#.turn?<lt,x >,1.0).(
       (check#.local#!<lt>,bwr)
      | (
        (check#.local#?<notx>,bwr).(
          ProcessBusyWait(zero, one, x, notx, turn, token
```

```
                          , proceed)
        )
        +
        (check#.local#?<x>,bwr).(
          (can#.proceed!<x>,1.0)
        )
      )
    )
  )

;

ProcessWhileFlag(zero, one, x, notx, flagx, flagnotx, turn
                 , token) =
  [check#][lfnotx]
  (
    (need#.get#!<flagnotx, x>,1.0)
    | (get#.flagnotx?<lfnotx, x>,1.0).(
      (check#.local#!<lfnotx>,1.0)
      | (
        (check#.local#?<one>,1.0).(
          (
            (need#.get#!<turn, x>,1.0)
            | [lt](get#.turn?<lt,x >,1.0).(
              (check#.local#!<lt>,1.0)
              | (
                (check#.local#?<x>,1.0).(
                  ProcessWhileFlag(zero, one, x, notx
                           , flagx, flagnotx, turn, token)
```

```
          )
          +(check#.local#?<notx>,1.0).(
            (
              (set#.flagx!<zero,x >,1.0)
              | (ok#.flagx?<zero, x>,1.0).(
                [proceed#](
                  ProcessBusyWait(zero, one, x, notx
                            , turn, token, proceed#)
                  | (can#.proceed#?<x>,1.0).(
                    (
                      (set#.flagx!<one, x>,1.0)
                      | (ok#.flagx?<one, x>,1.0).(
                        ProcessWhileFlag(zero, one
                        , x, notx, flagx, flagnotx
                        , turn, token)
                      )
                    )
                  )
                )
              )
            )
          )
        )
      )
    )
  )
 )
+(check#.local#?<zero>,1.0).(
  (is#.crit#!<x>,1.0)
)
```

```
      )
    )
  )
;

Process(zero, one, x, notx, flagx, flagnotx, turn, token) =
  (
    (set#.flagx!<one,x>,1.0)
    | (ok#.flagx?<one,x>,1.0).(
      ProcessWhileFlag(zero, one, x, notx, flagx, flagnotx
                       , turn, token)
      | (is#.crit#?<x>,1.0).(
        (is#.noncrit#!<x>,1.0)
        | (is#.noncrit#?<x>,1.0).(
          (
            (set#.turn!<notx,x>,1.0)
            | (ok#.turn?<notx,x>,1.0).(
              (
                (set#.flagx!<zero,x>,1.0)
                | (ok#.flagx?<zero,x>,1.0).(
                  nil
                )
              )
            )
          )
        )
      )
    )
  )
```

```
;


$

fixed;


[flag1#][flag2#][turn#]



[proc1#][proc2#]

[t#](
  Process(proc1#, proc2#, proc1#, proc2#, flag1#, flag2#
          , turn#, t#)
  |
  Process(proc1#, proc2#, proc2#, proc1#, flag2#, flag1#
          , turn#, t#)
  | Controller(flag1#, flag2#, turn#, proc1#, proc2#, t#)
)

$

$
is#.crit# < @1 > : @1_crit < 2 : (@1_crit'=1);
is#.noncrit# < @1 > : @1_crit < 2 : (@1_crit'=2);
```

B.3 Djikstra's Algorithm

```
CGv(v,value)=
  [process](need#.get#?<v, process>,1.0).(
    {(get#.v!<value, process>,1.0)}
    |  CGv(v,value)
  )
;

CSv(v, k_v) =
  [process][value](set#.v?<value, process>,1.0).(
    (kill(k_v),1.0)
    | {
      (ok#.v!<value, process>,1.0)
      | [k_new_v](
        CSv(v,k_new_v)
        | CGv(v, value)
      )
    }
  )
;


Controller(f0, f1, t,   token) =

  [flag_f0](
    CSv(f0, flag_f0)
    | CGv(f0, zero)
  )
```

```
  | [flag_f1](
    CSv(f1, flag_f1)
    | CGv(f1, zero)
  )
  | [flag_t](
    CSv(t, flag_t)
    | CGv(t, zero)
  )
;


CS(n,pid, procname,  b, c, b_of_i, c_of_i, flag) =
  (is#.crit#!<procname>,1.0)
  |(is#.crit#?<procname>,1.0).(
    (is#.noncrit#!<procname>,1.0)
    |(is#.noncrit#?<procname>,1.0).(

      (set#.c_of_i!<true,pid>,1.0)
      |(ok#.c_of_i?<true,pid>,1.0).(



        (set#.b_of_i!<true,pid>,1.0)
        |(ok#.b_of_i?<true,pid>,1.0).(

          nil
        )
      )
    )
```

```
  )
;

ForCycle(n,stop,pid,procname,j,b,c,b_of_i,c_of_i,c_of_j
         ,flag)=
  (check1#.check1#!<pid,j>,1.0)
  |(
    [otherj](check1#.check1#?<pid,otherj>,1.0).(
      (need#.get#!<c_of_j,pid>,1.0)
      |[mycj](get#.c_of_j?<mycj,pid>,1.0).(

        (check2#.check2#!<pid,( (! (j=pid)) & ( !(mycj)))>
                        ,bwr)
        |(
          (check2#.check2#?<pid,true>,bwr).(
            ProcessL1(n,pid,procname,b,c,b_of_i,c_of_i
                      ,flag)
          )
          +(check2#.check2#?<pid,false>,bwr).(
            ForCycle(n,stop,pid,procname,(j+1),b,c,b_of_i
                     ,c_of_i,(c+(j+1)),flag)
          )
        )
      )
    )
    +(check1#.check1#?<pid,stop>,1.0).(
      CS(n,pid,procname,b,c,b_of_i,c_of_i,flag)
    )
  )
```

```
;

ProcessL4(n,pid,procname,b,c,b_of_i,c_of_i,flag)=

  (set#.c_of_i!<false,pid>,1.0)
  |(ok#.c_of_i?<false,pid>,1.0).(
    ForCycle(n,(n+1),pid,procname,1,b,c,b_of_i,c_of_i
             ,(c+1),flag)
  )
;

ProcessL1(n, pid, procname,  b, c, b_of_i, c_of_i, flag)=
  (need#.get#!<flag, pid>,1.0)
  |[myflag](get#.flag?<myflag,pid>,1.0).(
    [b_of_flag](

      (match1#.match1#!<(!(myflag=pid)), (b+myflag),pid>,1.0)

      |(
        (match1#.match1#?<true, b_of_flag,pid>,1.0).(

          (set#.c_of_i!<true,pid>,1.0)
          |(ok#.c_of_i?<true,pid>,1.0).(

            (need#.get#!<b_of_flag,pid>,1.0)
            |[b_of_flag_value]
             (get#.b_of_flag?<b_of_flag_value,pid>,1.0).(
              (match2#.match2#!<pid,b_of_flag_value>,1.0)
```

```
                |(
                  (match2#.match2#?<pid,true>,1.0).(

                    (set#.flag!<pid,pid>,1.0)
                    |(ok#.flag?<pid,pid>,1.0).(
                      ProcessL1(n,pid,procname,b,c,b_of_i
                                ,c_of_i,flag)
                    )
                  )

                  +(match2#.match2#?<pid,false>,1.0).(
                    ProcessL1(n,pid,procname,b,c,b_of_i
                              ,c_of_i,flag)
                  )
                )
              )
            )
          )

          +(match1#.match1#?<false, b_of_flag,pid>,1.0).(
            ProcessL4(n,pid,procname,b,c,b_of_i,c_of_i,flag)
          )
        )
      )
    )
  )
;

Go() =
  [proc](operation#.ready#?<proc>,1.0).(
```

```
    (operation#.go#!<proc>,1.0)
    | Go()
  )
;

Process(n,pid, procname,  b, c, b_of_i, c_of_i, flag) =

  (operation#.ready#!<pid>,1.0)
  |(operation#.go#?<pid>,1.0).(


    (set#.b_of_i!<false, pid>,1.0)
    |(ok#.b_of_i?<false, pid>,1.0).(
      ProcessL1(n,pid,procname,b,c,b_of_i,c_of_i,flag)
    )
  )
;


SpawnProcesses(maxprocidx, m, pid, procbase,b, c,flag)=
  [cS#](
    (cS#.cS#!<pid>,1.0)
    |(
      [other](cS#.cS#?<other>,1.0).(
        Process(maxprocidx,pid,(procbase+pid),b,c,(b+pid)
        ,(c+pid),flag)
        |SpawnProcesses(maxprocidx, m,(pid+1),procbase,b
                        ,c,flag)
      )
```

```
      +(cS#.cS#?<m>,1.0).(
        Go()
      )
    )
  )
;

InitVector(v,index,max_val, val) =
  [k_proc](
    CSv(v+index, k_proc)
    | CGv(v+index, val)
  )
  |(
    [checkflag1#][val#](
      (checkflag1#.val#!<index>,1.0)
      |(
        (checkflag1#.val#?<max_val>,1.0).
          (finished#.initvector#!<v,(max_val-1)>,1.0)
        + [otherIdx](checkflag1#.val#?<otherIdx>,1.0).
          InitVector(v,index+1,max_val, val)
      )
    )
  )
;
Init(n,b,c,flag) =
  [k_flag](
    CSv(flag,k_flag)
    | CGv(flag, 1)
  )
```

```
  | InitVector(b, 1, n+1, true)
  | InitVector(c, 1, n+1, true)
;

$
fixed;

[proc#][b#][c#][flag#](
  (set#.maxProcNmb#!<2>,1.0)
  |[mpn](set#.maxProcNmb#?<mpn>,1.0).(
    Init(mpn,b#,c#,flag#)
    | (finished#.initvector#?<b#,mpn>,1.0).
      (finished#.initvector#?<c#,mpn>,1.0).
        SpawnProcesses(mpn,(mpn+1), 1, proc#, b#, c#
                       , flag#)
  )
)

$

$

is#.crit#<@1>:@1_crit < 2 : (@1_crit'=1);
is#.noncrit#<@1>:@1_crit < 2 : (@1_crit'=2);
```

B.4 Lamport's Bakery Algorithm

```
CGv(v,value)=
  [process](need#.get#?<v, process>,1.0).(
    {(get#.v!<value, process>,1.0)}
    |  CGv(v,value)
  )
;

CSv(v, k_v) =


  [process][value](set#.v?<value, process>,1.0).(
    (kill(k_v),1.0)
    | {
      (ok#.v!<value, process>,1.0)
      | [k_new_v](
        CSv(v,k_new_v)
        | CGv(v, value)
      )
    }
  )
;

Go() =
  [proc](operation#.ready#?<proc>,1.0).(
    (operation#.go#!<proc>,1.0)
    | Go()
```

```
  )
;


InitVector(v,index,max_val, val) =
  [k_proc](
    CSv(v+index, k_proc)
    | CGv(v+index, val)
  )
  |(
    [check1#][max_val#](
      (check1#.max_val#!<index>,1.0)
      |(
        (check1#.max_val#?<max_val>,1.0).
          (finished#.initvector#!<v,max_val>,1.0)
        + [otherIdx](check1#.max_val#?<otherIdx>,1.0).
          InitVector(v,index+1,max_val, val)
      )
    )
  )
;

Init(b, nmb, n)=
  InitVector(b, 1, n, false)
  | InitVector(nmb, 1, n, 0)
;


SpawnProcesses(b, nmb, procbase, idx, n,  token)=
```

```
  [check2#][max_val#](
    (check2#.max_val#!<idx>,1.0)
    |(
      (check2#.max_val#?<n>,1.0).(
        ProcessPhase1(n, b,nmb, (procbase+idx), idx
                      , (b+idx),(nmb+idx), token)

        | Go()
      )
      + [otherIdx](check2#.max_val#?<otherIdx>,1.0).(
        ProcessPhase1(n, b, nmb, (procbase+idx), idx
                      , (b+idx), (nmb+idx), token)
        | SpawnProcesses(b, nmb, procbase, (idx+1), n
                         , token)
      )
    )
  )
;

GetMaxRec(v,it,idx,v_of_it,max_it,var_max)=
  [new_max](

    (need#.get#!<v_of_it, idx>,1.0)
    | [theVal](get#.v_of_it?<theVal, idx>,1.0).(

      (update#.local_max#!<max(var_max, theVal)>,1.0)
      | (update#.local_max#?<new_max>,1.0).(
        [iterate#][index#](
          (iterate#.index#!<it>,1.0)
```

```
          | (
            [vi](iterate#.index#?<vi>,1.0).(
              GetMaxRec(v, it+1, idx, (v+(it+1)), max_it
                        , new_max)
            )
            +
            (iterate#.index#?<max_it>,1.0).(
              (return#.max#!<1+new_max,idx>,1.0)
            )
          )
        )
      )
    )
  )
;


Wait1(procname, idx,b_of_q, token) =
[lW1#]
(
  (need#.get#!<b_of_q, idx>,1.0)
  | [theVal](get#.b_of_q?<theVal, idx>,1.0).(
    (lW1#.procname!<theVal>,bwr)
    | (
      (lW1#.procname?<true>,bwr).(
        Wait1(procname,idx,b_of_q, token)
      )
      +
      (lW1#.procname?<false>,bwr).(
```

```
        (notify#.procname!<1>,1.0)
      )
    )
  )
)
;


Wait2(procname, idx, q, nmb_of_i, nmb_of_q,  token) =
[lW2#]
  (
  need#.get#!<nmb_of_i, idx>,1.0)
  |[v_nmb_of_i](get#.nmb_of_i?<v_nmb_of_i, idx>,1.0).(
    (need#.get#!<nmb_of_q, idx>,1.0)
    | [v_nmb_of_q](get#.nmb_of_q?<v_nmb_of_q, idx>,1.0).(
      (lW2#.procname!<( (!(v_nmb_of_q = 0))
                        & ((v_nmb_of_q < v_nmb_of_i)
                           | ((v_nmb_of_q = v_nmb_of_i)
                              & (q < idx))) )>,bwr)
      | (
        (lW2#.procname?<true>,bwr).(
          Wait2(procname, idx, q, nmb_of_i, nmb_of_q
                , token)
        )
        +
        (lW2#.procname?<false>,bwr).(
          (notify#.procname!<2>,1.0)
        )
      )
```

```
    )
  )
;

ProcessPhase2Rec(n,np1, b, nmb, procname, idx, q, b_of_i
                  , nmb_of_i,  token)=
[local#](
  (local#.iterate#!<q>,1.0)
  | (
    [v](local#.iterate#?<v>,1.0).(
      Wait1(procname, idx,(b+q), token)
      | (notify#.procname?<1>,1.0).(
        Wait2(procname, idx, q, nmb_of_i, (nmb+q)
              , token)
        | (notify#.procname?<2>,1.0).(
          ProcessPhase2Rec(n, np1, b, nmb, procname, idx
                           , (q+1), b_of_i, nmb_of_i, token)
        )
      )
    )
    +
    (local#.iterate#?<np1>,1.0).(
      (is#.crit#!<procname>,1.0)
      | (is#.crit#?<procname>,1.0).(
        (is#.noncrit#!<procname>,1.0)
        | (is#.noncrit#?<procname>,1.0).(
            (set#.nmb_of_i!<0, idx>,1.0)
            | (ok#.nmb_of_i?<0, idx>,1.0).(nil)
          )
```

```
        )
      )
    )
  )

;

ProcessPhase2(n, b,nmb, procname, idx, b_of_i,nmb_of_i
              , token) =
  ProcessPhase2Rec(n, (n+1),b, nmb, procname, idx, 1
                   , b_of_i, nmb_of_i,  token)
;

GetMax(v, idx, max_it)=
  GetMaxRec(v, 1, idx, (v+1), max_it, 0)
;

ProcessPhase1(n, b, nmb, procname, idx, b_of_i, nmb_of_i
              , token)=
  (operation#.ready#!<idx>,1.0)
  |(operation#.go#?<idx>,1.0).(
    (set#.b_of_i!<true, idx>,1.0)
    | (ok#.b_of_i?<true, idx>,1.0).(

      GetMax(nmb, idx, n)
      | [mynumber](return#.max#?<mynumber,idx>,1.0).(

        (set#.nmb_of_i!<mynumber, idx>,1.0)
        | (ok#.nmb_of_i?<mynumber, idx>,1.0).(
```

```
          (set#.b_of_i!<false, idx>,1.0)
          | (ok#.b_of_i?<false, idx>,1.0).(
            ProcessPhase2(n, b,nmb, procname, idx, b_of_i
                          , nmb_of_i,  token)
          )
        )
      )
    )
  )
;

$

fixed;
[n][nmb#][b#][proc#][t#](
  (set#.maxProcIdx#!<2>,1.0)
  |(set#.maxProcIdx#?<n>,1.0).(
    Init(b#, nmb#, n)
    |
    (finished#.initvector#?<b#, n>,1.0).
      (finished#.initvector#?<nmb#, n>,1.0).
        SpawnProcesses(b#, nmb#, proc#, 1, n, t#)
  )
)
$
$
is#.crit# < @1 > : @1_crit < 2 : (@1_crit'=1);
is#.noncrit# < @1 > : @1_crit < 2 : (@1_crit'=2);
```