Implementing Stochastically-Timed COWS for the Prism Probabilistic Model Checker.

Michele Sevegnani

Master of Science
School of Informatics
University of Edinburgh
2008

Abstract

This thesis presents the implementation of the COWS2Prism system, a compiler for stochastic COWS (Calculus for Orchestration of Web Services) into Prism. The process calculus COWS is intended to aid in the precise description of Web Services compositions. This places the present informal development approach associated with Web Services into a formal reasoning framework. The COWS calculus concerns itself with behavioural aspects although two timed extensions exist: Stochastic COWS and Timed COWS (for reasoning about non-functional aspects such as quality of service). In the dissertation we provide a syntax and a semantics for stochastic COWS, and then a detailed description of the design and implementation of the COWS2Prism system. We also give a well commented example of compilation of a COWS service into the Prism format.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Michele Sevegnani*)

To my family and all my teachers.

Table of Contents

Chapter 1

Introduction

In recent years, the increasing success of e-business, e-learning, e-government, and other similar emerging models, has led the World Wide Web, initially thought of as a system for human use, to evolve towards an architecture for service-oriented computing (SOC) supporting automated use. SOC advocates the use of loosely coupled 'services', to be understood as autonomous, platform-independent, computational entities that can be described, published, discovered, and assembled, as the basic blocks for building interoperable and evolvable applications. Current software engineering technologies for SOC, however, remain at the descriptive level and lack rigorous formal foundations. Many researchers have therefore put forward the idea of using process calculi that, due to their algebraic nature, convey in a distilled form the compositional programming style of SOC.

COWS (Calculus for Orchestration of Web Services) is a foundational language for specifying and combining service-oriented systems whose design has been influenced by WS-BPEL, the OASIS standard language for orchestration of web services. COWS combines in an original way a number of ingredients borrowed from well-known process calculi, e.g. asynchronous communication, polyadic synchronisation, pattern-matching, protection, delimited receiving and killing activities, while remaining different from any of them. The principles which have driven the design of COWS take inspiration from WS-calculus, a process language introduced in a previous work (Lapadula et al., 2006). COWS has proved to be sufficiently expressive both for modelling imperative and orchestration constructs, e.g. web services, flow graphs, fault and compensation handlers, and for encoding other process and orchestration

languages, e.g. Localised π-calculus, Orc, WS-calculus. Since its introduction, some mild linguistic extensions have been proposed to model timed activities, service discovery and service negotiation. Moreover, a number of methods and tools have been devised to analyse COWS specifications, such as a type system to check confidentiality properties, a stochastic extension to enable quantitative reasoning on service behaviours, and a logic and a model checker to express and check functional properties of services.

This dissertation presents a complete overview of the COWS2Prism system, an implementation of stochastic COWS into Prism. It consists of a compiler for models specified in High-Lan COWS, a functional-like language closing resembling the original COWS syntax. The corresponding output is the definition of a Continuous Time Markov Chain for the Prism probabilistic model checker. The following sections summarise the contents of this dissertation.

1.1 Background

Several proposals for the modelling of the primitives needed in the SOC paradigm have appeared in the literature. In Chapter 2, we briefly survey the most important of them, mainly focusing on process calculi based approaches. The related concepts of communication, synchronisation, concurrency and the notion of transition system are introduced. We then shift our attention to stochastic extensions of process calculi, highlighting the motivations leading to their development, and their peculiar characteristics, such as, for instance the concept of race condition. We specifically comment on COWS and stochastic COWS, providing a syntax and a labelled operational semantics for both of them. We also discuss the importance of the stochastic model checker Prism for the quantitative analysis of probabilistic systems modelling SOC scenarios. Moreover, the possible links with stochastic COWS are invedtigated. Finally, we list some related works.

1.2 COWS2Prism system design

Stochastic COWS is extremely appealing for implementation, thanks to its powerful primitives such as kill activity and protection, and the cleanness of the axioms and rules in the semantics. As a matter of fact, an implementation

can straightforwardly be obtained from the detailed information specifying the evolution of the modelled system provided by the labels in the semantic rules. In Chapter 3 we describe the design of a compiler capable of computing the transition system of a COWS service and translating it into a continuous-time Markov chain. Particular care is taken in the formal exposure of the functions and encodings corresponding to the conceptual sub-units of the compiler, i.e. fresh names generation, type inference and translation. The compiler is defined for specifications in a simplified version of stochastic COWS, called High-Lan COWS.

1.3 COWS2Prism system implementation

In Chapter 4 we comment on an implementation of the COWS2Prism system in the functional language OCaml. In particular, we describe how the implementation resembles the COWS2Prism system theoretical definition, and we analyse the advantages offered by the `ocamllex` and `ocamlyacc` tools provided by the standard OCaml distribution. Moreover, issues regarding computational complexity are considered. Where the implementation was more challenging, extracts of the source code are reported, in order to better explain our implementation choices. We also mention the data structures used in the various sub-units of the system. Chapter 5 presents an example showing the COWS2Prism system at work. Finally, Chapter 6 is devoted to the conclusion and future work.

Chapter 2

Background

Service Oriented Computing (SOC) paradigm is beginning to emerge as a widely accepted model for integrating disparate applications and systems. In particular, its most successful current realisation based on Web Services, is gaining popularity mainly because of its extraordinary interoperability characteristics. The key of this success has been the use of a publish-find-bind model based on open XML standards. The description of a web service is typically given in a WSDL document. WSDL is a XML-based language made for representing, in an abstract and structured way, the operations that a web service can execute. The information about the providers that give an implementation of a particular type of web service is stored in a UDDI service registry. After obtaining the description of the web service, the user can send requests to it, usually using SOAP messages over HTTP protocol. A more detailed introduction to these topics can be found in (Cerami, 2002). The widespread deployment of networked applications of this kind and adoption of the Internet has fostered an environment in which many distributed services are available. More and more often, SOC systems deliver application functionality as services to other services rather than services to end-user applications. As a result, there is a great demand for the automation of business processes and workflows among organisations and individuals, in order to take advantage of the opportunities of reusability and service composition offered by the SOC paradigm. Indeed, processes being built today need the business agility to quickly adapt to customer needs and market conditions. This would include incorporating new customers, partners, or suppliers used in a process. Solutions to such problems require *service composition* of concurrent and distributed services in the

face of arbitrary delays and failures of components and communications. The difference with respect to classic program or process composition is that the composed services are not statically designed, but on the contrary, they are constructed dynamically in terms of discovering the other services they need to include. In other words, the service paradigm provides the capabilities for dynamic run-time composition rather than requesting a statically planned architecture. Consider for instance the following wide-area computing problem presented in (Misra and Cook, 2006):

> A client contacts two airlines simultaneously for price quotes. He buys a ticket from either airline if its quoted price is no more than $300, the cheapest ticket if both quotes are above $300, and any ticket if the other airline does not provide a timely quote. The client should receive an indication if neither airline provides a timely quote.

As can be seen, the computational pattern involves the acquisition of data from one or more remote services, calculation with these data, and invocation of yet other remote services with the results. Notice also that these primitive operations are intrinsically part of the service composition paradigm. Nowadays, two terms are used to indicate composition of services: *orchestration* and *choreography*. Orchestration is about describing and executing a single view point model, whereas choreography is about specifying and guiding a global model. Though the difference between the two terms can be sometimes abused or blurred, substantially orchestration has a more centralised flavour, as opposed to the more distributed vision of choreography. Orchestration paradigms can be roughly categorised into three trends:

- technology-driven languages: all XML dialects and standardisation efforts (e.g. WS-BPEL (Andrews et al., 2003), XLANG (Thatte, 2001));
- model oriented: workflow aspects are prominent (e.g. Petri nets (Reisig, 1986; Peterson, 1981), YAWL (van der Aalst and Hofstede, 2002));
- process algebraic or messaging-based: the orchestration is ruled by communication primitives (e.g. CCS (Milner, 1980), π-calculus (Milner, 1999), Join-calculus (Fournet and Gonthier, 1996), Orc(Misra and Cook, 2006), and more recently COWS (Lapadula et al., 2007a)).

In the last few years, many researchers have exploited the studies on process calculi as a starting point to define a clean semantic model and lay rigorous methodological foundations for service-based applications and their composition.

2.1 Process Calculi

Despite their wide adoption, all the XML languages such as WS-BPEL are not provided with formal semantics, although they do have detailed (and often cumbersome) informal specifications. Even though it seems a problem only inside the academia, the consequences of this absence can be felt every day in the real word as costs for the organisations using and developing SOC systems. Indeed, as we learnt from Software Engineering, every flaw in the design and every delay in the debugging of the system are paid for in terms of monetary losses by the organisations. Therefore, formalisms that facilitate the modelling and simulation of the system (hopefully before it is physically deployed) and formal verification of the interactions among sites involved in the orchestration task should be received well among business organisations. For instance, process calculi permit us to implement a model of the system and perform both quantitative and qualitative analysis on it. As a consequence, it is easy to check if some desirable properties (e.g., liveness and fairness) are satisfied, even before the real system has been implemented. Clearly, this is a significant advantage over the technology-driven languages and it explains the effort of the research in this field in the last years.

Robin Milner developed his Calculus of Communicating Systems (CCS) over the years 1973 to 1980. The primitives and the concepts behind CCS served as foundations for most of the following theories. Indeed, the CCS approach to synchronisation of processes over complementary names and the restriction operator can widely be identified in most of the current calculi. A further important merit of CCS is that it served as the base for the development of the π-calculus: the first process calculi which introduced the concept of mobility. We will come back later to π-calculus.

Tony Hoare invented the language CSP (Communication Sequential Processes) (Hoare, 1985). In this case, the synchronisation paradigm is completely different from the one of CCS: Two processes can interact only if they are both

capable of actions on the same name. Moreover, the hiding construct has no counterpart in CCS. We will see later how CSP has strongly influenced the design of PEPA (Hillston and Thomas, 1998).

CCS lacks in representing mobile processes where the network topology dynamically changes. A first attempt to tackle this issue was an extension of CCS where names could be exchanged among processes. A further refinement has been carried out by Milner, Parrow and Walker in (Milner et al., 1992), where the π-calculus was proposed. The practical usefulness of the calculus has been demonstrated in application studies on mobile telecommunication networks and high speed networks. As a matter of fact, communication links are identified by names, and computation is represented purely as the communication of names across links. The combination of name communication and scope extrusion (namely the ability to dynamically change the scope of names) is the essential difference between the π-calculus and earlier processes calculi. These features, as said before, confer mobility (i.e. capability of changing interconnection topology) to the calculus as well as great expressiveness. As a consequence, several computational paradigms have been shown to be encodable in π-calculus. For instance, Milner (Milner, 1992) exhibited an encoding of the π-calculus in the λ-calculus.

More recently, several new formalisms have been proposed. Some prominent examples are the fusion calculus (Parrow and Victor, 1998), the ambient calculus (Cardelli and Gordon, 1998) and the Spi-calculus (Abadi and Gordon, 1997). The trend shows that new algebras tend to focus on a particular domain of application. For instance, bio-inspired languages (such as Brane-calculus, Beta-binders, Bio-ambients) to model biological phenomena or calculi expressly conceived to represent web-services interactions. In this last category we find, among the others, Orc, COWS, SCC.

Some central concepts are shared among all the calculi. One of the most important is the notion of transition system. A transition system is essentially a graph which can be syntactically derived from a term in a given calculus. The nodes stand for reachable states of the system, while the edges encode the action the system has to perform in order to pass from a state to the other. Intuitively, it captures all the possible behaviours of a system. In order to derive the transition system, some rules have to be specified so that only valid moves are allowed. This is accomplished by a set of axiom in the semantics of

the calculus which describe how a term can evolve to another term. Observe that a process calculus can have several semantics and consequently, several underlying transition systems. A basic classification of transition systems is built on the different type of semantics: Labelled or unlabelled.

Additional crucial concepts are the notions of bisimulation. Those relations are based on the requirement that any move of a certain process has to be matched by an analogous move of the bisimilar process. It is often possible to define a bisimulation relation between two processes in terms of graph theoretic properties. Hence, it can be verified by checking if some properties hold on the corresponding transition systems. It is worthwhile to highlight that the definition of bisimulation relations over processes in calculi equipped with mobility (such as π-calculus) are more involved than those over terms in CCS-like formalisms due to the fact that the issues of naming and name substitution have to be taken into account.

2.2 Stochastic Process Algebras

Process algebras extended with stochastic information have generated a lot of research in recent years. The standard approach is to introduce an additional parameter r for each action specified in the semantics in order to store information about its duration. The intended meaning is that the probability to leave the state before time t is governed by a negative exponential distribution. Formally, $F(t) = 1 - e^{-rt}$. The dynamic behaviour of a model in case of conflicting actions (i.e. more than one activity is enabled) is controlled by a race condition. This means that all the activities attempt to proceed but only the fastest succeeds. It is worthwhile to observe that the fastest activity could be different on successive occasions because of the nature of the random variables determining the durations of activities. As a consequence of the stochastic extension, the non-deterministic branching (as in CCS) is replaced by probabilistic branching. The probability that a particular activity (labelled with rate r) completes is defined as the ratio of r to the sum of the activity rates of all the enabled activities. To form the stochastic process a state is associated with each node of the graph, and the transitions between states are defined by the arcs of the graph. Typically, it is assumed that the model is finite so that the number of nodes in the derivation graph is finite. Since all activity durations

are exponentially distributed, the total transition rate between two states will be the sum of the activity rates labelling arcs connecting the corresponding nodes in the derivation graph.

Techniques to solve the underlying Markov process can be used to infer information about the temporal behaviour of the modelled system. Procedures to obtain both approximate and correct solutions are available. Moreover, an important rôle is played by simulation techniques. As a matter of fact, the numerical analysis of huge Markov processes can be computationally intractable. Therefore, simulation techniques are often the only approach to analyse the system. The main difference between simulators and solvers is that a simulator produces a single trajectory of the given system whereas a solver gives exact solution. Hence, in order to derive useful statistics about the time evolution of a given system many runs of a simulator have to be executed and then averaged. The more runs we perform the better we approximate the correct solution. However, in some domains such as computational system biology, the stochastic noise present in a single run of the simulator can be useful to understand border-line behaviours of the system. Widely adopted algorithms for this task are for example the Gillespie Algorithm and the related Tau-leap Algorithm.

The most important examples of stochastic calculi are PEPA, stochastic π-calculus (Priami, 1995). Since their introduction several extensions to other calculi have been proposed.

2.3 COWS

COWS (Calculus for Orchestration of Web Services) is a novel approach for orchestrating distributed systems. This recently proposed model is the result of an original combination of various constructs borrowed from other process calculi. COWS is mainly intended as a foundational language for SOC and therefore, its design has been strongly influenced by the principles underlying WS-BPEL. Despite this fact, it is not specifically tied to web services' current technology. An exhaustive presentation of the calculus and its features is given in (Lapadula et al., 2007a). The authors present the encoding of several orchestration constructs (e.g. fault and compensation handlers) and imperative constructs (such as matching and sequential composition) and the encodings

$$
\begin{array}{rcl}
s \in \mathcal{S} & ::= & u!w \mid g \mid s \mid s \mid \{\!|s|\!\} \mid \mathbf{kill}(k) \mid [d]s \mid S(n_1,\ldots,n_j) \mid S \\
g & ::= & \mathbf{0} \mid p?w.s \mid g + g
\end{array}
$$

Table 2.1: COWS syntax.

of three other orchestration languages. COWS has recently been extended with timed orchestration constructs (Lapadula et al., 2007b) in order to fully express the semantics of WS-BPEL. In (Prandi and Quaglia, 2007) instead, it is presented a stochastic extension.

2.3.1 Syntax

In what follows, we consider the monadic version of the calculus defined in (Prandi and Quaglia, 2007). The syntax of COWS is parametrised by four countable and pairwise disjoint sets: the set of *names* $\mathcal{N}$ (ranged over by m, n, o, p, m', n', o', p', …), the set of *variables* $\mathcal{V}$ (ranged over by x, y, x', y', …), the set of *killer labels* $\mathcal{K}$ (ranged over by k, k', …), and the set of *service identifiers* $\mathcal{I}$ (ranged over by S, S', …). The union $\mathcal{N} \cup \mathcal{V} \cup \mathcal{K}$ (denoted by $\mathcal{E}$) represents the set of the *entities* of the calculus. Identifiers u, v, w, u', v', w' are used to range over $\mathcal{N} \cup \mathcal{V}$, and d, d' to range over $\mathcal{E}$. The set $\mathcal{I}$ is defined as $\bigcup_{i=0}^{n} \mathcal{I}_i$ where $\mathcal{I}_i$ is the set of the i-ary identifiers. COWS computational entities are called *services*. They are inductively generated by the grammar given in Table 2.1, where, for some service s, a defining equation $S = s$ or $S(n_1,\ldots,n_j) = s$ is given. The set of all the defining equations is denoted with $\mathcal{D}$. Additionally, all n_i are assumed to be distinct, with $1 \leq i \leq j$. A service s is a structured activity built from basic activities, i.e. the empty activity $\mathbf{0}$, the kill activity $\mathbf{kill}(\_)$, the asynchronous invoke activity $\_!\_$, the receive activity $\_?\_$ and the service identifiers S and $S(\_)$, by means of prefixing $\_.\_$, choice $\_ + \_$, parallel composition $\_|\_$, protection $\{\!|\_|\!\}$ and delimitation $[\_]\_$.

Let us briefly present the intended interpretation of a COWS service. The empty activity $\mathbf{0}$ has to be considered as the service which can do nothing. Sometimes we freely omit from service syntax the trailing "$.\mathbf{0}$". Asynchronous invoke activity $u!w$ and receive activity $p?w.s$ are the communication primitives of a service. An input-guarded service $p?w.s$ waits for a possible communica-

$$g\triangleleft_g \qquad u!w\triangleleft_g \qquad \mathbf{kill}(k)\triangleleft_g \qquad \frac{s\triangleleft_g}{\{\!|s|\!\}\triangleleft_g} \qquad \frac{s\triangleleft_g}{[d]s\triangleleft_g} \qquad \frac{s_1\triangleleft_g \ \wedge \ s_2\triangleleft_g}{s_1|s_2\triangleleft_g}$$

Table 2.2: Predicate $s\triangleleft_g$.

tion over p with service $u!w'$ and then proceeds as s after the instantiation of the input parameter w. Entities p, u and w in $u!w$ and $p?w.s$ are called *endpoint* and *parameter* respectively. The delimitation $[d]$ can be seen as a scope declaration for entity d. A parallel composition $s_1|s_2$ expresses concurrent behaviour. Informally speaking, this service consists of s_1 and s_2 acting in parallel and interacting via shared links. Conversely, a service $s_1 + s_2$ can behave like s_1 or alternatively like s_2. The kill activity $\mathbf{kill}(k)$ is the capability to unconditionally terminate a service not surrounded by protection (i.e. $\{\!|s|\!\}$ cannot be terminated).

Note that in the original definition of the calculus given in (Lapadula et al., 2007a), communication endpoints involved in the request and invoke activities are identified by two distinct names called *partners* and *operations*. Although this naming mechanism is more flexible than the atomic naming used in most process calculi, we chose to simplify the notation by letting endpoints be denoted by single identifiers. Another syntactical deviation we adopt is to express recursive behaviours by means of service identifiers rather than by replication.

The only *binding* construct is delimitation: In $[d]s$ the occurrence of $[d]$ is a binding for d with scope s. An entity is *free* if it is not under the scope of a binder. It is *bound* otherwise. We write $\mathsf{fe}(s)$ and $\mathsf{be}(s)$ for the set of free and bound entities in s respectively. An occurrence of one term in a service is *guarded* if it is underneath a request activity. We extend the previous definition as follows:

Definition 1 (Guarded service)**.** A service s is a *guarded service* (written $s\triangleleft_g$) if all the possible occurrences of service identifiers in s are guarded. Predicate $\_\triangleleft_g$ is defined in Table 2.2.

Example 2. Consider service $s_1 = p!w|\{\!|S|p?w.R|\!\}$. As expected, predicate $s_1\triangleleft_g$ detects unguarded service identifier S. As a matter of fact, $S\triangleleft_g$ does not hold. Therefore, the rule for parallel composition cannot be applied and s_1 is not a guarded service. Now take instead service $s_2 = [x]p!w|p?x.R(p)$. Since s_2 is a

guarded service, the following derivation can be inferred

$$\frac{\dfrac{p!w \lhd_g \ \wedge\ p?x.R(p) \lhd_g}{p!w \mid p?x.R(p) \lhd_g}}{[x]p!w \mid p?x.R(p) \lhd_g}$$

For the sake of brevity we may sometimes write $s\{^{d'_1,\cdots,d'_j}/_{d_1,\ldots,d_j}\}$ for the simultaneous substitution of d_is by d'_is in the term s and use $[d_1,\ldots,d_j]s$ as a shorthand for $[d_1]\ldots[d_j]s$. Finally,

Definition 3 (Closed service)**.** A service s is a *closed service* if variables and killer labels in s are all bound.

Examples of closed services are $[x]p?x.\mathbf{0} \mid m!n$ and $[x]p?x.S(n_1,n_2) \mid [k]\mathbf{kill}(k)$.

2.3.2 Operational semantics

As in (Prandi and Quaglia, 2007), the operational semantics of COWS is defined only for closed services. Moreover, it is assumed that services occurring in defining equations are guarded, and that there is no homonymy either among bound entities or among free and bound entities. If a service does respect the latter property, we may sometimes call it a *homonymy free* service.

The labelled transition relation $\xrightarrow{\alpha}$ is the least relation over services induced by the rules in Tables 2.5 and 2.6 and by symmetric rules for the commutative operators of parallel composition and choice. Label α is generated by the following grammar:

$$\alpha \ ::= \ \dagger k \ \mid \ \dagger \ \mid \ p?w \ \mid \ p!n \ \mid \ p?(x) \ \mid \ p!(n) \ \mid \ p \cdot \sigma \cdot \sigma'$$

where, for some name n and variable x, σ ranges over ε, $\{^n/_x\}$, $\{^{(n)}/_x\}$, and σ' over ε, $\{^n/_x\}$. The meaning of labels $\dagger k$ and $\dagger$ is that a request for terminating a term from within the delimitation $[k]$ is being or it was executed, respectively. Label $p?k$ denotes computational steps corresponding to the execution of a request activity over endpoint p with parameter w. Similar interpretation is given to label $p!n$. Label $p \cdot \sigma \cdot \sigma'$ stands for executions of a communication over endpoint p. Particularly, component σ' keeps track of the substitution induced by the communication and σ records whether it has already been applied ($\sigma = \varepsilon$) or not. This sort of labels is meant to implement a *best-match* communication mechanism, i.e. if more than one matching receive activity is ready to process

$$\mathbf{kill}(k)\downarrow_k \qquad \frac{s\downarrow_k}{\{\!|s|\!\}\downarrow_k} \qquad \frac{s\downarrow_k}{[d]s\downarrow_k} \qquad \frac{s_1\downarrow_k \ \vee \ s_2\downarrow_k}{s_1\,|\,s_2\downarrow_k}$$

$$p?n.s\downarrow_{p?n} \qquad \frac{s\downarrow_{p?n}}{\{\!|s|\!\}\downarrow_{p?n}} \qquad \frac{s\downarrow_{p?n}}{[d]s\downarrow_{p?n}} \qquad \frac{s_1\downarrow_{p?n} \ \vee \ s_2\downarrow_{p?n}}{s_1\,|\,s_2\downarrow_{p?n}} \qquad \frac{s_1\downarrow_{p?n} \ \vee \ s_2\downarrow_{p?n}}{s_1 + s_2\downarrow_{p?n}}$$

Table 2.3: Predicates $s\downarrow_k$ and $s\downarrow_{p?n}$.

$$\begin{array}{rcll} halt(g) = halt(u!w) = halt(\mathbf{kill}(k)) & = & \mathbf{0} & \\ halt(s_1\,|\,s_2) & = & halt(s_1)\,|\,halt(s_2) & \\ halt(\{\!|s|\!\}) & = & \{\!|s|\!\} & \\ halt([d]s) & = & [d]halt(s) & \\ halt(S) & = & halt(s) & S = s \\ halt(S(m_1,\ldots,m_j)) & = & halt(s\{^{m_1,\ldots,m_j}/_{n_1,\ldots,n_j}\}) & S(m_1,\ldots,m_j) = s \end{array}$$

Table 2.4: Function $halt(\_)$.

a given invoke, then only the most defined one progresses. Labels like $p?(x)$, $p!(n)$ and $p\cdot\{^{(n)}/_x\}\cdot\sigma'$ are to be interpreted as corresponding labels $p?x$, $p!n$ and $p\cdot\{^n/_x\}\cdot\sigma'$. The additional parentheses only record that the scope of the entity is undergoing modification.

To define the labelled transition relation, we use some auxiliary functions. We write $s\downarrow_{p?n}$ if, for some s', service s has an unguarded subterm of the shape $p?n.s'$. Similarly, $s\downarrow_k$ means that some unguarded killer activity $\mathbf{kill}(k)$ is a subterm of s. Predicates $s\downarrow_{p?n}$ and $s\downarrow_k$ are defined inductively on the syntax of services in Table 2.3. Their respective negations are $s\not\downarrow_{p?n}$ and $s\not\downarrow_k$. We will also use function over services $halt(\_)$ defined in Table 2.4. It describes service behaviours correspondingly to the execution of kill activity: it takes a service s as an argument and returns the service obtained by only retaining the protected activities inside s. Finally, we use $d(\alpha)$ to denote the set of entities occurring in α, except for $\alpha = p\cdot\{^n/_x\}\cdot\sigma'$ or $\alpha = p\cdot\{^n/_x\}\cdot\sigma'$ which we let $d(p\cdot\{^n/_x\}\cdot\sigma') = d(p\cdot\{^n/_x\}\cdot\sigma') = \{n,x\}$ and for $\alpha = p\cdot\varepsilon\cdot\sigma'$ which is $d(p\cdot\varepsilon\cdot\sigma') = \emptyset$.

We comment on salient points of the operational semantics by starting with rules in Table 2.5. The execution of activity $\mathbf{kill}(k)$ forces termination of all unprotected parallel activities (rules (**kill**) and (**par kill**)) inside the scope of

$$\mathbf{kill}(k) \xrightarrow{\dagger k} \mathbf{0}\ (\textbf{kill}) \qquad p?w.s \xrightarrow{p?w} s\ (\textbf{req}) \qquad p!n \xrightarrow{p!n} \mathbf{0}\ (\textbf{inv})$$

$$\frac{g_1 \xrightarrow{\alpha} s}{g_1 + g_2 \xrightarrow{\alpha} s}\ (\textbf{choice}) \qquad \frac{s \xrightarrow{\alpha} s'}{\{\!|s|\!\} \xrightarrow{\alpha} \{\!|s'|\!\}}\ (\textbf{prot})$$

$$\frac{s_1 \xrightarrow{p!n} s_1' \quad s_2 \xrightarrow{p?n} s_2'}{s_1 \,|\, s_2 s_1 \xrightarrow{p\cdot\varepsilon\cdot\varepsilon} s_1' \,|\, s_2'}\ (\textbf{com n}) \qquad \frac{s_1 \xrightarrow{p!n} s_1' \quad s_2 \xrightarrow{p?x} s_2' \quad (s_1 \,|\, s_2) \not\downarrow_{p?n}}{s_1 \,|\, s_2 \xrightarrow{p\cdot\{n/x\}\cdot\{n/x\}} s_1' \,|\, s_2'}\ (\textbf{com x})$$

$$\frac{s_1 \xrightarrow{p\cdot\sigma\cdot\sigma'} s_1' \quad \sigma' = \{n/x\} \Rightarrow s_2 \not\downarrow_{p?n}}{s_1 \,|\, s_2 \xrightarrow{p\cdot\sigma\cdot\sigma'} s_1' \,|\, s_2}\ (\textbf{par conf}) \qquad \frac{s \xrightarrow{p\cdot\{n/x\}\cdot\{n/x\}} s'}{[x]s \xrightarrow{p\cdot\varepsilon\cdot\{n/x\}} s'\{n/x\}}\ (\textbf{del sub})$$

$$\frac{s_1 \xrightarrow{\dagger k} s_1'}{s_1 \,|\, s_2 \xrightarrow{\dagger k} s_1' \,|\, halt(s_2)}\ (\textbf{par kill}) \qquad \frac{s_1 \xrightarrow{\alpha} s_1' \quad \alpha \neq p\cdot\sigma\cdot\sigma' \quad \alpha \neq \dagger k}{s_1 \,|\, s_2 \xrightarrow{\alpha} s_1' \,|\, s_2}\ (\textbf{par pass})$$

$$\frac{s \xrightarrow{\dagger k} s'}{[k]s \xrightarrow{\dagger} [k]s'}\ (\textbf{del kill}) \qquad \frac{s \xrightarrow{\alpha} s' \quad d \notin d(\alpha) \quad s \downarrow_d \Rightarrow (\alpha = \dagger \vee \alpha = \dagger k)}{[d]s \xrightarrow{\alpha} [d]s'}\ (\textbf{del pass})$$

Table 2.5: Operational semantics of COWS (first part).

delimiter $[k]$. When $\dagger k$ reaches it, the killer label is deactivated by transforming it into $\dagger$ (rule (**del kill**)). The existence of delimitation $[k]$ is ensured by the assumption that the semantics is only defined for closed services. Sensitive code can be protected from killing by putting it in protection $\{\!|\_|\!\}$. The protected term, $\{\!|s|\!\}$ behaves like s as shown by rule (**prot**). Note that rule (**del pass**) defines an *eager* execution strategy of kill activities. This means that whenever a kill activity occurs unguarded within a service s delimited by d, service $[d]s$ can only execute actions of the form $\dagger k$ or $\dagger$. An invoke activity can only take place if its parameter is a name (axiom (**inv**)). A receive activity waits for a communication over endpoint p and then proceeds as s (axiom (**req**)). The execution of a receive permits to take a decision between alternative behaviours (rule (**choice**)). Variable instantiation can take place, involving the whole scope of variable x, due to a pending communication action of shape $p\cdot\{n/x\}\cdot\{n/x\}$ (rule (**del sub**)). Execution of parallel services is interleaved (rule (**par pass**)), but when a kill activity or a communication is performed. Communication allows the synchronisation of an invoke activity $p!n$ with either the best-matching request activity $p?n.s$ (rule (**com n**)), or with a less defined $p?x.s$ if a best-match is not offered by the locally available context (rule (**com x**)). Surrounding parallel

$$\frac{s \xrightarrow{p?x} s'}{[x]s \xrightarrow{p?(x)} s'}\ (\textbf{open req}) \qquad \frac{s_1 \xrightarrow{p!(n)} s_1' \quad s_2 \xrightarrow{p?(x)} s_2' \quad (s_1 \mid s_2) \not\downarrow_{p?n}}{s_1 \mid s_2 \xrightarrow{p\cdot\varepsilon\cdot\{n/x\}} [n](s_1' \mid s_2'\{n/x\})}\ (\textbf{close nx})$$

$$\frac{s \xrightarrow{p!n} s'}{[n]s \xrightarrow{p!(n)} s'}\ (\textbf{open inv}) \qquad \frac{s_1 \xrightarrow{p!(n)} s_1' \quad s_2 \xrightarrow{p?x} s_2' \quad (s_1 \mid s_2) \not\downarrow_{p?n}}{s_1 \mid s_2 \xrightarrow{p\cdot\{(n)/x\}\cdot\{n/x\}} s_1' \mid s_2'}\ (\textbf{close n})$$

$$\frac{s \xrightarrow{p\cdot\{(n)/x\}\cdot\{n/x\}} s'}{[x]s \xrightarrow{p\cdot\varepsilon\cdot\{n/x\}} [n]s'\{n/x\}}\ (\textbf{close del}) \qquad \frac{s_1 \xrightarrow{p!n} s_1' \quad s_2 \xrightarrow{p?(x)} s_2' \quad (s_1 \mid s_2) \not\downarrow_{p?n}}{s_1 \mid s_2 \xrightarrow{p\cdot\varepsilon\cdot\{n/x\}} s_1' \mid s_2'\{n/x\}}\ (\textbf{close x})$$

$$\frac{s\{m_1,\ldots,m_j/n_1,\ldots,n_j\} \xrightarrow{\alpha} s' \quad S(n_1,\ldots,n_j) = s}{S(m_1,\ldots,m_j) \xrightarrow{l\_dec(\alpha)} s\_dec(\alpha,s')}\ (\textbf{ser id}) \qquad \frac{s \xrightarrow{\alpha} s' \quad S = s}{s \xrightarrow{l\_dec(\alpha)} s\_dec(\alpha,s')}\ (\textbf{ser id0})$$

Table 2.6: Operational semantics of COWS (second part).

services are scanned to find a best match for $p!n$(rule (**par conf**)) until either a $p?n.s$ or the delimiter of the variable scope is encountered. In the first case the attempt to establish an interaction between $p!n$ and $p?x.s$ is blocked by the non applicability of the rules for parallel composition.

Rules listed in Table 2.6 deal with the management of the scope of binders and service identifiers. This technique closely resembles the analogous mechanism for closing and opening the scope used in the definition of the labelled transition system of the π-calculus. Basically, a binder is removed when an activity is *opened*. In this manner, possible synchronisations are allowed because they can pass over a parallel composition. Whether a communication takes place, the scope is *closed* by reintroducing the delimiter in the residual service. Rules (**open req**) and (**open inv**) open the scope of the parameter in a request and invoke, respectively. Note that this is recorded in the labels by surrounding the parameters with parentheses, i.e. $p?(x)$ and $p!(n)$. Rule (**close nx**) handles those scenarios when both n and x underwent a scope opening. As can be seen, variable x is instantiated in the receiving subterm and the scope is closed by reintroducing a delimiter $[n]$ in the residual. The instantiation of x is recorded in the label by setting element $\sigma = \varepsilon$. Rule (**close x**) manages in a similar way the cases in which only the scope of the request parameter x has been previously opened. The only difference is that no delimiter is reintroduced. Rule (**close n**)

is used when the scope delimiter for the invoke activity is within the scope of delimiter for the request activity, like e.g. in $[x]([n]p!n|p?x.s)$. In this case no delimiter is reintroduced and no instantiation is performed. As a matter of fact, $\sigma = \{^{(n)}/_{x}\}$ in the label. Observe that in all the last three rules a possible best-match can still be found in the surrounding parallel services. Rule (**close del**) executes the instantiation left pending by the application of rule (**close n**), reintroduces delimiter $[d]$ and sets $\sigma = \varepsilon$. Rules (**ser id**) and (**ser id0**) state that the behaviour of a service identifier is given by is defining service. Additionally, in the first rule formal parameters have to be substituted by actual parameters. Auxiliary functions $l\_dec(\_)$ and $s\_dec(\_,\_)$ assure that the resulting service respects the non-homonymy condition by decorating bound names. The details of their actual implementation will be exhaustively discussed in Chapter 4.

Transition systems generated by the operational semantics presented above enjoy several important properties. Let us list some definitions and set up some notational conventions.

Definition 4 (Computation step)**.** Reduction $s \xrightarrow{\alpha} s'$ is called a *computation step* if $\alpha = \dagger$ or $\alpha = p \cdot \varepsilon \cdot \sigma'$, for some p and σ'.

Definition 5 (Computation)**.** A sequence of connected transitions of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} s_3 \ldots$$

is called a *computation* from service s_0, where, for each i, $\xrightarrow{\alpha_i}$ is a computation step.

Definition 6 (Derivative)**.** Let s' and s be two services. s' is said a *derivative* of s if s' can be reached from s by a finite number of computation steps.

Definition 7 (Derivative set)**.** The *derivative set* of a service s (written $\Psi(s)$), is the set including s and all of its derivatives. A service s is *finite* if $\Psi(s)$ is finite.

The most important property is undoubtedly that transition systems associated with the semantics are guaranteed to be *finitely-branching*, i.e. every node has a finite number of successors. This is implied by the use of recursive definitions rather than replication, by the non-homonymy assumption and by the fact that service identifiers do not occur unguarded. The main advantage with respect to, for instance, the semantics given in (Lapadula et al., 2007a), is that Markovian techniques can effectively be applied by enriching the labels

$$req(p;\mathbf{kill}(k)) = req(p;u!w) = req(p;\mathbf{0}) = 0$$

$$req(p;p'?w.s) = \begin{cases} \rho(p) & \text{if } p = p' \\ 0 & \text{otherwise} \end{cases} \qquad req(p;\{\!|s|\!\}) = req(p;s)$$

$$req(p;g_1 + g_2) = req(p;g_1) + req(p;g_2) \quad req(p;p_1 \,|\, p_2) = req(p;p_1) + req(p;p_2)$$

$$req(p;[d]s) = \begin{cases} 0 & \text{if } p = d \text{ or } s \downarrow_d \\ req(p;s) & \text{otherwise} \end{cases} \qquad req(p;S) = req(p;s) \text{ if } S = s$$

$$req(p;S(m_1,\ldots,m_j)) = req(p;s\{m_1,\ldots,m_j/n_1,\ldots,n_j\}) \text{ if } S(n_1,\ldots,n_j) = s$$

Table 2.7: Apparent rate of a request

to include information about the duration of activities. This is indeed the approach followed by the authors in (Prandi and Quaglia, 2007) for the definition of a stochastic extension of COWS.

2.4 Stochastic COWS

In (Prandi and Quaglia, 2007), is presented a stochastic extension of COWS. Following the standard approach for stochastic extension of process calculi described in Section 2.2, an additional parameter $\lambda \in \mathbb{R}^+$ (called *rate*) is introduced for each action specified in the semantics in order to store information about its duration. The intuitive meaning is that the probability a computational state is successfully executed before time Δt is governed by a negative exponential distribution with parameter λ. The probability of a computational step $s \xrightarrow{\alpha} s'$ is defined as the ratio between its rate and the *exit rate* of service s i.e. the sum of the rates of all the activities enabled in s.

In the sequel, we consider a slightly modified version of the original stochastic COWS. Before embarking on the presentation of the stochastic semantics, we define some auxiliary functions. The *apparent rate* of request activities is specified by function $req : \mathcal{E} \times \mathcal{S} \to \mathbb{R}$. As can be understood from its definition in Table 2.7, $req(p;s)$ sums up the rates of all the request over endpoint p which are enabled in s. An analogous function (called $inv(\_;\_)$) computing the apparent rate of invoke activities is reported in Table 2.8. The apparent rate of α in service s (written $\sharp(\alpha,s)$) is computed as described in Table 2.9, using the two functions defined above. Finally, function $\rho : \mathcal{E} \to \mathbb{R}^+$ associates a stochastic

$$inv(p;\mathbf{kill}(k)) = inv(p;p?w.s) = inv(p;\mathbf{0}) = 0$$

$$inv(p;u?w.) = \begin{cases} \rho(p) & \text{if } p = u \\ 0 & \text{otherwise} \end{cases} \qquad inv(p;\{\!|s|\!\}) = inv(p;s)$$

$$inv(p;g_1 + g_2) = inv(p;g_1) + inv(p;g_2) \quad inv(p;p_1 \,|\, p_2) = inv(p;p_1) + inv(p;p_2)$$

$$inv(p;[d]s) = \begin{cases} 0 & \text{if } p = d \text{ or } s \downarrow_d \\ inv(p;s) & \text{otherwise} \end{cases} \qquad inv(p;S) = inv(p;s) \text{ if } S = s$$

$$inv(p;S(m_1,\ldots,m_j)) = inv(p;s\{^{m_1,\ldots,m_j}/_{n_1,\ldots,n_j}\}) \text{ if } S(n_1,\ldots,n_j) = s$$

Table 2.8: Apparent rate of an invoke

$$\sharp(\alpha,s) = \begin{cases} req(p;s) & \text{if } \alpha \in \{p?w, p?(x)\} \\ inv(p;s) & \text{if } \alpha \in \{p!n, p!(n)\} \\ [req(p;s), inv(p;s)] & \text{if } \alpha = p \cdot \sigma \cdot \sigma' \\ 0 & \text{otherwise} \end{cases}$$

Table 2.9: Apparent rate of α in service s.

rate to each entity. Sometimes, we use meta-variables λ, δ and γ to range over rates of kill, invoke and request activities, respectively.

In order to add stochasticity to the calculus, rules in the semantics presented in Tables 2.5 and 2.6 have to be adapted. This is actually done by substituting labels α with new *enhanced labels*, recording additional information about rates and choice. Namely, an enhanced label θ is a triple $(\alpha,\varphi,\varphi')$ prefixed by a choice-address ϑ. The first component of the triple is a label in the reduction relation defined by the COWS semantics. Elements φ and φ' can either be a rate or a pair of request-invoke rates in the form $[\gamma,\delta]$. The prefix ϑ is a string from the alphabet $\{+_0,+_1\}^*$ used to distinguish between the left and the right branch of a choice service. We write $\vartheta \cdot \vartheta'$ or $\vartheta\vartheta'$ to indicate the string resulting from the concatenation of strings ϑ and ϑ'. Sometimes, we omit the empty prefix by writing θ instead of $\varepsilon\theta$. Axioms (**kill**), (**req**), (**inv**) defining behaviours of kill, request and invoke activities, are updated as follows:

$$\mathbf{kill}(k) \xrightarrow{(\dagger k,\rho(k),\rho(k))} \mathbf{0} \quad p?w.s \xrightarrow{(p?w,\rho(p),\rho(p))} s \quad p!n \xrightarrow{(p!n,\rho(p),\rho(p))} \mathbf{0}$$

Rule (**par pass**) is modified by recording the apparent rate of α in s_2 in the φ'

component. It takes the shape shown below.

$$\frac{s_1 \xrightarrow{\vartheta(\alpha,\varphi,\varphi')} s_1' \quad \alpha \neq p\cdot\sigma\cdot\sigma' \quad \alpha \neq \dagger k}{s_1 \,|\, s_2 \xrightarrow{\vartheta(\alpha,\varphi,\varphi'+\sharp(\alpha,s_2))} s_1' \,|\, s_2}\ (\textbf{par pass})$$

Rules (**par kill**) and (**par conf**) are modified in a similar way:

$$\frac{s_1 \xrightarrow{\vartheta(\dagger k,\varphi,\varphi')} s_1'}{s_1 \,|\, s_2 \xrightarrow{\vartheta(\alpha,\varphi,\varphi'+\sharp(\dagger k,s_2))} s_1' \,|\, halt(s_2)}\ (\textbf{par kill})$$

$$\frac{s_1 \xrightarrow{\vartheta(p\cdot\sigma\cdot\sigma',\varphi,[\gamma,\delta])} s_1' \quad \sigma' = \{n/x\} \Rightarrow s_2 \not\downarrow_{p?n}}{s_1 \,|\, s_2 \xrightarrow{\vartheta(p\cdot\sigma\cdot\sigma',\varphi,[\gamma+\sharp(\alpha,s_2),\delta+\sharp(\alpha,s_2)])} s_1' \,|\, s_2}\ (\textbf{par conf})$$

Note that the apparent rate for killer and empty activities is $\sharp(\dagger k, s) = 0$ for some service s. Modifications necessary to compute the rate of a synchronisation between an invoke and a request activity affect rules (**com n**), (**com x**), (**close n**), (**close x**), and (**close nx**). Their definition is reported in Table 2.10. In this case, the strategy is to store both request and invoke rates γ and δ in the φ component of the enhanced label, and to update both request and invoke apparent rates γ'' and δ'' in φ'. This last step is accomplished by using function $\sharp(\alpha, s)$, where s is the subterm not performing α. Rule (**choice**) and its symmetric not listed in Table 2.5 are substituted by the following two rules.

$$\frac{g_1 \xrightarrow{\vartheta(\alpha,\varphi,\varphi')} s}{g_1 + g_2 \xrightarrow{+_0\vartheta(\alpha,\varphi,\varphi'+\sharp(\alpha,g_2))} s}\ (\textbf{choice0}) \qquad \frac{g_1 \xrightarrow{\vartheta(\alpha,\varphi,\varphi')} s}{g_1 + g_2 \xrightarrow{+_1\vartheta(\alpha,\varphi,\varphi'+\sharp(\alpha,g_1))} s}\ (\textbf{choice1})$$

All the other rules are transparent with respect to the extra information added in the enhanced label. This means that elements φ and φ' and the prefix ϑ are not affected by the application of these rules. It is worthwhile to note that the stochastic semantics is defined only for homonymy-free, guarded and closed services.

Finally, we extend Definition 4 to embrace the new features of stochastic COWS.

Definition 8 (Stochastic computation step)**.** Reduction $s \xrightarrow{\vartheta(\alpha,\varphi,\varphi')} s'$ is called a *stochastic computation step* if $\alpha = \dagger$ or $\alpha = p\cdot\varepsilon\cdot\sigma'$, for some p and σ'.

$$\frac{s_1 \xrightarrow{\vartheta(p!n,\delta,\delta'')} s_1' \quad s_2 \xrightarrow{\vartheta'(p?n,\gamma,\gamma'')} s_2'}{s_1 \mid s_2 \xrightarrow{\vartheta\vartheta'(p\cdot\varepsilon\cdot\varepsilon,[\gamma,\delta],[\gamma''+\sharp(p?n,s_1),\delta''+\sharp(p!n,s_2)])} s_1' \mid s_2'} \text{(com n)}$$

$$\frac{s_1 \xrightarrow{\vartheta(p!n,\delta,\delta'')} s_1' \quad s_2 \xrightarrow{\vartheta'(p?x,\gamma,\gamma'')} s_2' \quad (s_1 \mid s_2) \not\downarrow_{p?n}}{s_1 \mid s_2 \xrightarrow{\vartheta\vartheta'(p\cdot\{n/x\}\cdot\{n/x\},[\gamma,\delta],[\gamma''+\sharp(p?x,s_1),\delta''+\sharp(p!n,s_2)])} s_1' \mid s_2'} \text{(com x)}$$

$$\frac{s_1 \xrightarrow{\vartheta(p!(n),\delta,\delta'')} s_1' \quad s_2 \xrightarrow{\vartheta'(p?x,\gamma,\gamma'')} s_2' \quad (s_1 \mid s_2) \not\downarrow_{p?n}}{s_1 \mid s_2 \xrightarrow{\vartheta\vartheta'(p\cdot\{(n)/x\}\cdot\{n/x\},[\gamma,\delta],[\gamma''+\sharp(p?x,s_1),\delta''+\sharp(p!(n),s_2)])} s_1' \mid s_2'} \text{(close n)}$$

$$\frac{s_1 \xrightarrow{\vartheta(p!n,\delta,\delta'')} s_1' \quad s_2 \xrightarrow{\vartheta'(p?(x),\gamma,\gamma'')} s_2' \quad (s_1 \mid s_2) \not\downarrow_{p?n}}{s_1 \mid s_2 \xrightarrow{\vartheta\vartheta'(p\cdot\varepsilon\cdot\{n/x\},[\gamma,\delta],[\gamma''+\sharp(p?(x),s_1),\delta''+\sharp(p!n,s_2)])} s_1' \mid s_2'\{n/x\}} \text{(close x)}$$

$$\frac{s_1 \xrightarrow{\vartheta(p!(n),\delta,\delta'')} s_1' \quad s_2 \xrightarrow{\vartheta'(p?(x),\gamma,\gamma'')} s_2' \quad (s_1 \mid s_2) \not\downarrow_{p?n}}{s_1 \mid s_2 \xrightarrow{\vartheta\vartheta'(p\cdot\varepsilon\cdot\{n/x\},[\gamma,\delta],[\gamma''+\sharp(p?(x),s_1),\delta''+\sharp(p!(n),s_2)])} [n](s_1' \mid s_2'\{n/x\})} \text{(close nx)}$$

Table 2.10: Stochastic COWS operational semantics: communication rules

2.4.1 Stochastic analysis

In (Prandi and Quaglia, 2007), the authors describe a transformation technique of stochastic transition systems into Continuous Time Markov Chains. The main benefit of this procedure is to allow quantitative reasoning on the service generating the transition system undergoing the translation by means of CTMC analysis tool such as, for instance, the probabilistic model checker Prism. Let us recall a fundamental definition:

Definition 9 (Continuous Time Markov Chain (CTMC))**.** A CTMC is a triple $C = (Q, q_0, \mathbf{R})$ where Q is a set of *states*, q_0 is the *initial state*, $\mathbf{R} : Q^2 \to \mathbb{R}^+$ is the *transition matrix*. We write $\mathbf{R}(q_1, q_2) = r$ to mean that q_1 evolves to q_2 with rate r. A CTMC is *finite* if set Q is finite.

The idea behind the transformation is to infer a rate for a transition between two states in the chain from a label θ in the transition system. Formally, this is achieved by associating a CTMC to service a s (written $C(s)$), where the set of states Q is the derivative set of s (i.e. $\Psi(s)$), s is the initial state, and the transition matrix is computed as follows:

$$\mathbf{R}(s, s') = \sum_{s \xrightarrow{\theta} s'} \mu(\theta) \tag{2.1}$$

Function μ can be defined in several different ways, each one capturing a different analysis approach. The formula we use is:

$$\mu(\theta) = \begin{cases} \frac{\gamma}{\gamma}\frac{\delta}{\delta'} \min(\gamma', \delta') & \text{if } \theta = \vartheta(p, [\gamma, \delta], [\gamma', \delta,]) \\ \varphi & \text{if } \theta = \vartheta(\dagger, \varphi, \varphi') \end{cases} \tag{2.2}$$

It intuitively says that communication rate of the synchronisation between two services is taken to be proportional to the slowest one. Note that in order to have a computable procedure, the derivative set of service s has to be finite.

2.5 The Prism Probabilistic Model Checker

Prism (Kwiatkowska et al., 2002) is a probabilistic model checking tool initially developed at the University of Birmingham and now maintained at the University of Oxford (see (Parker et al., 2008)). It has been successfully used to analyse

performance, probabilistic termination, quality of service properties, and dependability for a wide range of systems, included polling systems, randomised distributed algorithms, workstation clusters and wireless cell communication. A typical model-checking session consists in giving as input a transition system describing a model and a specification of some property written in the probabilistic temporal logics **PCTL** and **CSL**; The resulting output is the probability that the model satisfies the given property. Some examples of properties we would wish to verify are the probability that a queue becomes full within t time units is less than or equal to 0.05 (expressed in **CSL** as $P_{\leq 0.05}\left[true\ U^{\leq t}\ full\right]$) or the probability that a queue is not full in the long run is greater than or equal to 0.99 (written $S_{0.99}[\neg full]$). Prism performs automatic analysis of such properties using either formal verification techniques based on numerical computation, or discrete-event simulation. The internal computations are performed by three different model checking engines. The first is based on symbolic model checking using multi-terminal binary decision diagrams (MTBDD), the second uses sparse matrices and full vectors, while the latter is a hybrid of the other two. The current implementation supports several probabilistic models: discrete-time Markov chains, CTMCs and Markov decision processes. Another important feature of Prism is the ability of directly check models defined by means of transition matrices.

2.6 Related Work

A timed extension of COWS (called C⏲WS) is described in (Lapadula et al., 2007b). The introduction of a wait activity allows a complete formalisation of the WS-BPEL semantics, in particular *wait*, *until* and *pick* timed constructs. The newly introduced activity $⏲_e$ specifies the time interval, whose value is given by evaluation of e, the executing service has to wait for. Moreover, this is the only activity consuming time during its execution. The labelled semantics of C⏲WS is extended with the addition of labels modelling time elapsing.

Several different approaches have been proposed for the simulation and analysis of COWS models. The first one was CMC (COWS interpreter and Model Checker) (Mazzanti, 2007): A web-based interface towards a remote on-the fly interpreter for the COWS language with checking capabilities of **UCTL** expressions. The tool is conceived for non-stochastic COWS, hence

quantitative analysis of the models is not supported. An interesting peculiarity of the interpreter is that no limitations on the number of states in the transition system are set. As a matter of fact, both finite and infinite transition systems are explored by the user by iteratively specifying the number of states each iteration has to consider.

A second approach has been chosen in (Prandi and Quaglia, 2007); It consists in applying continuous-time Markov chain based analysis to stochastic COWS terms. In practice, the CTMC associated with the transition system generated from a stochastic COWS process is manually computed and used as input for the probabilistic model checker Prism. In this way, properties of the system expressed formally in **CTL** are automatically analysed against the constructed model. This strategy is the one followed in our implementation of the COWS2Prism system, which is indeed intended as a tool for the automation of the tedious and error-prone procedure described above. We are not aware of the existence of tools for timed COWS.

Chapter 3

COWS2Prism System Design

The logical structure of the COWS2Prism system is presented in Figure 3.1. The *compiler* translates the source code into a run-time representation closely resembling the original COWS specifications. Moreover, details regarding stochastic rates are stored into the *environment*. Observe that a paramount rôle in this transformation is played by the *fresh names generator* and the *type system*. The *translation engine* computes the transition system corresponding to the input model and translates it into a Continuous Time Markov Chain.

3.1 The source language (High-Lan COWS)

We make two simplifications to the stochastic COWS presented by the authors in (Prandi and Quaglia, 2007). First, rates are removed from the constructs of the language and are instead handled separately. This modification makes service definitions in High-Lan COWS less cumbersome and more readable. Another additional benefit is that a name is always used with the same rate giving consistency to the definition. Furthermore, a minor improvement is that a global rate is easy to specify.

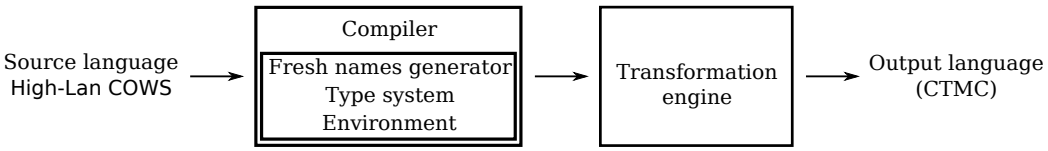

Figure 3.1: Logical structure of the COWS2Prism system.

$$
\begin{array}{rcl}
s \in S & ::= & e!e \mid g \mid s \mid s \mid \{\!|s|\!\} \mid \mathbf{kill}(e) \mid [e]s \mid S(e_1,\ldots,e_j) \mid S \\
g & ::= & \mathbf{0} \mid e?e.s \mid g + g \\
a \in D & ::= & S(e_1,\ldots,e_j) = s \mid S = s
\end{array}
$$

Table 3.1: High-Lan COWS syntax.

The second simplification we make is that the High-Lan COWS language does not support type annotations. As a consequence, services defined in a High-Lan COWS program are built up on generic entities and all the type constraints in the original COWS definition (see Table 2.1) are dropped (i.e. we do not distinguish among names, variables and killer labels). We call E (ranged over by e) the set of High-Lan COWS basic entities and S the set comprising all the High-Lan COWS services. A grammar for High-Lan COWS is given in Table 3.1.

3.2 Compiler

The compiler is the central component of the COWS2Prism system. It parses the syntactic definition of High-Lan COWS services listed in the input file and transforms them into a run-time representation usable as input for the translation engine. Additionally, the name-rate bindings specified in the definition are stored into the environment. All the modifications performed on a High-Lan COWS model are carried out by two separate compiler sub-units: the fresh names generator and the type system. The first one removes any homonymy among entities, while the second encodes a High-Lan COWS term into a closed COWS term.

Before embarking on with their formal description, we provide some useful definition. Furthermore, we recall some standard terminology and notation taken from (Sangiorgi and Walker, 2001).

Definition 10 (Program)**.** A *program* is a tuple $Z = \langle \sigma, \delta, \mathcal{E}nv \rangle$ where $\sigma \in S \cup \mathcal{S}$ and $\delta \in D \cup \mathcal{D}$. Proceeding from left to right, its components are called *service*, *definitions* and *environment*.

Conceptually, a program Z is the internal representation of a High-Lan

COWS model being used by the various sub-units of the COWS2Prism system. Observe that a program is in some way polymorphic because the first two components σ and δ can both be defined according to the High-Lan COWS syntax or to the COWS syntax. This gives us the advantage of uniform notation in our description. In the first case, we call Z a High-Lan COWS *program* otherwise it is said to be a COWS *program*. The sets of all High-Lan COWS and COWS programs are indicated by Z and $\mathcal{Z}$, respectively. The environment component is described later on in this chapter.

Definition 11 (Type environments)**.** An *assignment* of a type to a name is of the form $a : T$, where a is a name, called the name of the assignment, and T is a type, called the type of the assignment.

A *type environment* (or simply *typing*) is a finite set of assignments of types to names, where the names in the assignments are all different.

We use Γ, Λ to range over the set of type environments $\mathfrak{E}$. The notation $\mathsf{supp}(\Gamma)$ is used to indicate the *support* of Γ, i.e. the names in the environment. A type environment Γ can be thought of as a finite function from names to types. Therefore, we write $\Gamma(a)$ for the type assigned to a by Γ, assuming that $a \in \mathsf{supp}(\Gamma)$. The empty environment is indicated with $\varnothing$. To facilitate reading, we sometimes omit curly brackets outside a type environment. For example, we write $a : T, b : S$ for the type environment that assigns T to a, S to b and is undefined on the other names. We sometimes regard a type environment Γ as a set of elements in the form $a : T$. In this case, standard set operations (i.e. union ($\cup$), intersection ($\cap$), difference ($\setminus$) and symmetric difference (Δ)) have the expected semantics.

3.2.1 Environment

The environment (denoted by $\mathcal{E}nv$) is the COWS2Prism system component handling *name-rate* bindings. It provides them to the translation engine so that a finite stochastic transition system can be computed. In this context, a *name* is deemed as the string representation of an entity. To stress this fact, we will write $\widetilde{e}$ to indicate the string used for the naming of entity e. Note that function $\widetilde{\_}$ is defined for both High-Lan COWS and COWS entities. In particular, $\widetilde{e} = \widetilde{\widehat{e}}$, where $\widehat{e}$ is the COWS entity corresponding to High-Lan COWS entity e. Stochastic rates are non-negative real numbers as stated in previous chapter,

whilst entities are used in the High-Lan COWS input model as described in Table 3.1. According to Definition 11, the set of $\mathcal{E}nv$ names is $\widetilde{E}$ and the set of types is $\mathbb{R}^+$. Additionally, we write $\rho(e)$ to indicate the rate of entity e. Formally, it is a shorthand for the notation $\mathcal{E}nv(\widetilde{e})$.

3.2.2 Fresh names generator

In order to assure the non-homonymy condition is met in High-Lan COWS services, a procedure to generate fresh names has to be defined. Note that in this context, we use the word *name* to refer to strings identifiers for different entities $e \in E$. Without loss of generality, in the following we assume that services do not have free entities, i.e. we reinforce the closeness condition in Definition 3 over COWS services which, on the contrary, allows free names. To handle services not satisfying this assumption, two functions $close : S \to S$ and $close^{-1} : S \times E \to S$ can be defined:

$$\begin{aligned} close(s) &= \begin{cases} [e_1]\dots[e_j]s & \text{if } \mathsf{fe}(s) = \{e_1,\dots,e_j\} \\ s & \text{otherwise} \end{cases} \\ close^{-1}([e_1]\dots[e_j]s, N) &= s \qquad \forall e_i \in N \text{ with } 1 \le i \le j \end{aligned}$$

As can be seen, the syntactic modification introduced by function *close* always assures that no free entities occur in the resulting output service. Furthermore, function $close^{-1}$ removes this modification giving back the original term, if the set of entities given as input corresponds to its set of free names.

At this stage, the fresh names needed to refresh term s can easily be computed by counting the occurrences of a delimiter construct over the same entity. This is accomplished by the procedure described in Table 3.2. Note that $[\![\_]\!]_\_$ is a homomorphism on parallel composition, summation and protection operators and $\mathbf{0}$ is its neutral element. For this reason, their respective rules are omitted in the definition. By abuse of notation, we sometimes write $[\![\_]\!]$ for $[\![\_]\!]_\varnothing$.

Let us briefly explain the main features of the encoding. First of all, Λ is defined as an environment for entities $e \in E$, where types ϑ belong to the alphabet 0^*. As shown in Table 3.2, the refreshing encoding simply appends to each entity e in the input term s its string $\Lambda(e) = \vartheta$. The only exception is the rule handling delimiters. As a matter of fact, each occurrence of a delimiter introduces a fresh name by appending (using operator $\cdot$) an extra 0

$$
\begin{array}{rcl}
[\![e!e']\!]_\Lambda & = & \begin{cases} e \cdot \vartheta ! e' & \text{if } \Lambda(e) = \vartheta \\ e!e' \cdot \vartheta & \text{if } \Lambda(e') = \vartheta \\ e \cdot \vartheta ! e' \cdot \vartheta' & \text{if } \Lambda(e) = \vartheta \text{ and } \Lambda(e') = \vartheta' \\ e!e' & \text{otherwise} \end{cases} \\
[\![e?e'.s]\!]_\Lambda & = & \begin{cases} e \cdot \vartheta ? e'.[\![s]\!]_\Lambda & \text{if } \Lambda(e) = \vartheta \\ e?e' \cdot \vartheta.[\![s]\!]_\Lambda & \text{if } \Lambda(e') = \vartheta \\ e \cdot \vartheta ? e' \cdot \vartheta'.[\![s]\!]_\Lambda & \text{if } \Lambda(e) = \vartheta \text{ and } \Lambda(e') = \vartheta' \\ e?e.'[\![s]\!]_\Lambda & \text{otherwise} \end{cases} \\
[\![\mathbf{kill}(e)]\!]_\Lambda & = & \begin{cases} \mathbf{kill}(e \cdot \vartheta) & \text{if } \Lambda(e) = \vartheta \\ \mathbf{kill}(e) & \text{otherwise} \end{cases} \\
[\![S(e_1,\ldots,e_j)]\!]_\Lambda & = & \begin{cases} S(e_1,\ldots,e_i \cdot \vartheta,\ldots,e_j) & \text{if } \Lambda e_i = \vartheta \\ & \text{with } 1 \le i \le j \\ S(e_1,\ldots,e_j) & \text{otherwise} \end{cases} \\
[\![[e]s]\!]_\Lambda & = & [e \cdot \vartheta \cdot 0][\![s]\!]_{\Lambda'} \text{, where } \Lambda' = (\Lambda \setminus \{(e:\vartheta)\}) \cup (e : \vartheta \cdot 0)
\end{array}
$$

Table 3.2: Rules defining a refreshing encoding for High-Lan COWS services.

to it. Note that all successive occurrences of that name under the scope of the same delimiters are also modified in the same way. This because the associated string ϑ in the environment is also elongated by a trailing 0 symbol.

The initial environment for the encoding is indicated by Λ_0. It is defined by

$$\Lambda_0 = \{(e : \varepsilon) \mid e \in \mathsf{be}(s)\} \tag{3.1}$$

where s is the input service for the encoding and ε denotes the empty string.

The computational time complexity of the encoding is linear in the number of activities. Formally, if service s contains n activities, the execution of $[\![s]\!]_{\Lambda_0}$ is performed in time $O(n)$. This is explained by the fact that the procedure simply scans service s from left to right to refresh the names.

Finally, the procedure can be extended in a similar way to refresh a set of defining equations. Moreover, environment $\mathcal{E}nv$ has to be updated in order to have new bindings for the fresh names. In this manner, all the names in a High-Lan COWS program $Z = \langle s_0, D, \mathcal{E}nv \rangle$ can be refreshed as follows:

$$[\![Z]\!] = \left\langle s_1, [\![D]\!], [\![\mathcal{E}nv]\!]_{\Lambda_f} \right\rangle \tag{3.2}$$

$$
\begin{array}{rcl}
[\![D]\!] & = & \{[\![d]\!] \mid d \in D\} \\
[\![S = s]\!] & = & S = close^{-1}([\![close(s)]\!]_{\Lambda_0}, \mathsf{fe}(s)) \\
[\![S(e_1,\ldots,e_j) = s]\!] & = & S(e_1 \cdot 0,\ldots,e_j \cdot 0) = close^{-1}([\![close(s)]\!]_{\Lambda_0}, \mathsf{fe}(s)) \\
[\![\mathcal{E}nv]\!]_{\Lambda} & = & \bigcup_{e' \in \mathsf{supp}(\Lambda)} \bigcup_{i=1}^{n} \{(\widetilde{e} : \rho(e')) \mid e = e' \cdot \sigma_i \wedge \sigma_i = 0^i\} \\
 & & n = |\Lambda(e')|
\end{array}
$$

Table 3.3: Refreshing rules for High-Lan COWS defining equations and environment $\mathcal{E}nv$.

where $s_1 = close^{-1}([\![close(s_0)]\!]_{\Lambda_0}, \mathsf{fe}(s_0))$ and the encoding for defining equations and the environment are defined in Table 3.3. Note that environment Λ_f is the "union"of all the environments (denoted by Λ_f^i) produced by the last application of the encoding during the refreshing of service s_0 and D. Formally,

$$\Lambda_f = \{(e : \vartheta) \mid \Lambda_f^i(e) = \vartheta \ \wedge \ |\vartheta| = \max_j |\Lambda_f^j(e)|\} \tag{3.3}$$

with $i, j \ \in \{1,\ldots,|D|+1\}$.

Example 12. Suppose the components of High-Lan COWS program $Z = \langle t, \{d\}, \mathcal{E}nv \rangle$ are defined as below

$$
\begin{array}{rcl}
t & = & [x][a](a!x \mid [a](\mathbf{kill}(a) \mid p?x.[x][a]a!x)) \\
d & = & S(a) = a!p \mid [x]a?x.\mathbf{0} \\
\mathcal{E}nv & = & \{\widetilde{a} : \lambda_1, \widetilde{x} : \lambda_2, \widetilde{p} : \lambda_3\}
\end{array}
$$

Let us show the steps performed by the encoding to refresh service t. The set of free names is $\mathsf{fe}(t) = \{p\}$. Hence, applying function *close* we get

$$t_0 = close(t) = [p][x][a](a!x \mid [a](\mathbf{kill}(a) \mid p?x.[x][a]a!x))$$

At this stage, we show the steps leading to a refreshed term. Initially, the refreshing function is applied to t_0:

$$[\![t_0]\!]_{\Lambda_0} = [p0][\![[x][a](a!x \mid [a](\mathbf{kill}(a) \mid p?x.[x][a]a!x))]\!]_{\Lambda_1}$$

where $\Lambda_0 = \{(x : \varepsilon), (a : \varepsilon), (p : \varepsilon)\}$. As a matter of fact, $\mathsf{be}(t_0) = \{x, a, p\}$. According to the rules in Table 3.2, $\Lambda_1 = \{(x : \varepsilon), (a : \varepsilon), (p : 0)\}$.
Analogously, letting $t_1 = [x][a](a!x \mid [a](\mathbf{kill}(a) \mid p?x.[x][a]a!x))$ we get

$$[\![t_1]\!]_{\Lambda_1} = \ldots = [x0][a0][\![(a!x \mid [a](\mathbf{kill}(a) \mid p?x.[x][a]a!x))]\!]_{\Lambda_2}$$

where the updated environment Λ_2 is $\{(x:0),(a:0),(p:0)\}$.
Denoting $[a](\mathbf{kill}(a)\,|\,p?x.[x][a]a!x)$ with t_2, and further applying the refreshing procedure, we can write

$$[\![\, a!x \,|\, t_2 \,]\!]_{\Lambda_2} = [\![\, a!x \,]\!]_{\Lambda_2} \,|\, [\![\, t_2 \,]\!]_{\Lambda_2} = a0!x0 \,|\, [\![\, t_2 \,]\!]_{\Lambda_2}$$

Then, the right sub-term becomes

$$[\![\, t_2 \,]\!]_{\Lambda_2} = [a00]\,([\![\, \mathbf{kill}(a) \,]\!]_{\Lambda_3} \,|\, [\![\, t_3 \,]\!]_{\Lambda_3})$$

where $t_3 = p?x.[x][a]a!x$ and $\Lambda_3 = \{(x:0),(a:00),(p:0)\}$. Trivially, $[\![\, \mathbf{kill}(a) \,]\!]_{\Lambda_3}$ is transformed into $\mathbf{kill}(a00)$, while the fresh term corresponding to t_3 is given by

$$[\![\, t_3 \,]\!]_{\Lambda_3} = \ldots = p0?x0.[x00][a000]a000!x00$$

Summarising, the whole refreshed term is

$$t_4 = [p0][x0][a0](a0!x0 \,|\, [a00](\mathbf{kill}(a00) \,|\, p0?x0.[x00][a000]a000!x00))$$

Final environment Λ_f^1 is $\{(x:00),(a:000),(p:0)\}$. The last step consists in the application of $close^{-1}$ to get rid of the extra delimiter added in the first stage by *close*:

$$t_5 = close^{-1}(t_4,\{p\}) = [x0][a0](a0!x0 \,|\, [a00](\mathbf{kill}(a00) \,|\, p0?x0.[x00][a000]a000!x00))$$

As can be seen, service t_5 is a refreshed version of initial term t_1. The second component of Z is refreshed as follows

$$[\![\, d \,]\!] = S(a0) = a0!p0 \,|\, [x0]a0?x0.\mathbf{0}$$

and the final environment is $\Lambda_f^2 = \{(a:0),(p:0),(x:0)\}$. Finally, environment $\mathcal{E}nv$ can be updated as described in Table 3.3. More in details,

$$[\![\, \mathcal{E}nv \,]\!]_{\Lambda_f} = \{(\widetilde{a0}:\lambda_1),(\widetilde{a00}:\lambda_1),(\widetilde{a000}:\lambda_1),(\widetilde{x0}:\lambda_2),(\widetilde{x00}:\lambda_2),(\widetilde{p0}:\lambda_3)\}$$

where $\Lambda_f = \{(x:00),(a:000),(p:0)\}$ is the result of the merging of final environments Λ_f^1 and Λ_f^2.

3.2.3 Type System

In the following, we present an algorithm which infers the most general typing for a homonymy-free High-Lan COWS service, and proving its correctness.

The algorithm is similar in style to the one introduced in (Gay, 1993) for sort inference in the polyadic π-calculus. A crucial advantage given by our type system is that all High-Lan COWS services satisfying an inferred typing can straightforwardly be encoded into closed COWS services readily utilisable by the translation engine.

Types T are given by the following grammar

$$\tau \in \mathsf{T} ::= \mathsf{N} \mid \mathsf{V} \mid \mathsf{V_f} \mid \mathsf{K} \mid \mathsf{K_f} \mid \mathsf{R}_i$$

where $i \in \mathbb{N}$. Types N, V, K are meant to indicate COWS entities belonging to sets $\mathcal{N}$, $\mathcal{V}$ and $\mathcal{K}$ respectively, types $\mathsf{V_f}$ and $\mathsf{K_f}$ are used to denote unbound (i.e. free) variables and killer labels respectively, while R_i is the type for service identifiers in set $\mathcal{I}_i$. Following Definition 11, an environment Γ is formed by E and T as the sets of names and types respectively. Additionally, we define a special environment fail to denote failures in the type inference. It is assumed that supp(fail) = $\varnothing$.

We define operation $\uplus$ on environments as follows

$$\Gamma_1 \uplus \Gamma_2 = \begin{cases} \mathsf{fail} & \text{if } \Gamma_1 = \mathsf{fail} \\ \Gamma_1 \cup \Gamma_2 & \text{if } \mathsf{supp}(\Gamma_1) \cap \mathsf{supp}(\Gamma_2) = \varnothing \\ \Gamma_1 \setminus \{(e:\tau) \mid e \in \mathsf{supp}(\Gamma_2)\} \cup \Gamma_2 & \text{otherwise} \end{cases}$$

Therefore, the intended meaning of $\Gamma_1 \uplus \Gamma_2$ is to merge together Γ_1 and Γ_2, using type assignments in Γ_2 whenever there are conflicts between the two environments.

Auxiliary function $f : \mathfrak{E}^2 \to \mathfrak{E}$ is used to merge two environments, detecting potential conflicts. The merging strategy is implemented as described below:

$$f(\Gamma_1, \Gamma_2) = \begin{cases} \Gamma_1 \Delta \Gamma_1 \cup h(\Gamma_1, \Gamma_2) & \text{if } h(\Gamma_1, \Gamma_2) \neq \mathsf{fail} \\ \mathsf{fail} & \text{if } \Gamma_1 = \mathsf{fail} \vee \Gamma_2 = \mathsf{fail} \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

where Δ denotes the symmetric set difference operation and $h : \mathfrak{E}^2 \to \mathfrak{E}$ is

$$h(\Gamma_1, \Gamma_2) = \begin{cases} \mathsf{fail} & \text{if } \exists e.(\Gamma_i(e) = \mathsf{K_f} \wedge \Gamma_j(e) \neq \mathsf{K_f}) \\ \mathsf{fail} & \text{if } \exists e.(\Gamma_i(e) = \mathsf{R}_m \wedge \Gamma_j(e) = \mathsf{R}_n \wedge m \neq n) \\ \{(e:\tau) \mid \Gamma_1(e) = \Gamma_2(e)\} \cup \{(e:\mathsf{N}) \mid \Gamma_i(e) = \mathsf{N} \wedge \Gamma_j(e) = \mathsf{V_f}\} & \text{otherwise} \end{cases}$$

with $i, j \in \{1, 2\}$ and $i \neq j$. As can be seen in the above definition, conflicts arise whenever the two input environments do not agree in the typing of an

entity. The only exception is when one environment assigns type N to an entity, while the other $\mathsf{V_f}$. In this case, type N has higher priority, hence it survives after the merging, whereas $\mathsf{V_f}$ is discarded. Observe that the non-homonymy assumption assures us that conflicts of the kind $\mathsf{N}-\mathsf{V}$ are not possible, since entities typed as V cannot be in the same scope of N entities. We will reconsider this point in greater detail below.

Function $check : \mathfrak{E} \to \mathfrak{E}$ checks if the input environment contains free killer labels and free variables. In the first case, an unsolvable conflict is detected and environment fail is returned. In the second case, all the free variables are transformed into names. It is specified as follows

$$check(\Gamma) = \begin{cases} \mathsf{fail} & \text{if } \Gamma = \mathsf{fail} \\ \mathsf{fail} & \text{if } \exists e.(\Gamma(e) = \mathsf{K_f}) \\ \Gamma \uplus \{(e : \mathsf{N}) \mid \Gamma(e) = \mathsf{V_f}\} & \text{otherwise} \end{cases}$$

Definition 13. An environment $\Gamma \in \mathfrak{E}$ is said to be a *valid environment* if $\Gamma \neq \mathsf{fail}$ and if it does not contain type assignments such that $(e : \tau)$, where $\tau \in \{\mathsf{V_f}, \mathsf{K_f}\}$.

Since COWS services have no explicit results, our typing rules take the form $\Gamma \vdash s$, where Γ is a temporary environment used to derive a potential valid environment for service s via function *check*. We can read $\Gamma \vdash s$ as asserting that s uses its entities consistently with the types given in $\Lambda = check(\Gamma)$. Basically, each rule corresponds to one syntactic construct in the High-Lan COWS syntax. It is worthwhile to recall that service s is assumed not to have homonymy among its entities. Let us briefly explain the salient points of the inference rules.

The simplest High-Lan COWS service is the empty activity $\mathbf{0}$. It is not capable of any communication and hence, it is consistent with any environment:

$$\Gamma \vdash \mathbf{0} \ (\textbf{Nil})$$

The typing rule for service identifiers

$$\Gamma \cup \{(n_i : \mathsf{N}) \mid i \in \{1, \ldots, j\}\} \cup \{(S : \mathsf{R}_j)\} \vdash S(n_1, \ldots, n_j) \ (\textbf{Rec})$$

extends temporary environment Γ with types N for arguments $n_1, \ldots, n_j$ and type R_j for S, where j denotes its arity. Similarly, rule (**Rec0**) handles the typing of nullary service identifiers.

Asynchronous invoke activities $u!w$ are defined in the COWS syntax (see Table 2.1) over names and variables, whereas High-Lan COWS syntax (defined

in Table 3.1) only uses basic entities belonging to set E. This means that in COWS $u, w \in \mathcal{N} \cup \mathcal{V}$. Accordingly, typing rule (**Out**) has to assign type N or V to entities u and w. Considering the fact that only closed services can have a valid environment, the strategy we followed was to initially assign type $\mathsf{V_f}$ to both u and w in Γ. All the possible type modifications necessary to satisfy the closeness condition are postponed to the application of function *check*. The rule is then

$$\Gamma \cup \{(u : \mathsf{V_f}), (w : \mathsf{V_f})\} \vdash u!w \text{ (\textbf{Out})}$$

Kill activities are typed by:

$$\Gamma \cup \{(k : \mathsf{K_f})\} \vdash \mathbf{kill}(k) \text{ (\textbf{Kill})}$$

As can be seen, type $\mathsf{K_f}$ instead of K is assigned to entity k in Γ. We will explain below (see rule (**DelK**)) why we adopted this strategy. Note that no environment failure can arise in the previous four rules because they all consider terminal symbols in the High-Lan COWS grammar. Thus, Γ is always an empty environment.

The delimitation of entities is handled by three different rules: (**DelV**), (**Del**) and (**DelK**). The first one states that if $\Gamma \vdash s$ holds and entity d has not been previously declared in Γ or it was used with type $\mathsf{V_f}$, then $\Gamma \uplus \{(d : \mathsf{V})\} \vdash [d]s$ holds, i.e. type V is arbitrarily assigned to d. Similarly, the second rule is applied when entity d has previously been typed as N. In this case, no modifications are made on Γ, because the closeness property is already met. Rule (**DelK**) deals with the case in which entity d was previously typed as $\mathsf{K_f}$ in Γ. The result is that the type of d is set to K because d is bound in $[d]s$. Observe that the non-homonymy assumption saves us from considering cases when $\Gamma(d) = \mathsf{V}$ and $\Gamma(d) = \mathsf{K}$, because they can never occur.

Rule (**Prot**) states that $\Gamma \vdash \{\!|s|\!\}$ holds only if $\Gamma \vdash s$.

Three other rules are needed to type request activities $p?w.s$. Rule (**InpK**) is used whenever p or w have already been declared as $\mathsf{K_f}$ in Γ. Since this is not allowed by the COWS syntax, the typing system detects the conflict and thus, fail $\vdash p?w.s$ holds. Rules (**Inp**) and (**InpW**) both force the typing of p to N. The only difference is in the handling of entity w. In the first case w is already present in temporary environment Γ, then no modification has to be carried out. Therefore, $\Gamma \uplus \{(p : \mathsf{N})\} \vdash p?w.s$ holds. In the second case w is used for the first time and the same strategy used in rule (**Out**) is adopted.

$$\Gamma \vdash \mathbf{0}\ (\textbf{Nil}) \qquad \Gamma \cup \{(S : \mathsf{R}_0)\} \vdash S\ (\textbf{Rec0})$$

$$\Gamma \cup \{(n_i : \mathsf{N}) \mid i \in \{1,\ldots,j\}\} \cup \{(S : \mathsf{R}_j)\} \vdash S(n_1,\ldots,n_j)(\textbf{Rec})$$

$$\Gamma \cup \{(u : \mathsf{V_f}), (w : \mathsf{V_f})\} \vdash u!w\ (\textbf{Out}) \qquad \Gamma \cup \{(k : \mathsf{K_f})\} \vdash \mathbf{kill}(k)\ (\textbf{Kill})$$

$$\frac{\Gamma \vdash s}{\Gamma \vdash \{\!|s|\!\}}\ (\textbf{Prot}) \qquad \frac{\Gamma \vdash s \qquad \Gamma(d) = \mathsf{N}}{\Gamma \vdash [d]s}\ (\textbf{Del})$$

$$\frac{\Gamma \vdash s \qquad d \notin \mathsf{supp}(\Gamma) \vee \Gamma(d) = \mathsf{V_f}}{\Gamma \uplus \{(d : \mathsf{V})\} \vdash [d]s}\ (\textbf{DelV})$$

$$\frac{\Gamma \vdash s \qquad \Gamma(d) = \mathsf{K_f}}{\Gamma \uplus \{(d : \mathsf{K})\} \vdash [d]s}\ (\textbf{DelK})$$

$$\frac{\Gamma \vdash s \quad w \in \mathsf{supp}(\Gamma) \quad \Gamma(p) \neq \mathsf{K_f} \quad \Gamma(w) \neq \mathsf{K_f}}{\Gamma \uplus \{(p : \mathsf{N})\} \vdash p?w.s}\ (\textbf{Inp})$$

$$\frac{\Gamma \vdash s \qquad w \notin \mathsf{supp}(\Gamma) \quad \Gamma(p) \neq \mathsf{K_f}}{\Gamma \uplus \{(p : \mathsf{N}), (w : \mathsf{V_f})\} \vdash p?w.s}\ (\textbf{ImpW})$$

$$\frac{\Gamma \vdash s \qquad \Gamma(p) = \mathsf{K_f} \vee\ \Gamma(w) = \mathsf{K_f}}{\mathsf{fail} \vdash p?w.s}\ (\textbf{InpK})$$

$$\frac{\Gamma_1 \vdash s_1 \qquad \Gamma_2 \vdash s_2}{f(\Gamma_1,\Gamma_2) \vdash s_1 \mid s_2}\ (\textbf{Par}) \qquad \frac{\Gamma_1 \vdash s_1 \qquad \Gamma_2 \vdash s_2}{f(\Gamma_1,\Gamma_2) \vdash s_1 + s_2}\ (\textbf{Sum})$$

Table 3.4: Typing rules for High-Lan COWS services.

The typing rules for $s_1 \mid s_2$ and $s_1 + s_2$ must ensure that $s1$ and s_2 use their entities in a consistent manner. This task is accomplished by function f defined above. We therefore require

$$\frac{\Gamma_1 \vdash s_1 \qquad \Gamma_2 \vdash s_2}{f(\Gamma_1,\Gamma_2) \vdash s_1 \mid s_2}\ (\textbf{Par})$$

in the case of parallel composition. Rule (**Sum**) dealing with summation of services is similar.

The typing rules for High-Lan COWS services are summarised in Table 3.4. Note that the time complexity of the inference algorithm is linear in the number of activities defined in the input service.

Finally, we present a formal encoding of High-Lan COWS services into COWS services. Firstly, an auxiliary function $map : \mathsf{T} \to \mathcal{E} \cup \mathcal{I}$ acting as a mapping from types to COWS entities and service identifiers is defined as

$$
\begin{array}{rcll}
(\!|e!e'|\!)^A_\Lambda & = & \widehat{e}!\widehat{e'} & \widehat{e} \in map(\Lambda(e)) \quad \widehat{e'} \in map(\Lambda(e')) \\
(\!|e?e'.s|\!)^A_\Lambda & = & \widehat{e}?\widehat{e'}.(\!|s|\!)^A_\Lambda & \widehat{e} \in map(\Lambda(e)) \quad \widehat{e'} \in map(\Lambda(e')) \\
(\!|\mathbf{kill}(e)|\!)^A_\Lambda & = & \mathbf{kill}(\widehat{e}) & \widehat{e} \in map(\Lambda(e)) \\
(\!|S|\!)^A_\Lambda & = & \widehat{S} & \widehat{S} \in map(\Lambda(S)) \quad \widehat{S} \in \mathcal{I}_0 \\
 & & & \widehat{S} \in \mathsf{supp}(A) \\
(\!|S(e_1,\ldots,e_j)|\!)^A_\Lambda & = & \widehat{S}(\widehat{e_1},\ldots,\widehat{e_j}) & \widehat{e_i} \in map(\Lambda(n_i)),\ 1 \le i \le j \\
 & & & \widehat{S} \in map(\Lambda(S)) \quad \widehat{S} \in \mathcal{I}_j \\
 & & & \widehat{S}(\widehat{e_1},\ldots,\widehat{e_j}) \in \mathsf{supp}(A) \\
(\!|[e]\,s|\!)^A_\Lambda & = & [\widehat{e}]\,(\!|s|\!)^A_\Lambda & \widehat{e} \in map(\Lambda(e))
\end{array}
$$

Table 3.5: Formal encoding of High-Lan COWS services into COWS services.

follows:

$$
map(\tau) = \begin{cases} \mathcal{N} & \text{if } \tau = \mathsf{N} \\ \mathcal{V} & \text{if } \tau = \mathsf{V} \\ \mathcal{K} & \text{if } \tau = \mathsf{K} \\ \mathcal{I}_i & \text{if } \tau = \mathsf{R}_i \text{ with } i \in \mathbb{N} \end{cases}
$$

The encoding, indicated by $(\!|\_|\!)^{\_}_{\_} : S \times 2^{\mathcal{D}} \times \mathfrak{E} \to \mathcal{S}$, is defined in Table 3.5, where trivial rules for parallel composition, summation, protection and empty activities are not listed. Note that the encoding is defined for valid environments, i.e. Λ has to be a valid environment in $(\!|\_|\!)^{\_}_{\Lambda}$. The rôle of the encoding is to substitute generic High-Lan COWS entities (i.e. not typed) with COWS entities, according to the type directives stored in environment Λ. Moreover, it is checked that every service identifier is associated with a valid defining equation in set A. Observe that this step is indeed necessary because the High-Lan COWS syntax does not make this sort of assumption, while, on the contrary, the COWS syntax does. Set A is regarded as an environment where names are the left hand-sides of the equations and types are the right hand-sides. COWS entities and service identifiers are indicated in the encoding by the notation $\widehat{e}$ and $\widehat{S}$ respectively.

Lemma 14. *Whenever $\Gamma \vdash s$ holds, if $\Lambda = check(\Gamma)$ is a valid environment for High-Lan COWS service s, then, for some $A \subset \mathcal{D}$, COWS service $s' = (\!|s|\!)^A_\Lambda$ satisfies the following properties:*

(i) s' is a closed service;

(ii) there are no mismatching service identifiers in s';

(iii) a defining equation $a \in A$ is associated to each service identifier in s'

Proof. To prove the lemma, we prove each point separately.

(i) By Definition 3, we have to show that there are no free occurrences of killer labels and variables in service s'. Trivially, s' does not have free killer labels by construction of function *check*. Let us now assume Λ is a valid environment for service s, and $x_1, \ldots, x_j$ are all the free variables of s. This means that $\Lambda(x_i) = \mathsf{V_f}$, with $1 \leq i \leq j$. By definition of functions *check* and $\uplus$, this is impossible. As a matter of fact, the third case in the definition of *check* states that all the entities previously typed as $\mathsf{V_f}$ are instead typed as N. Therefore, there are no free variables occurring in s'.

(ii) Let us assume $\Lambda = \mathit{check}(\Gamma)$ is a valid environment for service s, and S_1, S_2 are two service identifiers in s such that $\Lambda(S_1) = \mathsf{R}_i$ and $\Lambda(S_2) = \mathsf{R}_j$, with $i \neq j$. Since service identifiers are terminal symbols in the High-Lan COWS grammar the only two rules we have to consider are (**Par**) and (**Sum**). In both cases, environment Γ is given by $f(\Gamma_1, \Gamma_2)$. Assuming $S_1 \in \mathsf{supp}(\Gamma_1)$ and $S_2 \in \mathsf{supp}(\Gamma_2)$, function f falls in its second case. Therefore, $\Gamma = \mathsf{fail}$ and consequently $\Lambda = \mathsf{fail}$ by definition of function *check*. This is a contradiction because fail is not a valid environment.

(iii) By construction of encoding $(\!|\_|\!)^{-}_{-}$.

□

Let us present some examples showing how the type system works.

Example 15. Consider the following High-Lan COWS service:

$$s_0 = \mathbf{kill}(k1) \,|\, [k2]\,[x]\,p?x.\,\mathbf{kill}(k2)$$

As can be seen, entity $k1$ occurs free, thus we expect the type system to fail. This is indeed shown by the derivation below:

$$\dfrac{k1 : \mathsf{K_f} \vdash \mathbf{kill}(k1) \qquad \dfrac{\dfrac{\dfrac{\dfrac{k2 : \mathsf{K_f} \vdash \mathbf{kill}(k2)}{k2 : \mathsf{K_f}, p : \mathsf{N}, x : \mathsf{V_f} \vdash p?x.\,\mathbf{kill}(k2)}\,(\mathbf{ImpW})}{k2 : \mathsf{K_f}, x : \mathsf{V}, p : \mathsf{N} \vdash [x]\,p?x.\,\mathbf{kill}(k2)}\,(\mathbf{DelV})}{k2 : \mathsf{K}, x : \mathsf{V}, p : \mathsf{N} \vdash [k2]\,[x]\,p?x.\,\mathbf{kill}(k2)}\,(\mathbf{DelK})}{\Gamma \vdash s_0}\,(\mathbf{Par})$$

where $\Gamma = \{k1 : \mathsf{K_f}, k2 : \mathsf{K}, x : \mathsf{V}, p : \mathsf{N}\}$ is the temporary environment. The final environment Λ is given by

$$\Lambda = \mathit{check}(\Gamma) = \mathsf{fail}$$

Example 16. Let us define a High-Lan COWS service as follows:

$$s_0 = \{\!|\, n1!n2 \,|\, p?x.\mathbf{0} \,|\!\}$$

The inferred environment is

$$\Lambda = \mathit{check}(\Gamma) = \{n1 : \mathsf{N}, n2 : \mathsf{N}, p : \mathsf{N}, x : \mathsf{N}\}$$

where temporary environment Γ is computed by the derivation below.

$$\dfrac{\dfrac{n1 : \mathsf{V_f}, n2 : \mathsf{V_f} \vdash n1!n2 \qquad \dfrac{\varnothing \vdash \mathbf{0}}{p : \mathsf{N}, x : \mathsf{V_f} \vdash p?x.\mathbf{0}}\,(\textbf{InpW})}{n1 : \mathsf{V_f}, n2 : \mathsf{V_f}, p : \mathsf{N}, x : \mathsf{V_f} \vdash n1!n2 \,|\, p?x.\mathbf{0}}\,(\textbf{Par})}{\underbrace{n1 : \mathsf{V_f}, n2 : \mathsf{V_f}, p : \mathsf{N}, x : \mathsf{V_f}}_{\Gamma} \vdash s_0}\,(\textbf{Prot})$$

This example shows how all the occurrences of free variables in temporary environment Γ are replaced with names by function *check*.

Example 17. Consider service $s = [x]p?x.S(p) \,|\, S(p1, p2)$. Here, service identifier S is used with two different arities. This means that a valid environment for service s cannot be inferred by the type system. As a matter of fact, function f fails to merge the two environments when rule (**Par**) is applied as it is shown in the following derivation tree:

$$\dfrac{\dfrac{\dfrac{p : \mathsf{N}, S : \mathsf{R}_1 \vdash S(p)}{p : \mathsf{N}, S : \mathsf{R}_1, x : \mathsf{V_f} \vdash p?x.S(p)}\,(\textbf{ImpW})}{p : \mathsf{N}, S : \mathsf{R}_1, x : \mathsf{V} \vdash [x]p?x.S(p)}\,(\textbf{DelV}) \qquad p1 : \mathsf{N}, p2 : \mathsf{N}, S : \mathsf{R}_2 \vdash S(p1, p2)}{\mathsf{fail} \vdash s}\,(\textbf{Par})$$

Hence, the final environment is $\Lambda = \mathit{check}(\mathsf{fail}) = \mathsf{fail}$.

Example 18. Applying the typing rules listed in Table 3.4 to $[k](\mathbf{kill}(k) \,|\, p!k)$ we get:

$$\dfrac{\dfrac{k : \mathsf{K_f} \vdash \mathbf{kill}(k) \qquad p : \mathsf{V_f}, k : \mathsf{V_f} \vdash p!k}{\mathsf{fail} \vdash \mathbf{kill}(k) \,|\, p!k}\,(\textbf{Par})}{\mathsf{fail} \vdash [k](\mathbf{kill}(k) \,|\, p!k)}\,(\textbf{DelV})$$

In this case function f reports the conflicting type assignments for entity k when rule (**Par**) is applied.

$$\Gamma \cup \{(S : \mathsf{R}_0)\} \vdash S = s \ (\textbf{Def0})$$

$$\frac{\Gamma \cup \{(n_i : \mathsf{N}) \mid i \in \{1,\ldots,j\}\} \cup \{(S : \mathsf{R}_j)\} \vdash S(n_1,\ldots,n_j) = s \ (\textbf{Def}) \quad \Gamma_1 \vdash a_1 \quad \ldots \quad \Gamma_n \vdash a_n \qquad n = |A| \quad a_i \in A \quad 1 \le i \le n}{\Gamma_1 \uplus \ldots \uplus \Gamma_n \vdash A} \ (\textbf{Defs})$$

Table 3.6: Typing rules for High-Lan COWS defining equations.

Example 19. The typing for service $[x]p?x.\mathbf{0} \mid p!n$ is given by

$$\dfrac{\dfrac{\dfrac{\dfrac{\varnothing \vdash \mathbf{0}}{p : \mathsf{N}, x : \mathsf{V_f} \vdash p?x.\mathbf{0}} \ (\textbf{InpW}) \qquad p : \mathsf{V_f}, n : \mathsf{V_f} \vdash p!n}{p : \mathsf{N}, x : \mathsf{V_f}, n : \mathsf{V_f} \vdash p?x.\mathbf{0} \mid p!n} \ (\textbf{Par})}{\underbrace{p : \mathsf{N}, x : \mathsf{V}, n : \mathsf{V_f}}_{\Gamma} \vdash [x]p?x.\mathbf{0} \mid p!n} \ (\textbf{DelV})$$

and by $\Lambda = \mathit{check}(\Gamma) = \{p : \mathsf{N}, x : \mathsf{V}, n : \mathsf{N}\}$. Application of rule (**Par**) shows the priority given by function f to type N over type $\mathsf{V_f}$ for the typing of entity p.

At this stage, it is useful to extend the typing system and the encoding to handle defining equations. In what follows, metavariables A, A', B and B' will be used to range over sets D and $\mathcal{D}$. Firstly, we introduce the typing rules listed in Table 3.6. The intended meaning of (**Def0**) and (**Def**) is analogous to the one of rules (**Rec0**) and (**Rec**) in Table 3.4. It is worthwhile to note that service s is ignored. Rule (**Defs**) collects all the typing environments of the defining equations belonging to set A. Whenever mismatching service identifiers are encountered, only the last one is considered, while the previous are discarded. This behaviour is due to definition of function $\uplus$. As a consequence, the inferred environment is always valid. The inference of this rule is computable because $A \subset D$ is assumed to be finite.

Finally, $(\!|\_|\!)^{\_}_{\_}$ is extended to encode a set A of High-Lan COWS defining equations into a set $A' \subset \mathcal{D}$ of COWS defining equations as follows:

$$(\!|A|\!)_{\Gamma} = \|A_0\|_A \ \cup \ \|A_n\|_A \tag{3.4}$$

where

$$\begin{aligned} A_0 &= \{\widehat{S} = s \mid S \in \mathsf{supp}(A) \wedge \widehat{S} \in \mathit{map}(\Gamma(S)) = \mathcal{I}_0\} \\ A_n &= \{\widehat{S}(\widehat{e_1},\ldots,\widehat{e_j}) = s \mid S(e_1,\ldots,e_j) \in \mathsf{supp}(A) \wedge \widehat{S} \in \mathit{map}(\Gamma(S)) = \mathcal{I}_j \wedge \widehat{e_i} \in \mathcal{N}\} \end{aligned}$$

with $1 \le i \le n$, and transformation function $\|\_\|_{\_}$ as shown in Table 3.7. The en-

$$
\begin{array}{rcll}
\|\widehat{S} = s\|_A & = & \widehat{S} = (\!|s|\!)^A_\Lambda & \Lambda = check(\Lambda') \quad \Lambda' \vdash s \\
\|\widehat{S}(\widehat{n_1},\ldots,\widehat{n_j}) = s\|_A & = & \widehat{S}(\widehat{n_1},\ldots,\widehat{n_j}) = (\!|s|\!)^A_\Lambda & \Lambda = check(\Lambda') \quad \Lambda' \vdash s \\
 & & & \Lambda(n_i) = \mathsf{N} \quad 1 \leq i \leq j
\end{array}
$$

Table 3.7: Transformation $\|\_\|\_$.

coding initially processes every defining equation by converting the left hand-sides into COWS. After that, it takes care of the encoding of the right hand-sides. The first phase is carried out with the creation of sets A_0 and A_n, whereas transformation function $\|\_\|\_\_$ terminates the transformation procedure. Note that the latter is defined only for valid environments, i.e. typing environment Λ inferred form a right hand-side has to be valid. Observe also that, equations specifying service identifiers with mismatching arities are removed. This is expressed by the equality constrains $map(\Gamma(S)) = \mathcal{I}_0$ and $map(\Gamma(S)) = \mathcal{I}_n$ in the definition of A_0 and A_n, respectively.

Example 20. Let service $s = [x](p?x.S(n_1) + p?x.S(n_2))$ be a High-Lan COWS service and A a set of defining equations:

$$A = \{S = n!m, S(n) = n!m\}$$

The environments inferred from s and A, are $\Gamma = \{(p : \mathsf{N}), (n_1 : \mathsf{N}), (n_2 : \mathsf{N}), (x : \mathsf{V})\}$ and $\Gamma_D = \{(S : \mathsf{R}_1)\}$, respectively. Note that the typing for the first defining equation is lost. Before converting s into a COWS service by means of encoding $(\!|\_|\!)^{\_}_{\_}$, set A has to be transformed. For this purpose we use the encoding $(\!|A|\!)_{\Gamma_D}$. We show all the intermediate steps. Initially, set $A_n = \{\widehat{S(n)} = n!m\}$ is created, where $\widehat{S} \in \mathcal{I}_1$ is a COWS service identifier and $\widehat{n} \in \mathcal{N}$ is a COWS name. After that,

$$A' = (\!|A|\!)_{\Gamma_D} = \|A_n\|_A = \{\widehat{S(n)} = (\!|n!m|\!)^A_\Lambda\} = \{\widehat{S(n)} = \widehat{n}!\widehat{m}\}$$

where $\Lambda = \{(n : \mathsf{N}), (m : \mathsf{N})\}$ and $\widehat{n}, \widehat{m} \in \mathcal{N}$ are COWS entities. Finally, service s can be translated:

$$(\!|s|\!)^{A'}_\Gamma = [\widehat{x}](\widehat{p}?\widehat{x}.\widehat{S}(\widehat{n_1}) + \widehat{p}?\widehat{x}.\widehat{S}(\widehat{n_2}))$$

where all the High-Lan COWS entities are converted into equivalent COWS entities by function map. In particular, $\widehat{x} \in \mathcal{V}, \widehat{p}, \widehat{n_1}, \widehat{n_2} \in \mathcal{N}$ and $S \in \mathcal{I}_1$.

$$\frac{s \lhd_g}{S(n_1,\ldots,n_j) = s \blacktriangleleft_g} \qquad \frac{s \lhd_g}{S = s \blacktriangleleft_g} \qquad \frac{a_1 \blacktriangleleft_g \wedge \ldots \wedge a_n \blacktriangleleft_g \qquad n = |A| \quad a_i \in A \quad 1 \leq i \leq n}{A \blacktriangleleft_g}$$

Table 3.8: Are right hand-sides all guarded?

3.2.4 Compiler: details

The compiler can be thought of as a black-box device accepting some sort of data and producing an output as the result of its internal computations on the given input. Before presenting the details of how its sub-components interact to produce the desired result, we describe function $\_ \blacktriangleleft_g$, an useful predicate over sets of COWS defining equations. It is introduced in Table 3.8. Expression $A \blacktriangleleft_g$ is verified only when all the right hand-sides s of the defining equations in set A are guarded services, where $A \subset \mathcal{D}$ and predicate $\_ \lhd_g$ is defined in Table 2.2.

Definition 21 (Valid COWS program)**.** Let A be a set of defining equations, $\{s_1,\ldots,s_n\}$ the set of its right hand-sides, and $Z = \langle s, A, \mathcal{E}nv \rangle$ a COWS program. We write s' to denote a service in the set $s \cup \{s_1,\ldots,s_n\}$. Program Z is a *valid program* if

(i) s' is a homonymy free and closed COWS service and it does not have mismatching service identifiers;

(ii) for every service identifier S occurring in s', $S \in \mathsf{supp}(A)$;

(iii) $A \blacktriangleleft_g$ holds;

(iv) $\mathsf{supp}(\mathcal{E}nv) = (\mathsf{fe}(s) \cup \mathsf{be}(s)) \cup \bigcup_{i=1}^{n} (\mathsf{fe}(s_i) \cup \mathsf{be}(s_i))$ i.e. a rate is associated to each entity used in Z.

The translation engine takes as input valid COWS programs and converts them into CTMCs. Therefore, the compiler can really be viewed as a procedure for their generation starting from the corresponding High-Lan COWS model specification. We define such a procedure in Table 3.10 by heavily relying on the results presented earlier in this chapter. Function $\hookrightarrow_v : \mathsf{Z} \to \mathcal{Z}$ is only defined for initial environments $\mathcal{E}nv'$ such that a rate is associated to every entity occurring in s' and A'. The first step performed by $\hookrightarrow_v$ is to refresh the input program

input: High-Lan COWS program $Z' = \langle s', A', \mathcal{E}nv' \rangle$

1. Program Z' is refreshed according to Equation 3.2. The resulting High-Lan COWS program is $Z'' = [\![Z']\!] = \langle s'', A'', \mathcal{E}nv \rangle$.
2. Typing environments for s'' and A'' are inferred. They are $\Gamma = check(\Gamma_t)$, where $\Gamma_t \vdash s''$ and $\Gamma_D = check(\Gamma'_t)$, where $\Gamma'_t \vdash A''$.
3. The set of defining equations A'' is encoded into COWS as described in Equation 3.4. The result is: $A = (\!|A''|\!)_{\Gamma_D}$.
4. High-Lan COWS service s'' is converted into COWS by the encoding described in Table 3.5. The result is $s = (\!|s''|\!)^A_\Gamma$.
5. Property $A \blacktriangleleft_g$ is tested.

output: COWS program $Z = \langle s, A, \mathcal{E}nv \rangle$

Table 3.9: Function $\_ \hookrightarrow_v \_$

Z. Subsequently, typing environments are inferred for service s'' and set of defining equations A''. Observe that the procedure is assumed to terminate with an error whenever $\Gamma_D = \mathsf{fail}$ or $\Gamma = \mathsf{fail}$. On the contrary, if the inference is successful, the computation of the output COWS program Z is carried on. Also when an invocation of $(\!|\_|\!)^{\_}_{\_}$ fails in the translation, $\hookrightarrow_v$ is interrupted. Finally, it is checked whether all the services used as right hand-sides are guarded. If A passes this test, COWS service Z is used as output value. The function has linear time complexity.

Lemma 22. *Let $Z' = \langle s', A', \mathcal{E}nv' \rangle$ be a High-Lan COWS program. Program $Z = \langle s, A, \mathcal{E}nv \rangle$, obtained by $Z' \hookrightarrow_v Z$ is a valid program.*

Proof. We prove each point in Definition 21 separately. In the following s' stands for a service belonging to set $s \cup R$, where $R = \{s_1, \ldots, s_n\}$ is the set of the right hand-sides of A.

(i) By construction of function $\hookrightarrow_v$ (step 1) and encoding $[\![\_]\!]_{\_}$, service s' is homonymy-free. Moreover, by construction of $\hookrightarrow_v$ (step 2) and Lemma 14,

s' is closed and no mismatching service identifiers occur in it. Finally, s' is a COWS service by construction of $\hookrightarrow_v$ (steps 3 and 4) and $(\!|\_|\!)^-_-$.

(ii) By construction of $\hookrightarrow_v$ (steps 3 and 4) and $(\!|\_|\!)^-_-$.

(iii) By construction of $\hookrightarrow_v$ (step 5) and $\_ \blacktriangleleft_g$.

(iv) By construction of $\hookrightarrow_v$ (step 1) and $[\![\_]\!]_-$.

□

3.3 Translation engine

The translation of a valid COWS program Z into a CTMC $\mathcal{C}(Z)$ is conceptually related to two different sub-procedures: The first one generates the stochastic transition system associated to Z, while the second transforms it into a transition matrix fully specifying $\mathcal{C}(Z)$.

Definition 23. The *labelled stochastic reduction relation* $\stackrel{\theta}{\Rightarrow}$ is the least relation over valid COWS programs such that:

$$\langle s, A, \mathcal{E}nv\rangle \stackrel{\theta}{\Rightarrow} \langle s', A, \mathcal{E}nv\rangle \quad \text{if } s \stackrel{\theta}{\rightarrow} s'$$

where $\stackrel{\theta}{\rightarrow}$ is the stochastic COWS reduction relation described in Section 2.4, but rules

$$\frac{s\{m_1,\ldots,m_j/n_1,\ldots,n_j\} \xrightarrow{\vartheta(\alpha,\varphi\varphi')} s' \quad A(S(n_1,\ldots,n_j)) = s}{S(m_1,\ldots,m_j) \xrightarrow{\vartheta(l\_dec(\alpha),\varphi\varphi')} s\_dec(\alpha,s')}\ (\textbf{ser id})$$

$$\frac{s \xrightarrow{\vartheta(\alpha,\varphi\varphi')} s' \quad A(S) = s}{s \xrightarrow{\vartheta(l\_dec(\alpha),\varphi\varphi')} s\_dec(\alpha,s')}\ (\textbf{ser id0})$$

As can be seen, the stochastic evolution of a program is fully driven by the evolution of its service component. The new definition of rules (**ser id**) and (**ser id0**) is used to handle the unfolding of recursion through the environment of defining equations A. Moreover, it is important to remark that function ρ used in the stochastic semantics of COWS to retrieve rates associated to entities is overridden by the homonym function defined in Section 3.2.1. In this case, rates are taken from global environment $\mathcal{E}nv$. We extend Definition 8 to programs:

input: COWS program $Z = \langle s_0, A, \mathcal{E}nv \rangle$

Define a CTMC $C = (Q, s_0, \mathbf{R})$ where $Q = \emptyset$ and $\mathbf{R} = \emptyset$
Initialise sets $Q = \mathcal{Q} = \{\omega(s_0)\}$
repeat
pic a COWS service $s \in \mathcal{Q}$ and remove it from $\mathcal{Q}$
for all (θ, s') such that $\langle s, A, \mathcal{E}nv \rangle \stackrel{\theta}{\Rightarrow} \langle s', A, \mathcal{E}nv \rangle$ is a stoch. red. step
if $\omega(s') \notin Q$
then add $\omega(s')$ to Q and $\mathcal{Q}$ and add $(s, \omega(s'), \mu(\theta))$ to $\mathbf{R}$
else update $(s, \omega(s'), r) \in \mathbf{R}$ with $(s, \omega(s'), r + \mu(\theta))$
until $\mathcal{Q} \neq \emptyset$

output: transition matrix $\mathbf{R}$

Table 3.10: Function $\_ \rightsquigarrow_{CTMC} \_$.

Definition 24 (Stochastic computation step)**.** Reduction $Z \xrightarrow{\vartheta(\alpha,\varphi,\varphi')} Z'$ is called a *stochastic computation step* if $\alpha = \dagger$ or $\alpha = p \cdot \varepsilon \cdot \sigma'$, for some p and σ'.

For the sake of brevity, we omit similar extensions of Definition 6 (derivative) and Definition 7 (derivative set).

The CTMC associated to a program Z is directly computed by procedure $\rightsquigarrow_{CTMC}: \mathcal{Z} \to \mathcal{C}(\mathcal{Z})$ in Table 3.10. Its definition is inspired by the work presented in (Deng and Papadimitriou, 1990). In that paper, the authors investigate a new algorithm for the exploration of an unknown graph. The algorithm driving the computation of function $\rightsquigarrow_{CTMC}$ is a modified version of a breadth-first graph traversal. Hence, the time complexity is $O(|\Psi(Z)| + |E_\Psi|)$, where the two variables are the set of nodes and edges of the Z-transition system, respectively. It is worthwhile to notice that, although this is a linear bound, the number of states can be exponential with respect to the syntactic length of the programs. The only modification with respect to the breadth-first traversal, is that multiple connections between nodes have to be handled. Another important point is that services are processed by function ω before being added to the Markov chain. Intuitively, $\omega(s)$ is a syntactic modification which removes unneeded empty activities and protections from s. It is specified in Table 3.11. Note that, $\rightsquigarrow_{CTMC}$ is not computable in general. As a matter of fact, it is only defined for programs having finite derivative sets.

$\omega(\{\!\{\{\!\{s\}\!\}\}\!\}) = \{\!\{\omega(s)\}\!\}$	$\omega(\{\!\{\mathbf{0}\}\!\}) = \mathbf{0}$
$\omega([d]\mathbf{0}) = \mathbf{0}$	$\omega(s \mid \mathbf{0}) = \omega(\mathbf{0} \mid s) = \omega(s)$
$\omega(p?u.s) = p?u.\omega(s)$	$\omega(s + \mathbf{0}) = \omega(\mathbf{0} + s) = \omega(s)$
$\omega(u!w) = u!w$	$\omega(\mathbf{kill}(k)) = \mathbf{kill}(k)$
$\omega([d]s) = [d]\omega(s)$	$\omega(S) = S$
$\omega(S(n_1,\ldots,n_j)) = S(n_1,\ldots,n_j)$	$\omega(\{\!\{s\}\!\}) = \{\!\{\omega(s)\}\!\}$

Table 3.11: Definition of function $\omega(\_)$.

Summarising, the COWS2Prism system is defined as follows:

$$Z_1 \hookrightarrow_v Z_2 \rightsquigarrow_{CTMC} \mathbf{R} \tag{3.5}$$

where Z_1 is the High-Lan COWS input service, Z_2 is the COWS service used by the COWS2Prism system for its internal computations, while the output $\mathbf{R}$ is the transition matrix of a CTMC generated from Z_2.

3.3.1 Recursion handling: discussion

At this stage, it is of paramount importance to discuss the management of recursion unfolding by the COWS2Prism system.

We already recalled that translation procedure $\rightsquigarrow_{CTMC}$ converges only on programs generating finite transition systems. However, the stochastic semantics only guarantees that a transition system is finitely branching. This is not enough because the set of states can still be infinite. Another important observation is that infinite behaviours can be described by finite state transition systems. Consider the following examples.

Example 25. We define a COWS program $Z = \langle s, A, \mathcal{E}nv\rangle$ where $s = S(p)$, $A = \{S(p) = [x]p?x.S(p) \mid p!n \mid q!n\}$ and $\mathcal{E}nv = \{(p : \lambda_1),(x : \lambda_2),(n : \lambda_3),(q : \lambda_4),\}$. It is immediate to see that Z gives rise to an infinite transition system, because at every recursive step an additional $q!n$ is added and it can never be consumed. Namely,

$$\begin{aligned} Z \;&\stackrel{\theta}{\Rightarrow} \langle S(p) \mid \mathbf{0} \mid q!n, A, \mathcal{E}nv\rangle \stackrel{\theta}{\Rightarrow} \langle S(p) \mid \mathbf{0} \mid q!n \mid \mathbf{0} \mid q!n, A, \mathcal{E}nv\rangle \\ &\stackrel{\theta}{\Rightarrow} \ldots \stackrel{\theta}{\Rightarrow} \langle S(p) \mid \mathbf{0} \mid q!n \mid \ldots \mid \mathbf{0} \mid q!n, A, \mathcal{E}nv\rangle \end{aligned}$$

Thus, the main loop in $\rightsquigarrow_{CTMC}$ keeps adding processes to sets Q an $\boldsymbol{Q}$ and never terminates.

Example 26. Let program $Z = \langle S(p), A, \mathcal{E}nv, \rangle$, where, $A = \{S(p) = [x]p?x.S(p) | p!n\}$ and $\mathcal{E}nv$ is a valid environment. In this case $\rightsquigarrow_{CTMC}$ converges, producing a one-state Markov chain. Initially, sets $Q = \mathcal{Q} = \{S(p)\}$, while $\mathbf{R}$ is empty. Note that the application of function ω does not affect $S(p)$. In the main loop, $S(p)$ is used to evaluate computational step $Z \stackrel{\theta}{\Rightarrow} \langle s_1, A, \mathcal{E}nv \rangle$, where $s_1 = S(p) | \mathbf{0}$. The empty activity in the parallel composition is removed with $\omega(s_1)$. Hence, Q is not changed because it already contains state $S(p)$, while element $(S(p), s_1, \mu(\theta))$ is added to $\mathbf{R}$ and $\mathcal{Q} = \emptyset$. The procedure terminates with output $\mathbf{R}$.

To tackle these issues several approaches are viable:

- restrict the calculus by removing recursive definitions;
- restrict the syntax of the calculus in such a way that infinite transition systems are impossible to be generated;
- ignore the problem.

In the first case, all infinite behaviours cannot be expressed in the reduced calculus. Since COWS was introduced to model interactions among web services, where infinite behaviours are common, this restriction introduces an unacceptable lack of expressiveness. Therefore, we discarded this solution.

In the second case, a possible solution is to forbid parallel composition in recursive definitions. This very approach was indeed adopted in (Deng et al., 2005) when the authors analysed the properties of transition systems generated by several different fragments of CCS. Despite the fact that this way of dealing with the problem appears appealing, we also notice that it is too restrictive. As a matter of fact, not only are services of the kind presented in Example 25 not legal, but also services like the one in Example 26 are forbidden. This is not desirable because we are losing the ability to express services giving finite transition systems.

The last proposal cannot really be considered a proper solution. Nevertheless, it is the approach we adopted in the COWS2Prism system implementation. Our choice is motivated by the fact that the major limitation in CTMC analysis is in the number of states of the chain. As a matter of fact, analysis based on CTMCs becomes unproductive for models with more than one million states (see (Schweitzer, 1990)). Therefore, the translation process in COWS2Prism

is interrupted whenever the number of states exceeds a fixed limit. In this manner, our program is assured to be always terminating.

Chapter 4

COWS2Prism System Implementation

The COWS2Prism system has been implemented in a functional language (OCaml), attaining to the design presented in the previous chapter. The system is a command-line application taking as input a text file describing a High-Lan COWS model. It consists of a single binary executable called `cows2prism`.

The purpose of COWS2Prism is to compute the CTMC corresponding to the stochastic transition system of a given High-Lan COWS service. The resulting transition matrix is stored in an `ASCII` file which can be accepted as input by the stochastic model checker Prism and then analysed. Additionally, a static type-checker accurately reports syntax and type errors before a given source file is executed. The complete specification of the High-Lan COWS language used for the definition of models in the input files is reported in Appendix A.

A general overview of the components organisation in the COWS2Prism system is depicted in Figure 4.1, while a detailed dependency graph between each source file (i.e. compilation unit) is drawn in Figure 4.2. The graph has been computed by `ocamldep`, the dependency analysis tool included in the OCaml distribution, and `ocamldot`, a graph generation utility. We also took advantage of the features of the first tool for the generation of compiling directives in `Makefile` format.

The `cows2prism` program is implemented in the `main.ml` file. However, the logic regarding the real functioning of the software is handled separately into several different implementation files. Therefore, the rôle of `main.ml` is limited to the organisation of the order of invocation of the several sub-components

Figure 4.1: COWS2Prism system components organisation.

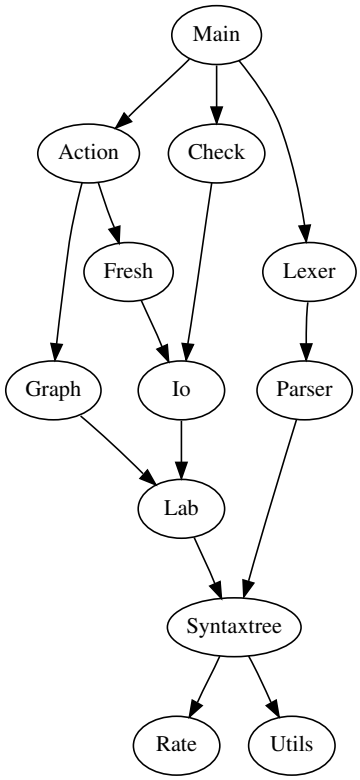

Figure 4.2: Compilation units

in the COWS2Prism system. Moreover, global data structures with in-place modification are initialised, e.g. hash-tables `rates` and `env` for the name-rate bindings and the defining equations, respectively.

In the remainder of this chapter we will explain the most important features of the modules forming the COWS2Prism system. Each component is presented in the order given in Figure 4.1, corresponding to the order of invocation in the main module. Thus, the flow of computation proceeds from one unit to the other in a *cascade* fashion. Finally, a brief description of the usage of the program is included.

4.1 Lexer

The lexer is the front-end component of the COWS2Prism system. It processes the source file (with extension `.cow`) producing a sequence of lexical units called *tokens* or *lexemes*. At this stage, the input file is considered as a raw sequence of characters and each token corresponds to a specific construct in the High-Lan COWS input. The recognition process is driven by the regular expressions listed in Appendix A.1. Observe that, despite the introduction of several new key-words in the source language, these rules closely resemble the High-Lan COWS syntax given in Table 3.1. The actual implementation (i.e. file `lexer.ml`) of the lexer in our system is generated by `ocamllex`, the standard OCaml lexical analyser generator. For this purpose, the lexical description is coded in file `lexer.mll`. We briefly comment on the most important sorts of token built during the lexing phase: service identifiers and entity identifiers. The firsts are generated by rule

$$SID \;=\; [\,'\mathbf{A}'-'\mathbf{Z}'\,]\;[\,'\mathbf{a}'-'\mathbf{z}'\;\;'\mathbf{A}'-'\mathbf{Z}'\;\;'\mathbf{0}'-'\mathbf{9}'\;\;'\_'\,]^{*}$$

stating that a string can be regarded as a service identifiers only if it begins with a capital letter. The seconds are recognised by the following regular expression:

$$EID \;=\; [\,'\mathbf{a}'-'\mathbf{z}'\,]\;[\,'\mathbf{a}'-'\mathbf{z}'\;\;'\mathbf{A}'-'\mathbf{Z}'\;\;'\mathbf{0}'-'\mathbf{9}'\;\;'\_'\,]^{*}$$

In this case, entity identifiers can only start with a lower-case letter. In both rules, the only legal characters are letters and '_'.

Finally, tokens corresponding to floating point numbers used for rates values are identified by

$$\begin{aligned} FLOAT \;=\;\; & \Big([\,'\mathbf{0}'-'\mathbf{9}'\,]^{+} \;\Big|\; \Big([\,'\mathbf{0}'-'\mathbf{9}'\,]^{+}\;'.'\,[\,'\mathbf{0}'-'\mathbf{9}'\,]^{+}\Big)\Big) \\ & \Big([\,'\mathbf{E}'\;'\mathbf{e}'\,]\;[\,'+'\;'-'\,]?\;[\,'\mathbf{0}'-'\mathbf{9}'\,]^{+}\Big)? \\ \Big|\;\; & \text{"}\mathbf{inf}\text{"} \end{aligned}$$

The actual transformation into OCaml `float` data type is performed by the library function `float_of_string`.

4.2 Parser

The parser is the sub-system assembling the tokens produced by the lexer so that they amount to syntactically correct sentences in the High-Lan COWS language. In our case, the syntactic assembly rules are defined by the Backus-Naur

Form grammar defined in Appendix A. The implementation file `parser.ml` is efficiently generated starting from YACC definition file `parser.mly` by the `ocamlyacc` tool. Let us explain the main points in the grammar definition.

A `.cow` program is composed by a mandatory list of rate declarations, an optional list of service defining equations and a service. This is given by the following fragment of the grammar:

$$\mathit{program} \ ::= \ \mathit{rates\_list} \ \ \mathit{opt\_def\_list} \ \ \textbf{in} \ \ \mathit{service}$$

Rate declarations are bindings between High-Lan COWS entities and floating point rates, recognised by lexing rules *EID* and *FLOAT* respectively. They are defined by:

$$\mathit{rate} \ ::= \ \textbf{rate} \ \ \mathit{EID} \ \ : \ \ \mathit{FLOAT} \ \ ;$$

Note that the last rate declaration has always to be a global rate declaration (like for example: `baserate : 1.5;`).

The defining equations list is formed by semicolon-separated equations defined by

$$\mathit{def} \ ::= \ \textbf{let} \ \ \mathit{SID} \ \ (\ \ \mathit{opt\_names\_list} \ \ ) \ \ = \ \ \mathit{service}$$

Service identifiers are tokenised by rule *SID*, whereas *opt_names_list* denotes a list of comma-separated *EID* lexemes. Since the latter is an optional element, service identifiers with no arguments are legal in the High-Lan COWS language.

For the missing grammar rules we refer the reader to Appendix A. We just mention that rule *service* is conceptually analogous to rule *s* in Table 3.1. The additional rules have only been added for the handling of unnecessary parenthesis.

Another important rôle of the parser is to translate the abstract entities induced by the grammar into actual OCaml values mimicking the High-Lan COWS specifications. This is indeed possible by associating to every rule in the grammar file `parser.mly` an OCaml function to be triggered whenever its semantic value is recognised in the input. This mechanism allows `ocamlyacc` to easily perform the required transformation. We will present the data types used in this task in the following section.

4.3 Data types

Before starting, it is worth to clarify that the same data types are used both in the High-Lan COWS and in the COWS implementation. This means that, from an OCaml prospective, there is no difference between the two formalisms. We adopted this approach to simplify the implementation of the translation $(\!|\_|\!)^{-}$ defined in Table 3.5. Therefore, in the following we may generally refer to *program*, *process* and *entity* without any distinction.

All data type declarations are contained in files `rate.ml` and `syntaxtree.ml`. The first states that a `rate` is implemented as a (`string`,`float`) pair, while the second collects all the other type declarations. The basic entities are implemented as `string`:

```
type name  =  Name of string
type var   =  Var of string
type klab  =  Klab of string
type id    =  Id of string
```

Note that, in COWS services, types `name`, `var`, `klab` and `id` are used to denote elements of sets $\mathcal{N}$, $\mathcal{V}$, $\mathcal{K}$ and $\mathcal{I}$ respectively. Following the same approach, different subsets of set $\mathcal{E}$ are implemented with different OCaml types. Entities in $\mathcal{N} \cup \mathcal{V} \subset \mathcal{E}$ are represented by type `entity1`, while entities in $\mathcal{N} \cup \mathcal{V} \cup \mathcal{K} = \mathcal{E}$ with type `entity`. This is shown below:

```
type entity1  =  EntN of name | EntV of var
type entity   =  Ent1 of entity1 | EntK of klab
```

As we explained above, the same type nesting is adopted for High-Lan COWS. However, all the details regarding the membership of an entity to a certain set are ignored. Therefore, there is no semantic difference between those types. The implementation of type `service` is as reported in Table 4.1. As can be seen, the distinction between simple services and guarded services introduced in COWS- High-Lan COWS grammars (see Table 2.1 and 3.1 respectively) is preserved. Moreover, we assume that values of type `service` are elements of S or $\mathcal{S}$. Implementations of `def` and `program` are:

```
type def      =  Def of id * name list * service
type program  =  Program of rate list * def list * service
```

Values of type `def` belong to sets D or $\mathcal{D}$. A `program` is the implementation of the abstract program in Definition 10, where the `rate` list is the environment

```
type service  =
                    Out of entity1 * entity1
                  | Guard of guard
                  | Par of service * service
                  | Prot of service
                  | Kill of klab
                  | Del of entity * service
                  | Rec of id * name list
and guard     =
                    Nil
                  | In of name * entity1 * service
                  | Sum of guard * guard
```

Table 4.1: OCaml definition of type `service`.

$\mathcal{E}nv$. It is worth to notice that at this stage, definitions and rates are both implemented as OCaml `list` data type.

The result of the parsing process is a value of type `program` (Z_1 in Equation 3.5) representing the OCaml implementation of the High-Lan COWS program in the original source file. It is then the static analysis phase that translates it into an equivalent COWS program implementation. We will describe the details of this operation in the next section.

4.4 Static analyser

The static analyser is the component of the COWS2Prism system that performs several preliminary checks and transformations on the input program before it is used by the translation engine. Additionally, it creates a global environment for the efficient handling of entity-rate bindings. Note that the system terminates whenever the static analyser fails, i.e. when some property does not hold for the input program. If this is the case, the system reports to the user a detailed message specifying the nature of the error. The implementation of the static analyser is based on the definition of function $\hookrightarrow_v$ described in Table 3.10. It is spread among three different files: `rate.ml`, `check.ml` and `fresh.ml`.

File `rate.ml` defines function `build_env` used for the creation of the global

environment. For efficiency reasons, it is implemented as a hash table data structure with in-place modification. In particular, we used the `(string,float) Hashtbl.t` type provided by the OCaml standard library. The `string` element standing for the name of the entity is used as a key to retrieve in constant time its rate, given by the `float` component.

File `check.ml` implements several checking functions for the input program. The most important of these is undoubtedly function `is_guarded`. As the name suggests, it tests if all the service identifiers occur guarded in the defining equations. Its implementation closely resembles predicate $\blacktriangleleft_g$ defined by the rules listed in Table 3.8. Another function is `is_defined`. It is used to check whether every service identifier is associated with a defining equation. Both functions force the COWS2Prism system to terminate whenever they fail. Finally, functions `warning_unused_rates` and `warning_unused_ids` are defined. The first scans the program looking for unnecessary rate definitions, while the second detects unused service definitions. Note that for these two functions, only a warning message is printed out when an error is found.

The fresh names generator and the type inference algorithm are both implemented by file `fresh.ml`. They are used in this order by the static analyser to modify the `program` generated in the lexing-parsing phase. The result of the transformation is a `program` value corresponding to a valid COWS program as stated in Definition 21. Considering that these two sub-component are of crucial importance in the COWS2Prism system architecture, we will comment more in detail on their implementation in the following.

The fresh names generator is the sub-component used to convert a `program` into a homonymy free equivalent one. It is implemented by function `freshen` by following the rules defining encoding $[\![\ ]\!]$ in Tables 3.2 and 3.3. Each `entity` being refreshed is modified by the addition of a string of zeros preceded by character `~`. Note that the latter is not allowed in the regular expression for *EID*. Hence, this assures that a name generated in this manner is different than all the other names already used in the source file. For example, a killer label `EntK (Klab "k")` becomes `EntK (Klab "k~00")` after the refreshing and string `"k~00"` was not previously used for any other entities. For the efficient handling of sets of entities, we used the set data structure provided by the OCaml standard library. Thanks to this choice, an element in the set can be retrieved in logarithmic time and the insertion of double elements does not affect the set. The code is the following:

```
module Entity_set = Set.Make (struct
  type t = entity
  let compare e1 e2 =
    compare (Io.string_of_entity e1) (Io.string_of_entity e2)
end);;
```

where `Set.Make` is the functor actually building the set structure. The elements are converted into strings by `Io.string_of_entity` an ordered by `compare`.

As described in the previous chapter, after a `program` has been refreshed, it is correctly typed. In our implementation of the COWS2Prism system, this task is performed by the type inference algorithm. Its OCaml implementation is function `fix`. It is defined exactly in the same way of the formal typing system given in Tables 3.4 and 3.6, and encoding $(\!|\_|\!)^{\_}$ in Table 3.5 and Equation 3.4. An important rôle in this function is played by several conversion functions. They are used to transform an `entity` in a different kind of `entity`. For instance, it is possible to transform a variable (i.e. `Ent1 (EntV Var "e")`) into a name (i.e. `Ent1 (EntN Name "e")`) with function `name_of_var`.

4.5 Translation engine

The translation engine is the back-end component of the COWS2Prism system that computes the final output by processing values of type `program` received from the static analyser. It corresponds to function $\rightsquigarrow_{CTMC}$ and therefore, its implementation is based on the algorithm presented in Table 3.10. This can be mainly seen, in file `action.ml`, where function `transition_system` is defined. As a matter of fact, this iterative function is almost identical to the main loop of $\rightsquigarrow_{CTMC}$. The only modification we introduced consists in a condition check that breaks the loop whenever the size of the transition system becomes intractable. In the current implementation of the COWS2Prism system, this happens when the number of states reaches the order of one million states. Note that the partial transition matrix computed till that point is returned as output. The stochastic reduction relation $\stackrel{\theta}{\Rightarrow}$ is implemented by function `trans`, set Q is an OCaml standard queue with in-place modification (i.e. a value of type `service Queue.t`) and sets Q and $\mathbf{R}$ are OCaml sets. Predicates used in the rules of the COWS semantics are implemented as specified in Chapter 2. Auxiliary functions for the management of labels and graphs algorithms are stored in files `rate.ml` and `graph.ml`, respectively. The major advantage offered by

$$
\begin{array}{rcl}
\textit{output} & ::= & \textit{INT}\ \ \textit{INT}\ \ \textbf{newline}\ \ \textit{transition\_list}\ \ \textbf{eof} \\
\textit{transition\_list} & ::= & \textit{transition} \mid \textit{transition}\ \ \textbf{newline}\ \ \textit{transition\_list} \\
\textit{transition} & ::= & \textit{INT}\ \ \textit{INT}\ \ \textit{FLOAT}
\end{array}
$$

Table 4.2: Grammar for CTMCs in Prism format.

the OCaml programming language is that the rules for the generation of the transition system can easily been implemented by means of pattern-matching construct. For example, rule (**kill**) is straightforwardly implemented as follows:

```
let rec trans (s:service) env rates =
  match s with
    | Kill(k) -> let lambda = get_rate rates (Io.string_of_klab k)
        in lab = Enhanced_lab ("",LabK k,Rho(lambda),Rho(lambda))
          in [(Guard Nil,lab)]
    | ...
    | ...
```

4.6 Output

The final output is a plain text file containing the transition matrix of the CTMC generated by the COWS2Prism system starting from an input `.cow` file. It is saved to disk in a format supported by Prism, as specified by the grammar in Table 4.2. The first line is always formed by two integers. The first one stands for the cardinality of the set of states of the CTMC, while the second indicates the number of transitions. Each transition is represented as a new line in which three values are written. The first two integers specify the starting state and the arriving state of the transition, respectively, while the last floating point value denotes its rate. Note that this format resembles the representation of **R** used by the translation engine. Therefore, the output process simply consist in an iteration over the elements of this set, using function `elements` defined in module `Set` of the OCaml standard library.

4.7 Usage

The COWS2Prism system can be executed by invoking the command-line program `cows2prism` as follows:

```
cows2prism [-v] FILE [-o FILE]
```

The first option enables verbose mode, while the second and the last specify an input and an output file, respectively. Note that only the second argument is mandatory. When `-o` option is not used, a file with the same name of the input file but the extension is used to store the output.

Chapter 5

Results and Evaluation

In this chapter, we illustrate an application of the COWS2Prism system to a simplified video on-demand scenario. The example consists of an user, a provider and two different digital video libraries. The user invokes the provider asking for the streaming of the desired video. The provider tries to get the video from one of the video libraries and sends it back to the user. The most interesting part of the orchestration is in the implementation of the provider service and its priority policy in the invocations. In our model, the two video libraries provide different typologies of service: One only distributes high-definition content, whereas the other is more focused on low bit-rate videos. Therefore, the provider first contacts the high quality library and then, if it fails, it uses the other to get a low quality version of the same video. If both the invocations fail, the user is reported with an error message. At the end of the transaction, the provider and the two libraries are reset to their initial states. Thus, further incoming user requests can be carried out. All the communications in the model are characterised by a rate, indicating how fast an action could be. An implementation of the considered scenario in High-Lan COWS is given in Appendix B.1. It is worthwhile to observe that the actual implementation presents some additional complications with respect to the description of the scenario we presented above. This because COWS is an asynchronous calculus. Therefore, two synchronisations are needed to mimic the behaviour of a synchronous communication. An alternative implementation in the Prism language is reported in Appendix B.2.

We devote the remaining part of the chapter to the detailed description of the computations executed by the COWS2Prism system during the translation

$$
\begin{array}{llcl}
d_1 & U(p,t) & = & p!t \,|\, [z,r]\, t?z.\, p!r \\
d_2 & V(p,v,r) & = & [ch,x,k]\,(p!ch \,|\, r!v \,|\, s_V) \\
 & & & s_V = r?x.\,(\mathbf{kill}(k) \,|\, \{\!|\, V(p,v,r) \,|\!\}) \\
d_3 & P(p,p_1,p_2) & = & [e,o,x_v,x_t,x_{ch_1},x_r]\, p?x_t.\, s_{P_1} \\
 & & & s_{P_1} = [k_1]\, p_1?x_{ch_1}.\, \underbrace{(ch_{s_1}?x_v.\, s_{P_3} \,|\, o!o \,|\, o?o.\, s_{P_4})}_{s_{P_2}} \\
 & & & s_{P_3} = \{\!|\, x_t!x_v \,|\, p?x_r.\, \underbrace{(ch_{s_1}!x_r \,|\, P(p,p_1,p_2))}_{s_{P_{10}}} \,|\!\} \,|\, \mathbf{kill}(k_1) \\
 & & & s_{P_4} = [k_2,x_{ch_2}]\,(\mathbf{kill}(k_1) \,|\, \{\!|\, \underbrace{p_2?x_{ch_2}.\, s_{P_6}}_{s_{P_5}} \,|\!\}) \\
 & & & s_{P_6} = ch_{s_2}?x_v.\, s_{P_7} \,|\, o!o \,|\, o?o.\, \underbrace{(\mathbf{kill}(k_2) \,|\, \{\!|\, s_{P_{11}} \,|\!\})}_{s_{P_8}} \\
 & & & s_{P_7} = \{\!|\, x_t!x_v \,|\, p?x_r.\, \underbrace{(ch_{s_1}!x_r \,|\, ch_{s_2}!x_r \,|\, P(p,p_1,p_2))}_{s_{P_9}} \,|\!\} \,|\, \mathbf{kill}(k_2) \\
 & & & s_{P_{11}} = x_t!e \,|\, p?x_r.\, s_{P_9} \\
 & s & = & U(p,t) \,|\, P(p,p_1,p_2) \,|\, \underbrace{V(p_1,v_{high},ch_{s_1}) \,|\, V(p_2,v_{low},ch_{s_2})}_{s_{M_1}}
\end{array}
$$

Table 5.1: Video on-demand example.

of the model into a CTMC. In particular, we show the intermediate steps performed by function $\rightsquigarrow_{CTMC}$ given in Table 3.10, when the input is program $Z = \langle s, \{d_1, d_2, d_3\}, \mathcal{E}nv \rangle$. The definitions of service s and defining equations d_1, d_2 and d_3 are shown in Table 5.1. As can be seen, Z corresponds to the formal High-Lan COWS specification of of the video on-demand scenario. Note that it is a guarded, closed and homonymy free program. Therefore, it can be used as input by $\rightsquigarrow_{CTMC}$ without further modifications. We assume the environment assigns to each entity e a rate in the form μ_e. Hence, $\rho(e) = \mu_e$. As described in Table 3.10, the first instructions performed by the algorithm are the initialisation of sets $\mathbf{R}$, Q and $\mathcal{Q}$. The first is empty while the others become $\{s\}$. The main part of the algorithm is an iteration over set $\mathcal{Q}$ until the fixed point is reached. We now describe the details of every iteration.

At the beginning of the first iteration s is removed from $\mathcal{Q}$ and then all the possible stochastic reduction steps are computed. The only action program Z can perform is the synchronisation over p. This action models the request of a video (with title t) by the user to the provider. Accordingly,

$$Z = \langle s, A, \mathcal{E}nv \rangle \xrightarrow{(p \cdot \varepsilon \cdot \{t/x_t\}, [\mu_p, \mu_p], [\mu_p, \mu_p])} \langle s_1, A, \mathcal{E}nv \rangle = Z_1$$

where $A = \{d_1, d_2, d_3\}$. The details of the derivation of service s' are given below. The left branch Φ is

$$\dfrac{\dfrac{p!t \xrightarrow{(p!t, \mu_p, \mu_p)} \mathbf{0}}{p!t \,|\, [z,r]\, t?z.\, p!r \xrightarrow{(p!t, \mu_p, \mu_p)} \mathbf{0} \,|\, [z,r]\, t?z.\, p!r}\ (\textbf{par pass})}{U(p,t) \xrightarrow{(p!t, \mu_p, \mu_p)} \mathbf{0} \,|\, [z,r]\, t?z.\, p!r}\ (\textbf{ser id})$$

while the right one Υ is

$$\dfrac{\dfrac{\dfrac{p?x_t.\, s_{P_1} \xrightarrow{(p?x_t, \mu_p, \mu_p)} s_{P_1}}{[x_t]\, p?x_t.\, s_{P_1} \xrightarrow{(p?(x_t), \mu_p, \mu_p)} s_{P_1}}\ (\textbf{open req})}{[e,o,x_v,x_t,x_{ch_1},x_r]\, p?x_t.\, s_{P_1} \xrightarrow{(p?(x_t), \mu_p, \mu_p)} [e,o,x_v,x_{ch_1},x_r]\, s_{P_1}}\ (\textbf{del pass})}{P(p,p_1,p_2) \xrightarrow{(p?(x_t), \mu_p, \mu_p)} [e,o,x_v,x_{ch_1},x_r]\, s_{P_1}}\ (\textbf{ser id})$$

Hence,

$$\dfrac{\dfrac{\Phi \qquad \Upsilon}{U(p,t)\,|\,P(p,p_1,p_2) \xrightarrow{\theta} \mathbf{0}\,|\,[z,r]t?z.p!r\,|\,[e,o,x_v,x_{ch_1},x_r]s_{P_1}}\ (\textbf{close x})}{U(p,t)\,|\,P(p,p_1,p_2)\,|\,s_{M_1} \xrightarrow{\theta} \mathbf{0}\,|\,[z,r]t?z.p!r\,|\,[e,o,x_v,x_{ch_1},x_r]s_{P_1}\,|\,s_{M_1} = s_1}\ (\textbf{par conf})$$

where $\theta = (p \cdot \varepsilon \cdot \{^t/_{x_t}\}, [\mu_p, \mu_p], [\mu_p, \mu_p])$. It is worthwhile to explain that the apparent request rate is μ_p, because only the synchronising activities over endpoint p are available in s. Applying function ω (i.e. $s'_1 = \omega(s_1)$), the algorithm discovers that the resulting service is not a node of the partial transition system built so far. Therefore, it adds $s'_1 = [z,r]t?z.p!r\,|\,[e,o,x_v,x_{ch_1},x_r]s_{P_1}\,|\,s_{M_1}$ to sets Q and $\boldsymbol{Q}$. Moreover, the transition matrix becomes $\mathbf{R} = \{(s, \omega(s_1), \mu(\theta))\}$, where $\mu(\theta) = \frac{\mu_p}{\mu_p}\frac{\mu_p}{\mu_p}\min(\mu_p, \mu_p) = \mu_p$.

The second iteration of the algorithm, adds service s_2 to sets Q and $\boldsymbol{Q}$. The transition matrix is extended with element (s_1, s_2, μ_{p_1}). In what follows, we will implicitly apply function ω to improve the readability of our presentation. Service s_2 is defined as follows:

$$[z,r]t?z.p!r\,|\,[ch_h]([e,o,x_v,x_r][k_1]s_{P_2}\{^{ch}/_{x_{ch_1}}\}\,|\,[x_h,k_h](ch_{s_1}!v_{high}\,|\,s_{V_h}))\,|\,V(p_2,v_{low},ch_{s_2})$$

It represents the first communication between the provider and the high-definition video provider. It conceptually means that future communications between the two services can take place over endpoint ch_{s_1}. This is authorised by acknowledgement ch. Note that entities ch, x and k become ch_h, x_h and k_h after the decoration performed by rule (**ser id**).

The third iteration discovers two new services s_3 and s_4:

$$s_3 = [z,r]t?z.p!r\,|\,[ch_h]([e,o,x_r][k_1](s'_{P_3}\,|\,o!o\,|\,o?o.s_{P_4})\,|\,[x_h,k_h]s_{V_h})\,|\,V(p_2,v_{low},ch_{s_2})$$
$$s_4 = [z,r]t?z.p!r\,|\,[ch_h]([e,o,x_v,x_r][k_1](ch_{s_1}?x_v.s_{P_3}\,|\,s_{P_4})\,|\,s'_{V_h})\,|\,V(p_2,v_{low},ch_{s_2})$$

where $s'_{V_h} = [x_h,k_h](ch_{s_1}!v_{high}\,|\,s_{V_h})$. The first is the result of the communication over ch_{s_1}. It stands for the successful transmission of video v_{high} from the library to the provider. This is shown by the instantiation of variable x_v in $s'_{P_3} = \{\!|\, t!v_{high}|p?x_r.s_{P_{10}} \,|\!\}\,|\,\mathbf{kill}(k_1)$. The second is generated by the synchronisation over endpoint o. This channel is internally used by the provider service to model the time-out triggering the termination of the communication with the

high-definition service. At the end of the iteration, the sets become:

$$\begin{aligned} \mathcal{Q} &= \{s_3, s_4\} \\ Q &= \{s, s_1, s_2, s_3, s_4\} \\ \mathbf{R} &= \{(s, s_1, \mu_p), (s_1, s_2, \mu_{p_1}), (s_2, s_3, \mu_{ch_{s_1}}), (s_2, s_4, \mu_o)\} \end{aligned}$$

The forth iteration derives service s_5. It is generated by the execution of reduction $\xrightarrow{(\dagger, \mu_{k_1}, \mu_{k_1})}$ which is triggered by the unguarded killer activity on k_1 in service s_3. It takes the form:

$$s_5 = [z,r]\, t?z.\, p!r \,|\, [ch_h]([e,o,x_r][k_1]\{\!|\, t!v_{high} \,|\, p?x_r.\, s_{P_{10}} \,|\!\}\,|\,[x_h,k_h]s_V) \,|\, V(p_2, v_{low}, ch_{s_2})$$

The sets are updated as usual. Analogously, the fifth iteration gives s_6 starting from service s_4.

$$s_6 = [z,r]\, t?z.\, p!r \,|\, [ch_h]([e,o,x_v,x_r][k_1]([k_2, x_{ch_2}]\{\!|\, s_{P_5} \,|\!\})\,|\, s'_{V_h}) \,|\, V(p_2, v_{low}, ch_{s_2})$$

The new configuration of the sets is

$$\begin{aligned} \mathcal{Q} &= \{s_5, s_6\} \\ Q &= \{s, s_1, s_2, s_3, s_4, s_5, s_6\} \\ \mathbf{R} &= \{(s, s_1, \mu_p), (s_1, s_2, \mu_{p_1}), (s_2, s_3, \mu_{ch_{s_1}}), (s_2, s_4, \mu_o), (s_3, s_5, \mu_{k_1}), (s_4, s_6, \mu_{k_1})\} = \mathbf{R}_1 \end{aligned}$$

When the node corresponding to service s_5 is considered, a communication over t can take place. This synchronisation models the transmission of the video from the provider to the user. The complete computation from s_5 is

$$s_5 \xrightarrow{(t \cdot \varepsilon \cdot \{v_{high}/z\}, \mu_t, \mu_t)} s_7 \xrightarrow{(p \cdot \varepsilon \cdot \{r/x_r\}, \mu_p, \mu_p)} s_8 \xrightarrow{(ch_{s_1} \cdot \varepsilon \cdot \{r/x_h\}, \mu_{ch_{s_1}}, \mu_{ch_{s_1}})} s_9 \xrightarrow{(\dagger, \mu_{k_h}, \mu_{k_h})} s_{10}$$

where services s_7, s_8, s_9 and s_{10} are defined in Table 5.2. The state of the sets after these five iterations is

$$\begin{aligned} \mathcal{Q} &= \{s_6\} \\ Q &= \{s, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}\} \\ \mathbf{R} &= \mathbf{R}_1 \cup \{(s_5, s_7, \mu_t), (s_7, s_8, \mu_p), (s_8, s_9, \mu_{ch_{s_1}}), (s_9, s_{10}, \mu_{k_h})\} = \mathbf{R}_2 \end{aligned}$$

Starting from service s_6, a communication with the second video library is established. This is shown by the following computational step:

$$s_6 \xrightarrow{(p_2 \cdot \varepsilon \cdot \{ch_l/x_{ch_2}\}, \mu_{p_2}, \mu_{p_2})} s_{11}$$

$$
\begin{aligned}
s_7 &= [r]p!r \mid [ch_h]([e,o,x_r][k_1]\{\!|p?x_r.s_{P_{10}}|\!\} \mid [x_h,k_h]s_V) \mid V(p_2,v_{low},ch_{s_2}) \\
s_8 &= [r,ch_h]([e,o][k_1]\{\!|s_{P_{10}}|\!\} \mid [x_h,k_h]s_V) \mid V(p_2,v_{low},ch_{s_2}) \\
s_9 &= [r,ch_h]([e,o][k_1]\{\!|P(p,p_1,p_2)|\!\} \mid [k_h](\mathbf{kill}(k_h) \mid \{\!|V(p_1,v_{high},ch_{s_1})|\!\})) \\
&\quad \mid V(p_2,v_{low},ch_{s_2}) \\
s_{10} &= [e,r,ch_h]([o][k_1]\{\!|P(p,p_1,p_2)|\!\} \mid [k_h]\{\!|V(p_1,v_{high},ch_{s_1})|\!\}) \mid V(p_2,v_{low},ch_{s_2})
\end{aligned}
$$

Table 5.2: Services s_7, s_8, s_9 and s_{10}.

At this stage, the iteration detects a branching point in the transition system. The left branch is the sequence of actions taking place when the transmission between the provider and the second video library is successful. It gives rise to the computation below:

$$
\begin{aligned}
s_{11} &\xrightarrow{(ch_{s_2}\cdot\varepsilon\cdot\{^{v_{low}}/_{x_v}\},\mu_{ch_{s_2}},\mu_{ch_{s_2}})} s_{12} \xrightarrow{(\dagger,\mu_{k_2},\mu_{k_2})} s_{13} \\
s_{13} &\xrightarrow{(t\cdot\varepsilon\cdot\{^{v_{low}}/_{z}\},\mu_t,\mu_t)} s_{14} \xrightarrow{(p\cdot\varepsilon\cdot\{^{r}/_{x_l}\},\mu_p,\mu_p)} s_{15}
\end{aligned}
$$

The right branch represents the operations performed when a failure happens. The computation is

$$
\begin{aligned}
s_{11} &\xrightarrow{(o\cdot\varepsilon\cdot\varepsilon,\mu_o,\mu_o)} s_{16} \xrightarrow{(\dagger,\mu_{k_2},\mu_{k_2})} s_{17} \\
s_{17} &\xrightarrow{(t\cdot\varepsilon\cdot\{^{e}/_{z}\},\mu_t,\mu_t)} s_{18} \xrightarrow{(p\cdot\varepsilon\cdot\{^{r}/_{x_l}\},\mu_p,\mu_p)} s_{19}
\end{aligned}
$$

Services s_i, with $11 \leq i \leq 19$ are given in Table 5.3. The updated sets are

$$
\begin{aligned}
\boldsymbol{Q} &= \{s_{15}, s_{19}\} \\
Q &= \{s\} \cup \{s_i \mid 1 \leq i \leq 19\} \\
\mathbf{R} &= \mathbf{R_2} \cup \mathbf{R_2'} \cup \mathbf{R_2''} = \mathbf{R_3} \\
\mathbf{R_2'} &= \{(s_6,s_{11},\mu_{p_2}),(s_{11},s_{12},\mu_{ch_{s_2}}),(s_{12},s_{13},\mu_{k_2}),(s_{13},s_{14},\mu_t),(s_{14},s_{15},\mu_t)\} \\
\mathbf{R_2''} &= \{(s_{11},s_{16},\mu_o),(s_{16},s_{17},\mu_{k_2}),(s_{17},s_{18},\mu_t),(s_{18},s_{19},\mu_p)\}
\end{aligned}
$$

The remaining computational steps model the termination of the invoked libraries. Both services s_{15} and s_{19} can perform a sequence of actions similar to the computation from s_8 to s_{10}. In this case, however, both the video libraries have to be terminated. Hence, all the possible computations are formed by four reductions. For instance, if the high-definition library is terminated before

$$
\begin{aligned}
s_{11} &= [z,r]\,t?z.\,p!r \mid [ch_h, ch_l]\,([e,o,x_v,x_r]\,[k_1]\,[k_2]\,\{\!|\, s'_{P_6} \,|\!\}\mid s'_{V_h} \mid [x_l,k_l]\,(ch_{s_2}!v_{low} \mid s_{V_l}))\\
s_{12} &= [z,r]\,t?z.\,p!r \mid [ch_h, ch_l]\\
&\quad ([e,o,x_r]\,[k_1]\,[k_2]\,\{\!|\, s_{P_7} \mid o!o \mid o?o.\,(\mathbf{kill}(k_2) \mid \{\!|\, s_{P_{11}} \,|\!\})\,|\!\}\mid s'_{V_h} \mid [x_l,k_l]\,s_{V_l})\\
s_{13} &= [z,r]\,t?z.\,p!r \mid [ch_h, ch_l]\,([e,o,x_r]\,[k_1]\,[k_2]\,\{\!|\, t!v_{low} \mid p?x_r.\,s_{P_9} \,|\!\}\mid s'_{V_h} \mid [x_l,k_l]\,s_{V_l})\\
s_{14} &= [r]\,p!r \mid [ch_h, ch_l]\,([e,o,x_r]\,[k_1]\,[k_2]\,\{\!|\, p?x_r.\,s_{P_9} \,|\!\}\mid s'_{V_h} \mid [x_l,k_l]\,s_{V_l})\\
s_{15} &= [r, ch_h, ch_l]\,([e,o,x_r]\,[k_1]\,[k_2]\,\{\!|\, s_{P_9} \,|\!\}\mid s'_{V_h} \mid [x_l,k_l]\,s_{V_l})\\
s_{16} &= [z,r]\,t?z.\,p!r \mid [ch_h, ch_l]\\
&\quad ([e,o,x_v,x_r]\,[k_1]\,[k_2]\,\{\!|\, ch_{s_2}?x_v.\,s_{P_7} \mid s_{P_8} \,|\!\}\mid s'_{V_h} \mid [x_l,k_l]\,(ch_{s_2}!v_{low} \mid s_{V_l}))\\
s_{17} &= [z,r]\,t?z.\,p!r \mid [ch_h, ch_l]\,([e,o,x_v,x_r]\,[k_1]\,[k_2]\,\{\!|\, s_{P_{11}} \,|\!\}\mid s'_{V_h} \mid [x_l,k_l]\,(ch_{s_2}!v_{low} \mid s_{V_l}))\\
s_{18} &= [r]\,p!r \mid [ch_h, ch_l]\,([e,o,x_v,x_r]\,[k_1]\,[k_2]\,\{\!|\, p?x_r.\,s_{P_9} \,|\!\}\mid s'_{V_h} \mid [x_l,k_l]\,(ch_{s_2}!v_{low} \mid s_{V_l}))\\
s_{19} &= [r, ch_h, ch_l]\,([e,o,x_v,x_r]\,[k_1]\,[k_2]\,\{\!|\, s_{P_9} \,|\!\}\mid s'_{V_h} \mid [x_l,k_l]\,(ch_{s_2}!v_{low} \mid s_{V_l}))
\end{aligned}
$$

Table 5.3: Services s_i, with $11 \leq i \leq 19$.

the other one, a possible computation from s_{15} is

$$
\begin{aligned}
s_{15} &\xrightarrow{(ch_{s_1}\cdot\varepsilon\cdot\{r/x_h\},\mu_{ch_{s_1}},\mu_{ch_{s_1}})} s_{20} \xrightarrow{(\dagger,\mu_{k_1},\mu_{k_1})} s_{21}\\
s_{21} &\xrightarrow{(ch_{s_2}\cdot\varepsilon\cdot\{r/x_l\},\mu_{ch_{s_2}},\mu_{ch_{s_2}})} s_{22} \xrightarrow{(\dagger,\mu_{k_2},\mu_{k_2})} s_{10}
\end{aligned}
$$

Note that all the computations converge to service s_{10}. As a matter of fact, this is the last service being added to set Q. The final output of function $\rightsquigarrow_{CTMC}$ is:

$$
\begin{aligned}
\mathbf{R} &= \mathbf{R}_3 \cup \mathbf{R}_{k_1} \cup \mathbf{R}_{k_2} \cup \mathbf{R}_{ch_{s_1}} \cup \mathbf{R}_{ch_{s_2}}\\
\mathbf{R}_{k_1} &= \{(s_{20}, s_{21}, \mu_{k_1}), (s_{24}, s_{27}, \mu_{k_1}), (s_{28}, s_{10}, \mu_{k_1})\}\\
\mathbf{R}_{k_2} &= \{(s_{25}, s_{26}, \mu_{k_2}), (s_{23}, s_{26}, \mu_{k_2}), (s_{29}, s_{10}, \mu_{k_2})(s_{22}, s_{10}, \mu_{ch_{s_2}})\}\\
\mathbf{R}_{ch_{s_1}} &= \{(s_{15}, s_{20}, \mu_{ch_{s_1}}), (s_{19}, s_{24}, \mu_{ch_{s_1}}), (s_{26}, s_{28}, \mu_{ch_{s_1}})\}\\
\mathbf{R}_{ch_{s_2}} &= \{(s_{15}, s_{25}, \mu_{ch_{s_2}}), (s_{19}, s_{23}, \mu_{ch_{s_2}}), (s_{27}, s_{29}, \mu_{ch_{s_2}}), (s_{21}, s_{22}, \mu_{ch_{s_2}})\}
\end{aligned}
$$

The CTMC has been generated by Prism using the output of the `cows2prism` program on source file `video.cow` (see Appendix B.1). The same result is obtained by running the native Prism model (see Appendix B.2). The Markov chain has 30 states and 34 transitions. Note that in both cases, an auto transition is automatically added by Prism to the final state s_{10}.

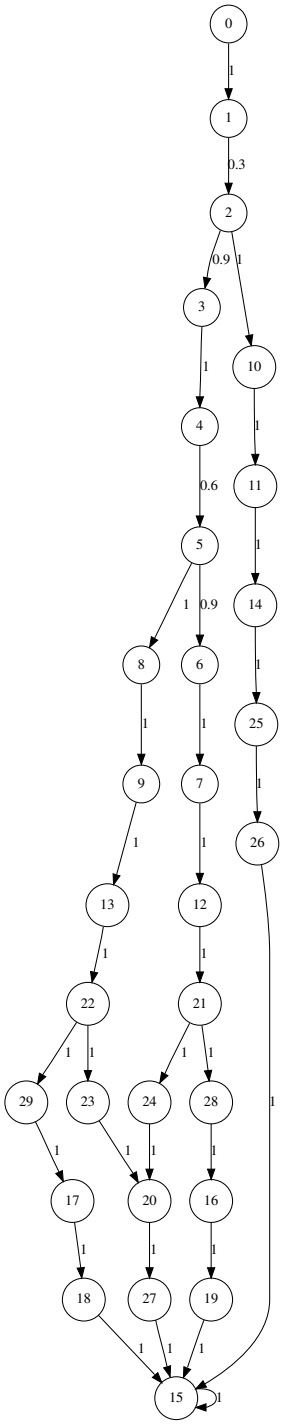

Figure 5.1: CTMC corresponding to the video on-demand example.

Chapter 6

Conclusion and Future Work

In this thesis we presented the COWS2Prism system, a compiler for COWS into Prism. We formally introduced its design and we explained the details regarding its implementation. Our tool supports orchestration models specified in the High-Lan COWS language, an untyped version of the calculus presented in (Lapadula et al., 2007a). The output is the specification of a continuous-time Markov chain written in the Prism format.

In Chapter 2 we gave a broad overview of the background and the related work. In particular, we described the service oriented computing paradigm and the most important concepts related to the web service technology such as choreography and orchestration. Moreover, we surveyed various computational models for concurrency, especially the ones historically used in the literature for the formalisation of communicating systems and more recently orchestration. We mainly focused our attention on stochastic process calculi. In the course of their presentation, the concepts of stochastic rate, synchronisation and race condition were introduced. Finally, an operational semantics and a stochastic extension of COWS were reported.

In Chapter 3 a formal description of the COWS2Prism system was introduced. We analysed all the sub-components (e.g. fresh names generator, environment, type system, ...) in great detail and we showed how they are jointly used in the whole system. Several examples were provided in order to highlight the crucial choices in our design. A considerable part of the chapter is devoted to the explanation of the differences between High-Lan COWS and COWS.

In Chapter 4 we described the implementation of the COWS2Prism system

in the functional language OCaml. Details regarding the data structures and the algorithms we adopted in the various compilation units were reported. Moreover, we mentioned the fact that `ocamllex` and `ocamlyacc` were used for the generation of the lexer and parser, respectively.

In Chapter 5 we showed the steps performed by the COWS2Prism system for the generation of the CTMC corresponding to a High-Lan COWS model. The example we chose formalises a simplified video on-demand scenario. Additionally, a Prism program generating the same CTMC is given.

In the course of the preparation we tried to develop a fully compositional encoding from COWS into other formalisms to try to overcome the state explosion problem. One candidate was PEPA since a complete set of analysis tools is provided with it. However, such a translation was impossible to find preserving the most interesting features of COWS e.g. killer activity and protection. A possible improvement of the COWS2Prism system could be to support other output formats, for other tool. As a matter of fact, the translation engine needs only few modifications to handle this. Another feature is the possibility to add simulation instead of exploring the entire state space. A more challenging development is to implement a preporocessor from a synchronous version of COWS into the standard asynchronous COWS considered in this thesis. As a matter of fact, we realised that during the modelling a lot of effort is employed for the specifications of actions that conceptually are atomic. We have to investigate the feasibility of this.

Appendix A

High-Lan COWS Grammar

program	::=	*rates_list opt_def_list* **in** *service*
rates_list	::=	*default* | *lambda_list default*
default	::=	**baserate** : *FLOAT* ;
lambda_list	::=	*rate* | *rate lambda_list*
rate	::=	**rate** *EID* : *FLOAT* ;
opt_def_list	::=	ε | *def_list* **in**
def_list	::=	*def* | *def* ; *def_list*
def	::=	**let** *SID* (*opt_names_list*) = *service*
service	::=	*base_service* | *complex_service*
base_service	::=	*EID* ! *EID* | { | *service* | } | **kill** (*EID*)
		| *base_guard* | *SID* (*opt_names_list*)
		| [*eid_list*] *base_service*
		| [*eid_list*] (*complex_service*)
complex_service	::=	*parallel_list* | *complex_guard*
base_guard	::=	**0** | *EID* ? *EID* . *base_service*
		| *EID* ? . (*complex_service*)
complex_guard	::=	*sum_list*
parallel_list	::=	*parallel* | *parallel* | *parallel_list*
		| *parallel* | *base_service* | *parallel* | (*complex_guard*)
parallel	::=	*base_service* | *base_service*
		| (*complex_guard*) | (*complex_guard*)
		| (*complex_guard*) | *base_service*
		| *base_service* | (*complex_guard*)
sum_list	::=	*sum* | *sum* + *sum_list* | *sum* + *base_guard*
sum	::=	*base_guard* + *base_guard*
opt_names_list	::=	ε | *names_list*
eid_list	::=	*EID* | *EID* , *eid_list*
names_list	::=	*EID* | *EID* , *names_list*

A.1 Regular Expressions

$$
\begin{aligned}
\mathit{SID} &= [\text{'A'} - \text{'Z'}]\ [\text{'a'} - \text{'z'}\ \text{'A'} - \text{'Z'}\ \text{'0'} - \text{'9'}\ \text{'\_'}]^{*} \\
\mathit{EID} &= [\text{'a'} - \text{'z'}]\ [\text{'a'} - \text{'z'}\ \text{'A'} - \text{'Z'}\ \text{'0'} - \text{'9'}\ \text{'\_'}]^{*} \\
\mathit{FLOAT} &= \Big([\text{'0'} - \text{'9'}]^{+} \ \Big|\ \big([\text{'0'} - \text{'9'}]^{+}\ \text{'.'}\ [\text{'0'} - \text{'9'}]^{+}\big)\Big) \\
&\quad \Big([\text{'E'}\ \text{'e'}]\ [\text{'+'}\ \text{'-'}]?\ [\text{'0'} - \text{'9'}]^{+}\Big)? \\
&\quad \Big|\ \text{"inf"}
\end{aligned}
$$

A.2 Notes

Although production rules for nonterminal symbols *eid_list* and *names_list* are identical, different semantic actions are associated with them. In the first case, each *EID* in the list is converted into an OCaml value of type `entity` in the form `Ent1(EntV(Var(_)))` (i.e. it is considered a variable). On the other hand, the second rule produces a `name` list.

Appendix B

Video on-demand example source files

B.1 High-Lan COWS source file

```
rate p: 1;
rate x_ch1: 0.3;
rate x_ch2: 0.6;
rate o: 0.9;
baserate: 1;

let U(p,t) = p!t | [z,r]t?z.p!r;
let V(p,v,r) = [ch,x,k]( p!ch | r!v | r?x.
                  ( kill(k) | {|V(p,v,r)|}) );
let P(p,p1,p2) =
  [e,o,x_v,x_t,x_ch1,x_r] p?x_t.[k1]p1?x_ch1.(ch_s1?x_v.
    ( {|x_t!x_v | p?x_r.( ch_s1!x_r | P(p,p1,p2) )|} | kill(k1) )
        | o!o | o?o.
          [k2,x_ch2]( kill(k1) | {| p2?x_ch2.(ch_s2?x_v.
            ( {|x_t!x_v | p?x_r.
                ( ch_s1!x_r | ch_s2!x_r | P(p,p1,p2) )|}
              | kill(k2) )
            | o!o | o?o.( kill(k2) | {|x_t!e | p?x_r.
              ( ch_s1!x_r | ch_s2!x_r | P(p,p1,p2) )|}) )|} ) )
in
U(p,t) | P(p,p1,p2) | V(p1,v_h,ch_s1) | V(p2,v_l,ch_s2)
```

B.2 Prism source file

```
ctmc

const double mu = 1;
const double mu_1 = 0.3;
const double mu_2 = 0.6;
const double mu_o = 0.9;

module User
u_state:[0..3] init 0;
[request_title]  u_state=0 -> mu:(u_state'=1);
[wait_video]     u_state=1 -> mu:(u_state'=2);
[video_received] u_state=2 -> mu:(u_state'=3);
endmodule

module Video_H
v_state_h:[0..4] init 0;
[wait_title_v_h] v_state_h=0 -> mu_1:(v_state_h'=1);
[send_v_h]       v_state_h=1 -> mu_x:(v_state_h'=2);
[wait_kill_h]    v_state_h=2&(v_state_l!=3&v_state_l!=4)
                 -> s1:(v_state_h'=3);
[wait_kill_h]    v_state_h=1&(v_state_l!=3&v_state_l!=4)
                 -> s1:(v_state_h'=4);
[kill_v_h]       v_state_h=3|v_state_h=4 -> k:(v_state_h'=0);
endmodule

module Video_L
v_state_l:[0..4] init 0;
[wait_title_v_l] v_state_l=0 -> mu_2:(v_state_l'=1);
[send_v_l]       v_state_l=1 -> mu_x:(v_state_l'=2);
[wait_kill_l]    v_state_l=2&(v_state_h!=3&v_state_h!=4)
                 -> s2:(v_state_l'=3);
[wait_kill_l]    v_state_l=1&(v_state_h!=3&v_state_h!=4)
                 -> s2:(v_state_l'=4);
[kill_v_l]       v_state_l=3|v_state_l=4 -> k:(v_state_l'=0);
endmodule
```

```
module Provider
p_state:[0..17] init 0;
[request_title]  p_state=0 -> 1:(p_state'=1);
[wait_title_v_h] p_state=1 -> 1:(p_state'=2);
[send_v_h]       p_state=2 -> 1:(p_state'=3);
[kill_1]         p_state=3 -> k_1:(p_state'=4);
[wait_video]     p_state=4 -> 1:(p_state'=5);
[video_received] p_state=5 -> 1:(p_state'=6);
[wait_kill_h]    p_state=6 -> 1:(p_state'=0);
[kill_o1]        p_state=2 -> mu_o:(p_state'=7);
[]               p_state=7 -> k_1:(p_state'=8);
[wait_title_v_l] p_state=8 -> 1:(p_state'=9);
[send_v_l]       p_state=9 -> 1:(p_state'=10);
[kill_2]         p_state=10 -> k_2:(p_state'=11);
[wait_video]     p_state=11 -> 1:(p_state'=12);
[video_received] p_state=12 -> 1:(p_state'=13);
[wait_kill_h]    p_state=13 -> 1:(p_state'=0);
[wait_kill_l]    p_state=13 -> 1:(p_state'=0);
[kill_o2]        p_state=9 -> mu_o:(p_state'=14);
[]               p_state=14 -> k_2:(p_state'=15);
[wait_video]     p_state=15 -> 1:(p_state'=16);
[video_received] p_state=16 -> 1:(p_state'=17);
[wait_kill_h]    p_state=17 -> 1:(p_state'=0);
[wait_kill_l]    p_state=17 -> 1:(p_state'=0);
[wait_kill_h]    p_state=0 -> 1:(p_state'=0);
[wait_kill_l]    p_state=0 -> 1:(p_state'=0);
endmodule
```

Bibliography

Abadi, M. and Gordon, A. (1997). A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. (2003). BPEL (Business Process Execution Language for Web Services) specifications version 1.1. `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`.

Cardelli, L. and Gordon, A. (1998). Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany.

Cerami, E. (2002). *Web Services Essentials*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Deng, X. and Papadimitriou, C. (1990). Exploring an unknown graph. *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 355–361 vol. 1.

Deng, Y., Palamidessi, C., and Pang, J. (2005). Compositional reasoning for probabilistic finite-state behaviors. In *In Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday, LNCS 3838*, pages 309–337. Springer.

Fournet, C. and Gonthier, G. (1996). The reflexive CHAM and the Join-calculus. In *POPL*, pages 372–385.

Gay, S. J. (1993). A sort inference algorithm for the polyadic π-calculus. In *POPL*

'93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 429–438, New York, NY, USA. ACM.

Hillston, J. and Thomas, N. (1998). A syntactical analysis of reversible PEPA models.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Kwiatkowska, M., Norman, G., and Parker, D. (2002). Prism: Probabilistic symbolic model checker. pages 200–204. Springer.

Lapadula, A., Pugliese, R., and Tiezzi, F. (2006). A wsdl-based type system for ws-bpel. In *Proc. of COORDINATION'06*, volume 4038 of *Lecture Notes in Computer Science*, pages 145–163. Springer.

Lapadula, A., Pugliese, R., and Tiezzi, F. (2007a). 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.

Lapadula, A., Pugliese, R., and Tiezzi, F. (2007b). C⊙WS: A timed service-oriented calculus. In *Proc. of 4th International Colloquium on Theoretical Aspects of Computing (ICTAC'07)*, volume 4711 of *Lecture Notes in Computer Science*, pages 275–290. Springer.

Mazzanti, F. (2007). CMC: an on-the-fly model checker and interpreter for COWS. `http://fmt.isti.cnr.it/cmc/`.

Milner, R. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer.

Milner, R. (1992). Functions as Processes. *Mathematical Structures in Computer Science*, 2(2):119–141.

Milner, R. (1999). *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, Cambridge, UK.

Milner, R., Parrow, J., and Walker., D. (1992). A Calculus of Mobile Processes, Part I and II. *Information and Computation*, 100(1):1–77.

Misra, J. and Cook, W. R. (2006). Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*.

Parker, D., Norman, G., Kwiatkowska, M., and Kattenbelt, M. (2008). Prism (Probabilistic Symbolic Model Checker). `http://www.prismmodelchecker.org/`.

Parrow, J. and Victor, B. (1998). The fusion calculus: Expressiveness and symmetry in mobile processes. In *Logic in Computer Science*, pages 176–185.

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ.

Prandi, D. and Quaglia, P. (2007). Stochastic COWS. In *Proc. 5th International Conference on Service Oriented Computing, ICSOC '07*, volume 4749 of *LNCS*.

Priami, C. (1995). Stochastic π-calculus. *Comput. J.*, 38(7):578–589.

Reisig, W. (1986). *Petrinetze*. Springer, 2 edition.

Sangiorgi, D. and Walker, D. (2001). *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press.

Schweitzer, P. (1990). *A survey of aggregation-disaggregation in large Markov chains*, chapter 4, pages 63–88. Marcel Dekker.

Thatte, S. (2001). XLANG: Web Services for Business Process Design. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`.

van der Aalst, W. and Hofstede, A. (2002). YAWL: Yet Another Workflow Language.